Karlsruhe Institute of Technology

Optimisation of LHCb Applications for Multi- and Manycore Job Submission

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

bei der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
DISSERTATION

von

M.Sc. Nathalie Rauschmayr

aus Schweinfurt

Datum der mündlichen Prüfung:	24.10.2014
Referent:	Prof. Dr. Achim Streit
Korreferent:	Prof. Dr. Ricardo Graciani Díaz
Betreuer am CERN:	Dr. Philippe Charpentier

Abstract

The Worldwide LHC Computing Grid (WLCG) is the largest Computing Grid and is used by all Large Hadron Collider experiments in order to process their recorded data. It provides approximately 400k cores and storages. Nowadays, most of the resources consist of multi- and manycore processors. Conditions at the Large Hadron Collider experiments will change and much larger workloads and jobs consuming more memory are expected in future. This has lead to a shift of paradigm which focuses on executing jobs as multiprocessor tasks in order to use multi- and manycore processors more efficiently. All experiments at CERN are currently investigating how such computing resources can be used more efficiently in terms of memory requirements and handling of concurrency. Until now, there are still many unsolved issues regarding software, scheduling, CPU accounting, task queues, which need to be solved by grid sites and experiments.

This thesis develops a systematic approach to optimise the software of the LHCb experiment [84] for multi- and manycore processors. This implies optimisation at the levels which are under control by LHCb's Workload Management System. First, this thesis analyses limitations of software and how to improve it by using intrusive and non intrusive techniques. In this scope, it discusses the applicability of parallelization concepts regarding High Energy Physics software. A parallel prototype is evaluated within extensive benchmarks. These include measuring memory reduction, runtime, hardware performance counters as well as tests on correctness of the output of data. Tools for automatic memory deduplication and compression are evaluated in the context of non intrusive optimisation. It also discusses how the change from 32- to 64-bit impacted LHCb software and how it can profit from the new platform model x32-ABI.

Executing jobs as parallel tasks must be also supported by the grid sites. Until now, it is an unsolved issue whether scheduling of multiprocessor tasks is subject to the Virtual Organization (VO) [100] or the grid site. Since the Virtual Organization has the insight into job parameters and past workloads, the thesis proposes a moldable job scheduler which optimises the job throughput of VO's task queues. Moldability implies that a job can be executed with an arbitrary number of processes and it is up to the scheduler to define the best value. Therefore, the thesis defines the scheduling problem which meets the requirements of LHCb jobs and evaluates different local search methods. The Worldwide LHC Computing Grid is a highly dynamic system which offers a large variety of different computing resources. Additionally, experiment conditions and software often change which have a significant impact on generated workloads. It is important, that a scheduler learns over time these changing conditions. Consequently, the thesis undertakes a detailed analysis of LHCb workloads and figures out how job requirements can be better predicted by a supervised learning algorithm.

Zusammenfassung

Das Worldwide LHC Computing Grid ist das größte Computing Grid und wird von allen Large Hadron Collider Experimenten genutzt, um deren Datenmengen zu bearbeiten. Es stellt ungefähr 400k Rechenkerne und Speicherressourcen zur Verfügung und besteht heutzutage hauptsächlich aus Multi- und Manycore Prozessoren. Die Bedingungen am Large Hadron Collider werden sich ändern und in Zukunft werden wesentlich größerer Workloads und speicherlastige Jobs erwartet. Das hat zu einem Paradigmenwechsel geführt, bei dem es darum geht, Jobs als parallele Tasks auszuführen, um Multi- und Manycore Prozessoren besser nutzen zu können. Alle Experimente am CERN untersuchen deshalb, wie solche Ressourcen bezüglich Speicher und konkurrierenden Zugriffe besser genutzt werden können. Bis heute gibt es noch viele ungelöste Probleme bezüglich Software, Scheduling, CPU Zeit Abrechnung, Prozess Warteschlangen. Diese Probleme müssen von Seiten der Ressource Provider sowie von den Experimenten gelöst werden.

Diese Thesis entwickelt einen systematischen Ansatz um die Software des LHCb Experiments [84] für Multi- und Manycore Prozessoren zu optimieren. Das impliziert, dass Optimierung auf allen Leveln durchgeführt werden muss, die unter der Kontrolle von LHCb's Workload Management System stehen. Zunächst wird die Thesis die Softwarelimits evaluieren und aufzeigen, wie Software durch intrusive und nicht intrusive Methoden optimiert werden kann. Im Rahmen dessen wird diskutiert, wie verschiedene Parallelisierungsmodelle in Software für Hochenergiephysik angewendet werden können. Es wird ein paralleler Prototyp vorgestellt, der in zahlreichen Benchmark-Tests evaluiert wird. Diese Tests beinhalten Messungen für Speicherreduktion, Laufzeit, Hardware Performance Counter so wie Tests auf Korrektheit der erzeugten Resultate. Im Kontext der nicht intrusiven Optimierung wird Speicherdeduplizierung und Komprimierung evaluiert. Es wird ebenfalls diskutiert, inwiefern sich der Wechsel von 32- auf 64-bit auf LHCb Software auswirkt und wie jene von der x32-ABI profitieren kann.

Die Ausführung von Jobs als parallele Tasks muss von Seiten der Ressource Provider unterstützt werden. Bis heute ist es ein ungelöstes Problem, ob das Scheduling von solchen Tasks von der VO (Virtual Organization) [100] oder dem Ressource Provider übernommen werden soll. Da die VO Kenntnis über Jobparameter und vorherige Workloads hat, schlägt die Thesis einen *moldable* Jobscheduler vor, der die Prozess Warteschlangen einer VO hinsichtlich Jobdurchsatz optimiert. *Moldable* bedeutet, dass ein Job mit einer beliebigen Anzahl an Prozessen ausgeführt werden kann und es ist Aufgabe des Schedulers die beste Anzahl zu definieren. Hierfür, definiert die Thesis das Schedulingproblem, welches die Anforderungen von LHCb Jobs repräsentiert, und evaluiert verschiedene lokale Suchmethoden. Das Worldwide LHC Computing Grid ist ein hoch dynamisches System, welches aus vielen verschiedenen Computersystemen besteht. Zusätzlich ändern sich häufig die Bedingungen an den Experimenten sowie die Software, was schließlich eine große Auswirkung auf die generierten Workloads hat. Es ist wichtig, dass ein Scheduler in der Lage ist, solche Veränderungen zu erkennen und über die Zeit zu lernen. Deshalb werden in dieser Thesis LHCb Workloads aus den letzten Jahren genau analysiert und es wird untersucht, wie Jobanforderungen mit einem Lernalgorithmus besser vorhergesagt werden können.

Acknowledgement

I would like to thank Philippe Charpentier and my professor Achim Streit for supervising my thesis. I also would like to thank Ricardo Graciani Díaz for evaluating my thesis as second reviewer. Special thanks goes to my colleagues from LHCb, CERN Openlab and CERN PH-SFT for their support and the fruitful discussions that I had with them. In this context, I am particularly grateful to Mario Úbeda García, Zoltán Máthé, Ben Couturier, Marco Clemencic, Rainer Schwemmer, Philippe Charpentier, Marco Cattaneo, Gloria Corti, Stefan Roiser, Stefan Lohn, Andrzej Nowak, Jakob Blomer, Axel Naumann and Pere Mató Vila. I also would like to thank my friends who encouraged me to not give up during the last three years.

List of Publications

Parts of the thesis have been presented and published at different scientific conferences and workshops. The following list shows the publications:

- N. Rauschmayr and A. Streit. Reducing the Memory Footprint of Parallel Applications with KSM. In: Facing the Multicore-Challenge III (2012), R. Keller, D. Kramer, J.-P. Weiss (Eds.), LNCS 7686, Springer, 2013, ISBN 978-3-642-35892-0, pages 48-59, DOI: 10.1007/978-3-642-35893-7_5

- N. Rauschmayr and A. Streit. Evaluation of x32-ABI in the Context of LHC Applications. In: International Conference on Computational Science (ICCS 2013). Procedia Computer Science: Volume 18, 2013, pages 2233-2240, ISSN 1877-0509, DOI: 10.1016/j.procs.2013.05.394

- Nathalie Rauschmayr and Achim Streit. Evaluating Moldability of LHCb Jobs for Multicore Job Submission. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013, pages 519-525, Sept 2013

- N Rauschmayr and A Streit. Preparing the Gaudi Framework and the Dirac WMS for Multicore Job Submission. Journal of Physics: Conference Series, 513(5):052029, 2014

- Mario Úbeda García and Víctor Méndez Muñoz and Federico Stagni and Baptiste Cabarrou and Nathalie Rauschmayr and Philippe Charpentier and Joel Closier. Integration of Cloud Resources in the LHCb Distributed Computing. Journal of Physics: Conference Series, 513(3):032099, 2014

Since the High Energy Physics (HEP) community is a quite large and global community, regular meetings and workshops are organized in order to discuss new results and ideas. The work has been presented in the following HEP workshops:

- GaudiMP - performance and KSM measurements. July 2012. Presented at Concurrency Forum, Geneva, Switzerland. `https://indico.cern.ch/event/198122`

- Optimisation of Gaudi applications for multi- and many-core CPUs. October 2012. Presented at 49th Analysis and Software Week, Geneva, Switzerland. `https://indico.cern.ch/event/159521/`

- Evaluation of x32-ABI in the context of CERN applications. October 2012. Presented at Concurrency Forum, Geneva, Switzerland. `http://indico.cern.ch/event/214319`

- Optimizing memory consumption within GaudiMP. February 2013. Presented at Annual Concurreny Forum, Chicago, United States. `https://indico.fnal.gov/getFile.py/access?contribId=0&sessionId=7&resId=0&materialId=slides&confId=6138`

- Optimizing memory consumption within GaudiMP. February 2013. Presented at 50th Analysis and Software Week, Geneva, Switzerland. https://indico.cern.ch/event/201535/
- ROOT and x32-ABI. March 2013. Presented at ROOT Users Workshop, Saas-Fee, Switzerland. https://indico.cern.ch/event/217511/contribution/26/material/slides/0.pdf
- Status of CPU concurrency R&D. May 2013. Presented at LHCb Computing Workshop, Geneva, Switzerland. https://indico.cern.ch/event/236650/
- Scheduling of Multicore Jobs. May 2014. Presented at HEPiX Spring Workshop 2014, Annecy-le-Vieux, France. http://indico.cern.ch/event/274555/session/14/contribution/21
- Optimization of LHCb Applications for Multi- and Manycore Job Submission. June 2014. Presented at CERN-IT-SDC White Area Meeting, Geneva, Switzerland. https://indico.cern.ch/category/2254/

Contents

1 Introduction 1
- 1.1 Research Questions 3
- 1.2 Contributions 4
- 1.3 Structure of the Thesis 4

2 Scientific Computing in the LHCb Experiment 7
- 2.1 The Large Hadron Collider 7
- 2.2 LHCb Detector and Physics 8
- 2.3 Data Flow 9
  - 2.3.1 Reconstruction 10
  - 2.3.2 Stripping 10
  - 2.3.3 Analysis 11
  - 2.3.4 Simulation 11
- 2.4 Software Framework Gaudi 11
- 2.5 Distributed Computing Infrastructure 12
  - 2.5.1 Grid Computing 12
    - 2.5.1.1 Worldwide LHC Computing Grid 13
    - 2.5.1.2 Workload and Data Management System DIRAC 16
  - 2.5.2 Cloud Computing 18
  - 2.5.3 Volunteer Computing 19
- 2.6 Workload Scale 19

3 Related Work 23
- 3.1 High Energy Physics Applications in the Multi- and Manycore Era 23
- 3.2 Memory Deduplication 26
- 3.3 Job Scheduling 26
  - 3.3.1 Optimising Scheduling Performance by Using Moldability of Jobs 27
  - 3.3.2 Runtime Prediction 28
  - 3.3.3 Backfilling 29

4 Problem Description in Detail 31
- 4.1 Experiment Upgrade 31
- 4.2 Current Job Model 32
- 4.3 Proposed Solutions 33

5 Optimisation with Non Intrusive Techniques 37
- 5.1 Memory Deduplication 37
- 5.2 The x32 Application Binary Interface 41
- 5.3 Memory Compression 45
- 5.4 Summary 47

6 Optimisation with Intrusive Techniques 49
6.1 Parallelization Principles . . . 49
6.2 Multiprocessing Approach . . . 51
6.3 Multithreaded Approach . . . 53
6.4 Perspectives of the Different Concepts . . . 54
6.5 Evaluation of a Parallel Software Framework Prototype . . . 54
6.5.1 Comparison with Serial Task Jobs . . . 54
6.5.2 Validation of Physics Results . . . 55
6.5.3 Speedup . . . 56
6.5.4 Memory Reduction . . . 59
6.5.5 Determining Bottlenecks . . . 61
6.5.6 Benchmark Results on Current Manycore Systems . . . 65
6.6 Summary . . . 68

7 Optimisation at the Level of Workload Scheduling 71
7.1 Impact of Multicore Jobs . . . 72
7.2 Moldable Job Model . . . 72
7.3 Objective Function . . . 74
7.4 Solving the Objective Function . . . 76
7.4.1 Local Search Methods . . . 77
7.4.1.1 Iterative Deterministic Approach . . . 78
7.4.1.2 Probabilistic Meta-heuristic Approach . . . 80
7.4.1.3 Combination . . . 82
7.4.2 IBM ILOG CPLEX CP Optimizer . . . 83
7.4.3 Summary . . . 84
7.5 History Based Estimation . . . 84
7.5.1 Defining the Feature Space of LHCb Jobs . . . 85
7.5.2 Consideration of Single Feature . . . 88
7.5.3 Consideration of Multiple Features . . . 91
7.5.4 Supervised Learning . . . 93
7.5.5 Summary . . . 96
7.6 Handling Uncertainties . . . 97
7.7 Summary . . . 100

8 Conclusion 103
8.1 Summary . . . 103
8.2 Outlook . . . 105

Bibliography 107

9 Appendix 119
9.1 Simulation Workloads MC11 . . . 119

List of Figures

1.1 The 20 most common CPU types in the Worldwide LHC Computing Grid at the Tier-1 level used by LHCb during reprocessing 2012 2

2.1 The four interaction points ATLAS, CMS, ALICE and LHCb (taken from [94]) 8
2.2 LHCb detector and its elements (taken from [93]) 9
2.3 Workflow in the LHCb experiment (status 2012) 11
2.4 Gaudi Architecture (taken from [73]) 12
2.5 Pull and Push principle 15
2.6 DIRAC Architecture (taken from [185]) 17
2.7 LHCb jobs in 2011, 2012, 2013 21

3.1 The 7 Performance Dimensions (taken from [129]) 25

4.1 Single and multicore jobs (taken from [90]) 32
4.2 Optimisation at different levels 34

5.1 Results for simulation job running with 2 workers (taken from [164]) 39
5.2 Results for simulation job running with 8 workers (taken from [164]) 40
5.3 Results for stripping jobs (taken from [164]) 41
5.4 Comparison of the different merging rates 585 MB/s (continous line) and 190 GB/s (dotted line) (taken from [164]) 42
5.5 Extract of the call graph of a simulation job 42
5.6 Comparison of memory consumption within reconstruction jobs (taken from [151]) 43
5.7 Comparison of memory consumption within stripping jobs (taken from [151]) 44
5.8 Difference in Memory and CPU time (in percentage) 45
5.9 Cumulated read and write accesses 46

6.1 Data parallelism in HEP software 50
6.2 Algorithms Parallelism in CMS Software (taken from [123]) 50
6.3 Overview of GaudiMP (taken from [152]) 52
6.4 The main steps of Gaudi applications and possible options of forking subprocesses 53
6.5 Overview of GaudiHive (taken from [98]) 54
6.6 Throughput depending on different memory thresholds (taken from [152]) . 55
6.7 Results obtained from the Kolmogorov-Smirnov test 56
6.8 Average parallelism and variance in parallelism (taken from [152]) 59
6.9 Overall memory footprint of reconstruction jobs 61
6.10 Speedup depending on the number of events (taken from [152]) 62
6.11 Total runtime while increasing the number of workers and events 63
6.12 System with 2 NUMA nodes 66
6.13 A and σ for parallel reconstruction jobs on an Intel Xeon (8 cores) 66

6.14 A and σ for parallel reconstruction jobs on an Intel Xeon (40 cores) 67
6.15 A and σ for parallel reconstruction jobs on an AMD Magnycours (48 cores) 68
6.16 Memory footprint of multicore jobs on the CERN cloud (Agent restarts indicated by blue lines) 68

7.1 Total memory footprint of a parallel reconstruction job 73
7.2 Sorted versus unsorted jobs 74
7.3 Minimal and maximal partition size 76
7.4 Workload per event of reconstruction jobs 77
7.5 Generated solutions by different sorting rules 79
7.6 Test results with $\alpha = 0.9$ and with restarts (blue lines) 81
7.7 Test results with $\alpha = 0.9$ and without restarts 82
7.8 Overview of the different features of a production 85
7.9 Multiplicity versus normalized time per event 85
7.10 File size versus processing time (2012) 86
7.11 Instantaneous luminosity for ATLAS, CMS, ALICE and LHCb (taken from [10]) 87
7.12 Overview of the different features of a job 87
7.13 Required time per event for reconstruction jobs 88
7.14 Memory requirements of reconstruction jobs 89
7.15 Memory requirements and generated workload of stripping jobs 90
7.16 Maximum likelihoods of simulation jobs (2011) with different event types and from different productions 91
7.17 Decision tree for obtaining a regression formula 94
7.18 Accumulated error for the prediction of runtime per event (Reconstruction) 95
7.19 Normalized CPU Time per event of all reconstruction jobs from 2012 sorted by run number 95
7.20 Accumulated error for the prediction of memory (Reconstruction) 96
7.21 Accumulated error for the prediction of runtime per event (Stripping) . . . 96
7.22 Accumulated error for the prediction of memory (Stripping) 97
7.23 Different mixtures of three independent jobs (taken from [163]) 98
7.24 Decisions made by the scheduler (indicated in red) (taken from [163]) . . . 98

List of Tables

2.1 HEP SPEC values for Intel Xeon X5650 (taken from [9]) 16

5.1 Memory reduction reached by memory deduplication in the different applications (taken from [164]) 40
5.2 Results for time measurements (taken from [151]) 44

6.1 Speedup values reached on different hardware configurations (with and without Intel Turbo Boost) 57
6.2 Results obtained from the Performance Monitoring Unit 64
6.3 Comparison of performance metrics between different parallel prototypes . 65

7.1 Loss in efficiency in % based on measured speedup curves 72
7.2 Results found by different sorting metrics 80
7.3 Results found by the Simulated Annealing algorithm 82
7.4 Results found by IBM Cplex Optimizer 84
7.5 Linear regression results for reconstruction jobs from the year 2012 92
7.6 Linear regression results for stripping jobs from the year 2012 93
7.7 Comparison of different approaches for handling wrong estimates 100

1. Introduction

The computing landscape can be characterized as an ever changing ecosystem with a wide variety of design challenges and deep structural revolutions. Change's pace outruns experiment phases - however, experiments live longer. In the past decades, CPU manufacturers were following the principle of increasing CPU performance by adding more transistors to a chip or by increasing the clock frequency. Moore's law [170] states that the number of transistors doubles roughly every two years. During the first decade of the 21st century a technical limit has been reached, meaning that it is not any longer feasible to further increase CPU performance by rising the clock frequency. This limit is also known as the power wall, which is according to [65] defined as:

> *"... difficulty of scaling the performance of computing chips and systems at historical levels, because of fundamental constraints imposed by affordable power delivery and dissipation."*

Generally speaking, CPU performance has met the point after which the power consumption would grow super linearly as explained in [65]. This represents a major constraint on a decade which rewards low consumption CPUs. Instead Moore's law manifests itself in increasing core counts. The multicore era started in the year 2001, when IBM released its first dual core processor (Power4) [12]. Manufacturers, like Intel and AMD, followed in 2005.

The Worldwide LHC Computing Grid (WLCG) is the largest Computing Grid and it is used by experiments at CERN in order to process the vast amounts of data that is for instance generated at the Large Hadron Collider [21]. Nowadays, all its resources consist of multicore CPUs because commodity hardware is deployed. In 2012, the most common types have been Intel Xeon CPUs which typically feature 4 to 8 cores (Fig. 1.1).

In the past, single threaded applications could profit from new and faster CPUs. Users had simply to buy the latest CPUs in order to speed up the execution of their software. Today this is not the case any more. Software must be able to run in parallel in order to profit from the latest processors. New problems arise, like concurrent accesses to computing resources and hardware components, like disc, RAM, I/O interfaces and memory bandwidth that are shared between cores. If accesses are not coordinated, it slows down overall performance because concurrent processes have to wait for resources becoming available. Obviously, such a coordination is not foreseen in single threaded applications. One of the main problems will be the memory ratio on future systems. Currently, cores in the WLCG are

typically equipped with 2 to 3 GB [119]. But it will not be feasible to provide 300 GB of RAM for a system with 100 cores. This problem is well described in [175]:

> *"... As the number of cores per socket increase, memory will become proportionally more expensive and power consuming in comparison to the processors. Consequently, cost and power-efficiency considerations will push memory balance (in terms of the quantity of memory put on each node) from the current nominal level of 0.5 bytes of DRAM memory per peak flop, down below 0.1 bytes/flop (possibly even less than 0.02 bytes/flop)."*

The number of cores on a die is limited by the memory wall [175] since bandwidth must be shared between multiple cores. As a result, cores operating at a high frequency are limited by slow memory accesses. The memory wall implies that multicore processors cannot provide an arbitrary large number of cores. On top of that, core design is meanwhile more driven by power than performance related aspects [175]. Consequently, manufacturers are forced to simplify the core design in order to put more cores on a chip. Each core is smaller, less powerful, contains less complex pipelines, but also consumes much less power. This results in a performance loss for single threaded applications. Cores are grouped into nodes where each one has its local memory. So in contrast to the core design, the overall processor design becomes more complex.

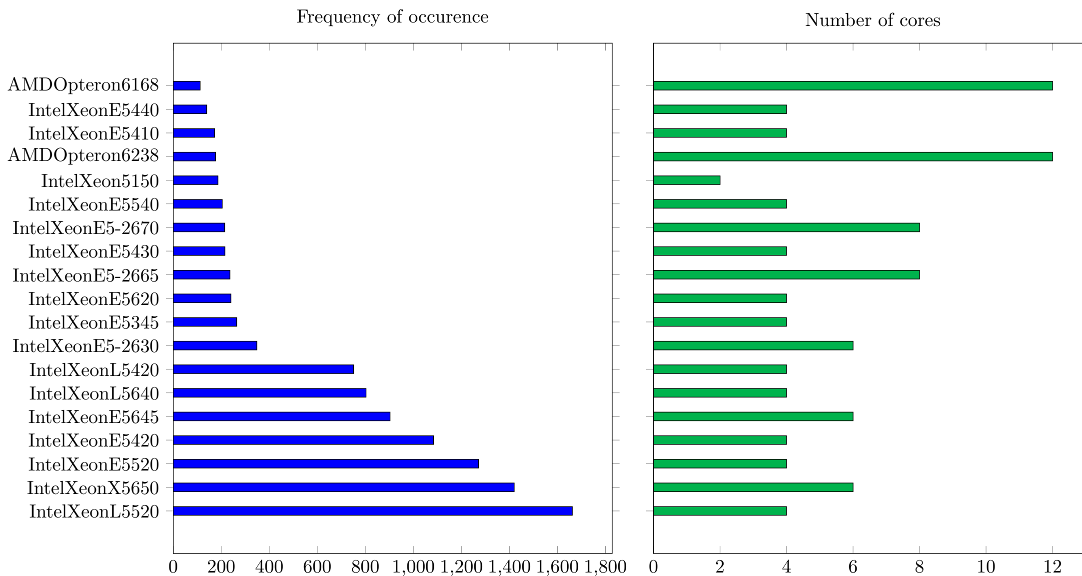


Figure 1.1: The 20 most common CPU types in the Worldwide LHC Computing Grid at the Tier-1 level used by LHCb during reprocessing 2012

In summary, applications which are not able to run in parallel will see a performance loss since single cores become less powerful. Software developers have to be aware of the underlying micro architecture in order to fully exploit the performance. As explained in [175], limitations in bandwidth as well as latency must be respected in the design of new algorithms. Currently, the abstraction layer between software and such complex hardware is missing. A large range of diverse computing resources is available nowadays: starting from symmetric multiprocessors, multi-socket CPUs and multicore processors, to GPUs, ARM for embedded and server systems and many more. This trend raises new questions, which have not played an important role in the past. For instance: What will be the memory ratio on manycore systems? How to handle concurrent access to RAM, caches and I/O interfaces? How to reduce performance loss due to effects of non uniform memory accesses (NUMA)?

The risk of leaving large potential untapped, forces software developers to rethink software models. Applications must be able to run in parallel on multiple cores with reduced memory consumption, such that they can profit from new hardware architectures. This is very challenging. In many cases, like in the LHCb experiment [171], a single threaded software model has been well conceived 20 years ago in the era of ever increasing clock frequency. The software framework itself has been developed over decades by many different developers [84] and it consists of millions of lines of code. It will not be easy to re-engineer the software and algorithms in order to make it scalable for large number of threads and processes. Therefore, the main aim is to apply changes which allow a parallel execution but do not require too many modifications at the core level of the software.

As a result the main research question of this thesis is, how multicore and manycore systems can be used more efficiently at the example of the LHCb experiment. The aim is to develop and evaluate techniques to adapt to the changing hardware landscape.

1.1 Research Questions

Given the main research question the following subquestions can be derived.

1. **What limits the software and the utilisation of computing resources?**
   Generally speaking, before undertaking any major decision on software parallelization the question arises how new developments and technologies can be applied to improve the single threaded applications. A deep understanding of the actual limitations is necessary. The knowledge of what are the boundaries of the software and what causes the bottlenecks becomes the baseline of the first part of the thesis.
2. **What are the performance impacts of multi- and manycore CPUs?**
   As explained before, chip producers are moving towards multi and manycore CPUs due to restrictions such as the power wall. The complexity of manycore CPUs is rapidly growing, providing features such as advanced vector registers, hyperthreading, Non Uniform Memory Access (NUMA), frequency scaling and many more. Each of the features, a priori has as many advantages but also side effects if applied incautiously. Therefore, knowing the baseline one could potentially apply the most relevant features to improve the general performance of the software at hand. One of the main drawbacks of manycore CPUs is the pressure on the memory resources, given their low memory per core ratio. This is problematic for High Energy Physics software which requires large memory footprints due to detector related data, complexity of reconstruction algorithms, tree buffers and many more. Bottlenecks of parallel software have to be understood and it must be evaluated how they can be minimized or removed. Extensive benchmark tests are required in order to understand performance differences between the various parallel software concepts and between different micro architectures. Many monitoring tools provided by the operating system do not allow a proper insight into hardware utilisation any longer due to the complexity of manycore systems. Consequently, low level measurements, based on hardware events, are required to understand limitations of software.
3. **How to apply multicore job submission within the Worldwide LHC Computing Grid?**
   Going from single- to multicore jobs is not an easy transition for the Worldwide LHC Computing Grid, since scheduling is subject to multiple VOs (Virtual Organizations) and grid sites with their quotas and shares. Consensus must be reached regarding how to deal with such jobs. It must be evaluated whether it is worth leveraging the scheduling at the level of the site or at the level of the VOs. Latter approach would

allow VOs to use their own schedulers, based on the performance of their software and not static quotas as it is currently the case.

1.2 Contributions

The thesis comprises the following new approaches and methods:

1. It presents the first systematic approach of optimising LHC experiment software by taking performance limitations at many different levels into account [164], [151], [152], [163]. Other LHC experiments are investigating as well improvements of their resource usage. However, improvements mostly focus on one given level. But for instance, studying the problem of multicore jobs must not neglect performance of parallel software. The thesis shows detailed benchmarking tests in order to evaluate and improve limitations of the parallel software prototype. It is still an unsolved issue within the HEP community whether scheduling of multicore jobs is a site or a VO (Virtual Organization) related problem. This work provides a scheduler for multicore jobs which can be integrated in the Workload Management System of a VO. It optimises schedules with respect to job throughput by taking into account worker node, job specific and prior job information [162], [161].
2. Optimising scheduling problems by using the moldability of jobs has been studied by [168], [82] and [169]. In contrast to these publications, the arrival and finish time do not matter for Computing Grid jobs. Instead, high job throughput must be guaranteed. Therefore the thesis has defined the scheduling problem and developed algorithms to solve it by using the moldability of jobs [162], [161].
3. Estimating the runtime of jobs submitted to the Computing Grid has been studied by many different research groups at CERN and collaborating institutes [174], [87]. A proper estimation is a precondition for multicore jobs. Since multi- and singlecore jobs shall share the same computing resources, backfilling must be applied which requires a proper runtime estimation [180]. This work analyses the impact of LHC parameters on the runtime and memory requirements of jobs. Since the LHCb detector supports luminosity levelling [10], a stable luminosity value is ensured during a LHC run. This means that the number of collisions per second remains stable during a run. Given that, it has been proven that runtime prediction of reconstruction jobs can be significantly improved [162], [161].

1.3 Structure of the Thesis

The structure of the thesis is the following:

Chapter 2: Scientific Computing in the LHCb Experiment

An overview of the LHCb experiment is presented in this chapter. It starts with an explanation of the Large Hadron Collider and the LHCb detector. Before describing LHCb's software framework, the related data workflow with the main types of grid jobs is presented. An overview of the distributed computing infrastructures used by the LHCb experiment is given as well as the workload scale is shown. This information is relevant for the understanding of the conducted research.

Chapter 3: Related Work

The thesis faces problems from many different research areas and related work is given in this chapter. It shows the attempts done by the HEP community to use multi- and many-core CPUs more efficiently. One important factor is the reduction of memory requirements

and the chapter shows related work for memory deduplication. The thesis also deals with the problem of scheduling multiprocessor tasks and related work will be given in the end of this chapter.

Chapter 4: Problem Description in Detail

In contrast to the introduction, this chapter focuses on the problem description from the experiment's point of view. The impacts of the LHC upgrade on the computing requirements as well as the limitations of the current job model are illustrated. Afterwards, a proposal for optimising the workflow is given that represents the baseline of this thesis. The work touches different research areas and further details are in the following chapters.

Chapter 5: Optimisation with Non Intrusive Techniques

This chapter describes different techniques that optimise execution of software in a transparent way. It shows this through the example of automatic memory deduplication and memory compression. The chapter also presents how advantages of 32- and 64-bit applications can be combined and how this impacts LHCb software. This chapter deals mainly with research question 1.

Chapter 6: Optimisation with Intrusive Techniques

As explained in research question 2, the impact of multi- and manycore CPUs must be understood as well as the limitations of parallel software. This is shown in this chapter. First, diverse parallelization principles and their usability with respect to HEP software are discussed. Different prototypes are presented and evaluations of benchmark results are shown.

Chapter 7: Optimisation at the Level of Workload Scheduling

This chapter focuses on research question 3. It evaluates the impact of multicore jobs and the importance of non linear speedup. It proposes a moldable job model as part of the VO's Workload Management System. It defines the objective function and evaluates how moldability can be used to optimise the scheduling. It shows how supervised learning can be applied to improve estimation of memory and runtime requirements over time. The chapter also evaluates the impact of input parameters on the scheduling decision. A detailed workload analysis is given as well as a proposal for the handling of uncertainties.

Chapter 8: Conclusion

The conclusion summarizes the most important results and their technical impacts. An outlook is given in the end.

2. Scientific Computing in the LHCb Experiment

The basics of the LHCb experiment and its data workflows are explained in this chapter. The LHCb experiment is one of the Large Hadron Collider experiments at CERN [171]. Section 2.1 and 2.2 give an overview about the Large Hadron Collider (LHC), the LHCb experiment and the related physics. Section 2.3 explains the data flow, starting from collisions to the different processing steps until the first physics analysis. Section 2.4 gives details about LHCb's software framework. Section 2.5 shows the distributed computing infrastructures used, in order to process the large amount of data. An overview about past workloads is given in section 2.6.

2.1 The Large Hadron Collider

The Large Hadron Collider is a proton-proton collider [147], located near Geneva, 100 meters below ground of 27 km circumference. Its maximum design energy is 7 TeV per beam ($7 \cdot 10^{12}$ eV), which results in a maximum center of mass energy of 14 TeV for proton-proton collisions. Protons are pre-accelerated in previous steps and when they have an energy of 450 GeV, they are injected into the large LHC ring to be further accelerated. LHC consists of two beam lines: one circulating clockwise the other one anticlockwise. Collisions take place in four interaction points, where the experiments ATLAS [83], CMS [76], ALICE [51] and LHCb [171] are located (Fig. 2.1). In these points the beam lines are focused and the smaller the cross section, the higher the probability that collisions will take place. Protons are injected into the LHC in form of bunches and the intersections are called bunch crossings. Due to the collisions the beam looses protons such that the rate of collisions per bunch crossing will decrease over time. However, the LHCb experiment applies mechanisms in order to keep this rate at a level of about 2 collisions per bunch crossing [108]. When two protons collide new particles are produced and the larger their colliding energy has been the more particles will be generated. Each beam consists of about 1400 bunches of protons and a bunch crossing occurs every 50 ns. After the long shut down 1 (LS1) this rate will be reduced to 25 ns with 2808 bunches. The LS1 started in 2013 and lasts until the end of 2014. During this time period the accelerator is upgraded, so that it can run with its maximum design energy. In 2012, the LHC was running at a center of mass energy of 8 TeV (4 TeV per beam).

In the optimal case the beam lines would collide frontally, which is difficult to achieve in a ring accelerator like LHC. However, it is easier to reach large energies in an accelerator

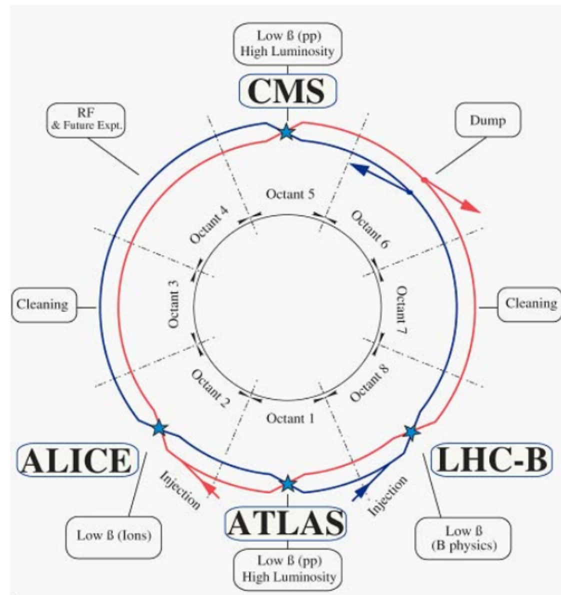


Figure 2.1: The four interaction points ATLAS, CMS, ALICE and LHCb (taken from [94])

where particles can increase their energy in each turn. Radially accelerated particles emit the so called synchrotron radiation such that they loose energy: the larger the acceleration the larger the loss. In order to keep this small, protons have to be used instead of electrons and positrons. They are about 2000 times heavier and loose therefore less energy due to synchrotron radiation. Nevertheless, the disadvantage of protons is that they are not elementary particles, because they are made of quarks and gluons. Hence, from the computing perspective it is more difficult to analyse and reconstruct such information. This is the main reason, why research groups are working on the next concept of collider, the International Linear Collider [41], in which electrons and positrons are accelerated linearly.

2.2 LHCb Detector and Physics

The LHCb experiment explores the difference between matter and antimatter. For this it analyses the difference between mesons that contain a b (beauty) quark and antimesons that contain an anti-b quark. B quarks are used because their difference (asymmetry) between quark and anti-quark is larger than with ordinary matter. Just after the big bang, matter and antimatter were evenly distributed. When the universe started to expand, the composition changed such that matter dominated. In order to find an explanation for this process, energy density in the LHC collisions is the same that has existed a hundredth of a billionth of a second after the Big Bang [42].

The LHCb detector is a forward spectrometer and measures consequently particles moving only in one direction. It consists of many sub detector layers (Fig. 2.2), which are responsible to find the properties and trajectories of particles. The sum of all information allows the particle identification. A very detailed description of the detector can be found on the official LHCb website [42]. The Vertex Locator (VELO) surrounds the collision point and it is responsible for measuring the trajectories of particles close to the interaction point. It allows the separation of primary and secondary vertices. A vertex is the point from where particles are emerged, either from a collision (primary vertex) or from the decay of a short lived particle like the B mesons. The Rich-1 detector is located behind the Vertex Locator and allows to identify low-momentum particles. RICH stands for Ring Imaging Cherenkov. Such detectors use the Cherenkov radiation of particles going through a dielectric medium

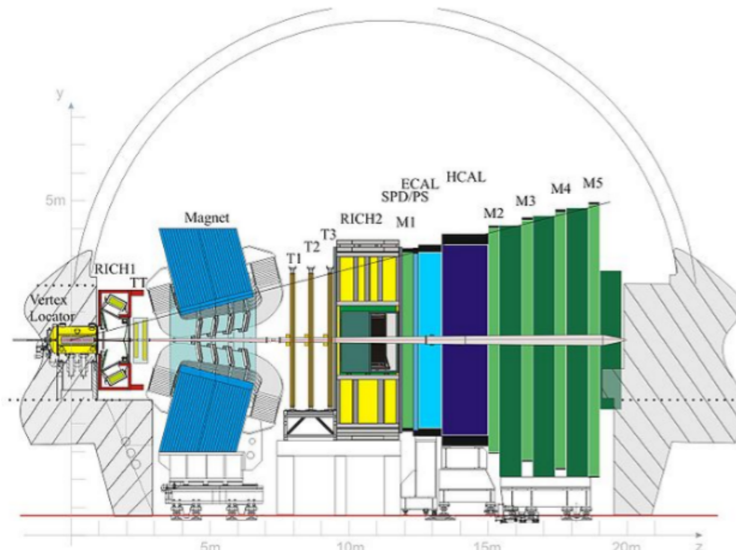


Figure 2.2: LHCb detector and its elements (taken from [93])

which allows to determine the speed of particles. Rich-1 is followed by the Tracker Turicensis (TT) which is part of the Main Tracker and measures the transverse-momentum of particles. The three Tracking Stations (T1-T3) which are also part of the Main Tracker are located behind the large LHCb dipole magnet. They locate charged particle tracks and can measure their momentum using the bending by the magnetic field. Another Rich detector is located behind the Main Tracker and in contrast to Rich-1, it is responsible to identify particles with high momentum. The next layer consists of the electromagnetic and hadronic calorimeters (ECAL/HCAL), which stop particles and measure their energy. The last layers are the Muon chambers, that identify Muons, the only charged particles not stopped by the calorimeters. The detection of Muons is important since they are part of many B meson decays.

The detector is specialised for B physics and the main goal is to search for CP violation in B meson decays. B physics is term for the study of behaviour of B mesons. B stands for Beauty (quark) and the assumption is that the difference between matter and anti-matter can be understood by evaluating the differences between the beauty and its anti-beauty quark. B mesons consist of a beauty, also known as bottom quark, and either an up-, down-, strange- or a charm-quark. The LHCb detector is designed to record exactly these kind of particles. CP stands for conjugation symmetry parity and it basically defines the difference of behaviour between a particle and its anti-particle. The CP violation is the broken symmetry between them. In 1964, James W. Cronin and Val L. Fitch were the first ones who observed such a violated symmetry in the decay of strange particles (containing a strange quark) [79]. They analysed neutral kaons, which were generated at the Alternating Gradient Synchrotron in the Brookhaven Laboratory. They could observe that a certain amount of kaons did not decay like predicted by the CP symmetry. In 1999, the KTEV experiment at Fermilab [1] and the NA48 experiment at CERN [2] could also prove the CP violation of neutral kaons. CP violation can also be observed for B mesons which are heavier than kaons. In 2004, this has been proven the first time in the BaBar experiment at the Stanford Linear Accelerator Center [13] and Belle experiment at KEK (Japan) [30].

2.3 Data Flow

The LHCb detector records several million collisions per second which correspond to an amount of information of more than hundreds of GB per second. An efficient and fast

computing is important, in order to deal with such an amount of information generated by the detector. Due to limited capacities not all this information can be stored. Filter mechanisms are applied which discard collisions corresponding to well known physics processes. This is done by a hardware and a software trigger. Latter one runs on a large computing farm which provides 1300 nodes and about 26000 logical cores. It is also called the High Level Trigger (HLT). It performs parts of reconstruction on the events in order to allow a proper selection. The maximum output rate of the hardware trigger is limited to 1 MHz [52] and is reduced to about 5 KHz by the software trigger. It means that about 5000 events per second are finally recorded in raw files [60]. An event contains the information of the LHCb detector corresponding to a single beam crossing. It takes approximately 60 to 100 kB and normally contains detector responses that are result of more than one collision. As next, raw files are created, which contain about 50k to 60k events and measure up to 3 GB. The first step in the processing chain is the reconstruction of these raw events [60]. An overview of the data workflow is shown in Fig. 2.3.

2.3.1 Reconstruction

The raw files contain digitized information from the electronic signals recorded by the detector. One must reconstruct which particle has caused which particle trajectory and which group of particles belong to a collision. The reconstruction consists of several steps and a detailed description can be found in [92]. First the software is initialized and decodes the raw buffers from the input file. In the next step, tracks must be found via pattern recognition algorithms. Therefore, information from the Vertex Locator (Velo) and the Main Tracker are collected. Track states are then identified, which are according to [145] defined by the charge, the momentum, the (x,y) position and the tangent direction of a particle. A final fit is applied on those track states, which is done via a Kalman filter [103]. According to [145] five classes of tracks exist, which are long, upstream, downstream, Velo and T tracks. Having a majority of certain tracks gives a hint on the event type. The next step is the reconstruction based on the fitted tracks. Information obtained from the Rich, Muon and Calorimeter systems is used in order to identify particles. The Rich reconstruction is based on a likelihood method which determines the probability distribution of finding hit pixels in the Rich detector. The observed hits are analysed and a particle hypothesis is made which is then modified in order to maximize the likelihood. During the muon reconstruction, tracks are simply extrapolated and observed hits in the muon chambers are matched to the tracks which are the closest ones. Data obtained from the calorimeter must be clustered and then hypothesis are made about the cluster origin. The combination of all pieces of information allows the identification of particles. The reconstruction software consists of about 200 different algorithms, whose computational complexity vary a lot. Many of those algorithms are dependent on the output of other algorithms. The complexity of reconstruction is correlated with the conditions at the LHC. The larger the collision energy the more complex the reconstruction becomes because more particles are generated. The complexity of reconstruction is also correlated to the number of initial collisions taking place per bunch crossing. More detailed information can be found in section 7.5.

2.3.2 Stripping

Each physics analysis is performed on a much smaller data sample than what is recorded. Therefore a further data reduction is necessary, called stripping. That takes place after the reconstruction step [60]. This procedure reduces the amount of data to roughly 10^6 to 10^7 events per year by selecting the reconstructed events into different physics analysis streams for each type of analysis [146]. Several stripping lines being part of a stream contain the sequence of selections in order to create candidates and filter out unimportant

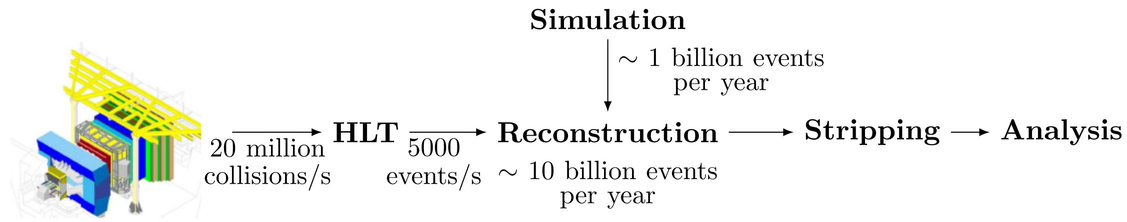


Figure 2.3: Workflow in the LHCb experiment (status 2012)

information. Such a reduction can be done, because certain decay products are more relevant than others. The ones which are caused by well known physics can be discarded without problems. Stripping jobs produce dozen output streams each grouping the events with similar type.

2.3.3 Analysis

The output of stripping jobs is merged to larger files. These files are replicated and stored at CERN, at different Tier-1 and special Tier-2 sites. These sites are computing sites participating in the Worldwide LHC Computing Grid, that provide apart from computing also disc storage facilities. More information about the different sites in the Worldwide LHC Computing Grid can be found in section 2.5.1.1. The output of stripping jobs is used by physicists in order to do their analysis. These analysis represent the user jobs [60]. These jobs can only be executed at sites, which have all required input files. This is one reason, why replication of data and distributed computing is necessary. It increases reliability and it allows a better load balancing of user jobs which are submitted by physicists from all over the world.

2.3.4 Simulation

Apart from information recorded by the detector, collisions can be also generated by simulation [60]. Proton-proton collisions are simulated according to the known or expected production models, generating new particles that are propagated through the detector. Afterwards the response of the detector is simulated, events are triggered and stored. Simulation helps to understand the experimental conditions, to compare results obtained from real collisions with models and therefore to interpret them. In High Energy Physics Monte Carlo methods (MC) are used, which are based on the law of large numbers. This means that many simulations are required in order to numerically solve a problem. In LHCb MC simulations are used for performance studies, explanation of unexpected effects and as estimation of design choices [85].

2.4 Software Framework Gaudi

LHCb computing tasks are mainly based on the object-oriented software framework Gaudi written in C++ and Python [59]. As explained in the Gaudi User Guide [73] the basic requirements of physicists are to have tools for event data processing, analysis and visualization which should be easily extendible. Therefore, a software architecture has been developed which is based on components. These can interact with other components and have interfaces provided by different methods. Fig. 2.4 shows the architecture of the software framework. The main components are the algorithms, converters and data objects. Algorithms are responsible for processing input data and generating new output data. Therefore, they use services in order to receive and to store data. For example, the persistency service facilitates the read and write access to disc. Other services are the messaging

service, a configuration service analysing the job options, a service generating histograms and many more. This software architecture allows a clear separation of algorithms and data. Data are for example the recorded particles, the reconstructed tracks and detector related information. They all are stored as objects in different data stores, like the transient event store. The framework is easily extendible in the sense that physicists can add new components, for example new algorithms for reconstructing particle decays. At the same time, the component based patterns have to be followed in order to ensure future flexibility of the framework.

The job options are Python scripts which define the configuration of the application. They define the actual processing chain, for instance which kind of Gaudi services and algorithms have to be used. While the components are implemented in C++, configuration is done via a Python interface that simplifies and abstracts the usage of the core software. Gaudi consists of a large software stack, which includes many external, CERN specific and some LHCb specific packages and each of them consists again of sub modules. Like many other HEP experiments, LHCb also uses the ROOT framework, in particular for I/O, histogramming and mathematical functions [35].

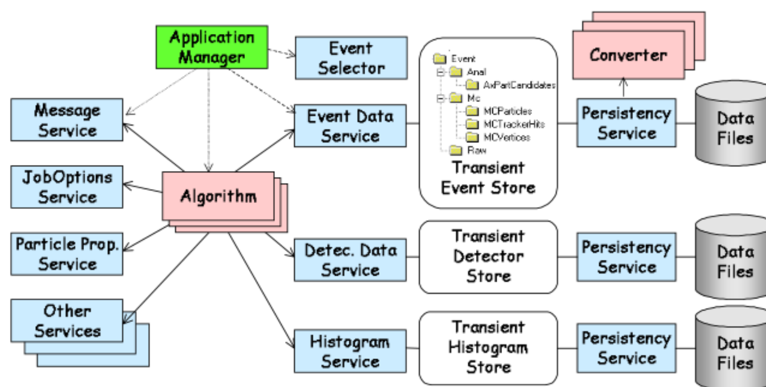


Figure 2.4: Gaudi Architecture (taken from [73])

2.5 Distributed Computing Infrastructure

In distributed systems, computing facilities are connected to construct a larger system. These facilities are geographically distributed. Distributed computing is often applied in areas that need to solve large computational tasks which can be split into subtasks.

2.5.1 Grid Computing

Grid Computing is a distributed computing system, where resources are located at geographically separated locations. These are normally loosely coupled which requires that tasks do barely communicate with each other. Grid Computing is used in application areas which deal with a lot of data and which cannot have all computer and storage units in one location. Ian Foster proposes a three point check list, which defines Grid as [99]:

> *"A Grid integrates and coordinates resources and users that live within different control domains - for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system."*
>
> *"A Grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery,*

> *and resource access. [...] it is important that these protocols and interfaces be standard and open. Otherwise, we are dealing with an application-specific system."*
>
> *"A Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts."*

A Grid allows to utilise resources located somewhere else and it provides a less expensive solution compared to supercomputing [133]. Grid Computing can be divided in data and computational Grid. First one serves as a distributed data storage where users can access and modify large amounts of data. File naming conventions, a secure and efficient data transport must be guaranteed as well as data must be replicated to allow better load balancing. The main goal of computational Grids is to solve a large task by subdividing it into smaller pieces and executing them at different computing elements. These pieces do in general not require communication between each other. A Grid must provide functionalities, protocols and interfaces, to allow a unified access for all users.

One of the first Grid Computing projects started in 1995. The aim of the I-WAY project (Information Wide Area Year) was to connect and unify several supercomputer centres and virtual environments [181]. It was later on adopted by the Globus toolkit [29], which is nowadays the common standard for the development of Grid applications. Nowadays, Grid Computing is not only applied in research but also in many other fields like finance, web services and industry. For instance, the financial sector uses Grid Computing to execute Monte Carlo simulations for pricing and scenario analysis [120].

Grid Computing increases reliability due to the non existence of a single point of failure. A failing job can be simply send to a different worker node. Resources can be easily upgraded without affecting the overall system because jobs just have to be executed at a different site. Unlike a supercomputer, where this would lead to the impossibility of executing tasks.

2.5.1.1 Worldwide LHC Computing Grid

The LHC experiments generate in total 15 PB of data per year. At the time when experiments have been designed, it has not been possible to process these amounts in one large supercomputing center and it has been required to build a distributed computing infrastructure. The recorded events are independent and can easily be computed in parallel. So jobs do not have to communicate with each other and can therefore be processed at any place as long as the input files are provided. Distributed computing resources can be used since universities collaborating with the experiments have access to computing facilities. As a result, all available resources have been integrated into the Worldwide LHC Computing Grid in 2002 [75]. The Grid also serves as storage for files, where multiple copies are placed. This ensures that physicists from all over the world can access them in near real-time.

The Worldwide LHC Computing Grid (WLCG) consists of three main layers, which are the Tier-0, Tier-1, Tier-2 level. CERN's data center represents the Tier-0 as the source of all datasets from all LHC experiments. It is responsible for distributing and replicating raw data. The following sites represent the Tier-1 level:

- TRIUMF (Canada)
- KIT (Germany)
- PIC (Spain)

- IN2P3 (France)
- INFN (Italy)
- NDGF (North Europe)
- NIKHEF, SARA (Netherlands)
- ASGC (Taipei)
- RAL (UK)
- FNAL-CMS, BNL-ATLAS (US)
- RRC-KI, JINR (Russia)

CERN is connected via an optical fibre link to these sites and it provides a bandwidth of 10 GB/s to each of them. Files are transferred with the Grid File Transfer Service [3]. The Tier-1 level differs from the Tier-2 in the sense that the sites are also responsible for providing tape storage facilities. As a result, reconstruction and stripping jobs are generally executed at the Tier-1 sites since these jobs require input files and have to store large output files. Tier-2 sites provide computing power and are typically used for simulation and user analysis jobs. By the end of an experiment year, all experiments start their reprocessing productions in which datasets from the whole year are re-reconstructed. This normally produces very large workloads. In such circumstances Tier-2 sites are used by LHCb as well in order to execute reconstruction and stripping jobs.

In order to access the distributed computing infrastructure users must acquire a certificate issued by certification authorities. This is managed by subscribing to a Virtual Organization (VO). A VO represents a group of people or institutes sharing resources like CPU and storage [100]. The VO manages the rights of users, validates certificates, defines requirements and goals. A user has to obtain a certificate and transform it to a Grid certificate. It allows, to create a proxy with a limited validity and to send jobs to the Computing Grid.

The middleware is an abstraction layer between operating system and application. It is the core element of a Computing Grid [46], since it provides basic services and protocols. For example, WLCG uses middleware developed in Europe (gLite, EMI) or in US (OSG). Unicore is a middleware which offers a wider range of services. It provides security and workflow mechanisms, global data management and load balancing [50].

Pull versus Push Principle

One of the main challenges in Grid Computing is the scheduling of jobs. On one side there are the Workload Management Systems of the experiments which create, monitor and send jobs to the grid sites. On the other side there are the batch systems at the grid sites which take care of scheduling and monitoring jobs on their worker nodes. The push principle sends jobs directly to the scheduler which then allocates resources (early binding) (Fig. 2.5). The pull principle sends pilot jobs. As soon as they are safely running on a computing resource they request real jobs through a scheduler (late binding) (Fig. 2.5). The advantage of the first approach is that the grid sites can control the workloads. However, once the job is submitted it cannot change its priority in this model. Problematic is also, that failures often occur during the start of a job which then requires a rescheduling. Late binding minimizes this problem, because jobs will only be requested when the environment has been set up on the worker node. It also allows a better load balancing, because jobs will be picked which better match the allocated resources. Nowadays, most of the experiments use the pilot job model which implements the pull principle. It allows the experiments to control the match making of their tasks. However, it implies some problems in the context of multicore jobs which will be explained in section 4.2.

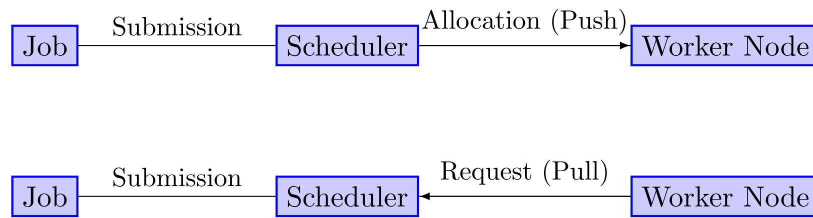


Figure 2.5: Pull and Push principle

High Throughput Computing versus High Performance Computing

With increasing number of cores on modern CPUs, applications are facing problems which are well known in the field of high performance computing (HPC). Supercomputers tightly connect a large number of nodes, such that fast inter node communication can be facilitated. The aim is to perform large amounts of floating operations per second (FLOPS) for executing compute intensive tasks [14], which cannot be executed anywhere else. Therefore, data and/or tasks are separated and processed in parallel on different nodes. Applications need to scale well on a large number of processes.

Jobs executed within the LHC Computing Grid are mainly oriented on high throughput computing (HTC). The main metric is the number of executed jobs per certain time period [14]. In addition, jobs are loosely coupled and do in general not communicate with each other. This allows using computing systems distributed over the whole world. While HPC focuses on CPUs providing a lot of FLOPS, HTC relies on increasing the amount of CPUs where each of them does not necessarily provide many FLOPS. In the context of high throughput computing efficient scheduling strategies also play a major role. If a job performs well but the remaining time of a schedule is not used at all, then the throughput would not improve compared to a job which needs more time to be executed. The Grid is a highly dynamic system with a lot of dissimilar resources. Each site can deploy its own policies and can add/remove resources. As described in [148]:

> *"Such an environment has to employ opportunistic scheduling: Resources are used as soon as they become available and applications are migrated when resources need to be preempted. The applications that most benefit from opportunistic scheduling are those that require high throughput rather than high performance. "*

[148] proposes and evaluates a matchmaking framework, in which requirements are matched to resources. Requirements are defined by several attributes which can contain strings, numbers or more complex expressions. Such matchmaking algorithms are used by batch systems for High Throughput Computing, for instance HTCondor [34].

Since the start of the multicore era, paradigms from the field of HPC play a more important role in HTC. The question is now, how applications can be parallelized in order to exploit the full potential of multicore processors and what must be undertaken such that they can scale on manycore systems as well.

HEP SPEC Benchmark

The Worldwide LHC Computing Grid provides a large variety of different computing resources. These differ in the number of job slots, clock frequency, size of RAM, micro architecture and they are produced by different CPU manufacturers. For the experiments it is important to have a metric in order to compare different worker nodes. For this purpose, HEP SPEC benchmarks have been defined which is a subset of the SPEC benchmark test

HS06	Speed (MHz)	L2+L3 cache (KB)	Cores (runs)	RAM (GB)	Mainboard type	Site
213	2666	3072+24576	24 HT on	48	Dell C6100 (2 sockets)	DESY-HH
212	2666	3072+24576	24 HT on	72	Dell 0F0XJ6	KISTI
210	2666	3072+24576	24 HT on	48	Dell 0D61XP	GRIF-IRFU
207	2670	1536+12288	24 HT on	72	Dell C6100	CC-IN2P3
174	2666	3072+24576	12 HT off	24	HP ProLiant BL460c G6	PIC
172	2667	3072+24576	12 HT off	24	Dell C6100	UVIC
161	2667	1536+12288	12 HT off	24	HP DL170e G6	Australia-ATLAS

Table 2.1: HEP SPEC values for Intel Xeon X5650 (taken from [9])

suite [9], [118]. SPEC stands for Standard Performance Evaluation Corporation which defines a number of tests in order to evaluate the performance of a computing system. A mix of diverse tests is necessary which stress different components, like I/O, bus rates, memory, network, CPU etc. Additionally, compiler options and operating system impact the runtime of tests as well. The higher the SPEC value, the more powerful a system is. This value is also necessary to normalize CPU time and to compare similar tasks executed at different systems.

However, the High Energy Physics community derived a subset of these tests, which better reflects the needs of HEP software. The aim of these benchmarks is providing a measure which scales with HEP software. Having such a normalization value is very important in the Computing Grid since it provides a large variety of different resources. Pilot jobs must indicate the required CPU time as HEP SPEC value and this allows a proper matching by the grid site. When experiments evaluate their computing requirements for future experiment years, they have to indicate their expected workloads in HEP SPEC seconds (HS06.s), hours (HS06.h), years (HS06.y). Knowing the HEP SPEC value of a CPU allows the grid sites to determine the amount of required CPUs for providing given computing capacity. Tab. 2.1 shows results of the HEP SPEC benchmarks measured on Intel Xeon X5650 CPUs. The final HEP SPEC value is influenced by many configurations, like the size of cache, amount of RAM and hyperthreading. As Tab. 2.1 shows, the results differ significantly ranging from 161 HS06 to 213 HS06.

2.5.1.2 Workload and Data Management System DIRAC

The LHCb experiment uses 6 Tier-1 sites and about 83 Tier-2 sites. A Workload Management System is essential in order to submit jobs to different grid sites and monitor their execution. The LHCb experiment is following the principle to use all possible computing resources with minimal effort. Hence, a Distributed Infrastructure with Remote Agent Control system (DIRAC) has been developed which is according to [24] a complete solution for workload and data management on computational grids. The LHCb experiment uses this framework, but extends it with further functionalities. The key components are (Fig. 2.6):

- Resources
- Services
- Agents

- Interfaces

Resources present all computing elements provided by the Computing Grid, Cloud facilities, dedicated CPU clusters or volunteer computing. BOINC is a platform for volunteering computing, where one can subscribe and let jobs run on a conventional personal computer. This has been made available for DIRAC in 2013 [22] and allows LHCb to use additional CPUs. Within DIRAC resources can be also storage elements, which can be accessed via protocols like gridftp, ftp and http. Services are used for reporting and retrieving information that is generally stored in a database and can only be accessed by authorized users. As an example: the job monitoring service tracks the status of all jobs and stores changes in the database. The third element are the agents, which are processes running closely together with services. While services track the state of something, agents are able to perform different operations based on the information obtained from services. A job agent is responsible to pull tasks from the central task queue and to run them on a worker node. In addition, it has to ensure the correct execution of jobs. The last component are the interfaces which allow to use DIRAC functionalities. This can be done via an API, the command line or via a web interface. This allows to monitor the system status in an easy way and to obtain an overview about currently running tasks. The framework itself provides more components like configuration, monitoring and logging services. In addition, a protocol for secure communication is provided (DISET - DIRAC Secure Transport) which allows authentication of users. DIRAC is based on the pull job scheduling paradigm

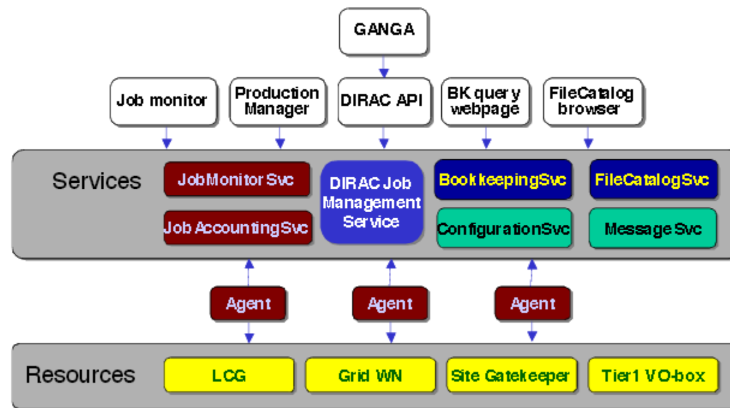


Figure 2.6: DIRAC Architecture (taken from [185])

which means that the computing resource is seeking for new jobs to be executed [67]. A scheduler which only checks the availability of computing resources and directly sends a job realises the push paradigm. In this situation all information about available resources must be collected at one place and this can lead to scalability issues. The pull paradigm does not have such a problem since a computing element simply has to request a new job from the task queue. Hence, load balancing can be realised more easily since a powerful resource will simply request more jobs. DIRAC uses pilot jobs in order to match jobs from the task queues. Pilot jobs are python scripts, which are sent and executed on a worker node. They represent empty resource reservation containers which ensure matching of appropriate tasks on computing elements [70]. In addition, they do the setup of all required components, they perform system checks and ensure that the environment is safe. In case of problems, tasks will not be fetched and therefore they are not impacted. Incidents are then reported to the Workload Management System. If tests have been successful, a pilot job starts a Job Agent, which is responsible to pick tasks from the queue. For this purpose, it connects to the central DIRAC service. The pilot job mechanism allows late binding of jobs, such that tasks are chosen which are more suitable for a computing resource [70]. It

also reduces failures because jobs are only picked when the pilots are safely running [70]. They also allow to run multiple payloads.

Before simulation, reconstruction and stripping jobs can be generated, the so called productions must be created. They define the job options and the set of files to be processed. Each job can run a different application or a sequence of applications with diverse configurations. The job options define the input files for a job, the amount of events, the name of output files, the sequence of algorithms and many more. It is also possible to execute several applications within a job, like it is the case for simulation jobs. First collisions are created, simulated, then events are reconstructed and stripped. Jobs from the same production have options and application version in common. The production manager is responsible to create productions and to define its properties. When a production is ready, grid jobs are created and sent to the task queue.

2.5.2 Cloud Computing

Cloud computing refers to a network of computing resources, which users can access via services. The name cloud derives from the fact, that the complexity of IT infrastructure is hidden by virtualization. This allows to execute applications within a virtual environment which can be configured as required. The users do not need to take care about the hypervisor's operating system. Other components like network or storage are virtualized as well. Users can request resources, but they don't see where their data is actually stored or which server their application is running on. They can just see it as a cloud. To request resources, services are provided. These are [68]:

- **Infrastructure as a Service (IaaS):**
  It provides access to hardware resources, like CPUs and storage. They offer functionalities to run and stop operating system instances. For example, Amazon EC2 and OpenNebula offer such service.
- **Platform as a Service (PaaS):**
  It provides access to computing platforms including an execution environment. It is mostly used by developers, in order to implement software over the Internet.
- **Software as a Service (SaaS):**
  It provides access to software which normally requires licenses. Instead of obtaining the full license users pay on a per-use basis.

The LHCb experiment profits from the emerging so-called institutional Cloud Computing infrastructures [31] [25], which sit in between public and private infrastructures. Some members of the WLCG are becoming Cloud Computing providers and at this early stage their resources are available for those who are ready to experiment with them. The rationale behind this movement at the WLCG is motivated by economic factors, since a virtualized infrastructure allows a significant reduction in costs derived from headcount and maintenance. This landscape represents a change of paradigm with many serious implications as well as a shift of responsibilities. Out of those, the most outstanding one is the fact that now the historical responsibility of troubleshooting the resources locally is now with the experiments which have to monitor their resources. Where Grid resources are best troubleshot in case of problems, the Cloud resources are cheaper if terminated and re-instantiated.

At the present, traditional grid sites such as CERN, PIC, RAL, LAL provide IaaS. The heterogeneity of resources present in the Grid was soon present on the Cloud, as the different cloud platforms used by the sites. As an example, OpenStack is used by CERN and RAL [32], OpenNebula is used by PIC and StratusLab by LAL. The LHCb experiment via the VMDIRAC module, interacts with the different APIs provided EC2 for OpenStack

framework and OCCI on a homogeneous manner. This module allows the experiments to schedule VMs dynamically based in the load, pre-stablished temporal restrictions, and downtimes.

2.5.3 Volunteer Computing

In 2013, LHCb investigated the usage of resources provided by volunteer computing [22]. Personal computing facilities like workstations and laptops are often idle and can be used as additional resources by the experiment. This is most suitable for jobs requiring a lot of CPU and generating only small data like for instance Monte Carlo productions. LHCb uses the Berkeley Open Infrastructure for Network Computing (BOINC) platform which takes care of initializing and executing applications [56]. In a first step, a user must download the BOINC client. Jobs are executed within a virtual machine, which ensures the correct software environment. This requires that in addition a hypervisor is installed by the user. In volunteer computing many issues can occur because resources are unmanaged and unreliable. Questions arise like how to monitor and account jobs [22] or what to do in case the user switches off the system. Simulation jobs run normally for several hours since they process up to 500 events. In order to reduce execution time, jobs executed on such computing resources process only 50 events. Additionally, work is currently ongoing such that simulation tasks automatically adapt their size and safely shut down if necessary.

2.6 Workload Scale

The larger the collision energy, the more particles are generated. This increases the size of events, because more information is recorded. At the same time, the combinatorics of reconstruction rise leading to a larger computational complexity. The energy of LHC is increased each year, in order to search for new particles and physics processes.

Run 1

In 2011, LHC has been operating at a center of mass energy of 7 TeV, which corresponded to a beam energy of 3.5 TeV. At LHCb an average collision multiplicity of about 1.5 has been measured during the year. It means that about 1.5 collisions took place per bunch crossing. The HLT trigger produced an output rate of 3 kHz and in total 600 TB of raw files have been produced over the whole year. The mean event size was about 53 kB. In addition, 10k simulation jobs were running permanently on average. Summing all jobs up, a total workload of more than 130 kHS06.y has been generated in 2011. This corresponded to about 12.000 required cores [111].

In 2012, the energy of LHC has been increased by 0.5 TeV leading to a beam energy of 4 TeV. The detector recorded about 20 million collisions per second. The average collision multiplicity increased up to 1.7. Due to a novel triggering technique the HLT output rate has improved up to 5 kHz. The LHCb experiment has recorded 2.06 PB of raw files, which is significantly larger than the amount taken in 2011. According to [110], 39% of available computing power has been used for Monte Carlo productions, 47% for real data processing and 13% for user analysis. A total workload of 173 kHS06.y has been generated.

Fig. 2.7 shows how the workload evolved from 2011 until 2013. In 2011, Monte Carlo simulations contributed with 74% to the workload. Since the amount of data was relatively small, real data processing accounted only for 14.9% (6.3% for reconstruction, 5.6% for reprocessing and 3.0% for stripping). Reprocessing takes place in the end of an experiment year, which reprocesses data from the whole year. As it can be seen in Fig. 2.7, it normally lasts from mid September until the end of December. In 2012, much more data has been generated and as a result, the workload for reconstruction, stripping and reprocessing

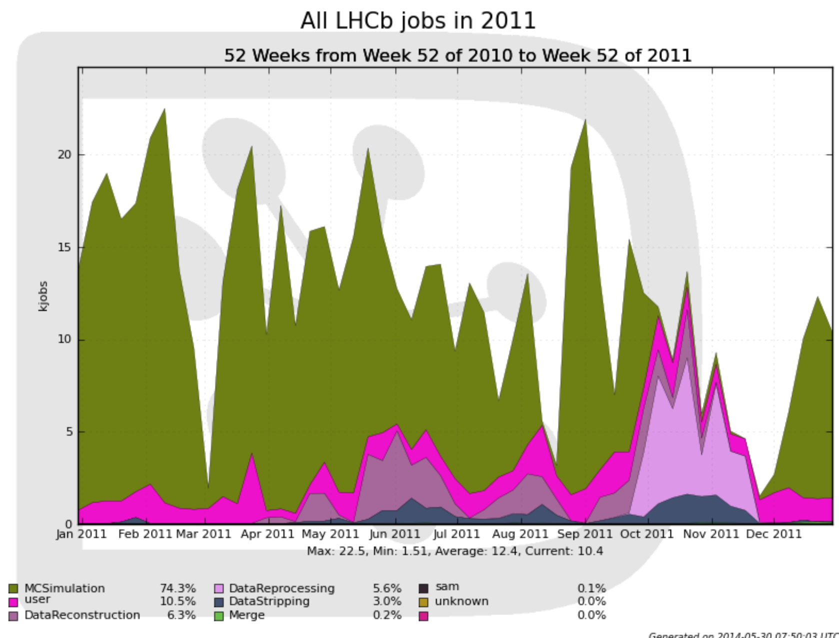
has significantly risen. In February 2013 the Large Hadron Collider has been shut down. As Fig. 2.7 shows no reconstruction jobs have been submitted after mid of March 2013. Simulation jobs contributed with about 80% to the total workload of 2013.

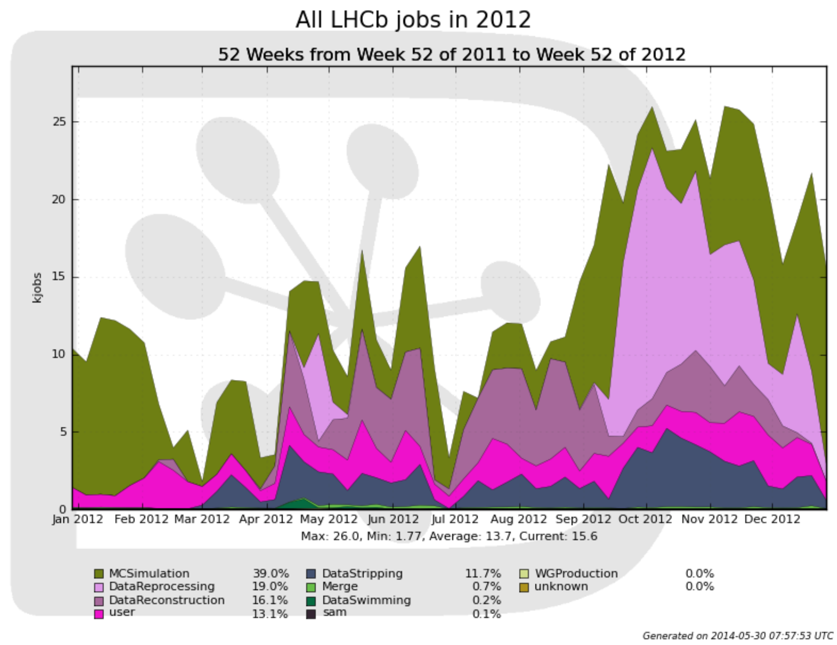
Run 2

In 2015, LHC will be restarted with 13 TeV center of mass and much larger workloads are expected. As described in detail in [74] the HLT output rate will be increased up to 12.5 kHz while 2.5 kHz will be directly reconstructed on the computing farm. Bunch spacing will change from 50 ns to the nominal 25 ns. At LHCb an increase of up to 50% of luminosity is expected which also means that the HLT output rate will rise [72]. It is estimated that the rate will approximately double and will reach about 700 MB/s. Furthermore, two output streams will be made available: one for data which must be processed immediately and one for data which has to be parked and processed later when resources are available. It is expected that after 2015 not enough computing resources will be available in order to process immediately the larger amount of data. A workload of 185 kHS06.y is assumed for the year 2015.

Long Shutdown 2

The long shutdown 2 (LS2) will take place from 2018 until 2019 and the LHCb detector will be upgraded during this time period. The main goal of the upgrade is to remove the hardware trigger entirely [54]. This presents a large computing challenge, since the HLT trigger needs to process a data rate of 40 MHz instead of 1 MHz. Sub detector elements are upgraded, in order to cope with the higher data rate. As explained in [54] readout and electronic systems will be completely redesigned, in order to provide 40 MHz to the HLT trigger. The output rate of the software trigger will increase from 12.5 kHz to 20 kHz.





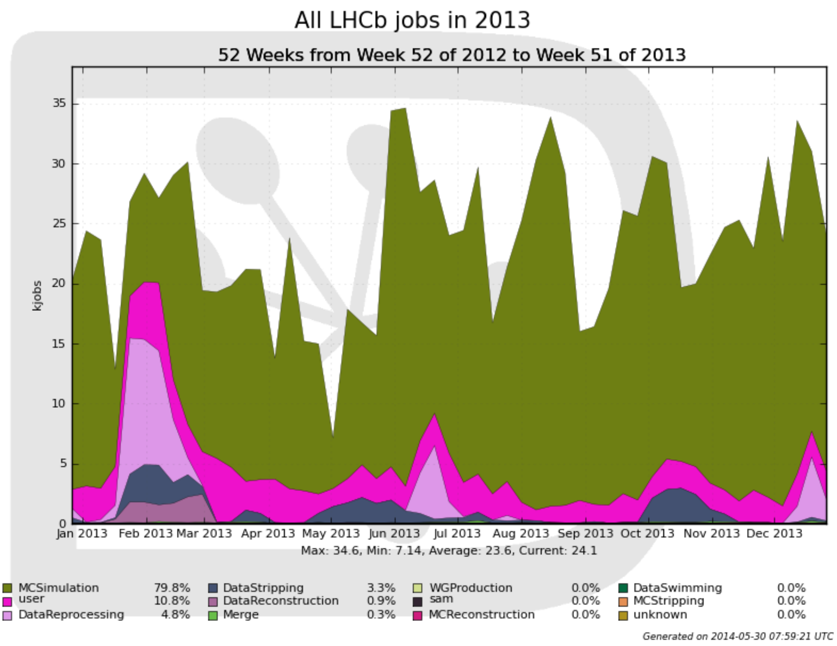


Figure 2.7: LHCb jobs in 2011, 2012, 2013

3. Related Work

As explained in section 2.5 efficient resource usage becomes more difficult since the start of the multicore era and VOs face challenges from High Performance as well as from High Throughput Computing. Consequently, the thesis touches topics from many different areas and most of them have their origin in High Performance Computing, like estimation of runtime, multicore usage in a node or moldable job scheduling. Related work is given in the following sections. It starts with an overview of the computing ambitions within the HEP community in section 3.1. Afterwards, section 3.2 presents related work for memory deduplication. Section 3.3 focuses on related work in the area of scheduling of multiprocessor tasks and improving estimation of job requirements.

3.1 High Energy Physics Applications in the Multi- and Manycore Era

Multicore R&D Project

All experiments at CERN are facing the same challenge and several groups are working towards better utilisation of multi- and manycore CPUs. This resulted in an R&D project which started in 2009 [27]. The aim was to regularly discuss new technologies and concepts between experts from different experiments. A summary of the whole project can be found in [190]. As shown in this presentation, the Copy-On-Write principle is very promising in the context of HEP software. It allows to automatically share read-only memory between forked processes. This alleviates the problem that the throughput of HEP applications might be limited by RAM size. As stated in [190], I/O presents another bottleneck, since multiple processes write multiple output streams. The R&D project also discussed whole node scheduling. As explained in [190] a new processing model requires a new model for allocation of computing resources. The aim was to allocate resources, such that parallel applications use all processing units on a system. However, providing a whole node or only a subset of multiple cores is a decision made by the grid site [17]. The R&D project also discussed the applicability of multithreaded software frameworks in HEP which aims to run algorithms in parallel. More details will be discussed in section 6.1.

Concurrency Forum

After the R&D project had finished in 2011, the concurrency forum has been created where the focus is much broader and not only relying on the parallelization of software

[36]. Topics discussed in this forum are for example vectorizing code, new types of hardware like Graphical Processing Units (GPUs) and CPUs for embedded systems like ARM, multithreaded software frameworks, identification of software bottlenecks, new CPUs and platform models. A status of the forum from 2013 can be found in [134]. As explained in this presentation the computing resources will not follow with the increasing requirements of simulation and data processing. Consequently, software has to become faster. Significant speedups can be achieved by using vectorized libraries and should be leveraged by HEP software. [134] also illustrates that a combination of several techniques will be required in the long term future. For instance, combining multithreaded with multiprocessing approaches that makes use of vectorized libraries and profits additionally from accelerators like GPUs. However, such efforts are only worthwhile if they will be supported by the grid sites.

Multiprocessing Approaches

In the beginning many experiments were applying the multiprocessing approach, in order to reduce memory consumption of jobs. The model of the parallel prototypes are different since the underlying software frameworks are not the same. The ATLAS experiment has developed a prototype [86], where each worker processes different events and they are forked after the main loop has started. Therefore, it is necessary to restart the main loop in order to reset event counters and histograms. They also investigated via simulation how singlecore and multicore jobs can be scheduled on the same resources. However, they assume in their simulations that the number of job slots is always the number of processes which is required by the multicore job. CMS presents a multiprocessing approach in [123], which is based on a similar principle. First the program starts the initialization and as soon as forking is applied all files are closed and reopened. Before starting the child processes a range of events is determined that has to be processed by each worker. They open then the input file and process only a part of it. They make clear that a multithreaded framework is necessary when latency and memory ratio shall be further optimised. Therefore, it will be required to explore sub-event parallelism, where algorithms are executed in parallel.

Multithreaded Approaches

The CMS experiment focuses on a multithreaded version of its software framework. One of the most computing intensive tasks is the tracking which determines the tracks of particles. The main goal is to parallelize the track reconstruction. CMS uses several technologies, like OpenCL [48], OpenMP [49] and Intel TBB [37] to run the tracking in parallel threads [116] and nearly linear speedups have been achieved. However, they could also observe that running such tasks on a GPU is very expensive due to the slow data transfer between CPU and GPU. They evaluated that processing less than 10k tracks on a GPU is not worth [19]. ATLAS and LHCb share the software framework, that is further developed into a multithreaded software model. The aim is to run different algorithms in parallel and to scale up to hundreds of cores [117]. More explanation about this model is shown in section 6.3.

The 7 Performance Dimensions

[129] describes that modern processors provide seven performance dimensions (Fig. 3.1). They can be distinguished in intra-core and extra-core dimensions.

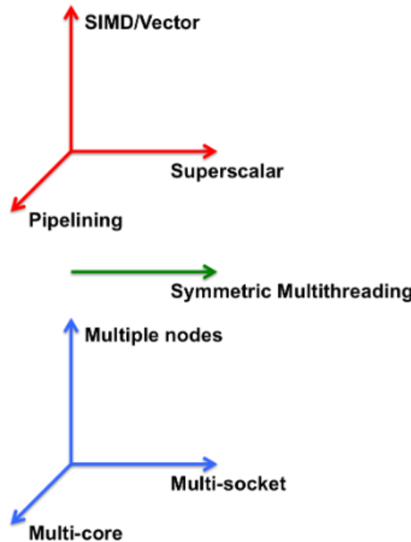


Figure 3.1: The 7 Performance Dimensions (taken from [129])

Single threaded applications can profit from features like pipelining, superscalar and SIMD/ Vector in order to increase their performance. Pipelining increases the instruction throughput, since multiple operations (fetch, decode, execute, access and write back) can be performed at the same time. Superscalar processors facilitate instruction level parallelism, since pipelines are replicated and it allows to fetch, decode and execute several instructions at a time. SIMD stands for single instruction multiple data, and allows to execute an instruction on multiple data. This is facilitated by vector units. SSE (Streaming SIMD Extension) is an extension of the instruction set in order to perform SIMD operations and to profit from a performance boost. Intel firstly introduced AVX (Advanced Vector Extension) with the Sandy Bridge micro architecture, where vector registers have been increased from 128 to 256 bit. If a software consists of instructions that are executed several times on different objects, then it can use vector units in order to perform these tasks faster. Either developers must write intrinsics code or compilers must produce assembler code that can profit from vector units. If a CPU provides 4-wide vector units then the maximum speedup can be 4. Consequently, many LHC experiments are also focusing on vectorization since this can give an additional performance boost for serial tasks [116], [97]. Such optimisations become more difficult with the rising complexity of modern CPUs. Since the start of the multicore era, new dimensions have been introduced, which are multi-core, multi-socket, multiple nodes (Fig. 3.1). These require a parallel execution of applications.

WLCG Multicore Task Force

Executing jobs as multiprocessor tasks requires changes in the software but it must be also supported at the grid sites. The complexity of resource configuration increases and this can lower the CPU utilization [44]. Hence, the main objective is to define procedures for managing and scheduling resources more appropriately and to derive metrics for proper accounting of CPU usage. Especially, latter point plays a major role at certain grid sites, since funding often depends on how efficiently the system has been utilised. If multicore jobs decrease the CPU utilisation, this might lead to budget cuts. [44] also indicates that scheduling multicore jobs is not only a site but also a VO problem. They must ensure an internal dynamic scheduling of jobs with different requirements. The different approaches proposed by the VOs are presented as introduction into chapter 7.

The goal of this thesis is to evaluate the applicability of different parallel concepts with respect to HEP software. It is shown how to improve the multiprocessing prototype of the

LHCb software framework which is currently a full working prototype. Optimisation is applied with respect to memory footprint and limitations are evaluated within extensive benchmarks.

3.2 Memory Deduplication

Section 4.1 shows, that the prime reason for executing jobs as multiprocessor task is to reduce the overall memory footprint of HEP software mainly because memory to CPU power ratio on future manycore CPUs will decrease. Since this is a common problem, many different research groups are working on tools for automatic memory sharing and reduction. For example, duplicated memory is a significant problem in the context of virtualization, when several instances of an operating system are started loading similar libraries. A duplicated memory page can be easily removed by storing only one copy in shared memory. Since the start of the multicore era, memory deduplication is subject of research and many tools have been developed by different communities which can significantly improve memory performance [132]. As stated in [132]:

> *"... there is compelling evidence that exploiting similarities found in the contents of system memory is a promising research approach for improving the resilience and memory bandwidth of extreme scale systems."*

If the contents of two pages are identical, they will be shared. This is known as content based page sharing. A hash function is applied on the data, in order to determine whether two pages are identical. This mechanism is used by the Kernel Same Page Merging tool (KSM) [58] and can significantly improve the overall memory footprint. [58] explains the detailed implementation of that tool and they evaluate how many Mega Bytes per second can be shared. It has been implemented for the hypervisor KVM but can also be used in other contexts, like parallel applications. VMware ESX applies the same technique [191]. [63] proposes a novel technique of creating a mergeable cache, which applies a similar technique. Equal memory blocks are automatically detected and merged. They show that this reduces the amount of off-chip memory accesses and rises consequently application performance. They have evaluated this for few benchmarks and they can show a speedup of 2.5 on average.

This thesis evaluates how memory deduplication can be used to reach further improvements in memory reduction and how it affects the runtime of the applications.

3.3 Job Scheduling

Batch systems coordinate and monitor tasks submitted by users. They allocate resources, apply limits, manage the running environment and handle priority queues. Current batch systems used by the grid sites are for instance HTCondor, SLURM, Univa Grid Engine and Torque. [105] explains that Torque is suffering scalability problems if it has to handle more than 4000 jobs and if a system provides large number of cores. SLURM is used at many super computer centers because it can easily scale up to 65k worker nodes [105]. Additionally it supplies many scheduling functionalities. The Univa Grid Engine does not show any scalability problems with 20k job slots [124]. HTCondor is a reliable batch system, which could prove to scale up to 30k simultaneous jobs [124]. The core element of a batch system is the scheduler which gives reservation to jobs waiting in a queue and it defines their start date. Scheduling is subject to optimisation of certain performance criteria like for instance:

- **Makespan:** Time required to process all jobs in the schedule.
- **Turnaround Time:** Duration of task submission to its finalization.

- **Waiting Time:** Duration of task submission to its start of execution.
- **Job Throughput:** Amount of jobs processed within a schedule.
- **Utilisation:** How much the CPU has been utilised.

Tasks have different properties, like core count, runtime, bandwidth and memory. For instance, users can indicate a minimum and maximum number of processes and a scheduler can modify this parameter in order to improve performance criteria. According to [95] parallel tasks can be classified as evolving, rigid, malleable or moldable. In the first two cases, it is not subject to the scheduler to change the number of processors required by a job. In the latter two cases, the scheduler can modify this value in order to optimise performance. Section 7.2 will explain the different types and their applicability on LHCb jobs. Mechanisms for multitask scheduling must be provided by the pilots sent from the VO when jobs are executed in parallel. The aim is to determine for each job the best degree of parallelism. This is subject to moldable job scheduling.

3.3.1 Optimising Scheduling Performance by Using Moldability of Jobs

[168] proposes an iterative algorithm, which assigns an additional core to the job which profits the most from it. The authors can prove that it always finds the minimum average turnaround time (the time from submission until completion of a job). [127] applies the same technique, however they treat jobs as malleable. This means that a job can change its degree of parallelism during runtime and this is initiated by a scheduler. They implement an agent-based distributed resource management scheme and compare it against a centralized resource management system. An agent assumes that applications are malleable and it evaluates whether it is more beneficial for the overall gain in speedup when cores are moved from one application to another one [127]. In other cases, makespan is the criteria to optimise which leads to a two dimensional strip packing problem [187], [131]. The point in time when the last job finishes shall be as early as possible.

[89] and [91] postulates that speedup curves are sufficient to determine the best degree of parallelism for each job. [89] defines an average and variance in parallelism to describe and predict speedup curves. This allows to determine the point when cores cannot be used efficiently any longer such that an upper limit can be derived. They show that strategies which respect either average in parallelism or variance in parallelism achieve similar results. [91] evaluates the trade-off between efficiency and speedup which are important parameters for processor allocation. They postulate that the knee of a speedup curve is the point which maximizes the benefit per unit cost. They evaluate the exact location of the knee by inferring the average parallelism of an application.

[82] and [81] propose to modify jobs at submit time because information about the current system workload can be taken into account. Users specify certain properties which might then be adapted by the scheduler if needed. They also show that most of the workloads executed at supercomputing centers are moldable even though users tend to submit jobs with partition sizes as a power-of-2. [81] presents a scheduling strategy based on the moldability of jobs. They show that their strategy can significantly improve turnaround time of jobs. They tested it in a variety of scenarios and could evaluate that the performance of the strategy decreases with rising system load. Another metric for optimisation is the stretch of tasks which is according to [169] the time a job spent in the system normalized by its execution time. [169] infers from measurements that this allows a more efficient usage of cluster systems. Many different metrics can be taken into account and depend mainly whether user or system criteria play a major role.

It is part of this thesis to define the scheduling problem for LHCb jobs. In contrast to the referred work, the problem is subject to high throughput computing. User metrics like

waiting time or turnaround time, that play often a major role in supercomputing centers, are less relevant in this context. The goal is to derive an optimisation algorithm based on the moldability of jobs which meets the requirements of high throughput. In contrast to other VOs the thesis proposes that scheduling of multicore jobs should be done by the VO and it should take application and job specific parameters into account.

3.3.2 Runtime Prediction

Scheduling requires knowledge about runtime of jobs, which is typically not present up-front [183]. Consequently, further optimisation can be undertaken by allowing the system to learn over time. This is also known as supervised learning and represents a large research area. A common technique is to group jobs based on certain similarities, for example tasks submitted by different users. [179] and [106] define templates in order to detect similar jobs. The novelty presented in [179] is the definition of search techniques in order to find similarities. The right balance between number of categories and jobs must be found. As explained in [179], if there are too many categories, each one will not contain a sufficient amount of jobs. If there are only few categories they might be too generic and group too many unrelated jobs. They construct a search space based on the given workloads and evaluate different search techniques like, Greedy and Genetic algorithm, in order to find good templates. Therefore, the algorithms compute the effectiveness of a template. [179] tested this approach on diverse workloads obtained from 4 different supercomputing centers. Their results show that this approach decreases the error in prediction by 14 to 49 percent compared to [106]. In [106] a historical profiler is proposed which estimates the runtime of jobs and the related uncertainty. Therefore, they categorize jobs by user, name and number of processors.

[130] proposes an approach that discovers correlations between parameters and runtime. They derive a subset of historical data via filtering and apply linear regression. They record for each job information like number of processors, average CPU load, average bandwidth and latency. They show that runtime prediction can be significantly improved when data is filtered beforehand. If many different features are involved more complex techniques can be applied like neuronal networks [121]. A vector of features is transformed via several hidden layers to an output value. [121] evaluates this on a large scale parallel benchmark whose execution time could have been predicted within an error of 5 to 7%.

The CMS experiment at CERN is also investigating how to improve the estimation of resources requested by user and reconstruction jobs as stated in [174] and [87]. Since user estimates are normally not reliable, an automatized way must be provided. [174] shows that pattern analysis can help in differentiating user behaviour. They propose that this can be used to indicate a maximum job runtime with high confidence interval. [87] analyses runtime estimation for reconstruction workflows. They show that the CMS experiment sees a wide range of luminosity values during a LHC run, which leads to a wide range of job lengths. Especially, jobs processing high luminosity data require a lot of time. They developed a job splitting algorithm which estimates runtime per event based on the luminosity. The splitting helps to achieve more uniform distributed job runtimes. Jobs with high luminosity data will have less events to process than jobs with low luminosity data.

The thesis investigates which kind of features are strongly correlated with job requirements. In contrast to [179], where the authors have to define the similarity between jobs first, it can be easily deducted for LHCb jobs since there are only three main types. Instead of focusing on generic parameters like size of executable, staged file and many more, as proposed in [179], this work focuses more on physics related job parameters. More features will be evaluated than it is proposed in [87]. And in contrast to other publications, estimation will not only be evaluated for runtime but as well for memory.

3.3.3 Backfilling

A moldable job scheduler computes an efficient offline schedule, which might become invalid when jobs finish earlier or later than estimated. Hence, mechanisms must be provided during runtime for filling gaps. Backfilling techniques have become a very popular approach, which allows to execute jobs out of order. They can significantly rise system's utilisation. A lot of research has been done in this field. The two types are [180]:

- **Conservative backfilling:** Jobs get a reservation which guarantees that no job in the queue will be postponed. It ensures that only jobs will be backfilled which do not postpone any tasks in the waiting queue.
- **Aggressive backfilling:** It only ensures that the first job in the queue will not be delayed. This can lead to repetitive delays for less prioritized tasks in the queue.

Both rely on the fact that the backfilled job shall not postpone the first task in the queue. EASY stands for Extensible Argonne Scheduling sYstem and is a scheduler that has been developed for the IBM SP [143]. It uses a backfilling strategy that belongs to the second category. It moves jobs ahead in the queue when resource utilisation can be improved [177]. Since this favours smaller jobs, it might lead to longer delays for large tasks.

The conservative approach versus EASY backfilling is analysed in [193]. They can show that system utilisation can be the same, but the estimation of time when a job can start is more precise with the conservative algorithm. Many different variants exist which try to select jobs depending on certain criteria, like for example Shortest Job Backfilled First or First Come First Serve. Since backfilling is based on accurate user estimates, the impact of wrong estimates is studied in [184]. They propose to automatically generate predictions instead of relying on users. They show that the average wait time of EASY backfilling can be improved by 25%. They also evaluate that using a Shortest Job Backfilled First strategy can improve the average slowdown up to 47 percent. [78] also studies the impact of runtime estimation on scheduling performance. They show that a more precise estimation can improve average slowdown and wait time for short jobs. They also evaluate that jobs with better prediction see a larger improvement in performance.

One can summarize that the major problem is the runtime prediction in scheduling and backfilling. In many cases, runtime estimation is subject to an underlying distribution as stated in [144]. Some values are more likely than others and the aim is to choose the one with the highest probability. [144] evaluates a probabilistic backfilling which assumes a certain distribution of runtime values. The aim is to backfill tasks if the probability is minimal that the first job in the queue will be postponed. They use dynamic programming in order to determine the overall probability of free slots and job completion times. They show that this approach significantly improves system utilisation compared to EASY backfilling.

This thesis undergoes a detailed workload analysis and can confirm in section 7.5 that the required time per event fits a distribution. This allows to conclude that using a backfilling strategy as proposed in [144] is the most suitable one. This thesis will evaluate how this approach can be applied in the context of a moldable job model. A detailed overview of the probabilistic backfilling strategy is given in section 7.6.

4. Problem Description in Detail

After having described the LHCb experiment and its related data workflow in detail, the aim of this chapter is to introduce the problem of multi- and manycore CPUs from the perspective of LHCb computing. The upgrade of the Large Hadron Collider puts experiments under pressure because larger data rates and more complex events are generated. This is explained in section 4.1. The current job model supports a suboptimal utilisation of resources which presents a limitation. Details are shown in 4.2. Finally, a proposal of optimisations is given in section 4.3. It shows what can be undertaken, in order to improve utilisation of multi- and manycore CPUs and it represents the baseline of the thesis.

4.1 Experiment Upgrade

The LHC experiments are not only facing problems related to future manycore CPUs but also due to the upgrade of the LHC. In 2015, LHC will operate at a center of mass energy of 13 TeV. At higher energies more particles are produced and therefore more information must be stored within one event. The larger the event the more complex the reconstruction. This results in larger amount of raw data as well as higher computational complexity.

Apart from that, an increase in the average multiplicity is expected. Multiplicity indicates how many collisions took place per bunch crossing. The larger the total cross section of the two beam lines, the more pile-up interactions are produced which indicate the number of interactions in one event. The higher this value the more complex it becomes to compute which particles belong to which decay and collision. Besides a larger computational complexity, it also results in higher memory requirements. This becomes problematic when facing at the same time a decreased memory ratio on future CPUs. In the worst case, not all available processing units can be used or jobs suffer performance penalties due to swapping memory pages to the file system. This has already become a problem in the High Level Trigger, where due to the change from 32- to 64-bit the memory requirements of the trigger software has significantly risen because the software uses millions of pointers. Changes had been necessary in order to be able to run as many trigger instances as processing units were available [102]. This problem also occurs with respect to certain LHCb jobs executed in the LHC Computing Grid and it will aggravate in the near future.

Furthermore, the budget cuts taking place in many collaborating institutes as a consequence of the financial crisis of 2007 are not negligible. At many Tier-1 and Tier-2 sites capacity will only barely increase in the coming years as they operate at best at constant

budget. This impacts the planned operations of WLCG [166]. Summing it up, it is more important than ever to use available resources as efficiently as possible in order to be able to process future workloads on modern manycore processors.

4.2 Current Job Model

The job model foresees to execute each job as a single threaded application on one processing unit. Neither this model nor the software itself exploits the full potential of multi- and manycore CPUs. However, scheduling is rather simple since just the time dimension matters. Job requirements like memory and runtime are not automatically predicted. This is done by hand and generally much more CPU time is requested than actually needed. Runtime is a critical parameter, because a job will be interrupted by the batch system of the grid site, when it does not meet the deadline. If it finishes too early, either another job will be matched or the worker node will be released. In latter case, it is then up to the grid site to backfill the node. Memory is less critical, since a job can still show good performance as long as swapping is enabled at the grid site. If it is not supported, certain sites interrupt jobs when they address more than 4 GB or require more than 2 GB of physical memory. When this happens, jobs have to be restarted at sites which do not apply these restrictions. As a result, there is a lot of room for improvement in the current job model in order to use and request resources more appropriately.

A parallel execution of jobs can improve utilisation of multicore CPUs and lower the overall memory footprint. Parallel tasks are able to share detector related information, like detector description, run conditions, magnetic field map and job options. This data is large and is accessed in read only mode. Since it can be shared, the total memory footprint grows sublinearly with increasing number of parallel processes. Additionally, performance impacts due to concurrent accesses to file system and network resources can be coordinated when jobs are executed as multiprocessor tasks.

Multicore jobs are currently only allowed at certain grid sites, where queues have been set up for testing purposes. Tests executed by the ATLAS and CMS experiment have clearly shown, that the largest problem is the runtime estimation of jobs. The aim is to run multi- and singlecore tasks on the same resources. The finishing time of jobs must be defined, in order to reserve a multicore job slot (Fig. 4.1). Since time is in general significantly overestimated, backfilling cannot be applied leading to degraded resource utilisation [90].

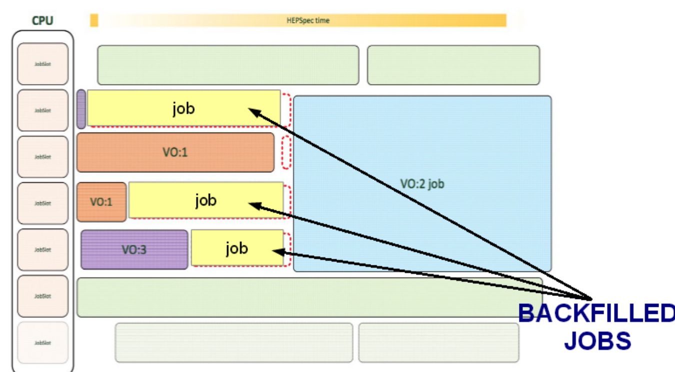


Figure 4.1: Single and multicore jobs (taken from [90])

Practically, grid sites can just see pilots running on their worker nodes, but not which tasks have been picked by them. This leads to the problem, that the local batch system cannot optimise its task queues since it does not know which job is going to be executed. Consequently, the main problem of the current job model is, that only the experiments have

knowledge about job specific parameters and are able to estimate workloads. However, the grid sites providing the computing facilities rely on information given by the experiments.

Therefore, the thesis proposes a job model, which applies optimisation at different levels, including parallel execution of LHCb jobs, scheduling mechanisms for multiprocessor tasks and creating a history based estimation for job runtimes. It will be evaluated how job requirements can be estimated more precisely by using information about prior jobs. This is still an unsolved issue in many experiments. The LHCb experiment stores all metadata and provenance information about jobs in databases, like the Bookkeeping [135] and the DIRAC Accounting [71]. Much information is available, like for example the conditions during an LHC run, the size of the input and output files, average luminosity and many more. This can be used to find correlations between features and estimate future job requirements.

4.3 Proposed Solutions

Optimisation has to remove several bottlenecks at the same time. There is no point in having a good parallel software concept if throughput deteriorates due to under- or overestimated job requirements. The same applies for the speedup of a job. Mechanisms must be applied if an application does not scale beyond a certain degree. Imagine the case that a worker node provides 10 cores, a job might be executed with 10 processes. Having 100 processing units, the software will likely not scale sufficiently well as it is shown in section 6.5. Due to synchronization and serialization of objects, software does often not scale linearly with the number of cores. On top of that, the Grid offers a high variety of computing resources which differ in manufacturer, micro architecture, number of sockets, job slots and many more. Each of them scales differently depending on configuration and current workload. As it has been shown in section 2.5 (Tab. 2.1), identical CPUs have quite different HEP SPEC values depending on their configurations. Section 6.5 evaluates parallel software on different multi-socket CPUs and depending on the number of interconnects between sockets or features like frequency scaling software scales differently. Besides, software modifications and experiment conditions have a large impact on runtime, speedup and memory footprint of applications. These parameters can often change and it implies that learning algorithms should be implemented. These are able to regularly evaluate jobs and to adapt to new conditions. This is discussed and evaluated in section 7.5. Since the LHCb experiment runs 10k jobs every day, it is not feasible to do such estimation by hand. Because of these issues, the thesis explores a workflow model where optimisation takes place at many different levels (Fig. 4.2).

Intrusive Optimisation

The biggest challenge is the parallelization of software, such that the different jobs like reconstruction, stripping and simulation can be executed with several processes. As it will be evaluated in section 5.1, LHCb applications allocate a lot of memory, but access most of it in read-only mode. As a result, memory can be shared between parallel instances. An efficient technique to do so is the Copy-On-Write principle (COW) which keeps a memory page in shared memory as long as no process writes to it [66]. It is the aim to use this feature for lowering the memory requirement of the applications. This implies modifications within the framework which are preferred such that the core software can remain the same. This can guarantee that the key features of the software like transparency and flexibility are still provided to the users. This is of course a trade off between the resulting scaling behaviour and the impact on the software model. Different parallel concepts are evaluated in section 6.1 and the current status of the implementations is shown in section 6.5.

Non Intrusive Optimisation

In addition, software can also be optimised in a non intrusive manner by using tools for memory deduplication or compression. Especially in the context of cloud computing it is worth profiting from such tools since virtual machines can be arbitrarily configured. In the last years, many functionalities for memory reduction have been developed by different communities. The most promising ones and related problems are shown in sections 5.1 to 5.3. Such tools do not only allow an improvement in memory requirement and/or runtime, but also help to get a better understanding of the application's behaviour.

Workload Scheduling Optimisation

As shown in Fig. 4.2, changes are also required at the level of scheduling. This deals with the problem of mixing jobs in an appropriate way, such that efficiency does not significantly decrease due to the non linear scaling of applications. Therefore, a scheduler must make decisions (Fig. 4.2). Since VOs have knowledge about workloads and prior job runtimes, such scheduling should be part of the VO's Workload Management System. This will be detailed in section 7.2. A scheduler can determine the degree of parallelism of each job based on its properties and the state of the worker node and based on information obtained from prior jobs. LHCb applications are moldable, which means that they can be in principle executed with an arbitrary number of processes. This property can be used to optimise scheduling decisions and therefore to increase the overall job throughput. This is evaluated in section 7.4. A scheduler must predict requirements, like memory and runtime. The aim is to use a history based estimation where information can be obtained from databases, like Bookkeeping [135] and Accounting [43]. Though, it must be investigated which job parameters have a large impact on its resulting memory and runtime. Data mining techniques can be applied in order to model a hypothesis. However, for making good decisions the scheduler has to learn over time how different software versions scale on different micro architectures. Job meta data must be stored in the databases, they must be measured and fed back to the algorithm. Optimisation can also be applied at this level, since a better resource allocation improves throughput as well. This will be discussed and evaluated in section 7.5.

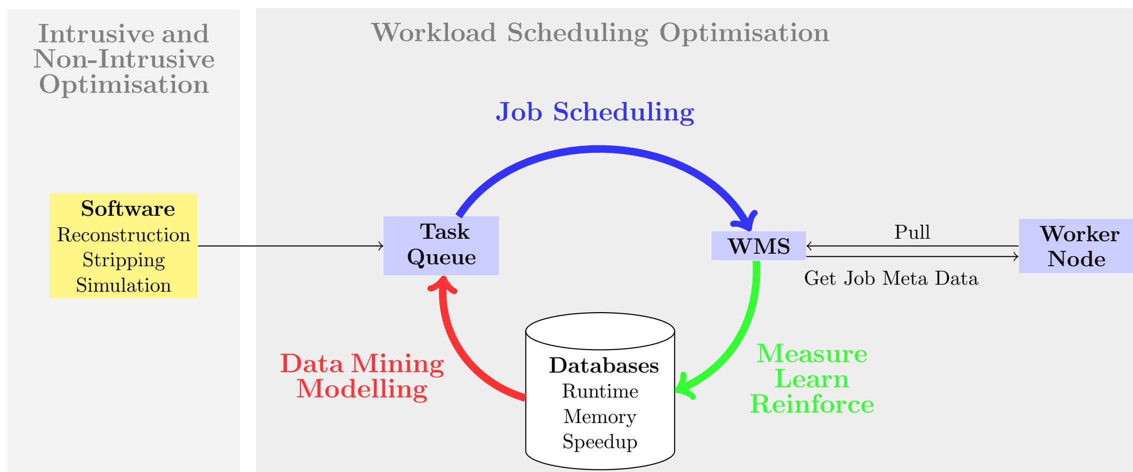


Figure 4.2: Optimisation at different levels

The status of the work has been regularly presented within the LHCb collaboration and other HEP meetings, like LHCb Software and Analysis Week, the Forum on Concurrent Programming Models and Frameworks, ROOT Users Workshop and LHCb Computing Workshop [154], [155], [156], [157], [159], [153], [158], [162], [161]. Results of the work have

also been published and presented in different symposia, workshops and conferences [160], [164], [151], [152], [163].

5. Optimisation with Non Intrusive Techniques

This chapter will focus on how single tasks can be optimised by using external tools provided by the operating system or by the compiler. There is a large number of functionalities and it would go beyond the scope of the thesis to evaluate all of them. The chapter will focus on recent developments which are promising in the context of LHCb applications with respect to memory. Since the start of the multicore era, many new tools have been developed by different communities. In the current state of the Worldwide LHC Computing Grid such technologies cannot be used in a straightforward way due to missing support. Many tools require special kernel versions, configurations or root rights. Nevertheless, a transition of paradigm is currently taking place leading from Grid to Cloud Computing [64]. This opens new possibilities, since jobs are executed within a virtualized environment where the job owner has full control over the virtual machine. Consequently, the operating system within the virtual machine can be configured such that the software profits the most from it.

Section 5.1 evaluates automatic memory deduplication. Section 5.2 discusses advantages of 32- and 64-bit LHCb applications with respect to memory and it shows how LHCb software can profit from x32-ABI. Section 5.3 evaluates memory compression in the context of LHCb jobs.

5.1 Memory Deduplication

Developers in the field of Cloud Computing share the same problems with respect to required memory and future memory ratio. If several instances of the same operating systems are created, many system and kernel libraries can be actually shared. Since they are used in read only mode, they can be kept in shared memory and used by all instances at the same time. This reduces significantly the overall memory footprint. It has lead to the development of the Linux module KSM (Kernel Samepage Merging), which allows to automatically detect and share equal memory pages [58]. It has been originally developed for the hypervisor KVM but can also be used at the level of applications. The module is based on the Copy-On-Write principle and as explained in section 6.1, pages remain in shared memory as long as there is no process writing to it. It has been evaluated in the context of LHCb applications because it allows to share memory without modifying the software itself. Results have been presented in [164], [154] and this section mainly refers to these publications.

The KSM thread runs in background and scans regularly pages which must have the standard size of 4 kB. If it detects equal pages, one copy is kept in shared memory and the rest is removed. The CPU consumption required by the KSM thread might become quite large dependent on how frequently pages are scanned. It is worthwhile to tune this parameter in order to avoid a slow down of the main applications. The parameters *pages_to_scan* and *sleep_millisecs* define the maximum merging rate, which can be computed as:

$$\frac{\text{page_size} \cdot \text{pages_to_scan}}{\text{sleep_millisecs}}.$$

By default these parameters are set to 100 pages and 20 ms which results in a merging rate of 19.5 MB/s. A kernel interface allows to register pages which have to be scanned by the KSM module. After this, a check sum is computed which later on allows to identify whether the content has changed or not. If this is not the case, the content is then bytewise compared with already registered pages. Two internal data structures facilitate the handling of registered pages, the so called stable and unstable tree [58], which are implemented as a red-black tree. Such trees support binary search and also allow an efficient modification with a complexity of $\mathcal{O}(\log_2 n)$, where n is the size of the dataset [61]. Pages which have been shared by the KSM thread are registered in the stable tree. Potential candidates for sharing are taken into account by the unstable tree, in case the content of the memory page has not changed for a certain period of time.

Requirements

As already described in the previous subsection, memory pages must be registered. This can be done via the madvise kernel interface, which is part of the Linux kernel since version 2.6.32 [125]. It is a tool for memory handling, which allows to indicate whether a page is mergeable and must be scanned by the KSM thread. A page can then be measured as follows:

- **Sharing:** The reduction of memory in number of pages
- **Unshared:** Presents the number of pages whose content is not equal to any of the registered pages
- **Shared:** The amount of pages that are currently shared
- **Volatile:** Number of pages whose content changes too often and cannot be registered within the stable nor unstable tree

An Example: *A process allocates an array containing* 10^7 *zeros*

Number of pages: $(10^7 \cdot \frac{\textit{sizeof}\ (\text{int})}{1024}\ \text{kB}) \cdot \frac{1}{4\ \text{kB}}$ *= 19531*
Sharing: 19530 pages
Shared: 1 page

Since all pages have the same content, KSM has only to keep one single memory page which is accounted as shared and 19530 pages can be removed. From this example, it can be deducted that a high ratio between sharing to shared indicates a good efficiency.

In order to use KSM from within the software, madvise calls must be inserted. This can be done directly inside the framework or via a malloc hook. Latter option intercepts malloc calls from a process and allows to modify them. As a result, all allocated memory pages would be accounted as mergeable and have to be scanned by the KSM thread. For testing purposes, the merging rate must be adapted to the rate with which memory is allocated by an application. Stripping jobs allocate for example about 40 MB/s during the initialization

period. Having 8 workers leads to an allocation rate of $8 \cdot 40$ MB/s $= 320$ MB/s in the case of forking subprocesses before the actual initialization. Within the tests, the values have been set once to 20 ms and 3000 pages leading to a rate of 585 MB/s. Afterwards, the rate has been risen up to 20 ms and 10^6 pages leading to 190 GB/s. All 4 different categories of pages have been monitored during the execution of the tests and results are shown in the following subsection.

Benchmark Results

The KSM module was tested on an Intel Xeon processor (L5520) with 8 physical cores and hyperthreading disabled. The operating system was Scientific Linux 6 [18]. A version of LHCb's parallel prototype was used [137], in which subprocesses are forked before the initialization. During the period of initialization many libraries and datasets are loaded that can be shared between parallel processes. This can be achieved by using KSM and can be compared later on with an improved implementation of the parallel prototype. Tests have been executed with different number of 2, 4 and 8 worker processes.

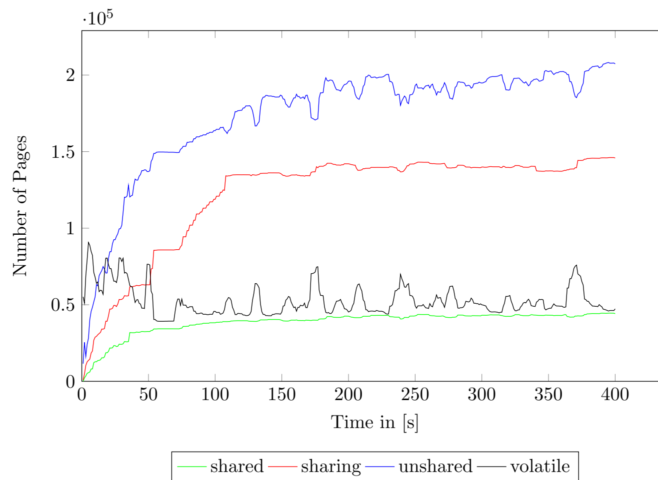


Figure 5.1: Results for simulation job running with 2 workers (taken from [164])

Fig. 5.1 shows the results for a parallel execution of a simulation job with 2 worker processes. After the initialization period the parameters shared and sharing reach a stable value. This is due to the fact that LHCb applications allocate hundreds of MBs in the beginning but then do not modify it during the main loop. Datasets like detector description, magnetic field map and run conditions are used in read only mode. Such pages can remain in the shared memory and this can consequently save up to hundreds of MBs. The volatile and unshared pages' parameters are counteracting, since a positive peak on one side causes a negative peak on the other side. These fluctuations are caused by pages to which tasks write quite often, but which cannot be merged due to frequently changing content. It is likely that a single peak corresponds to a preallocated file buffer, which is listed as volatile and is accounted as unshared afterwards. Pages accounted as volatile or unshared do not reduce the overall memory footprint.

Reaching the stable value for shared memory is delayed when more worker processes are initialized. Fig. 5.2 shows the same test case but executed with 8 workers. In contrast to the 2 worker case, the stable value is now reached after 200 seconds and the number of volatile pages rises significantly in the beginning (up to factor 5). This is related to the larger amount of allocated pages, which KSM cannot handle at once. Consequently, many pages are accounted as volatile before they are scanned and categorized as unshared. Stripping jobs are in general memory bound, since they deal with several output streams

	serial mode	2 workers	8 workers
Simulation	183 MB (22%)	623 MB (33%)	2659 MB (48%)
Reconstruction	100 MB (8%)	448 MB (21%)	2297 MB (33%)
Stripping	165 MB (13%)	890 MB (26%)	3864 MB (32%)

Table 5.1: Memory reduction reached by memory deduplication in the different applications (taken from [164])

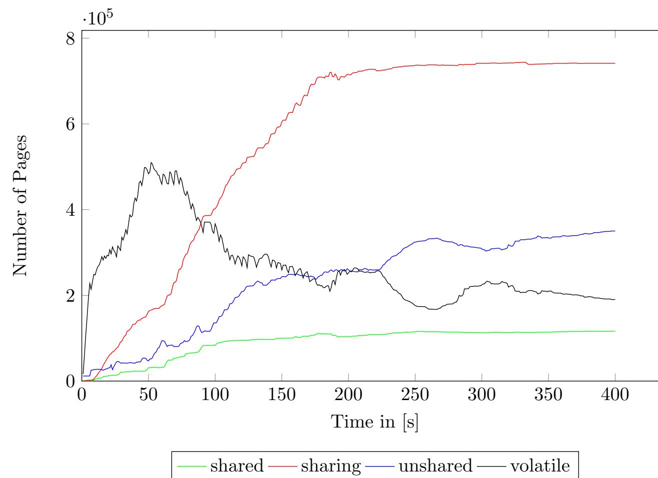


Figure 5.2: Results for simulation job running with 8 workers (taken from [164])

at the same time. Fig. 5.3 shows how much memory can be actually reduced by KSM for different parallel test cases of stripping jobs. Not only the initialization period lasts longer for more workers but also the peak of volatile pages rises significantly. In serial mode a stable value is reached after 80 seconds, with 4 worker processes after 200 and with 8 after 450 seconds. On top of that it has been evaluated whether results can be improved when the merging rate is adapted. It has been set to two very different values (190 GB/s and 585 MB/s). Fig. 5.4 shows that the maximum value of shared pages does not differ significantly and can also not be reached earlier. Furthermore, the peak of volatile pages still occurs and does not vary a lot. Consequently, setting the rate larger than the one at which memory is allocated does not affect much the final result.

Evaluation of the Kernel Samepage Merging Tool

Memory deduplication can notably improve the memory footprint of LHCb applications and Tab. 5.1 shows the results for the absolute and relative difference. As shown, many pages can already be shared in a serial execution of the applications. This is caused by empty file buffers, which are allocated but not used at all. Applications are analysed in detail in order to determine a minimum value of memory sharing. During the execution of an application, its malloc calls are intercepted. This allows to track the status of the dynamically allocated memory, which is stored on the heap. Libunwind is used to allow introspection of function calls and to resolve symbols [47]. Knowing the dependencies between functions, a call graph can be derived (Fig. 5.5). It is a directed graph, where each node represents a function. Fig. 5.5 shows an extract of the call graph obtained from a simulation job. Knowing which objects are accessed in read only mode, one can determine the minimum amount of shared memory. For instance, elements from the conditions database (CondDBAccessSvc) and all its subsequent items can be shared. They contain information about the conditions of an LHC run. As shown in Fig. 5.5 this counts ap-

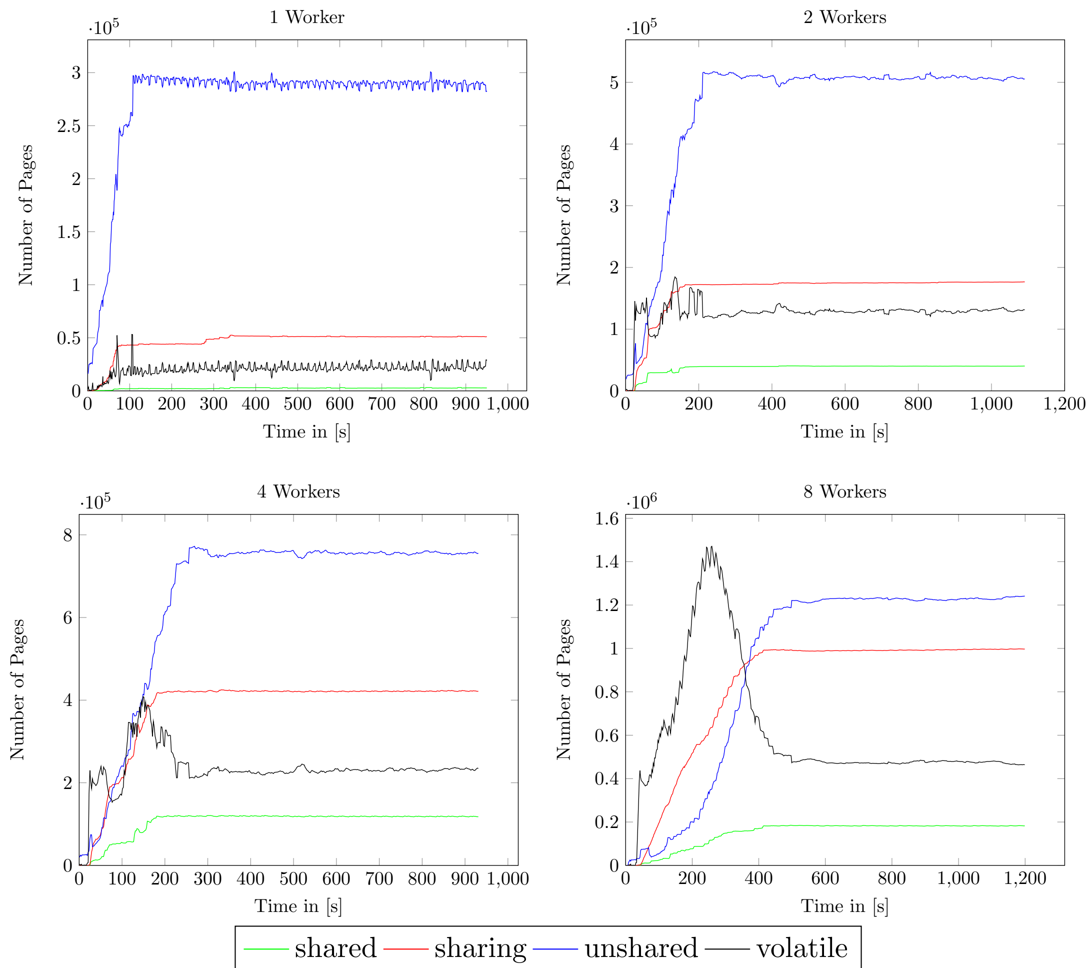


Figure 5.3: Results for stripping jobs (taken from [164])

proximately 150 MB. Having the complete call graph allows to determine, which instance requested how much memory and what can be in principal shared. An amount of about 230 MB has been evaluated. This results in about 230 MB $\cdot (n-1)$ of data what can be at least shared between n workers. Consequently, the more tasks the larger the reduction becomes. As it can be seen in Tab. 5.1 the KSM tool reaches at least this minimum value, which would be 230 MB for 2 workers and 1610 MB for 8 workers.

Comparing and sharing pages can be a CPU intensive task and might affect the performance of other applications. However, it appeared that the impact is less than 5% when an appropriate merging rate is configured. LHCb applications allocate most of memory during the initialization period. Afterwards, they reach a steady state during which most of the pages are accessed in read only mode.

It has been shown that the memory footprint can be reduced by a large factor and that the impact on CPU time is relatively small. Since LHCb applications use most of the memory pages in read only mode, it is worthwhile applying tools for memory deduplication. This can be either enabled by the VO in case it has sufficient rights or by the grid sites.

5.2 The x32 Application Binary Interface

A significant increase in memory usage has been observed when LHCb software moved from 32- to 64-bit applications. Nowadays, applications can be compiled as 32- or 64-bit, since the x86-64 architecture supports both types. x86-64 is an extension of the old x86

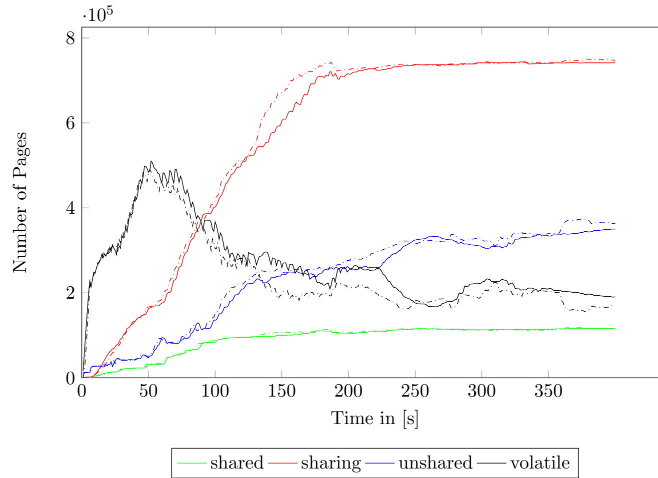


Figure 5.4: Comparison of the different merging rates 585 MB/s (continous line) and 190 GB/s (dotted line) (taken from [164])

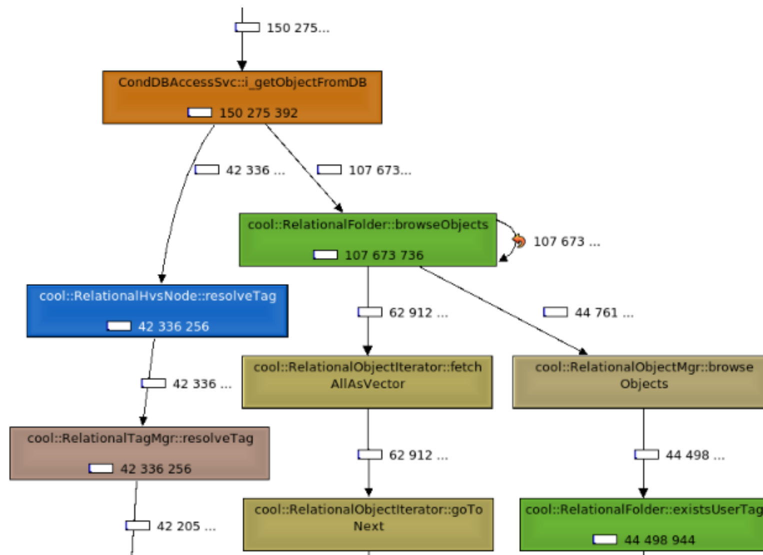


Figure 5.5: Extract of the call graph of a simulation job

architecture which supported a word size of 32-bit. The addressable memory is limited to 4 GB (2^{32}) which has become a problem with growing memory capacity of modern computers. In order to allow a task to address more than 4 GB, the word size has been extended to 64-bit. The new instruction set did not only provide a larger word size but also supported hardware features of modern CPUs in a better way. As a result, 64-bit applications can profit from faster system calls, better floating point performance and larger CPU registers [107]. Larger registers allow a faster computation of 64-bit integer values. As explained furthermore in [107], 64-bit applications return floating point values via SSE registers instead of loading them into the x87 register [142].

In general, 64-bit applications can perform better, since they take advantage of modern hardware features. However, if the memory footprint rises too much due to extended data types it can have a negative impact as it is analysed and presented in [194]. Performance can be decreased due to an increased number of cache misses, page faults, higher memory contention or paging. Consequently, there has been a lot of research done on how to profit from advantages of 32- and 64-bit application at the same time. One of these is the x32

Application Binary Interface (x32-ABI) developed by [107]. In 2012, this platform model has been officially introduced into the Linux kernel. It reduces the size of pointers and C-data type long to 32-bit. It is based on x64 instruction set such that x32 applications can profit from new hardware features in the same way. Since certain instructions use pointers in memory, 28 out of 300 system calls had to be modified according to [107]. It has been tested in the context of several CERN related applications and results have been presented in [151], [158] and [153]. This section mainly refers to this publication and presentations.

Requirements

The Application Binary Interface x32 has been firstly introduced into Linux kernel 3.4, but it is disabled by default. In order to use this platform model the kernel must be recompiled with the flag `CONFIG_X86_X32`. This allows it to distinguish system calls made by 64-, 32- and x32-applications. All these types can coexist on the same system, but cannot be linked against each other. Each of them has a different ELF-header (Executable and Linkable Format). The header of an x32-application looks like the following:

```
ELF 32-bit LSB shared object, x86-64, version 1 (SYSV)
```

It shows, that the binary uses 32-bit pointers but is based on the x86-64 instruction set. Furthermore, compiler, related c-libraries and tools must support x32-ABI, in order to generate such binaries. Support has been firstly introduced in glibc 2.16, gcc 4.7 and binutils 2.22 [15], [20], [16].

Benchmark Results

The x32-ABI has been evaluated for LHCb's reconstruction and stripping applications with respect to memory and CPU time.

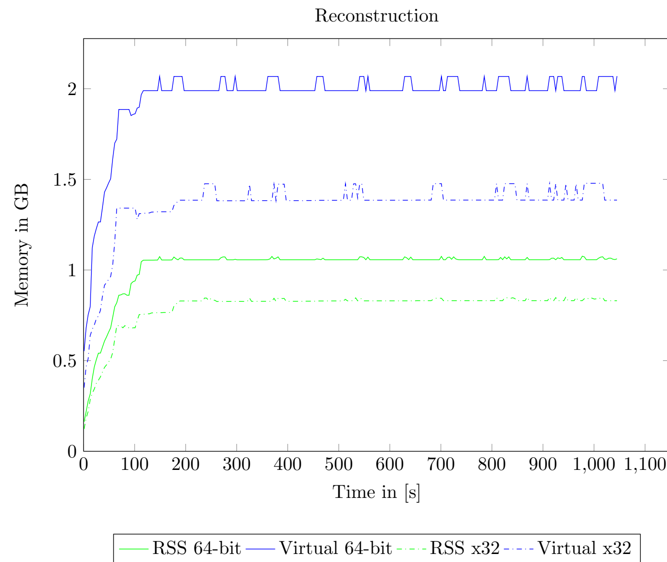


Figure 5.6: Comparison of memory consumption within reconstruction jobs (taken from [151])

Reconstruction

The memory requirements of reconstruction jobs increased by a factor of 1.6 due to the change from 32- to 64-bit. Instead of requiring about 700 MB, they allocate at least

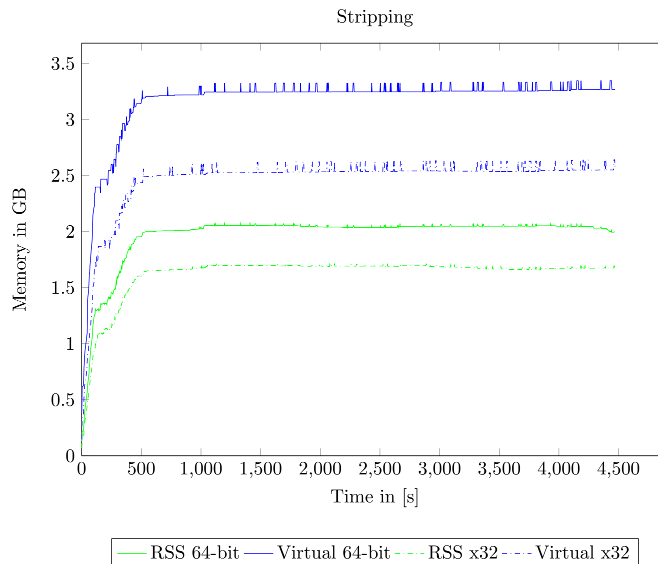


Figure 5.7: Comparison of memory consumption within stripping jobs (taken from [151])

1.1 GB of physical memory. Fig. 5.6 shows that reconstruction jobs can reduce their physical memory requirements by about 230 MB when they are recompiled as x32 binary. This corresponds to a reduction of about 20%. As it will be explained in section 6.5.4 applications normally address by far more memory than actually needed and this can be also seen in Fig. 5.6. Since virtual memory is limited to 4 GB at certain grid sites, it is also important to decrease it. The reduction is about 28% compared to the correspondent 64-bit application. The impact of x32-ABI on the CPU time of reconstruction jobs is minimal. Only a slight improvement of 2.7% has been observed (Tab. 5.2).

Stripping

The standard stripping case for 2012 data starts with 2.0 GB of physical memory consumption that increases during the main loop of the application. As can be seen in Fig. 5.7 physical memory can be reduced by about 17% and virtual memory by 21%. Only 2.5 GB of virtual memory are addressed instead of 3.2 GB. Physical memory consumption decreases from 2.0 GB to 1.65 GB during the main loop. CPU performance is 2.1% slightly worse in the case of the x32-binary. However increasing the number of events shows that this value can be reduced to 1.6% (Tab. 5.2).

	Reconstruction	Stripping	
Number of Events	1000	1000	10000
Average Total Time (64) in [s]	580.91	554.85	3701.09
Average Total Time (x32) in [s]	565.05	566.59	3760.27
Difference in %	2.7	-2.1	-1.6

Table 5.2: Results for time measurements (taken from [151])

Evaluation of x32-ABI

Reduction of memory requirements comes for free with x32-ABI and in general x32-applications run faster. But currently, there are still unsolved issues which might not allow each task to profit from this interface [107]. This is for example due to loop unrolling. The x32-ABI currently fully unroll loops even though it is not necessary. On top of that, data alignment changes, so if an application has been optimised for 64-bit it might

show less performance in the case of x32. Some instructions require a zero extension of pointers in the case of x32 in order to be 64-bit compatible. The Gnu C compiler does this via an additional system call which is not very efficient. The x32-ABI is still under development and future changes might solve these issues. The main disadvantage of x32-ABI is that the addressable memory is limited to 4 GB. But it is not really an issue for jobs executed in the Computing Grid, since the common rule is anyway not to address more than that.

The main advantage of x32-ABI is that applications can profit from a performance boost and memory reduction at the same time. This comes for free assuming that grid sites provide the right kernel version and that software has been recompiled as x32 binary. Problematic is the fact, that x32-ABI is currently not supported by Redhat and will consequently not be part of a future Scientific Linux version [18].

5.3 Memory Compression

Another approach for reducing the memory footprint of applications is to compress pages. ZRAM, initially called Compcache, is a tool that allows to do this [8]. It works as a virtual swapping device and it uses the swapping mechanism. Instead of writing pages to disc they are compressed and stored inside the RAM. However, this approach is mainly a trade off between memory reduction and additional CPU time. Accessing compressed pages requires a decompression and results in large delays.

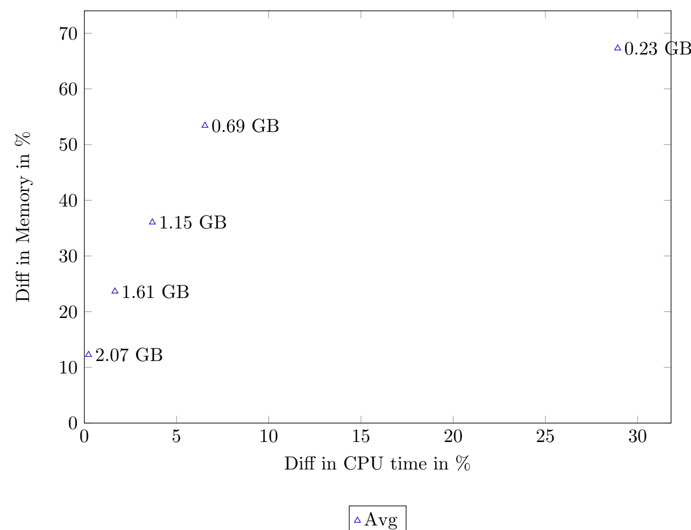


Figure 5.8: Difference in Memory and CPU time (in percentage)

Requirements

In order to use ZRAM, a virtual block device must be first generated, which is part of the normal RAM. Swapping must be enabled for it, while it must be disabled for all other devices. Cgroup can be used in order to limit task resources, like memory and CPU time [33]. As long as a task does not reach the memory limit the allocated memory is counted as proportional and resident set size. After the limit is reached pages are then accounted as Swap and are automatically transferred to the corresponding ZRAM device. ZRAM compresses the content of these pages and stores them inside the RAM.

Benchmark Results

ZRAM has been tested for serial stripping jobs, since it is the memory bounded application of the LHCb experiment. The memory limit was set to 90%, 70%, 50%, 30%, 10% of the maximal memory footprint. These values correspond to 2.07 GB, 1.61 GB, 1.15 GB, 0.69 GB and 0.23 GB. The actual reduction in memory can be defined as the difference between the size of swapped pages and compressed memory. Fig. 5.8 shows the overall average memory footprint reached during the runs of the different cases. The curve follows a logarithmic shape which means the more pages are swapped out the larger the additional CPU time becomes. It is then more likely that pages that have to be often accessed are compressed. But it is remarkable that a significant amount of memory can be reduced in return of only a small impact on the CPU time. So for example, about 50% can be saved with an additional CPU time of only 5%. This leads to the conclusion that many pages are actually not much used during the main loop of the application.

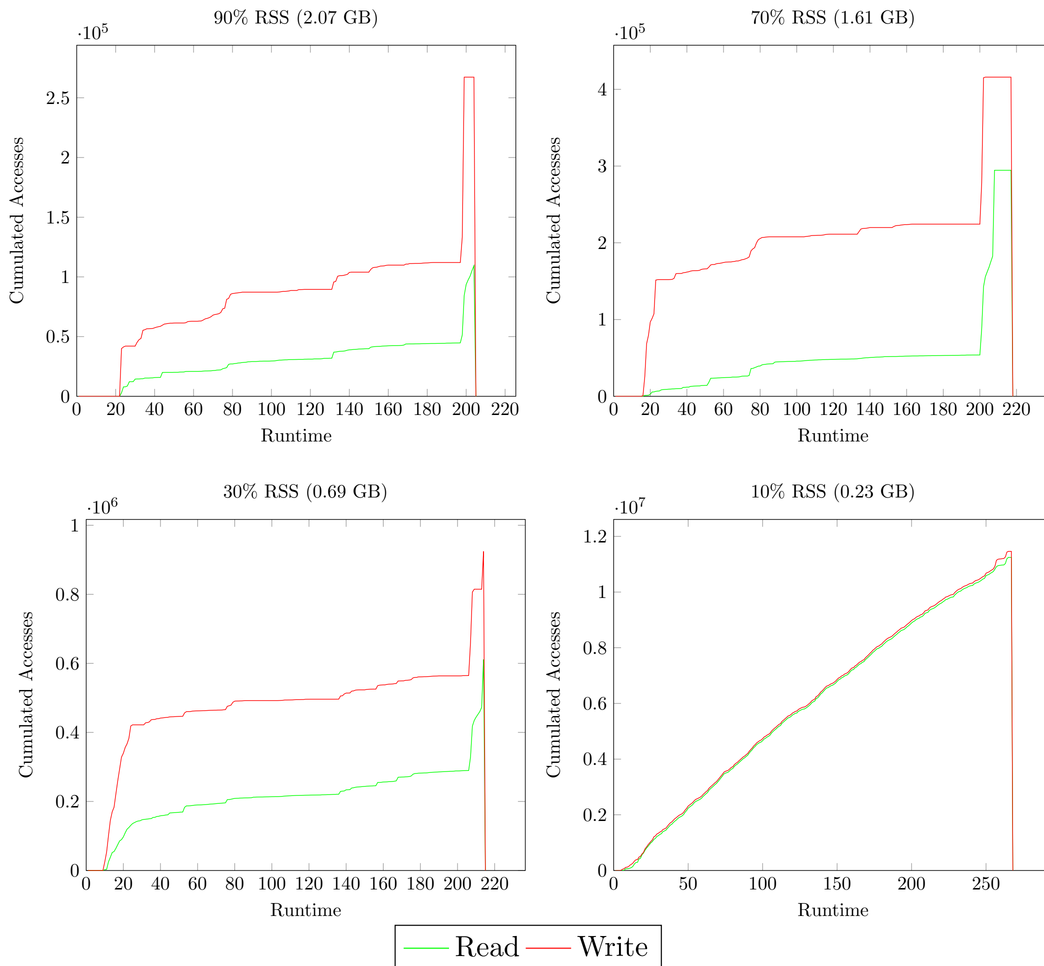


Figure 5.9: Cumulated read and write accesses

Fig. 5.9 shows on the y-axis the cumulated read and write accesses of the test runs. The x-axis indicates the time. It is obvious that read and write accesses increase linearly, when the threshold is too low (10% of RSS). This indicates that certain pages have been swapped out, which have to be accessed frequently in read and write mode. If for example, the detector description is compressed performance will significantly degrade. This dataset is used by all events and consequently it is quite often accessed. When stripping jobs write all output streams an evident peak in memory occurs in the end. This happens because validation takes place before files are finally written to disc. Consequently, memory footprint grows by mutliple times. This can also be seen in Fig. 5.9. The lower the threshold, the larger is

the processing time as it is indicated by the x-axis of Fig. 5.9 and the larger the cumulated read and write accesses. It can be seen that more pages are written than actually read when the threshold is lower. This is compatible with the assumption that many compressed pages are actually not often accessed. The conclusion is that applying a hard memory limit and forcing a process to simply swap pages to the disc would not have a large impact on its performance. The main advantage of memory compression is that it allows a grid site to arbitrarily reduce the memory requirements of jobs.

5.4 Summary

The chapter has shown that memory deduplication significantly reduces the overall memory footprint. In the case of 8 worker processes up to 50% can be achieved. It has been presented, that one of the major problems for LHCb software has been the transition from 32- to 64-bit. Memory requirements have risen by a factor of 1.6. However, the chapter has proven that the new platform model x32-ABI helps to compensate this problem. A reduction of up to 25% has been evaluated. Memory compression allows to apply an arbitrary limit at the cost of additional CPU time. The chapter has evaluated the ratio and unlike expected a large amount can be reduced with only a low overhead in CPU time. For instance, up to 30% reduction has been achieved and at the same time the job required only 5% more in runtime. This concludes, that a lot of allocated memory is actually not used at all.

These results also help to understand applications' behaviour. They show that there is still plenty of room for improvement in the single threaded LHCb applications like for example memory handling and data structure layout. As described in [112]:

> *"... it is often possible to re-engineer codes to achieves significant speedup (2x to 5x unoptimized speed) using simple program transformations. Parallelizing sub-optimal serial codes often has undesirable effects of unreliable speedups and misleading runtimes."*

That's why, many more tools and compiler options are evaluated by different research teams at CERN. For example, it could be also observed that different malloc implementations can improve run time of applications [138]. So the main question is not only how to use multicore CPUs more efficiently, but also how to let single task applications perform better by using other performance dimensions like vector registers as explained in section 3.

6. Optimisation with Intrusive Techniques

Using external tools or recompiling the software can notably change its execution time and memory footprint, as it has been presented in the previous chapter. However, the largest impact can be achieved by modifying the software itself. Section 6.1 discusses the main paradigms like data and task parallelism in HEP software. Advantages of using threads or processes are discussed as well. Prototypes and their perspectives are presented in section 6.2, 6.3 and 6.4. Detailed benchmarking results for a multiprocessing prototype are shown in section 6.5 since this is currently a full working parallel prototype of the LHCb software framework.

6.1 Parallelization Principles

The principle of parallel applications is to divide a large problem into smaller chunks and to process them concurrently. This separation can be applied at different levels.

Data Parallelism

If an application is processing a large amount of data, the data can be split into sub datasets [126]. Each task is operating on a different chunk of data. Data parallelism is commonly used in the field of Computational Fluid Dynamics, where fluid problems are numerically solved with methods like Finite Element (FEM), Finite Volume (FVM) or Finite Difference (FDM) [195]. The input for these methods are geometrical representations of objects which can be very large. In order to speed up the solving process, these objects are partitioned. Such applications run highly parallel on supercomputers. Data parallelism can be also applied in HEP software. For instance, a LHCb reconstruction job processes on average 50k events. Since events are independent, they can be easily distributed on different worker threads. Each task executes the same set of algorithms but on different data (Fig. 6.1).

Task Parallelism

If an application executes a large set of independent algorithms, task parallelism can be applied [126]. Each worker thread operates on the same data, but executes different algorithms. It is often the case that there are dependencies between the input and output of algorithms. A direct acyclic graph (DAQ) helps to determine such dependencies and to calculate the concurrency level. HEP software consists of large set of algorithms. For

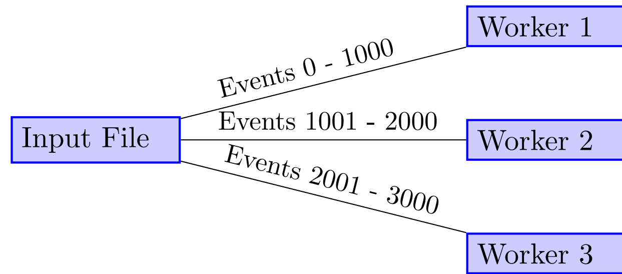


Figure 6.1: Data parallelism in HEP software

instance, the LHCb reconstruction software contains about 200 algorithms [23]. However, measurements in LHCb and CMS have shown that the number of data independent algorithms is very small leading to a low concurrency level. Fig. 6.2 shows results from the CMS experiment. They have evaluated, that the concurrency level ranges between 1 and 16 during the processing of one event. According to [123], the concurrency level is high in the beginning because many calibrations algorithms can run in parallel. During the tracking the level decreases due to the dependencies between the tracking algorithms. As a result pure task parallelism in HEP software cannot achieve large speedups.

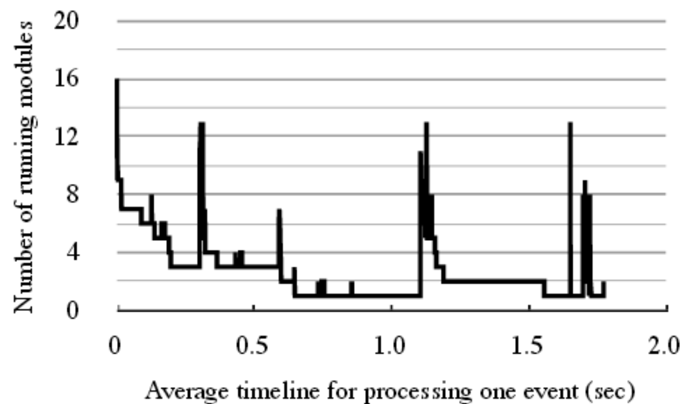


Figure 6.2: Algorithms Parallelism in CMS Software (taken from [123])

Instruction Level Parallelism

Modern micro processors provide several pipelines in order to fetch, decode and execute instructions. This introduces another level of parallelism, the so called instruction level parallelism (ILP) [150]. If a software provides an appropriate layout it can profit from this parallelism. When instructions are independent they can be overlapped and executed in parallel. Compiler optimises code in order to increase ILP. For instance, this can be achieved by loop unrolling. However, HEP software suffers in general from a low ILP as it is evaluated in section 6.5.

Processes versus Threads

Parallelizing the software can be achieved by using multiple processes, threads or a mixture of both. The advantage of processes is that the address spaces are separated [149]. Shared memory pages are automatically handled by the Copy-On-Write mechanism of the Linux kernel [66]. However, communication must take place via sockets since a process cannot directly access the memory space of another task. Even though there are possibilities to use memory mapped files as shared memory between processes, few problem arise which will

be discussed in section 6.2. Threads share the same address space and system resources. Therefore, threads can communicate with each other via shared memory. However, locks must be applied, when threads are accessing or modifying objects at the same time [149]. Consequently, all libraries included by an application must ensure thread safety.

Using threads or processes depends a lot on the user and software requirements. In general, threads might scale better since they are more light weight and can use shared memory for communication. When a context switch is performed, the virtual address space of a process does not remain the same. However, for a thread it does. Context switches take place, when a CPU is idle due to a pending task and another task is then placed [165]. Context switches require more CPU cycles in the case of processes than in the case of threads. Nevertheless, the speedup of a parallel program depends a lot on the implementation itself and how many locks have to be applied.

6.2 Multiprocessing Approach

A parallel prototype for the software framework Gaudi has been developed (Fig. 6.3), which is based on the multiprocessing approach (GaudiMP) [137] [152]. Since events are independent they can be easily computed in parallel. As a result, data parallelism can be applied where datasets are split and distributed on different processes. The main loop of the application is split into several parts which are computed by different workers. The processing of events represents the parallel part of the application. As a consequence, the more events the larger the main loop. This implies that speedup improves because the impact of serial parts like initialization and finalization becomes smaller. An evaluation is given in section 6.5. The performance limiting part starts when events have been computed and must be sorted and merged into a single output file. This is done by a single serial task, the writer process. Events must be read from the disc which can be done by a separate process in order to coordinate the access to data. Since reading goes rather quickly it does not represent a performance bottleneck.

The main advantage of the multiprocessing approach is that it can be applied on the software framework in a rather transparent way. Processes must be only configured differently, such that modifications at the core level of the software can be avoided. As shown in Fig. 6.3 one reader, writer and several worker processes exist. All these tasks obtain in the beginning the same Gaudi configuration, which is defined via the job options. Afterwards, they are reconfigured in order to keep just the relevant setup. Consequently, worker processes have only knowledge about the algorithms but not about the output and input streams.

Python's Global Interpreter Lock

Since different configurations are applied via Python scripts within the software framework, GaudiMP has been mainly implemented in this scripting language. However, this results in another problem. Due to Python's Global Interpreter Lock (GIL) [80] it is rather difficult to run parallel Python threads. GIL is implemented as a mutex which means that threads cannot access objects at the same time. This is necessary since Python's memory management is not thread safe. As explained in the official Python documentation [101], it could happen that for example reference counters of objects are not correct when multiple threads access them. In worst case an object will be deleted which is still required by another thread. Consequently, the Global Interpreter Lock ensures that objects are locked.

Serialization and Deserialization

Using processes instead of threads results in the problem that communication cannot be realised via shared memory. Since processes cannot share virtual tables, objects must

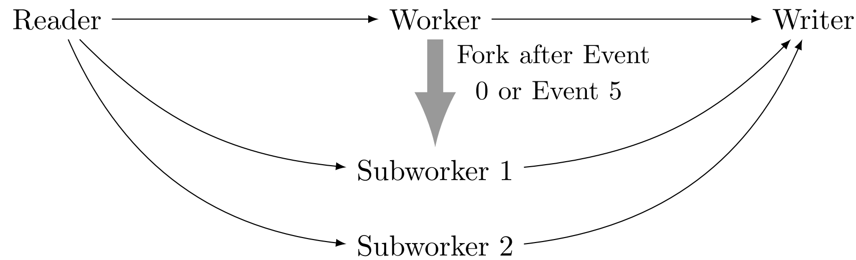


Figure 6.3: Overview of GaudiMP (taken from [152])

be serialized and deserialized which is a time intensive task. Events are read as ROOT TBuffer objects [35] and consist of virtual functions. If an object is created by process 1 in shared memory the virtual table points to the virtual address space of process 1, which is not shared. Consequently, process 2 cannot access the information to which the virtual table points. Serialization and deserialization is a function of the number of events and it significantly impacts the speedup. One reconstruction job for example processes around 50k events, which must be serialized and deserialized between reader and worker and between worker and writer. This represents the main limitation in GaudiMP.

Late Forking

Executing applications in parallel impacts their overall memory footprint. Due to the Copy-On-Write principle of Linux memory pages are automatically shared as soon as child processes are forked [66]. When a process intends to write to a shared page this area of memory is copied to its own address space. This results then in an increase of the overall memory footprint. During the initialization of the applications a lot of datasets are loaded which are accessed in read only mode during the main loop. This applies to:

- Detector description
- Magnetic fieldmap
- Conditions of the LHC run
- Configuration of the Gaudi framework

The overall memory footprint can be improved by forking child processes as late as possible. This could be either done during the initialization or in the beginning of the application main loop. The best result can be achieved when a few events have been already processed. It can happen that condition datasets have to be reloaded during the processing of the first event. However, if the main loop has already begun when the fork is applied (3rd red arrow of Fig. 6.4), the application needs to be restarted. Otherwise, histograms would be shared and contain wrong counters. It has been decided to fork first the reader, writer and one worker processes in the beginning. The remaining subworkers are forked by the main worker before event 0 has been loaded but after the initialization period. This is indicated by the 2nd red arrow of Fig. 6.4. Consequently, they can share all data that has been loaded during the initialization. This counts approximately 300 to 500 MB. With increasing degree of parallelism the overall memory footprint grows sublinearly. An earlier prototype of GaudiMP contained a fork before the initialization (1st red arrow of Fig. 6.4) and consequently did not reach a significant memory reduction.

Even though a transparent usage is ensured in GaudiMP, few modifications were necessary at the core level of the software in order to allow the full functionality of the different applications. First, threads have to be joined and restarted after the fork of the subworkers. During the initialization period of the applications, threads are created which are

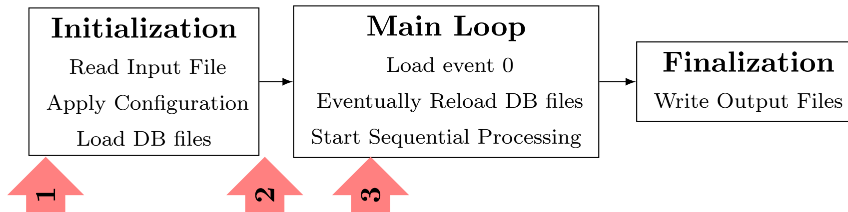


Figure 6.4: The main steps of Gaudi applications and possible options of forking subprocesses

responsible to read files from databases. However, if new processes are forked, threads just disappear and locks are not released.

Stripping

Stripping jobs require few modifications at the core level of the software, since they have to split the input files into important and non relevant decay products. The algorithms ensure the splitting procedure and forward the necessary information to the output streams. The problem is that writer and worker processes are configured differently within GaudiMP. Since the writer does not obtain the configuration of algorithms, it does not know which events can be discarded. Consequently, the services had to be extended in order to store and send this relevant information.

Simulation

The concept of reader changes in the simulation software since input files are not required. Instead, the reader process serves as generator for random seeds, which guarantees reproducibility and independent sets of random numbers. If random numbers were generated within the workers, each of them would create the same numbers and therefore events would be duplicated.

6.3 Multithreaded Approach

Few years ago, a new development was started which aims to parallelize the Gaudi framework via threads (GaudiHive) [98]. The aim is to have a highly scalable software framework. Instead of using only event parallelism, where datasets are split and distributed on different workers, algorithms are running in parallel. This is also known as task parallelism, where each thread executes different algorithms on the same data. If the dependencies between different algorithms are known beforehand, it is possible to execute independent ones at the same time. Fig. 6.5 shows the architecture of GaudiHive, which aims to extend the Gaudi framework in such a way that transparent usage is still guaranteed. The algorithm scheduler is responsible for fetching instances from the pool of algorithms, taking events and creating tasks. Furthermore, the AlgorithmPool facilitates the handling of non thread safe resources. If such a case is identified within an algorithm, only one instance is created as explained in [98]. This allows a lock free implementation, but guaranteeing thread safety is still one of the major issues.

Creation of tasks is done via Intel Threading Building Blocks (TBB), which provides an abstraction layer for modelling and scheduling tasks. Dependencies are defined via Direct Acyclic Graphs (DAG), which allows to determine when new algorithms can be launched [117]. As mentioned in section 2.3 LHCb's reconstruction application consist of about 200 different algorithms. However, analysis has shown that only up to 4 algorithms might be able to run at the same time [117]. This requires that events and algorithms

are duplicated in order to improve the scalability of the software. Replicating events is managed by the Whiteboard service (Fig. 6.5) which contains according to [117] several event stores. The execution context determines which event has to be read or written and tracks the status of each duplicated event. Currently, GaudiHive supports a minimal version of the reconstruction application consisting of only 20 algorithms. Many more developments and changes are required in future [98].

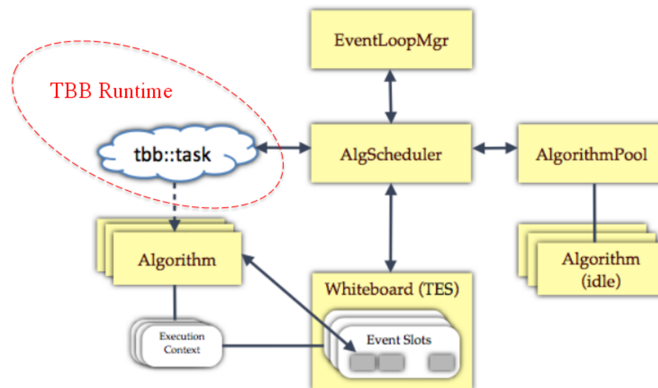


Figure 6.5: Overview of GaudiHive (taken from [98])

6.4 Perspectives of the Different Concepts

The impact of these approaches on the software framework differs quite a lot, as stated in the previous subsections. Even though, the aim is to keep a transparent usage and all the key features of Gaudi, essential modifications will be necessary when the framework shall scale on future manycore CPUs. [136] proposes to add concurrency at the level of events, algorithms and subalgorithms and therefore to combine multiprocessing with multithreaded approaches. This means to use data parallelism in order to split events on different workers and within the workers multiple threads take care of a parallel processing of different tasks. The third step, called subalgorithms parallelism, includes parallelism at the level of I/O.

6.5 Evaluation of a Parallel Software Framework Prototype

This section will focus on an evaluation of the current parallel prototype in respect to speedup, performance bottlenecks and comparison with single task jobs. Since the multithreaded approach does not work for real reconstruction and stripping jobs, the tests will focus on GaudiMP. The section will mainly refer to results presented in [152], [154], [155], [156] and [157].

6.5.1 Comparison with Serial Task Jobs

Currently, jobs are executed as serial tasks within the Worldwide LHC Computing Grid. Parallel processing helps to overcome memory limitations but also impacts the efficiency. It is difficult to achieve linear speedup due to serial parts, like initialization, finalization, synchronization and communication. However, having a lot of uncoordinated serial tasks rises concurrency and cause loss in performance. Consequently, the aim is to first evaluate how much better GaudiMP scales compared to single task jobs with respect to memory limitations. In the test each job has processed the same amount of events. The parallel job has processed all events with 8 worker processes, while in the other case 8 separated

Gaudi instances have been initialized, whereat each of them has processed $\frac{1}{8}$th of the input file. At the same time, the memory limit has been decreased step by step, in order to evaluate when the parallel prototype provides an overall better scaling. Throughput has been chosen as metric, which indicates how many events are processed within a given time period.

Fig. 6.6 shows the results obtained from reconstruction and stripping jobs. In the first case there is no significant difference in the throughput, as long as the limit is larger than 1 GB. Consequently, overhead due to serialization and deserialization does not have a large impact on the overall performance. Beyond 1 GB per process GaudiMP scales by far better since the overall memory footprint can be reduced via sharing. In the case of stripping jobs the difference is more significant and beyond 1.5 GB GaudiMP can provide better event throughput. If the memory limit is too low, the parallel prototype also looses performance.

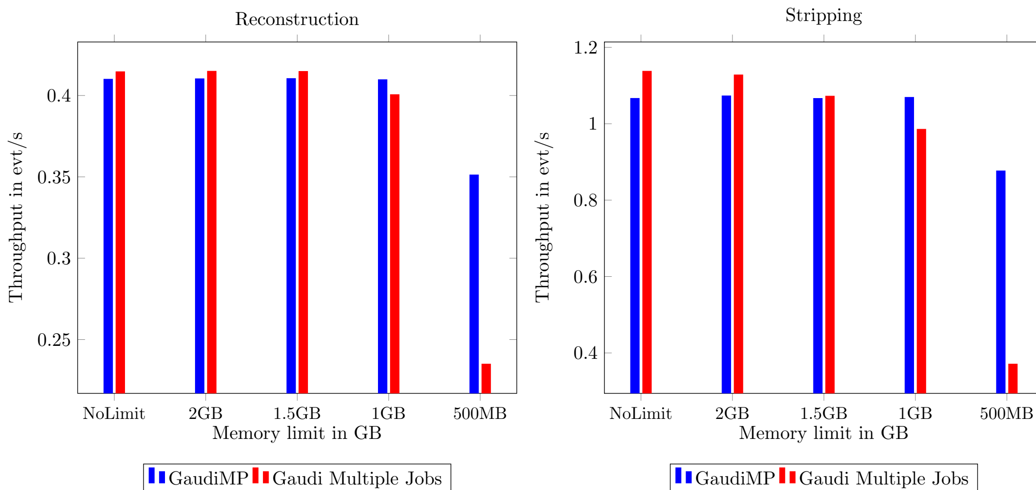


Figure 6.6: Throughput depending on different memory thresholds (taken from [152])

6.5.2 Validation of Physics Results

A non negligible criteria is the correctness of output files. The generated results shall not differ from the ones produced by a serial task execution. Tools have been implemented which compare ROOT histograms and the so called DST-output files [178]. A reconstruction job generates a ROOT file, which contains general information recorded by each sub detector element, like number of registered particles and clusters. These information are stored as histograms. The reconstructed events are written to a DST file, which is a directory like data structure emulating the Transient Event Store of the Gaudi Framework. It stores all information about the raw event and the decay products. Stripping jobs produce several DST output streams.

It must be ensured that the generated ROOT and DST files are correctly produced within a parallel execution. For the first case, histogram paths are evaluated and their entries are compared. In the next step, a Kolmogorov-Smirnov test [176] is applied in order to check whether the content of histograms match. It is a statistical test which compares the empirical cumulative distribution functions (CDF) of two datasets. A CDF defines the probability of a random variable X being smaller or equal than a certain value x. For discrete random variables it is defined as [192]:

$$F(x) = P(X \leq x) = \sum_{x_i < x} p(x_i).$$

An empirical CDF is an approximation to an unknown CDF. It calculates the number of x_i entries smaller than a certain value x and divides it by the number of observations. The Kolmogorov-Smirnov test determines the maximum distance between the two empirical CDFs $\widehat{F}_A$ and $\widehat{F}_B$, which is defined as [173]:

$$M = \max_x |\widehat{F}_A(x) - \widehat{F}_B(x)|.$$

The user has to define the threshold for the maximum acceptable difference. A perfect match is required for the histograms produced by a serial and a parallel reconstruction and stripping job. Comparing DST files requires to rerun all the events for determining the order. In a next step, events from a serial and parallel job are sorted and information are transformed to ASCII format. Afterwards, event by event can be compared based on the decrypted information. On top of that, file summary records have to be compared which are basic counters stored additionally in the output files. These are information provided by the HLT trigger, which indicate the occurrence of certain event types [6].

```
------------------------------------------------------------
Summary of histos which failed Kolmogorov Test
------------------------------------------------------------
<class '__main__.TH1D'>        /stat/Brunel/MemoryTool/Total Memory [MB] :     K-Test Result :  0.0546463301138466
<class '__main__.TH1D'>        /stat/Brunel/MemoryTool/Virtual mem, all entries :      K-Test Result :  0.0134348133777156
<class '__main__.TH1D'>        /stat/CaloMoniDst/ElectronMon/14 :      K-Test Result :  0.9999999997843159
<class '__main__.TH1D'>        /stat/CaloMoniDst/PhotonMatchMon/1 :    K-Test Result :  0.9999999997843159
<class '__main__.TH1D'>        /stat/CaloMoniDst/PhotonMon/14 :        K-Test Result :  0.9999999997843159
<class '__main__.TH1D'>        /stat/OT/OTTimeMonitor/30 :     K-Test Result :  0.9999960893050661
<class '__main__.TH1D'>        /stat/PROTO/ChargedProtoANNPIDMoni/Long/ElectronANN :   K-Test Result :  0.0000000000000000
<class '__main__.TH1D'>        /stat/PROTO/ChargedProtoANNPIDMoni/Long/MuonANN :       K-Test Result :  0.9999999968458320
<class '__main__.TH1D'>        /stat/Timing/CaloEventProcTime/overallTime :    K-Test Result :  0.4004710362237014
<class '__main__.TH1D'>        /stat/Timing/ProtoEventProcTime/overallTime :   K-Test Result :  0.9999999997843159
<class '__main__.TH1D'>        /stat/Timing/RichEventProcTime/overallTime :    K-Test Result :  0.9999999997843159
<class '__main__.TH1D'>        /stat/Timing/TrackEventProcTime/overallTime :   K-Test Result :  0.1640791977266520
<class '__main__.TH1D'>        /stat/Timing/VertexEventProcTime/overallTime :  K-Test Result :  0.4004710362237017
------------------------------------------------------------
```

Figure 6.7: Results obtained from the Kolmogorov-Smirnov test

Differences should not occur, as long as histograms are produced in an additive manner. Nevertheless, certain ones fail the Kolmogorov test as shown in Fig. 6.7. Histograms containing timing results cannot be the same, since the runtime of parallel and serial jobs differ. The same applies for memory values. But it appears that certain histograms fail the test, like CaloMoniDst, Proto and OT. In these cases the problem occurs, that the histograms are not summarized in an additive manner and therefore a correct result cannot be produced. However, as long as the number of workers is known, the effect is reproducible. The comparison of DST files did not show any discrepancies.

6.5.3 Speedup

The main metric for parallel processing is speedup, since it indicates how an application scales with the number of cores [139]. It can be either sublinear, linear or superlinear. In latter case an application would run more efficiently in parallel than in serial mode. This can happen due to caching effects, which allow another thread to access data sets faster since they have been preloaded by another task. Nevertheless, this case is very rare. Linear speedup indicates, that an application scales exactly with the number of processes. So, if a job runs with 4 processing units its runtime would be $\frac{1}{4}$th of its serial time. In many cases, applications show nearly linear speedup for small number of processes and then speedup evolves sublinearly. In general, speedup can be calculated as [139]:

$$S(n) = \frac{time(1)}{time(n)}, \tag{6.1}$$

where the serial runtime is divided by the time needed in parallel mode. Consequently, efficiency can be defined as [139]:

$$E(n) = \frac{S(n)}{n}, \tag{6.2}$$

Processes	Without	With	Difference in %
2	1.9833	1.9680	0.77
3	2.9007	2.8560	1.89
4	3.7806	3.7342	1.23
5	4.7419	4.4575	5.99
6	5.7120	5.2071	8.84
7	6.6551	6.0035	9.79
8	7.5271	6.7262	10.64

Table 6.1: Speedup values reached on different hardware configurations (with and without Intel Turbo Boost)

where n indicates the number of used processing units and $S(n)$ is the speedup. So if an application runs on 10 cores but provides only a speedup of 5, then 50% of the efficiency is lost. In this context it must be also respected which kind of time is measured on a CPU. The three basic types are:

1. System time
2. User time
3. Wall clock time

The first value indicates how much time the CPU spent in kernel space for executing system calls for example. User time shows how much time the CPU spent in user space for executing the actual process. CPU time is the sum of both values. Wall clock time is CPU time plus the time the processing unit was idle due to pending tasks or has been blocked by other tasks. The Linux time module sums up the total CPU time required by parent and forked subprocesses. Different time values can be chosen in order to measure speedup of an application. If it is I/O-bound it is worth measuring it's wall clock time, since it takes I/O activities into account. On the other side, wall clock time can easily be affected by other processes.

Furthermore, it is important to configure a CPU appropriately in order to obtain proper benchmark results. Frequency scaling is a technique applied by many CPU manufactures to let applications profit from a performance boost in case the overall workload on the CPU is small. The Intel Turbo Boost technology defines a TDP-value (thermal design power), which presents a global limit for all cores [38]. As long as this limit is not reached, a core is allowed to increase its frequency, for example up to 3.4 GHz on an Intel Core i7. Consequently, if a machine is idle a process can run with the maximum frequency. If there is an high workload all processes have to lower their frequencies due to thermal effects. Without switching off this technology measurements are influenced by the speedup of the machine. In order to switch off Intel Turbo Boost on an Intel Core i7, the model specific register 0x1a0 must be set to 0x4000850089. Tab. 6.1 shows the effect of frequency scaling on the speedup values, which have been measured with a parallel reconstruction job. An Intel Xeon (L5520) has been used, whereat Intel Turbo Boost can increase maximum performance by 133 MHz for 4 and 3 cores and by 266 MHz for 1 and 2 cores. In the test case in which Turbo Boost has been switched off, the frequency has been set to 2 GHz. The higher the workload on the machine the lower the frequency and the worse the scaling. In the case of 8 processes a significant difference of 10 percent can be observed, as Tab. 6.1 shows. Many different formulas exist, in order to obtain speedup curves and the most common ones will be shown in the following subsections.

Amdahl's Law

The most famous law for parallel processing is Amdahl's Law [55]. It basically says, that the execution time of parallel applications is limited by their serial parts. It also expresses that software can never be completely parallel since parts like initialization are always required. Speedup is defined as:

$$S(n) = \frac{1}{s + \frac{1}{n}(1-s)}, \tag{6.3}$$

where s is is the serial part of an application and n the number of used processing units. Nevertheless, Amdahl's Law is not applicable in certain contexts. For example, it does not respect that a software might scale superlinearly. The best achievable speedup is linear according to the law. It also ignores that the parallel part of an application might grow, when more processing units are available. Consequently, the main objective of this law is that an application cannot scale infinitely because the serial part will become dominating.

Gustafson's Law

Gustafson's Law is an extension of Amdahl's Law and it expresses that a sufficiently large problem can be parallelized in an efficient way [114]. It also implies that the same application could run within the same amount of time but with a larger workload. It defines speedup as:

$$S(n) = n + (1-n) \cdot s, \tag{6.4}$$

where s denotes the serial part of an application. The more processing units are used the larger the parallel part can become. It expresses the same like Amdahl's Law but it focuses on the parallel portion of an application.

Downey Speedup Model

Amdahl's and Gustafson's Law separate a program into its serial and parallel part. However, in many cases such a clear distinction is not easy to apply and as a result using these laws is not appropriate [128]. The Downey speedup model tries to avoid these issues. It defines an average parallelism of software based on measurement results. Consequently, it helps to determine how efficiently an application can use processing units and when it is not worth assigning more processes. The Downey speedup model distinguishes between a high and a low variance model [88]. The variance indicates, how well a software scales with the number of cores. A value of 0 presents linear speedup and infinite would be the worst case. The low variance model, implies that the variance of speedup is smaller than 1 and the model is then defined as:

$$S(n) = \begin{cases} \frac{An}{A+\sigma(n-1)/2} & 1 \leq n \leq A \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} & A \leq n \leq 2A-1 \\ A & n \geq 2A-1, \end{cases} \tag{6.5}$$

where A is the average parallelism and σ the variance in parallelism. The high variance model implies that σ is larger than 1 and it is determined as:

$$S(n) = \begin{cases} \frac{An(\sigma+1)}{\sigma(n+A-1)+A} & 1 \leq n \leq A + A\sigma - \sigma \\ A & n \geq A + A\sigma - \sigma. \end{cases} \tag{6.6}$$

Parameter A can be understood as an upper limit, beyond which an application does not allow further scaling. This defines the point, when new technologies must be introduced into the software framework in order to allow further improvements. Speedup values must be obtained, in order to solve those equations. Finding proper values for the parameters A and σ which fit best the set of given points, presents a two dimensional minimization problem. Fig. 6.8 shows the predicted speedup curves for the different types of LHCb jobs. The average parallelism is 43.0 for reconstruction, 21.93 for simulation, and 29.52 for stripping. It means that simulation jobs can scale up to 21 cores. This shows, that new parallelization concepts must be realized when the software framework shall scale beyond 20 processing units.

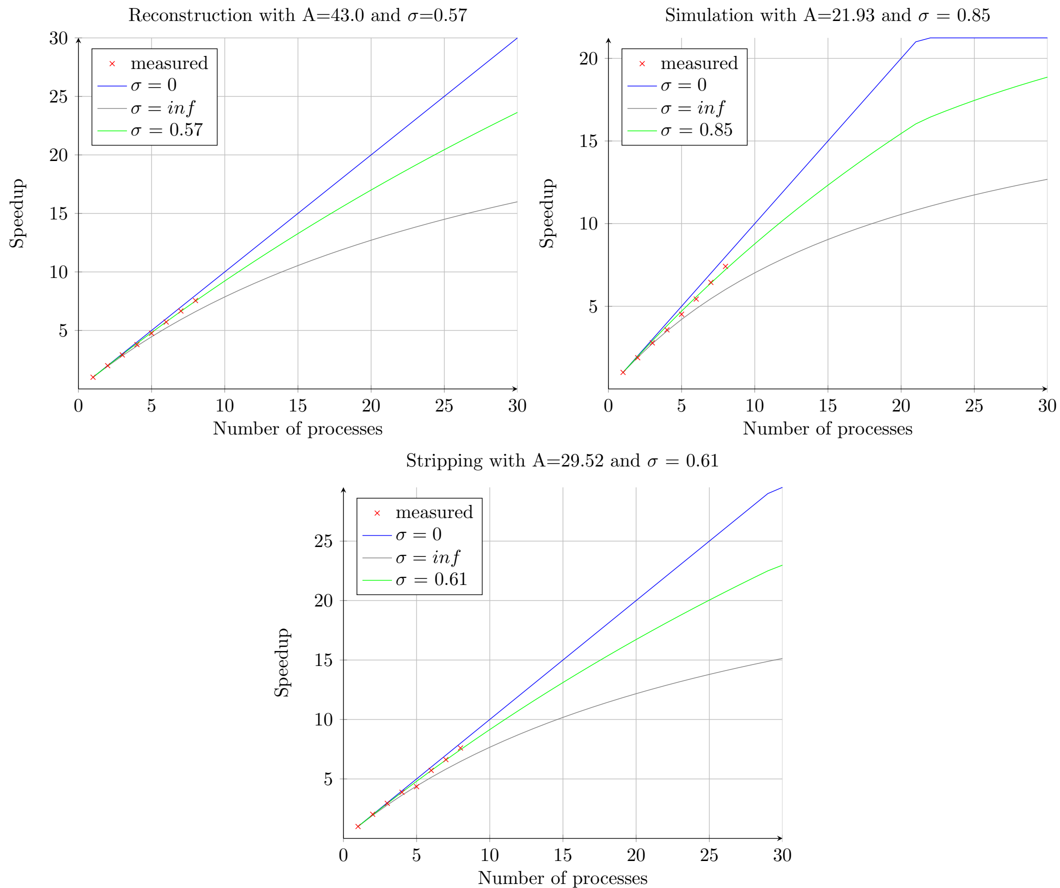


Figure 6.8: Average parallelism and variance in parallelism (taken from [152])

6.5.4 Memory Reduction

Memory is another important metric for benchmarking the parallel prototype, since this is the main reason for a parallel execution of jobs. Memory is handled by the operating system and just a certain size of memory can be allocated. This is indicated by the page size, which is normally 4 kB large [189]. Other sizes exist, the so called huge pages, which measure generally a few Mega Bytes. This can increase performance since an application can allocate large amount of memory with fewer overhead [189]. Normally, a process can address much more memory than is actually provided by the system. Due to the change from 32- to 64-bit a task would be able to address 2^{64} bytes. Different memory values can be monitored, which will be explained in the following subsections.

Virtual Address Space

At compile time virtual addresses are sequentially assigned to a program [4]. The Translation Lookaside Buffer (TLB) is responsible for translating those addresses from virtual to physical memory addresses, while a program is running. This concept has several advantages. A program can allocate continuous blocks of memory and does not have to respect limitations of RAM size. Furthermore, each process has its own virtual space and therefore they are encapsulated. When an application is running not all virtual addressed memory is actually allocated within the RAM. This leads to the so called page faults: A task tries to access a page which has not been mapped into physical memory yet. The Linux call mmap allows for example to map a file into memory. Therefore, virtual memory is requested but not physically allocated. Such calls are also used to extend automatically the heap of an application.

Resident Set Size (RSS)

The resident set size indicates the amount of memory which a process has allocated inside the RAM [4]. However, it does not take into account that certain libraries are shared between several tasks. Consequently, the total RSS value might become larger than the actual size of RAM. Especially, since the multicore era this parameter is not reliable any longer. When a parallel application can share a significant amount of memory between threads and tasks, it will not be respected by the RSS value.

Proportional Set Size (PSS)

Another value is the proportional set size, which solves the issue with shared pages [4]. If a page is shared it is divided by the number of instances which share it. Consequently, it allows to measure memory reduction of parallel applications. Furthermore, the total PSS value cannot be larger than the actual size of the RAM. The following example shall illustrate the 3 different parameters.

> **An Example:** *A 4 kB page is shared by 4 tasks. The values for task 1 would be:*
> *VSS: 4 kB*
> *RSS: 4 kB*
> *PSS: 1 kB*

These values can be obtained from the smaps file which is located in the process directory under Linux [4]. It shows all memory mappings and indicates the size of heap and stack. In case, more memory must be allocated, the operating system can swap pages to the disc. This results in a performance loss, since accessing the disc requires more CPU cycles. Certain grid sites in the Worldwide LHC Computing Grid even disable swapping, which means that jobs are automatically interrupted as soon as they run out of memory. Currently, the memory peak of each job is monitored and stored in database. However, virtual memory is counted which does not allow to draw many conclusions from. The problem is that LHCb applications are designed such that they request normally much more memory than actually needed as shown in section 5.2. This is then accounted in the virtual but not in the physical memory. Consequently, LHCb jobs appear with a much larger memory footprint. However, they can also run on machines which provide only a fraction of memory.

Fig. 6.9 shows the overall memory footprint of reconstruction jobs in which the fork of subprocesses has taken place at different time slots. If jobs are not executed in parallel, no sharing can be achieved and the memory footprint will grow linearly. A slight memory

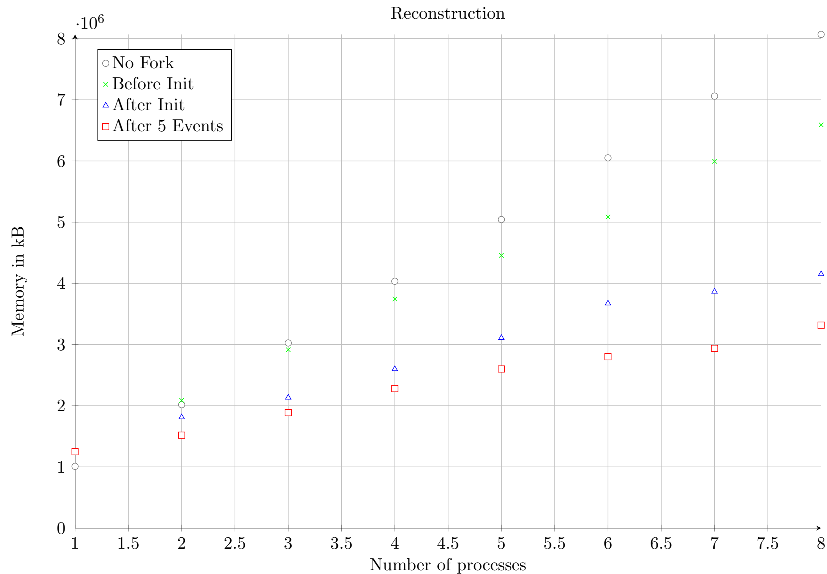


Figure 6.9: Overall memory footprint of reconstruction jobs

overhead is introduced in the case of 1 worker compared to serial execution which is caused by the reader and writer process. If processes are forked before the initialization phase, memory consumption can be slightly improved about 18% in the 8 worker test case. The best sharing can be achieved by forking subworkers after the main loop has started, which results in 58% reduction in the case of 8 workers. As already explained previously, many changes at the core level are required in order to reset counters and histograms. Results achieved by forking processes after the initialization phase but before the first event achieves a memory reduction of about 48% in the 8 worker case. It is a significant improvement compared to the old prototype, which forks processes before the initialization.

6.5.5 Determining Bottlenecks

It is important to determine the performance bottlenecks of GaudiMP in order to reach further improvements. As already mentioned in section 6.2, serialization of objects limits the speedup. Since it depends on the parallel part it cannot be easily optimised: the more events are processed the more objects must be serialized. It could be improved by applying a different writer concept. The ROOT framework provides a new functionality which allows to write files in parallel. This allows according to [69], that each task can write and store its part to local disc and in the end it is uploaded to a server, which merges the content. As explained in [69] merging can be done in parallel to the processing by using this functionality. This means that files could be written by the worker processes and consequently the writer becomes unnecessary. This would skip the serialization of events between workers and writer. Nevertheless, this approach requires many modifications at the core level of the software and has not been applied yet.

The serial parts of the application present another bottleneck. First the main instance of Gaudi has to be initialized, which forks the child processes. In a next step, these must start the core services, apply the configuration and reconfigure themselves before the remaining subworkers can be forked. This fixed serial part can be minimized by increasing the parallel part of Gaudi. Fig. 6.10 shows the speedup of a parallel reconstruction job. The case, in which 10k events are processed shows a better scaling than the one with 1k events. This is due to the minimization of serial parts. Furthermore, an upper limit can be determined,

by summing up only the time taken for processing events. This excludes any serial parts like synchronization or finalization. As shown in Fig. 6.10 it does not represent linear scaling. This is caused by concurrent accesses during the processing step.

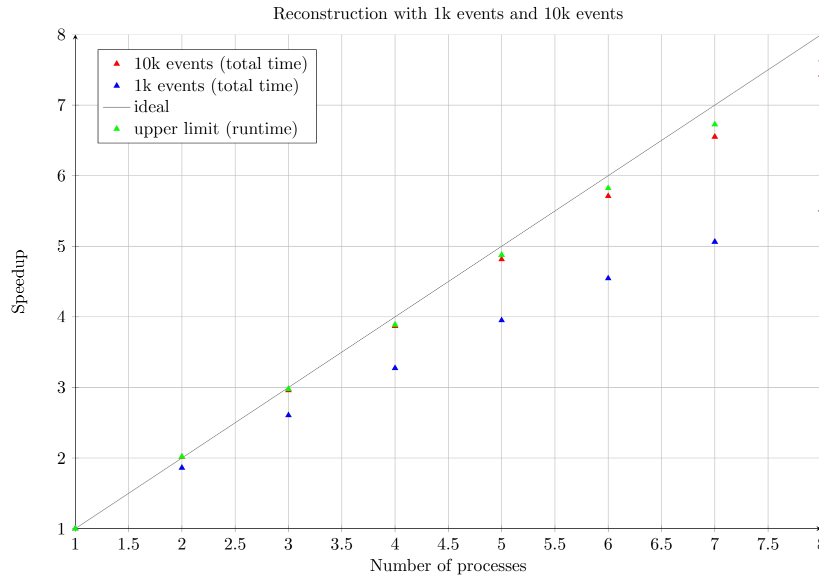


Figure 6.10: Speedup depending on the number of events (taken from [152])

Problematic is the compression of output files that can notably impact the overall performance. Events are not compressed and written, before the main loop has finished. The larger the compression factor the larger the overall processing time becomes (Fig. 6.11). A case has been defined, in which each worker processes 1000 events. The number of workers has been continuously increased such that the total number of events increased up to 48k. This test case illustrates well the principle of Gustafson's Law: Having more processing units allows to process a larger amount of workload and speedup is linear when runtime remains the same. An AMD Magny-Cours having 48 cores has been used as testing environment. Fig. 6.11 shows that the overall runtime can remain similar in the case of uncompressed files but not in the case of compressed files. In latter case, the writer becomes a bottleneck because writing and compressing files is a serial task. The more workers are initialized, the larger the processing time for compression becomes. The workers cannot be joined before the writer has finished.

Performance Monitoring of Software

The complexity of CPUs has risen a lot in the past and the performance of software is often limited by the underlying architecture. In order to better determine and understand bottlenecks, manufacturers introduced the so called Performance Monitoring Unit (PMU) [26]. It provides counters for basic operations like accessing the cache, RAM, fetching and decoding instructions and many more. It allows to monitor the behaviour of a CPU during program execution and to understand the limitations of software. Counters can be measured in every CPU cycle but this causes a large overhead. Instead, the sampling rate must be adjusted such that the PMU is only read after N occurrences of an event. Furthermore, PMU events are CPU specific and the same hexcode might monitor different events on different CPUs. On Intel Xeon Phi, events are indicated by an 8-bit umask and an 8-bit event code. The first one determines the logic unit, while the latter one defines the event itself. On an Intel Xeon Phi the number of executed instructions is for example indicated by the hexcode 0x00 0x16.

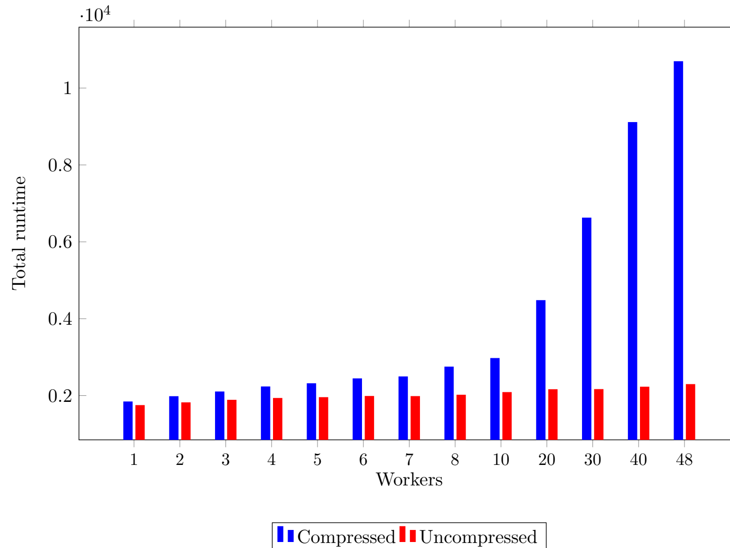


Figure 6.11: Total runtime while increasing the number of workers and events

Metrics

The most common metric for measuring performance is instruction per cycle (IPC) and its counter metric cycle per instruction (CPI). It defines the number of completed instruction per cycle and due to complex pipelining techniques on modern CPUs a value way larger than one is possible [172]. Instructions go through several steps, before they are actually executed. These steps are fetch, decode, execute and store. Instructions are preloaded based on predictors (speculative execution) [172]. For instance, on conditional branches it is unknown which branch will be taken. Consequently, it might happen that wrong instructions are loaded, which is detected in the following steps and the pipeline must then be emptied. Instructions that made it through all steps, are called retired. The efficiency of software can be computed by counting the number of retired instructions and unhalted CPU clocks:

$$IPC = \frac{Retired\ instructions}{CPU\ clocks}.$$

It shows, whether the overall performance of software is good and the distribution of CPI values over the code helps to find code sections which require lot of computing time. As it can be seen in Tab. 6.2 very small IPC values are reached. This is a well known phenomena in HEP software which suffers normally a large number of load and store instructions and 10% of branches as stated in [122]. Branches occur due to conditional jumps, like switch case statements. The CPU predicts a branch and preloads the instructions. If it turns out during the decoding stage, that the wrong branch has been taken, the pipeline has to be unloaded and CPU cycles are lost. If a software consists of too many branches, it can lead to poor runtime. In HEP software branches are partly introduced due to poor data layout, object hierarchy and inheritance [122]. The IPC value decreases with rising number of worker processes which is the effect of the non linear scaling factor.

Another performance metric is memory bandwidth, which can be computed via the number of DRAM accesses, the number of read and written bytes and the number of CPU clocks. If the bandwidth is far below the system's capacity, then an application must be optimised in respect to how data is accessed, prefetched and streamed. However, this metric only plays a major role in applications which are highly memory bound. A more important metric for LHCb software is how often cache misses occur. Common datasets like detector description and magnetic field map are accessed by each single event, but they are too large to be stored in cache. L1 cache is divided into an instruction and a data cache.

Good spatial locality implies that the data cache is quite often accessed and not many misses occur. The data cache miss ratio is defined as [5]:

$$DC\ miss\ ratio = \frac{DC\ misses}{DC\ accesses}.$$

The data cache miss rate is determined by:

$$DC\ miss\ rate = \frac{DC\ misses}{Retired\ instructions}.$$

The L2 miss rate and ratio is not computed in the same way. First the number of accesses must be determined, which is the sum of writebacks from L1, which occurs when data is evicted from L1 to L2 and the number of requests. The L2 miss ratio is defined as [5]:

$$L2\ miss\ ratio = \frac{L2\ misses}{L2\ requests + L2\ writebacks}.$$

And the L2 miss rate is [5]:

$$L2\ miss\ rate = \frac{L2\ misses}{Retired\ instructions}.$$

The L3 miss ratio can be defined as [5]:

$$L3\ miss\ ratio = \frac{L3\ misses}{L3\ requests}.$$

The L3 data cache miss rate can be computed as [5]:

$$L3\ miss\ rate = \frac{L3\ misses}{Retired\ instructions}.$$

As Tab. 6.2 shows, the DC and L2 miss rate and ratio does not deteriorate when the number of worker processes increases. The number of misses and accesses increases in a linear manner, which results in a similar ratio. A DC miss rate of 0.014 implies that a data cache miss occurs after every 71 (0.014^{-1}) instructions, respectively every 161 in the case of L2. The L3 miss rate and ratio significantly increases with the number of worker processes. It is related to the fact that number of cache misses and requests scale with different factors. This can be an effect related to the L3 cache being shared between 6 cores on the testing environment. This can lead to an increased cache pollution when the number of workers becomes larger.

The DC, L3 and L2 ratio implies that out of all cache accesses only a small fraction have been missed. This leads to the conclusion that the data inside the cache is accessed very often. Since not all data fits inside a significant amount of misses still occurs. Many more

	serial mode	8 workers	48 workers
IPC	0.89	0.74	0.72
DC miss ratio	0.02	0.02	0.02
DC miss rate	0.014	0.014	0.013
L2 miss ratio	0.16	0.15	0.14
L2 miss rate	0.007	0.007	0.007
L3 miss ratio	0.066	0.076	0.123
L3 miss rate	0.001	0.001	0.003

Table 6.2: Results obtained from the Performance Monitoring Unit

metrics are available to evaluate performance bottlenecks. However discussing all of them would go beyond the scope of this thesis.

	Prototype 1	Prototype 2	Improvement in %
DC misses	$1711 \cdot 10^9$	$1611 \cdot 10^9$	6.2
L2 misses	$967 \cdot 10^9$	$907 \cdot 10^9$	6.6
L3 misses	$28 \cdot 10^6$	$27 \cdot 10^6$	3.7
Instructions	$122 \cdot 10^9$	$115 \cdot 10^9$	6.0

Table 6.3: Comparison of performance metrics between different parallel prototypes

Comparison of Different Parallel Prototypes

Section 6.2 has presented the multiprocessing prototype of the LHCb software framework. An older version of GaudiMP only supported the creation of subprocesses before the initialization period of the application (Prototype 1). The current prototype forks child processes after the initialization finished but before the main loop starts (Prototype 2). This impacts the amount of pages being shared between child processes and therefore it also impacts performance metrics. A comparison is given in Tab. 6.3, where different versions of GaudMP have been executed with 48 worker processes. Prototype 2 requires overall less instructions since subprocesses are created later in runtime and an improvement of 6% has been observed. In addition, less cache misses occur due to better memory sharing. DC misses decreased by 6.2%, L2 misses by 6.6% and L3 misses by 3.7%.

6.5.6 Benchmark Results on Current Manycore Systems

The scaling factor of an application is not only influenced by the way how software is parallelized but also how the underlying micro architecture is designed and performs with increasing workloads. As mentioned in the introduction Non Uniform Memory Accesses (NUMA) occur on manycore systems and they can have a significant performance impact. When multiple cores share the same resources, like bandwidth and memory, this can become a limitation. Consequently, the core count cannot be very large. In order to avoid this problem, the concept of Non Uniform Memory Access is applied which has its origin in Symmetric Multiprocessors (SMP). SMPs are systems which contain multiple independent processors. Each processor is assigned to a separate socket, but all them share the same memory. In contrast to SMPs, multicore processors have the core logic of a processor replicated multiple times on the same chip. Cores are located normally on the same socket. SMPs and multicore processors face the same problem, namely that memory becomes a limitation. Nowadays, multicore systems with large number of cores are in general subdivided into different nodes and sockets.

NUMA systems consist of several nodes and each node has its own memory (Fig. 6.12). Cores can access memory from other nodes, but the delay is larger than accessing the local memory. A task can also access remote caches, load data and store it inside its local cache. NUMA systems can be cache coherent (ccNUMA) or non cache coherent (nccNUMA) [188]. In first case, controllers need to communicate changes of cache data in order to guarantee coherence. This increases communication and can negatively impact performance.

NUMA can notably degrade speedup of multithreaded applications when threads share the same memory but run on different nodes. In such circumstances, disabling NUMA can increase performance and minimize traffic between nodes. Context switches can also have a negative impact. For instance, a process is executed on core 0 and all its data is inside the local memory. If it is waiting for I/O, a context switch will be performed, such that another task can run on core 0. Once the I/O request has been resolved, the process can be restarted and this might happen on a different NUMA node. However, its data still relies in the local memory of core 0. Consequently, it requires to access the memory

in a non uniform way because the data is in a remote memory. In order to avoid such performance loss, tasks must be pinned to a certain core or node. CPU affinity can notably increase performance and this has been done for the following evaluations. GaudiMP has been executed on different manycore systems in order to evaluate its speedup.

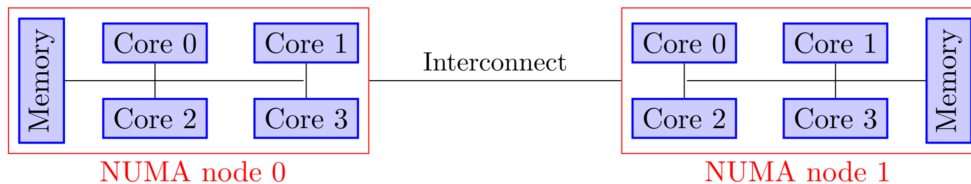


Figure 6.12: System with 2 NUMA nodes

Intel Xeon with 8 cores

As already shown in the introduction, Intel Xeon E5 family and the 5000 sequence are the most common server CPUs in the Worldwide LHC Computing Grid. A parallel reconstruction has been executed on an 8-core Intel Xeon L5520 processor running at 2.26 GHz which belongs to the 5000 sequence and is based on the Nehalem EP architecture [40]. This type of CPU has been released in the year 2009. The system provides 24 GB RAM, two 32 kB L1 caches per core, one 256 kB L2 cache per core and one 8192 kB L3 cache per node. Hyperthreading and frequency scaling are disabled in order to minimize side effects. Fig. 6.13 shows the speedup curve obtained from these measurements. Accordingly, the parallel reconstruction job does hypothetically not scale beyond 23 cores. For verification purposes additional measurements on a system providing more than these number of cores have to be undertaken.

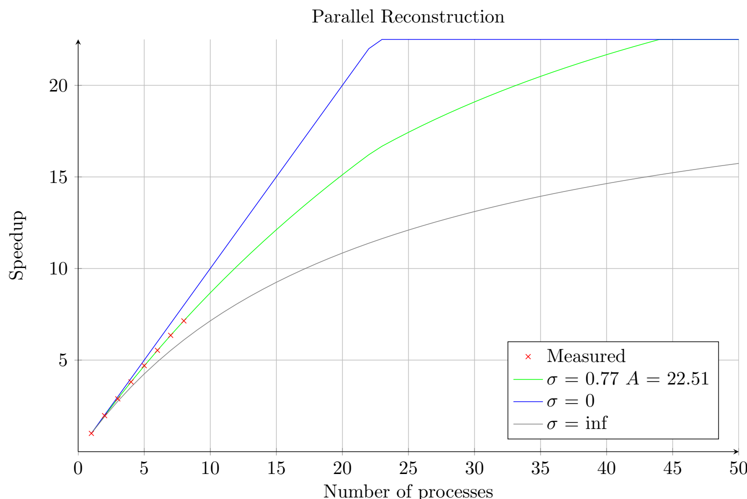


Figure 6.13: A and σ for parallel reconstruction jobs on an Intel Xeon (8 cores)

Intel Xeon with 40 cores

The same testcase has been executed on a 4 socket Intel Xeon E7-4870 system, where each socket consist of 10 real cores. It belongs to the E7 family, is based on the Westmere EX architecture and has been released in the year 2011 [39]. Executing GaudiMP with 40 processes results in a better speedup than estimated (Fig. 6.14). It is about 26.7, but 21.6 has been predicted by the speedup curve obtained in the previous subsection. Hence,

better values for A and σ can be achieved on this machine than in the previous case. Differences are mainly caused due to the underlying architecture. The main difference between Nehalem EP and Westmere EX chips is the manufacturing process which is 45 nm in the first case and 32 nm in the latter one. Nehalem EP chips are less power efficient, which might result in poorer performance when workload is increasing. Furthermore, the processors differ in their sockets. Intel Xeon L5520 uses a socket of type LGA 1366, which is used in server systems with large RAM. Intel Xeon E7-4870 uses the improved version LGA 1567, which provides more Quick Path Interconnect links. This also impacts the overall performance.

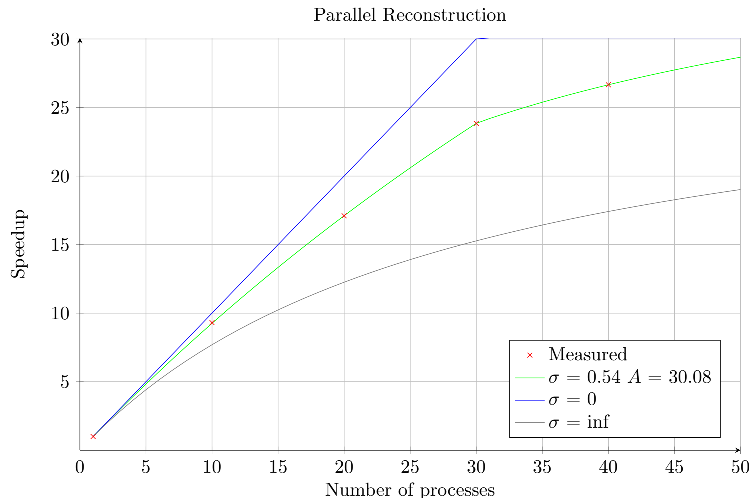


Figure 6.14: A and σ for parallel reconstruction jobs on an Intel Xeon (40 cores)

AMD Magnycours 48 cores

AMD already produced processors with large core counts in early years. However, due to the missing hyperthreading technology processors are not more powerful than the Intel variants with less cores. The AMD Magny-Cours has been released in the year 2010 and can support up to 48 cores. It belongs to the 6100 series of AMD Opteron processors and is also based on 45 nm manufacturing process. Running the same parallel reconstruction job on this manycore system shows a slightly worse scaling than on the Intel 40 core machine. An average parallelism of 27.12 and a variance in parallelism of 0.78 can be achieved (Fig. 6.15). However, the speedup is better than the predicted curve obtained from the 8 core Intel Xeon system. It estimates a speedup of 21 in case of 40 cores, while the AMD test system reaches a speedup of 24. It becomes clear, that not only the software itself matters but also the underlying micro architecture and its scaling behaviour.

CERN Cloud

CERN has its own cloud infrastructure, where experiments can run tests or production jobs on [31] [62]. Since more than one core can be requested at a time, it allows to do measurements with multicore jobs. Modifications in DIRAC have been necessary, such that the job agent is able to execute a task in parallel. The job agent is running inside a virtual machine and picks up the jobs which are assigned to the CERN cloud. The memory footprint has been monitored over a wide range of jobs and Fig. [31] shows the results. When a job finishes, the memory footprint becomes zero and the job agent has to pick up the next job. As soon as it starts, the memory footprint increases and remains at a certain level. The difference between PSS and RSS values indicate the memory saving due

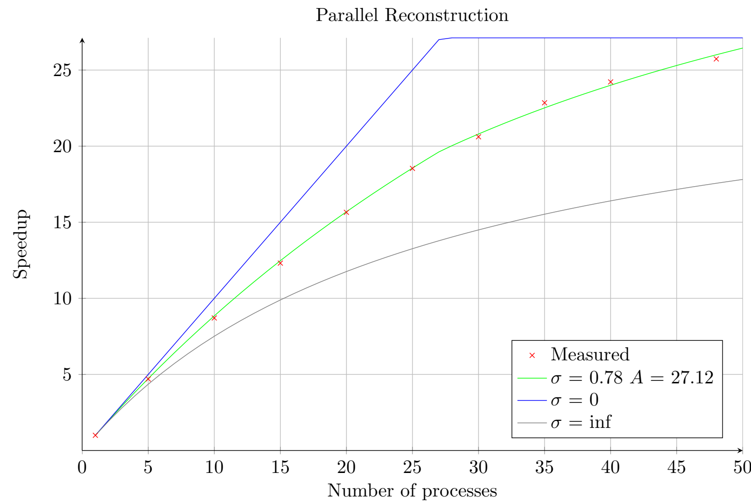


Figure 6.15: A and σ for parallel reconstruction jobs on an AMD Magnycours (48 cores)

to parallel software prototype (GaudiMP). It is obvious, that the range is nearly similar in each job and the reduction is about 30%. Multicore jobs were executed with 4 worker processes. Better results can be achieved when jobs are executed with a larger degree of parallelism.

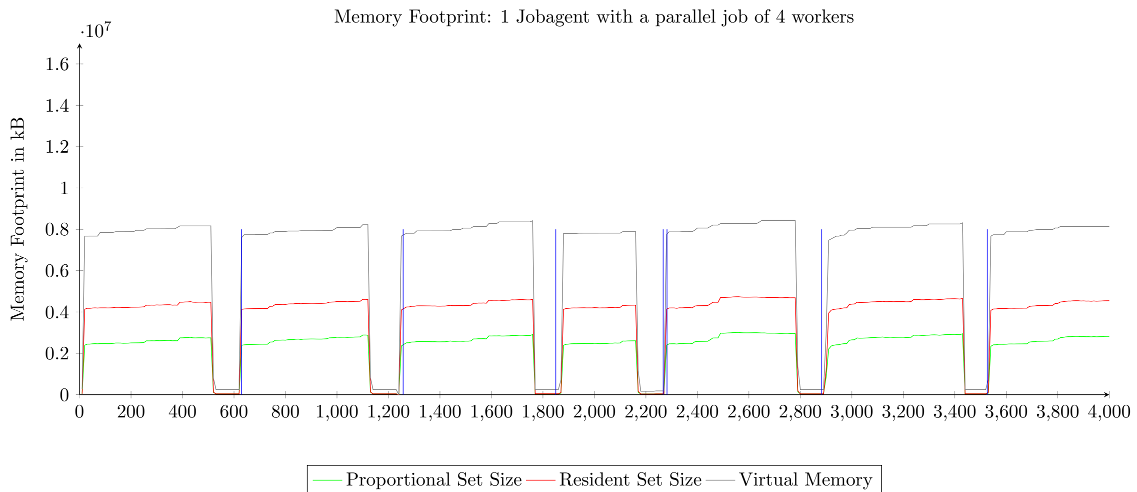


Figure 6.16: Memory footprint of multicore jobs on the CERN cloud (Agent restarts indicated by blue lines)

6.6 Summary

This chapter has discussed the main parallel concepts and their applicability on HEP software. Two parallel prototypes and their limitations have been discussed. The main criteria of the framework like flexibility and transparency are still provided within the multiprocessing prototype (GaudiMP) of the LHCb software framework. Impacts on the core level of the software have been minimal, since the fork of subprocesses takes place before the first event. The previous subsections have given an evaluation of this parallel prototype with respect to memory reduction and speedup. It has been shown, that postponing the creation of subprocesses can notably improve the overall memory footprint. A reduction of about 50% has been evaluated with 8 worker processes. However, speedup deteriorates

a lot when GaudiMP is executed with more than 20 worker processes. On top of that, the parallel prototype has been tested on diverse multi-socket CPUs and it has been shown, that speedup differs.

Consequently, the next step is to define the best degree of parallelism for a job. The following chapter deals with the problem of scheduling multiprocessor tasks.

7. Optimisation at the Level of Workload Scheduling

Virtual Organizations (VOs) and grid sites need to find a solution on how to submit multicore jobs to the Computing Grid such that CPU utilisation does not deteriorate. In the beginning experiments were relying on whole node allocation [17]. However, studies have shown that such an approach can lead to bad system utilisation. As explained in [17] it causes resource partitioning because singlecore jobs would not be allowed to run on nodes reserved for multicore tasks. From the perspective of a grid site it is worthwhile to allow resource sharing between single and multicore jobs. It must be ensured that none of these job types starves. It is still an unsolved issue, whether scheduling of multicore jobs is subject to the VO or the grid site. The CMS experiment suggests an internal scheduling [119] [90], while the ATLAS experiment relies on scheduling done by the grid site [86]. ATLAS proposes to execute their applications always with a certain number of processes and to only request exactly this number of job slots. The aim of internal scheduling is to simplify resource provision for grid sites. They supply arbitrary multicore job slots and VOs need to handle the matching of their jobs on these job slots.

The aim of this work is to go beyond this scope by first characterizing the scheduling problem and then to optimise it with respect to job throughput. It does not only matter that jobs run in a shorter time period, but also how efficiently a job uses the CPU. As shown in section 6.5 the parallel prototype of the LHCb software framework does not scale linear with the number of cores. It reaches a certain degree, beyond which it cannot use additional cores efficiently any longer. This complicates the scheduling, since jobs must be limited in the number of processes they request. Even though it is easiest executing a job with the number of cores provided by a grid site, throughput deteriorates when a job is not able to scale sufficiently. This chapter will show the related problems and possible solutions. It starts with an evaluation of the impact of multicore jobs with respect to total throughput and this is presented in section 7.1. Section 7.2 discusses a new job model and section 7.3 deduces the objective function. Section 7.4 illustrates methods for solving efficiently the scheduling problem. Since an estimation of job requirements is necessary as input for the solving procedure, section 7.5 discusses how this can be improved. Section 7.6 discusses the impact of wrong predictions and possible improvements.

7.1 Impact of Multicore Jobs

Each type of job has other system requirements and scales differently. While stripping jobs are normally memory bound, simulation jobs are CPU bound. Memory footprint as well as run time of these jobs also differ and depend on many diverse parameters. Consequently, mechanisms must be applied to automatically select the best degree of parallelism for each job which can be done by a scheduler. Not limiting the number of processes can result in a large loss in efficiency when a system provides many processing units. This can be computed as:

$$1 - \frac{S(n)}{n}. \tag{7.1}$$

In the case of linear scaling $S(n)$ is equal n and the loss becomes zero. Applying this formula on the speedup curves obtained from the different LHCb job types results in values presented in Tab. 7.1. Assigning 30 processes nearly generates a loss up to 37% of CPU time. Accordingly, throughput decreases drastically if jobs are not limited in their degree of parallelism. Determining the best degree is mainly a trade off between efficiency and the job requirements like memory requirements, runtime reduction and available job slots.

Number of processes	Reconstruction	Stripping	Simulation
10	11.1	8.5	12.3
20	20.8	16.4	22.8
30	27.4	23.4	37.1

Table 7.1: Loss in efficiency in % based on measured speedup curves

7.2 Moldable Job Model

[95] classifies parallel tasks depending whether a job can change its number of processes and whether this is initiated by the system or user. They distinguish between the following four types:

- **Rigid**: The degree of parallelism is fixed and it is determined by the user at the submission time. Jobs cannot be executed with fewer or more processes.
- **Evolving**: The job modifies its degree of parallelism during runtime and this is initiated by application's requirements. The system must provide the additional resources otherwise the job would interrupt.
- **Moldable**: An application can be executed with an arbitrary number of processes. Before the job is finally started, a scheduler determines the best degree of parallelism. The number of processes cannot change during the execution of the job.
- **Malleable**: A job can change its degree of parallelism during execution. Changing the number of processes is requested by a scheduler.

The parallel prototype of the LHCb software framework (GaudiMP) does not support each type. Currently, GaudiMP jobs can be rigid or moldable, since they can be executed with an arbitrary number of processes. The implementation could possibly support malleable jobs, too. This would require a signal handler, which notifies worker processes to finalize. The most appropriate type is a moldable job scheduler because the number of processes is defined by a scheduler and not by a user. Since 10k jobs are running each day, job properties and the correspondent schedules must be determined in an automatic way. This kind of scheduler has been proposed in [162] and details are shown in the following sections.

Aim of a Moldable Job Scheduler

The Workload Management System of DIRAC sets up task queues for each newly submitted production. Productions can have different sizes, but they typically contain hundreds to thousands of jobs. A moldable job scheduler can optimise such task queues based on the moldability of jobs. Pilots sent to the worker nodes, set up the environment and pick up tasks from the task queues. Accordingly, the pilot knows the properties of the worker node like memory capacity, processing units and reserved time period. This information can be forwarded to the moldable job scheduler which can then define job properties and optimise schedules. This optimisation happens in an offline manner, because the whole schedule is computed before the first job from the schedule is actually started. Parameters like exact runtime are not known beforehand and must be estimated. Consequently, schedules must be recomputed in an online scenario because jobs finish earlier or later than expected. This will be subject of section 7.6.

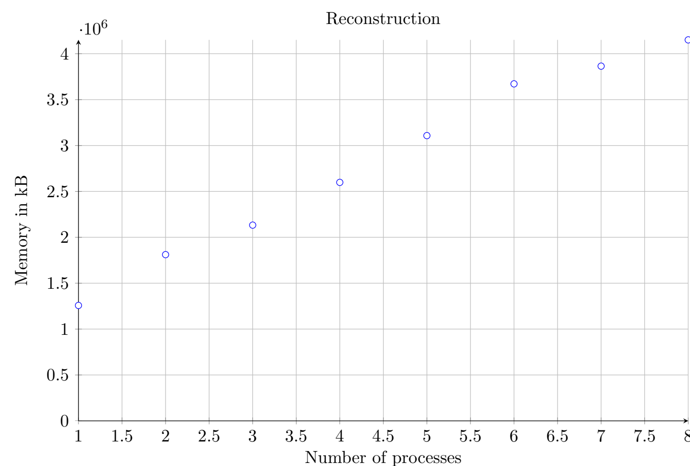


Figure 7.1: Total memory footprint of a parallel reconstruction job

Each job can be represented as a rectangle, where one dimension corresponds to the number of processes and the other one is equal the expected runtime. Workload can be seen as the product of both dimensions and it increases with rising degree of parallelism due to non linear scaling. Assuming sublinear scaling, the optimal per job efficiency [168] is reached when a job is executed with only one process. Even though, a job can be executed with an arbitrary number of processes, a minimum degree exists which is defined by its memory requirements. Knowing how much memory can be shared with rising partition size (Fig. 7.1), the minimum number of required processes n can be computed as following:

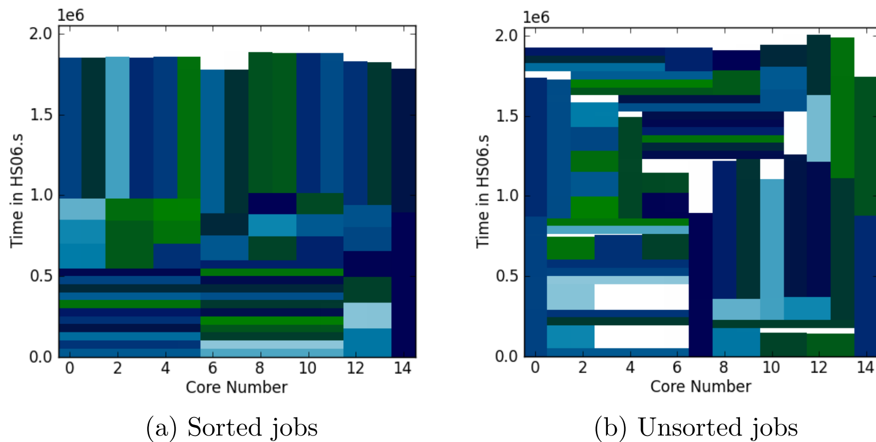
$$\frac{\text{Memory footprint}}{n} \leq \frac{\text{Size of RAM}}{\text{Number of job slots}}.$$

Assuming a system which provides 40 cores with hyperthreading enabled and 80 GB of RAM, then each logical process should not require more than 1 GB. Given the plot, shown in Fig. 7.1, the job must be executed with at least 2 processes in order to meet this requirement. Then, the total footprint is 1.8 GB, which is divided by 2 processes and results in 900 MB for each. Currently, computer systems in the WLCG are on general equipped with 2 to 3 GB per core.

Even there exists a minimum partition size for each job, it might be worthwhile to assign more cores to a job if this can improve the overall throughput. This scheduling problem presents a typical packaging problem, where the aim is to place as many items as possible within a given container. The moldable job model has the following steps:

1. The capacity of the worker node is estimated by computing the reserved time with the number of cores. Jobs are picked from the queue until the estimated total workload has exceeded the capacity of the worker node. Due to this overcommitment executing all selected jobs within the schedule presents a non reachable optimum.
2. Memory requirements of each job is estimated and the minimum partition size is computed.
3. A start schedule is defined.
4. Modifying the partition size of single jobs and redefining schedules. If overall throughput has improved, the solution is kept.
5. Repeat step 4 until a certain amount of iterations has been reached or no further improvement can be achieved.

It is actually irrelevant whether a task is scheduled in the very beginning or shortly before the deadline, because its creation or submit time is not important. In fact, the production as a whole matters and the production can be closed only when the latest job finishes. Given that, tasks can be randomly placed inside a schedule. Certain rules can be applied in order to define more compact schedules and accordingly increase throughput. Tasks can be placed with large degree of parallelism first, since this allows to keep fragmentation low (Fig. 7.2). In addition jobs can be placed in the lowest position and either oriented on the right or the left side. In the following a heuristic is used which puts items in the leftmost position [77].



(a) Sorted jobs (b) Unsorted jobs

Figure 7.2: Sorted versus unsorted jobs

7.3 Objective Function

The objective function is the mathematical representation of the scheduling problem. It is used for the optimisation process. The main metric for LHCb jobs is high throughput and the question is which parameters influence it. The aim is to process as many jobs as possible within a certain time period and tasks which do not meet the deadline are interrupted by the resource provider.

Gaps can occur in a schedule because of the unavailability of tasks, which are sufficiently small. The loss can be computed by subtracting the sum of workloads from the total capacity C. The workload of a job j can be computed as:

$$\frac{time_j(1)}{S_j(n)} \cdot n, \tag{7.2}$$

where n is the number of processes, $S_j(n)$ the correspondent scaling factor and $time_j(1)$ its serial runtime. A job can run in shorter time, when it is executed on more cores. But as a result it also requires more resources. The workload can even increase, if a job is not able to scale with the number of cores. As already shown in the introduction of this chapter, non linear scaling is one parameter which negatively affects throughput. It causes loss of efficiency and consequently introduces overhead. This overhead can be computed by multiplying the number of processes n with the difference of actual and ideal parallel runtime. If a job j requires a serial runtime $time_j(1)$, it ideally requires a parallel runtime $\frac{time_j(n)}{n}$ when it runs in parallel on n cores. But software rarely scales linear with the number of cores due to synchronization between parallel processes and other factors. For LHCb jobs the real parallel runtime is always larger than $\frac{time_j(1)}{n}$. This results into the following objective function:

$$\underbrace{C - \sum_{j=1}^{J} \frac{time_j(1)}{S_j(n)} \cdot n}_{\textit{Lost due to gaps}} + \underbrace{\sum_{j=1}^{J} \left(\frac{time_j(1)}{S_j(n)} - \frac{time_j(1)}{n} \right) \cdot n}_{\textit{Lost due to non linear scaling}}, \tag{7.3}$$

where $S_j(n)$ is the speedup, n the number of cores a job j has used, $time_j(1)$ is the serial runtime, J is the number of jobs in the queue and C defines the capacity of the worker node. The first term computes the amount of gaps, which is the total capacity minus the sum of workloads of all jobs. The second term calculates the amount of lost CPU time due to non linear scaling. The aim is to minimize this function and it can be simplified as:

$$C - \sum_{j=1}^{J} time_j(1). \tag{7.4}$$

It looses the dimension presented by the numbers of processes. It shows that the throughput does not change as long as the same jobs are set within a schedule independently from their degree of parallelism. As an example: Assuming 10 jobs available and out of these only job 1 to 5 fit inside the schedule. If the number of processes of each task is modified, but still results in the fact that only the same 5 tasks can be placed, then the overall throughput will not change. As a result, increasing throughput can only be achieved by setting more or different combinations of jobs within the schedule. However, the latter equation can violate the constraint, that each job has a minimum degree of parallelism and that the number of job slots cannot be exceeded. The objective function has two extrema which are presented by the following two cases (Fig. 7.3):

1. Jobs run with minimum partition size (n = 1)
2. Jobs run with maximum partition size (n = max)

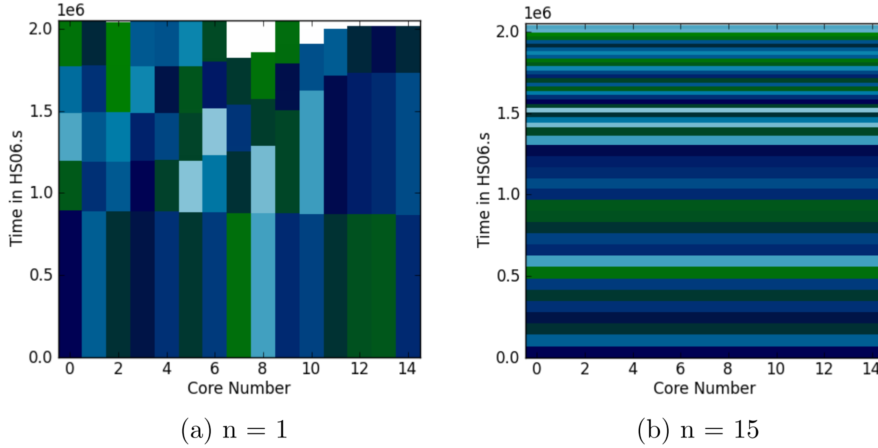
As already described, executing a job on one core results in optimal per job efficiency [168]. This is the case for the first scenario. Throughput can only be decreased by fragmented schedules and the second part of the objective function is zero:

$$\sum_{j=1}^{J} \left(\frac{time_j(1)}{S_j(n)} - \frac{time_j(1)}{n} \right) \cdot n = 0, \tag{7.5}$$

where n and S_j is equal 1. The second scenario results in a large loss due to non linear scaling, but the schedule is barely fragmented. The following applies:

$$\lim_{n\to\infty} \sum_{j=1}^{J} \left(\frac{time_j(1)}{S_j(n)} - \frac{time_j(1)}{n} \right) \cdot n \to \infty \tag{7.6}$$

while n is equal the number of available cores. Both parts of the objective function are counteracting and the aim is to find the best balance between both.



(a) n = 1 (b) n = 15

Figure 7.3: Minimal and maximal partition size

7.4 Solving the Objective Function

Scheduling multiprocessor tasks belongs to the class of $\mathcal{NP}$-complete problems, as described in [104]. $\mathcal{NP}$ is defined as a problem which can be solved by a non deterministic Touring machine in polynomial time. Cases which are $\mathcal{NP}$-complete, belong to the $\mathcal{NP}$ and $\mathcal{NP}$-hard problems. This means that the optimal solution cannot be found within a reasonable amount of time. The larger the problems size the more complex it becomes to find the optimum.

The described scheduling of LHCb jobs does not only face the difficulty of multiprocessor task scheduling but in addition jobs can be executed with an arbitrary number of processes. This results in an immense search space. The aim is, to find an optimisation algorithm which can find good solutions within a reasonable amount of iterations. This section will evaluate deterministic and probabilistic meta-heuristic local search methods and compare them with results obtained from IBM ILOG CPLEX CP Optimizer [7].

Defined Testcases

This subsection will specify a few test cases, which are used to evaluate different optimisation algorithms for the given scheduling problem. A worker node provides a particular number of cores and is available for a certain period of time. Cases have been defined, where the number of processing units vary from 8, 15, 30 to 50 cores. Time is indicated by HEP SPEC seconds (HS06.s). The time interval has been set to 2.5 million HS06.s for the first two cases and 1.5 million HS06.s for the last two cases. The reduction has been necessary otherwise these tests would not be solvable with IBM ILOG CPLEX CP Optimizer. The time limits of the test cases correspond to a value of about 69 hours and 41 hours on an Intel Xeon CPU which typically provides about 10 HS06 per core (as

previously shown in table 2.1). These are characteristic time values requested by the LHC experiments for running their serial tasks.

Taking the last test case and assuming a task with a duration of 10k HS06.s requiring only one core: there are $p = 1.5 \cdot 10^6 \cdot 50$ possibilities to start the job and $p - (1 \cdot 10 \cdot 10^3 \cdot 50) = 745 \cdot 10^5$ possibilities to place the job such that it finishes in time. Dependent on the position of the first task, a second one has a reduced amount of possibilities. Given the fact that a job can be in addition executed with an arbitrary number of processes, another degree of freedom is introduced. This results in an immense search space. So one goal is, to reduce the amount of possibilities and to find reasonable solutions with less overhead. One way is for example to define the order in which jobs are placed inside the schedule. If jobs with a defined property have to be placed in certain job slots, then this drastically reduces the amount of possibilities. Given the above example and assuming that jobs requiring one core have to be placed first, then this leads to only $1 \cdot 50 = 50$ possible permutations. The job can be set at time 0 in one of the 50 slots.

The first step is to specify job properties as described in section 7.2. Memory and runtime requirements can be predicted within a given range dependent on job type and options. Therefore, a maximum likelihood estimation can be applied on values obtained from prior similar jobs since the normalized time per event fits a Gaussian distribution (Fig. 7.4). Given the HEP SPEC value of a worker node, runtime can be estimated by:

$$time = \frac{\text{Number of Events} \cdot \text{Maximum Likelihood}}{\text{HEP SPEC Value}}.$$

A more detailed description of estimating job requirements and related problems can be found in section 7.5. Memory has been limited to 1.2 GB per process within the test cases. This is a realistic value on CPUs with hyperthreading enabled which consequently provide more logical than physical cores [182]. Therefore, the available memory per core decreases. Typically, systems are overbooked by a factor of 1.5.

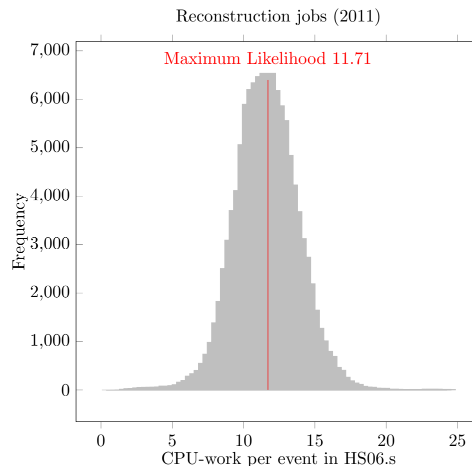


Figure 7.4: Workload per event of reconstruction jobs

7.4.1 Local Search Methods

Solving scheduling problems belongs to the field of discrete optimisation. The three main types of solving such problems are [186]:

- Local search
- Constraint programming

- Integer programming

Local search starts in a random solution and then goes iteratively through the search space seeking for better solutions. Such methods normally get stuck in local optima, but in general reasonable results can be achieved within a few iterations. Constraint programming constructs a tree with all possible solutions. Constraints must be defined which allow to reduce the search space. This is done via constraint propagation which removes branches violating constraints. This technique guarantees to find the global optimum, but dependent on the problem size it results in a large computational complexity and memory footprint. Integer programming is very similar to constraint programming, but more limited [140]. Integer programs allow only linear inequalities as constraints. The following subsection shows a deterministic and non deterministic local search method for solving the given objective function.

7.4.1.1 Iterative Deterministic Approach

A deterministic local search always ends up in the same solution when the identical start point and search direction are chosen. Hill Climbing is such an algorithm, which randomly launches a solution and goes from there to the next local optimum. Results can only be improved by either selecting a different start point or directing the search else wise. Therefore, the neighbourhood must be defined such that the algorithm can decide to which neighbour solution it has to move. A solution presents a certain schedule and a better result is found when the job throughput of a new schedule is better than the previous one. Neighbour solutions are determined in the way how new schedules are generated. This is achieved by altering the properties of jobs which can be done in many different manners. Either one or several jobs at a time are modified and either one or several additional processing units are assigned. Evaluating all possible modifications would go beyond the scope of this work. Based on results presented in [168], this thesis proposes to give in each iteration an additional processing unit to a single job and to select the next candidate dependent on a certain order. The iterative deterministic approach has the following steps:

1. Order the list of jobs
2. Create initial start solution
3. Pick the next job from the list of candidates and increase its partition size by one
4. Create new solution
5. If solution has improved accept it, if not remove the job from the list of candidates
6. Repeat step 3-5 until no better solution can be found

Different criterion are evaluated for step 1. The question is, which task will profit more from an additional core and can improve job throughput at the same time. These might be jobs which provide good scaling and therefore have a good efficiency ratio or large jobs which can reduce their runtime the most. To investigate this, an inefficient start schedule has been created. It is then evaluated which job order allows a faster convergence to the final optimum. Jobs are sorted by the following criterion:

- Best Speedup
- Largest decrease in runtime
- Position in job list (FCFS)

The first criterion considers only the scaling behaviour of different job types but not their size. Jobs with high efficiency are chosen first. Since scaling normally grows sublinearly, this criterion favours jobs with small partition size. The second metric respects the scaling factor and the size of a job. If a job generates a large workload it will profit more from an additional core. Tasks can also be chosen randomly, like in the order they arrived. Fig. 7.5 shows results obtained from the Hill Climbing algorithm using different sorting metrics. The y-axis indicates the loss in CPU time compared to the total CPU time available. The aim is to reach the least loss with as little iterations as possible. It can be seen that the local search converges more quickly, when tasks are chosen by their decrease in runtime or scaling. As previously explained, jobs with large degree of parallelism are placed first. Consequently, tasks which are not placed within the schedule have small degree in parallelism and are selected first by such sorting metrics.

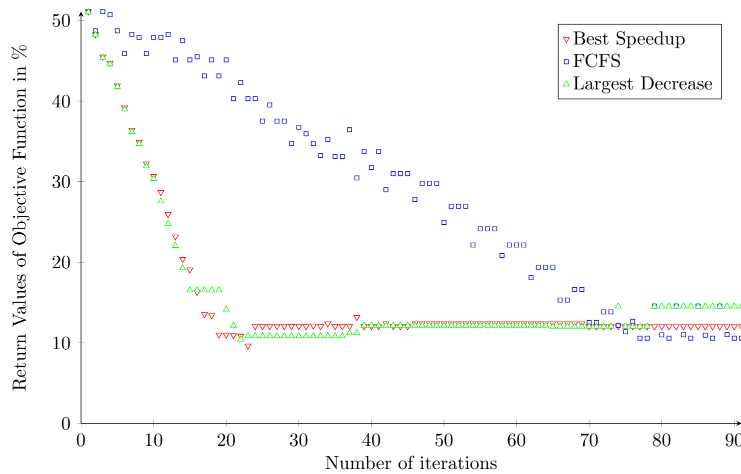


Figure 7.5: Generated solutions by different sorting rules

Step 5 of the algorithm can be modified, in order to improve the local search. The question is, when it becomes more reasonable to remove a candidate from the list and to not consider it any longer for modifying its degree of parallelism. It might be that assigning one more processing unit to a job does not improve the overall throughput, but it does with two more units. If a candidate would be directly removed, the second solution cannot be found. As Fig. 7.5 shows, modifying jobs with large partition size will most likely not affect the overall throughput. Consequently, candidates will be removed when their modification did not help to find a better solution and when their parallel time is less than 20% of their serial time. So the following two criteria have been chosen:

- Remove the selected candidate directly (criterion 1)
- Keep it for a few more iterations (criterion 2)

Tab. 7.2 shows the results achieved by the Hill Climbing algorithm with different configurations. The column optimum shows the quality of the best schedule which could have been generated with a certain configuration. It indicates the return value of the objective function compared to the total capacity provided by the worker node: the smaller the better it is. Criterion 1 stops the local search earlier and requires accordingly less amount of iterations leading to worse final solutions. However, criterion 2 allows a better exploration of the search space and reaches results up to 30% better. As an example, an optimum of 7.9%, 5.6% and 5.4% can be found for the test case with 50 cores and criterion 2. A solution of 25.1%, 41.5% and 42.1% is found with criterion 1. The column iteration indicates when the optimum has been found. As previously discussed, sorting jobs by speedup and

Sorted by	Cores	Criterion 1		Criterion 2	
		Optimum	Iteration	Optimum	Iteration
Position in job list	8	5.2	2	4.1	17
	15	3.5	38	2.7	24
	30	10.5	78	8.6	125
	50	25.1	119	7.9	219
Decrease in runtime	8	5.2	2	3.5	81
	15	3.5	4	3.3	4
	30	10.4	23	4.5	84
	50	41.5	86	5.6	147
Speedup	8	5.2	13	2.5	40
	15	3.3	4	3.3	4
	30	9.6	24	9.6	24
	50	42.1	86	5.4	154

Table 7.2: Results found by different sorting metrics

decrease in runtime converges more quickly. It takes for example 125 iterations in the test case with 30 cores and criterion 2 to find the optimum instead of 84 and 24 iterations like in the other test cases. The different configuration impact the direction of the local search and it does not result in the same optimum as can be seen in Tab. 7.2.

7.4.1.2 Probabilistic Meta-heuristic Approach

The main disadvantage of deterministic local search methods is, that they reach local optima from where they cannot return. In order to overcome this issue, worse solutions must be accepted in order to allow the algorithm to find different local optima. One approach is Simulated Annealing, which is based on the concept that heated solids cool down and reach a steady state which represents a minimum energy configuration. A very detailed description can be found in [167], [113]. The slower the cooling process and the higher the start temperature, the better the final solution. The key elements are:

- Start and end temperature
- Cooling function
- Randomness of solutions
- Number of generated solutions per temperature step

Temperature influences the acceptance probability: the higher the probability the more likely it is that a worse solution will be accepted and the better the search space can be explored. This is determined by:

$$p = e^{-(E_{new}-E)/\theta},$$

where E_{new} represents the energy of the current iteration, E the energy of the last accepted solution and θ the current temperature. The energy defines the quality of a solution and it is defined by the objective function. If the start temperature and as a result the acceptance probability are very high the search becomes a random search. If they are quite low the algorithm cannot come out of local optima. As a result, choosing appropriate values is quite important for the quality of the solution and the computational complexity of the algorithm. The second condition is the cooling function. It determines how quickly the search converges to a certain optimum. Commonly used is a geometric decrement like the following:

$$t = \alpha \cdot t.$$

The larger α the slower the system cools down and according to [113] values between 0.8 and 0.99 are appropriate. In the beginning the decrements will be larger than towards the end. Randomness is another key parameter in Simulated Annealing. In each temperature step random solutions must be generated. In the context of the given scheduling problem randomness is applied at two levels:

- Choose a random job
- Assign a random number of processing units

The value must be greater than the requested minimum partition size and smaller than the number of processing units. Another important parameter is the amount of generated solutions per temperature step. In the given problem it is defined by the number of jobs available.

The main challenge is defining appropriate values for the cooling function, the amount of generated solutions and the start and final temperature [113]. It should be chosen such that the computational complexity does not become an issue. Having 1000 jobs and 1000 temperature steps already leads to 1 million iterations. On top of that, the Simulated Annealing algorithm can get stuck in a certain configuration, from where it cannot return to a different state. This requires then a restart from the point, which has been found as best solution so far.

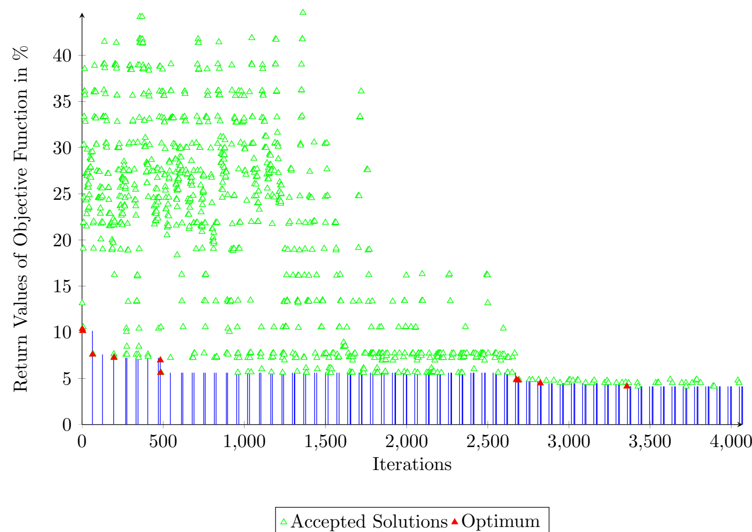


Figure 7.6: Test results with $\alpha = 0.9$ and with restarts (blue lines)

If a better schedule can be found, the solution is accepted and in case it is worse, it will be accepted with a certain probability. When the list of candidates is empty, the system cools down and starts the next cycle. Fig. 7.6 presents results achieved by the Simulated Annealing algorithm. A random start point is generated and a restart is performed as soon as the algorithm got stuck in a certain state. This happens, when critical configurations exist from where the algorithm cannot return such that it always produces the same solution. If this occurs, the acceptance probability becomes one and the newly generated solution which is the same like the previous one will be always accepted. This results in an infinite loop. In order to avoid that, the algorithm must be reset to the last known optimum when it has not generated a different solution after a certain number of iterations. Such a scenario depends a lot on the formulation of the objective function. If the search space contains many plateaus, it is more likely that a Simulated Annealing algorithm will get stuck. Indeed, many restarts are required as shown in Fig. 7.6. This is related to

Cores	Loss in throughput	Iterations	Improvement
8	2.5%	18096	0%
15	2.3%	32246	0.4%
30	4.5%	32246	0%
50	4.6%	54681	0.8%

Table 7.3: Results found by the Simulated Annealing algorithm

the problem, how jobs are placed within the schedule. It has been previously shown that more compact schedules can be generated when jobs with large partition size are placed first. Assuming the scenario, all jobs are assigned the maximum number of processes: a random modification of one single job will not impact the schedule any longer, since it will be always the last item which is going to be placed. Since the task queue has been slightly overcommitted, the randomly modified job will not be placed at all. Accordingly, the same job throughput will be achieved in each iteration.

In order to solve this issue, the last optimum is memorized and used as restart configuration. Fig. 7.6 shows that many worse solutions are randomly generated, going up to about 45% loss in throughput. After 1500 iterations the algorithm converges slowly to a final state. After 4000 iterations it reaches the local optimum of 4.1%. The algorithm would not easily find better solutions when no restarts are applied (Fig. 7.7). Too many worse solutions are generated and the algorithm does not manage to converge quickly.

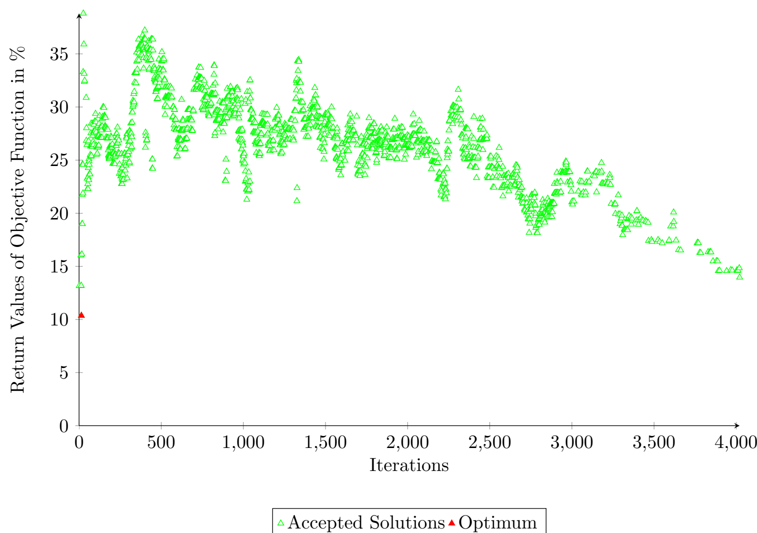


Figure 7.7: Test results with $\alpha = 0.9$ and without restarts

7.4.1.3 Combination

Better results can be reached when the deterministic and probabilistic approach are combined. First, the Hill Climbing algorithm finds the best local optimum starting from a certain initial solution. This state is then used by the Simulated Annealing algorithm as a start and restart configuration. Applying this on the proposed moldable job scheduler reaches the following results for the 4 test cases (Tab. 7.3). Slightly better solutions can be found compared to the iterative algorithm, but at the same time more iterations are required.

Simulated Annealing can find the global optimum when acceptance probability and number of iterations are infinite. In the worst case, it requires more computational time than

exhaustive search, which figures out all possible combinations. Randomness is the key element, such that solutions are not reproducible as long as no random seeds are used.

7.4.2 IBM ILOG CPLEX CP Optimizer

As previously described, constraint programming is another technique to solve scheduling problems and to find the optimal solution. One of the most efficient commercial tools is IBM ILOG CPLEX CP Optimizer [7]. It builds a tree, which consist of all possible combinations and where each leaf presents a certain schedule. Some combinations may violate constraints and can therefore be removed from the tree. This is done via constraint propagation which evaluates the feasibility of solutions and removes branches of the tree. This helps to reduce the search space and to find the global optimum within less computational time. The scheduling problem generates an immense search space, since a job can be arbitrarily placed within a schedule and can be executed with any number of processing units. The throughput of a schedule can be improved by:

- Position of a job within a schedule
- Increasing/decreasing jobs' partition size

The first approach is the most common one practically used [96]. It is the aim to evaluate the global optimum found by IBM ILOG CPLEX CP Optimizer when the position of jobs within a schedule is modified. First, constraints must be defined. The total workload of jobs cannot exceed the capacity provided by the worker node. Consequently, the first limitation can be expressed as:

$$\sum_{j=1}^{J} \frac{time_j(1)}{S_j(n)} \cdot n \leq C. \tag{7.7}$$

The number of processing units cannot be exceeded at any time. This presents the second constraint:

$$\left(\sum_{j=1}^{J} n \cdot job_j.running(t) \right) \leq nCores \qquad \forall t \in T : 0 \leq t \leq t_{max}. \tag{7.8}$$

Executing the defined test cases with the given constraints results in an enormous search space. Since the complete tree containing all combinations are kept in memory, it leads to a memory consumption of more than 44 GB. The problem is, that the constraints do not reduce the search space sufficiently enough. As a result, optimality cannot be proven with IBM ILOG CPLEX CP Optimizer due to the hardware limitations. Therefore, the best solution found so far must be accepted as the global optimum. Memory has been limited to 23 GB of RAM and additional 5 GB of swap area. Results found by IBM ILOG CPLEX CP Optimizer and the number of explored branches are presented in Tab. 7.4. As it can be seen, many more iterations are necessary but at the same time better results can be found. It requires several days to explore these amounts of branches. But the runtime of the optimiser is basically limited by the hardware and it terminates when it runs out of memory. The Hill Climbing approach requires only a few minutes to complete when it is configured as presented in section 7.4.1. The runtime of the Simulated Annealing algorithm can be more time consuming than exhaustive search dependent on the configuration of the parameters. It is in any case worthwhile, to stop the algorithm when a result has been found that can be considered as good enough.

Cores	Loss in throughput	Explored branches
8	1.6%	$25 \cdot 10^9$
15	1.5%	$75 \cdot 10^9$
30	2.0%	$43 \cdot 10^9$
50	1.7%	$89 \cdot 10^9$

Table 7.4: Results found by IBM Cplex Optimizer

7.4.3 Summary

It is important, that schedules for jobs running in the Computing Grid are quickly generated. However, the quality of solutions depends mainly on the available computational time. Even though results found by IBM ILOG CPLEX CP Optimizer are better, the calculation period is large. A local search which combines a deterministic and probabilistic method is the best approach in order to define how to schedule multicore jobs. As shown in section 7.4.1 jobs with good efficiency ratio should be considered first for modifying their degree of parallelism.

One of the most important steps of a moldable job scheduler is the estimation of job requirements as explained in section 7.2. Scheduling decisions are based on the properties like predicted runtime and memory. A bad estimate will have a negative impact on the scheduling since tasks finish later or earlier than expected. Currently, such an estimation is not provided by the Workload Management System. However, the generated workloads must be better understood in order to optimise them. As a result, the question is not only how to schedule jobs but also how to improve prediction and backfill schedules. This will be discussed in the following sections.

7.5 History Based Estimation

The aim of a history based estimation is to analyse prior jobs and derive the meta data that influences job requirements the most. Since the LHCb experiment runs ten thousands of jobs each day, a lot of statistical information can be obtained. Especially, runtime of reconstruction jobs is strongly correlated to physics related meta data. A moldable job scheduler needs to estimate runtime, memory and the scaling factor of jobs. The scaling factor (speedup) can be derived from runtime and the number of used cores.

Data Mining Methodologies

The research area of data mining explores data and tries to find patterns, like correlations or groups of clusters [109]. Based on small training sets, certain information can be deducted and applied on new data. Since this research area deals typically with large amounts of information, statistical methods are often used, like for example histograms, clustering, regression. More advanced techniques are neuronal networks, where an input vector is mapped via several hidden layers on an output vector. Decision and classification trees present another common technique [109]. They are used to learn decision rules and to predict values. A branch is a combination of nodes and indicates the information flow starting from the root node. Each leaf contains a decision criterion.

The complexity and the application of these techniques depend mainly on the problem itself. For example, clustering techniques and decision trees are more appropriate for classification problems, while statistical tools help to give a rough estimate of numerical values. The following subsection will focus on the description of the feature space of LHCb jobs.

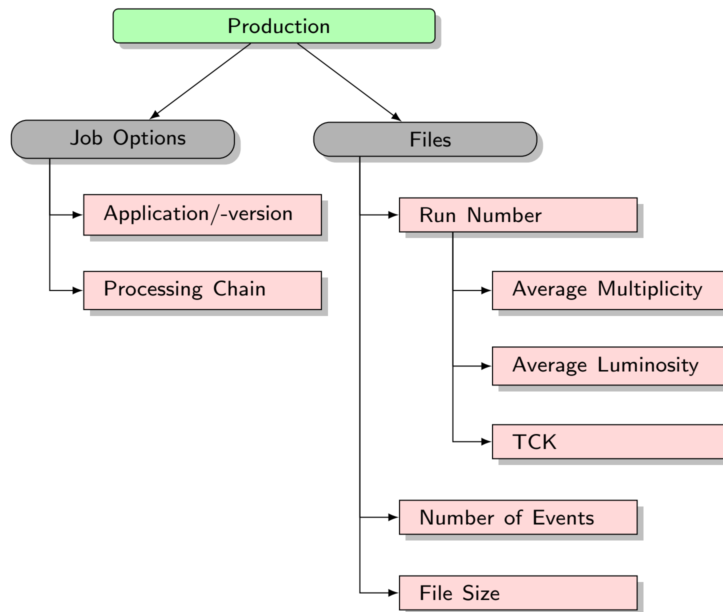
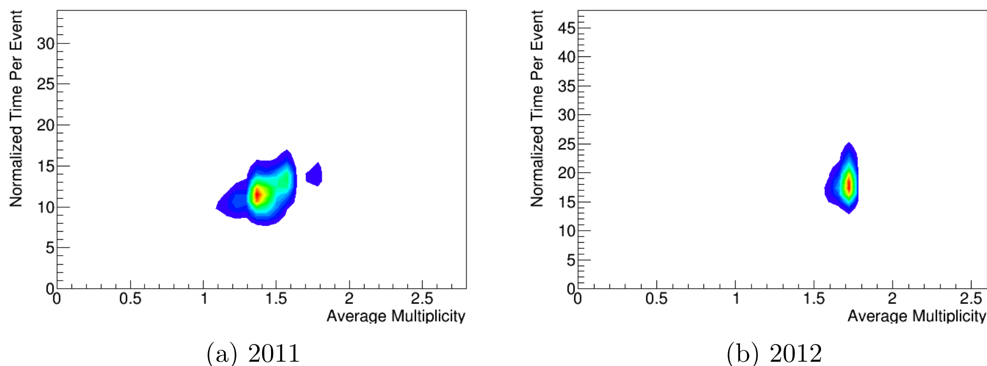


Figure 7.8: Overview of the different features of a production

7.5.1 Defining the Feature Space of LHCb Jobs

Within the LHCb experiment productions are created which contain a set of files and the so called job options. Jobs based on a production are then created and sent to the Computing Grid. A job can be defined and categorized by a lot of parameters and certain parameters are again subdivided by more categories. The most important ones are explained in the following and are shown in Fig. 7.8 and Fig. 7.12.

The two beam lines collide every 50 ns and about 2 to 3 collisions occur per bunch crossing. This value is indicated by the **average multiplicity**. The detector registers the generated particles and stores them as event. An event presents therefore a snapshot of the detector. A well known correlation is: the larger the multiplicity the more complex the reconstruction of an event [87]. This is caused by the fact that the combinatorial complexity increases. This can be seen in Fig. 7.9a and Fig. 7.9b. They show the average multiplicity compared to the normalized time per event from the year 2011 and 2012. In 2012 an increase of the beam energy has taken place which resulted in a larger average multiplicity and more complex events. As it can been seen in Fig. 7.9a and Fig. 7.9b the average multiplicity has slightly increased from 1.5 to 1.8. The required processing time per event increased from 11 HS06.s to 17 HS06.s.



(a) 2011

(b) 2012

Figure 7.9: Multiplicity versus normalized time per event

Not all information can be stored on tape, since several million collisions occur per second. Therefore, a hardware and software trigger are responsible for filtering out less relevant information. The configuration of the triggers are indicated by an ID which is the so called **trigger configuration key (TCK)** [45]. It allows to determine which data set has been recorded under which kind of trigger conditions. The database also stores information about the output rate of the hardware trigger (**avL0PhysRate**)[45], the software trigger (**avHLTPhysRate**)[45] and its software version (**programVersion**)[45]. Events are then summed up to raw files, whose size is limited up to 3 GB. The larger the file size, the larger the total processing time. Fig. 7.10 shows the total processing time compared to the size of raw files. It can been seen that the file size is limited to 3 GB. Until this point the total processing time shows nearly linear scaling to the file size. The dispersion is caused by the different characteristics of each event: the smaller the file, the less events are stored inside and the larger the impact of variations in processing time per event becomes.

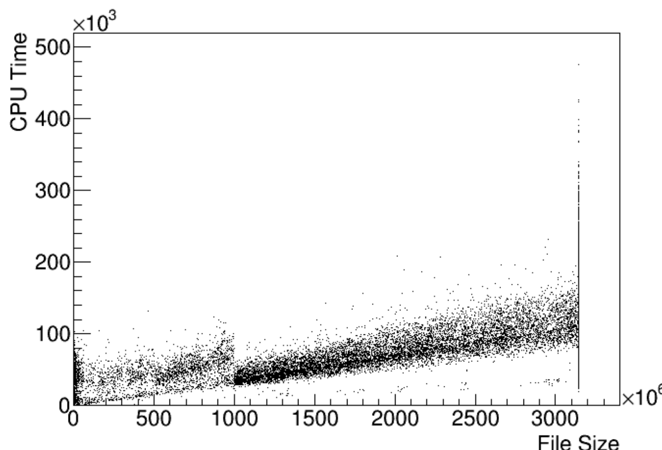


Figure 7.10: File size versus processing time (2012)

LHC experiments require constant conditions for their analysis. But the density of the beam decreases during an LHC fill. Experiments apply the concept of runs that subdivides an LHC fill into smaller slots. It can be assumed, that the conditions at LHCb were stable during a run. An LHCb run is stored in database as an ID (**run number**)[45]. It indicates, when the data taking has been started and stopped and this might be up to one hour. Each raw file belongs to a certain LHCb run. **Average multiplicity** and **average luminosity** are stored in a database for each run [45]. Luminosity indicates the number of collisions per second and is strongly correlated with the multiplicity. In general, the luminosity decreases during a run since the beam looses protons. Nevertheless, it can be assumed that the luminosity remains the same, because the overlap of the beam lines changes during a run (luminosity levelling [53]). This technique is applied in the LHCb experiment, to keep the luminosity at a certain level. Experiments like ATLAS and CMS do not support it and Fig. 7.11 shows the instantaneous luminosity during a run.

During an LHC fill the magnet is configured such that the magnetic field has a certain direction (up or down). This information is stored in the run database that contains information about all runs. Further experiment conditions recorded in the database are for example, the energy of the LHC and the Velo position. The Velo is moveable, such that it can be positioned very close to the beam. When the beam is injected it is normally opened and during a stable beam it is closed.

Productions contain a set of files, they define the job options and they are indicated by a **Production ID**. Each file consist of a certain **number of events** and it allows a rough estimate of the overall processing time. Jobs from the same production execute the

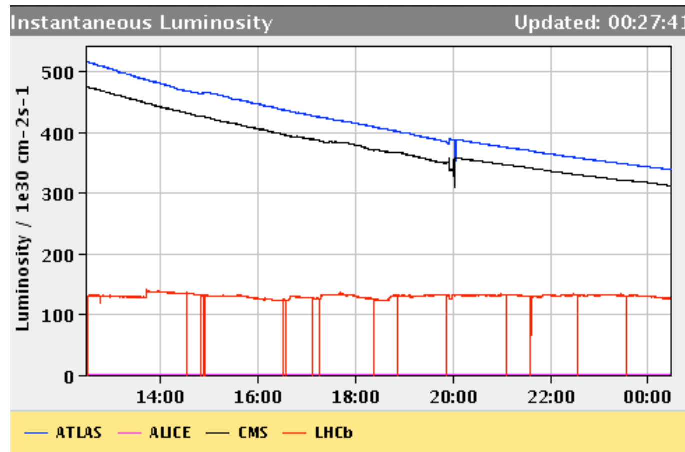


Figure 7.11: Instantaneous luminosity for ATLAS, CMS, ALICE and LHCb (taken from [10])

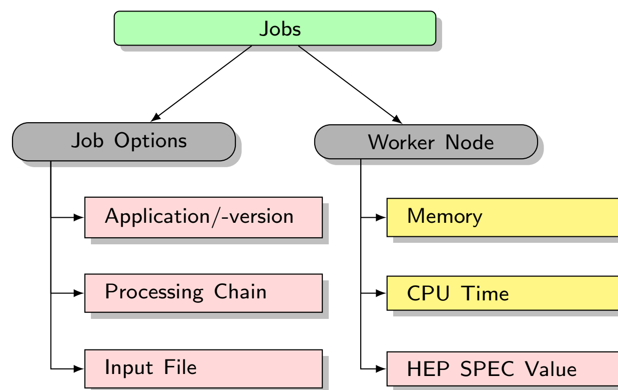


Figure 7.12: Overview of the different features of a job

equivalent processing chain with the same application version. The **type of application** and its **version** is another category, since it allows to define whether a job is memory or CPU bound. As soon as a pilot job is executed on a worker node, a normalization factor is computed which is indicated by the **HEP SPEC value**. Normalized processing time allows a fairer comparison of jobs running on different CPU types. Other worker node specific information, like cache size, CPU type and grid site are stored as well in the database. This allows for example to compare same CPU types between different grid sites. The features of a job are presented in Fig. 7.12. Each job belongs to one production, runs at a certain worker node and processes a certain set of files. As soon as a job has finished its memory footprint and required runtime are stored in the database. These are the values which have to be estimated for new submitted jobs.

Derived Features

In order to make jobs more comparable further features can be derived. The **normalized CPU time per event** indicates the required processing time per event and it is the product of the normalization factor and CPU time divided by the number of events. The obtained probability density function of this feature corresponds normally to a Gaussian distribution: some events are more computational complex, others are less complex. Another feature is the **average event size** which is the file size divided by the number of events. It can give a rough estimate on the complexity of events: very small events are in general less computational complex.

7.5.2 Consideration of Single Feature

If no job meta data is available, requirements can be predicted by calculating the average values taken from prior jobs. The most common technique is the maximum likelihood estimation (MLE) which can also be used if only a few observations are available. It maximises the likelihood function under the assumption of a certain distribution [115]. In the case of runtime prediction a Gaussian distribution can be assumed and the likelihood function can be expressed as:

$$\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(\mu - x_i)^2}{2\sigma^2}}, \tag{7.9}$$

where μ is the mean of the distribution, σ is the standard deviation, x_i corresponds to the observations and n is the number of measurements. As described in section 7.4 runtime can be predicted by taking the normalization factor of a CPU, the number of events and the maximum likelihood into account.

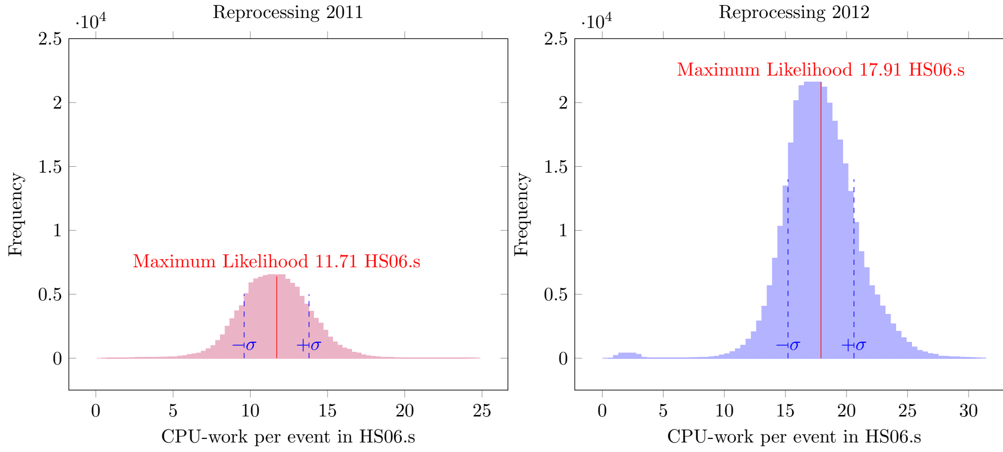


Figure 7.13: Required time per event for reconstruction jobs

Reconstruction

This has been computed on the reprocessing productions from the year 2011 and 2012 (Fig. 7.13). In such productions raw files from the complete experiment year are reprocessed with the same software version and job options. Fig. 7.13 shows that the maximum likelihood has increased from about 11.71 HS06.s to 17.91 HS06.s and also the amount of data has significantly increased. This is related to the different experiment conditions in the year 2012. LHC has been running with higher energy and therefore more complex events have been recorded at a larger data rate. The standard deviation is about 2.1 in the year 2011 and about 2.7 in 2012. This means that runtime is on average wrongly estimated by 2.7 HS06.s.

Fig. 7.14 shows that the memory requirements of reconstruction jobs do not fit a Gaussian distribution any longer. Applying the maximum likelihood estimation under the assumption of a Gaussian distribution might cause significant errors. Nevertheless, memory is not as critical as runtime and an overestimation will not have any negative impact. Fig. 7.14 shows that the maximum likelihood is 1759 MB (1.7 GB) with a standard deviation of about 45 MB. As a result, estimates are on average wrong by a value of 45 MB. The standard deviation appears to be rather low and the main reason is that virtual memory is recorded in the database instead of the physical memory consumption. Consequently, correlations might not be visible, because LHCb applications address a lot of memory

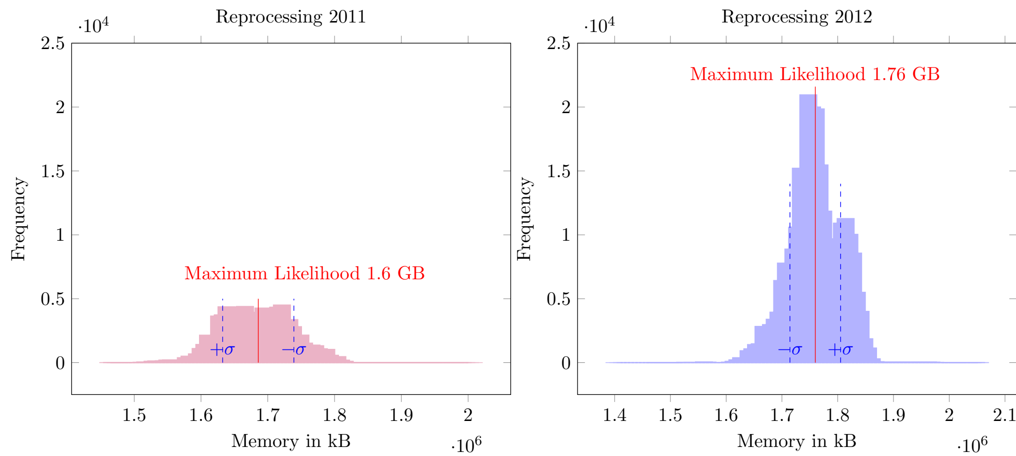


Figure 7.14: Memory requirements of reconstruction jobs

buffers which are sometimes not used at all. Many jobs appear therefore with the same consumption, but the real physical memory consumption is much smaller.

Stripping

The computational complexity of stripping jobs has not significantly increased as shown in Fig. 7.15. A slight increase from 5.26 HS06.s per event to 5.87 HS06.s can be observed and this is mainly caused by different stripping options in the correspondent year. The characteristics of events does not influence the computational complexity of these jobs since they only separate the information and put them into different output streams. It can be seen that more data has been processed by stripping jobs in the year 2012 compared to 2011.

Fig. 7.15 also shows the memory requirements of stripping jobs from the year 2011 and 2012. The memory footprint has significantly increased from 2.1 to 3.3 GB and is mainly related to different software versions and stripping options. Until the mid of 2011, LHCb used its own service for writing objects to disc (POOL) which is based on ROOT data structures [11]. However, due to changes in the ROOT framework this service has become very inefficient in the context of I/O and memory usage [11]. Since the end of 2011, the POOL service is no longer available and files are directly written via the ROOT framework. Stripping jobs write several output streams and for each of them many internal buffers are allocated during the application main loop. Due to a poor configuration of these buffers, much more memory has been requested by stripping jobs than actually needed. This is the main reason, why such an increase in memory was observed in 2012. A fix has been applied in summer 2012 that solved this memory issue [28].

Simulation

The generated workload of simulation jobs depends on many more parameters that change every year. They simulate only certain decays, while the real events present a mixture of different decay products. Certain decays are rather complex and generate huge workloads of hundreds of HEP SPEC seconds per event. Only a few hundred events can be simulated compared to reconstruction and stripping tasks. Fig. 7.16 shows the maximum likelihoods of each different event type. As it can be seen the majority of simulation jobs produces a workload of 500 to 700 HS06.s per event. Apart from the event type, further job options are relevant for the prediction of workload. Users can apply selection criteria that allow to either choose all events or to extract only special ones. They can adjust the crossing angle between the two beam lines or the energy of the protons. This can lead to larger number

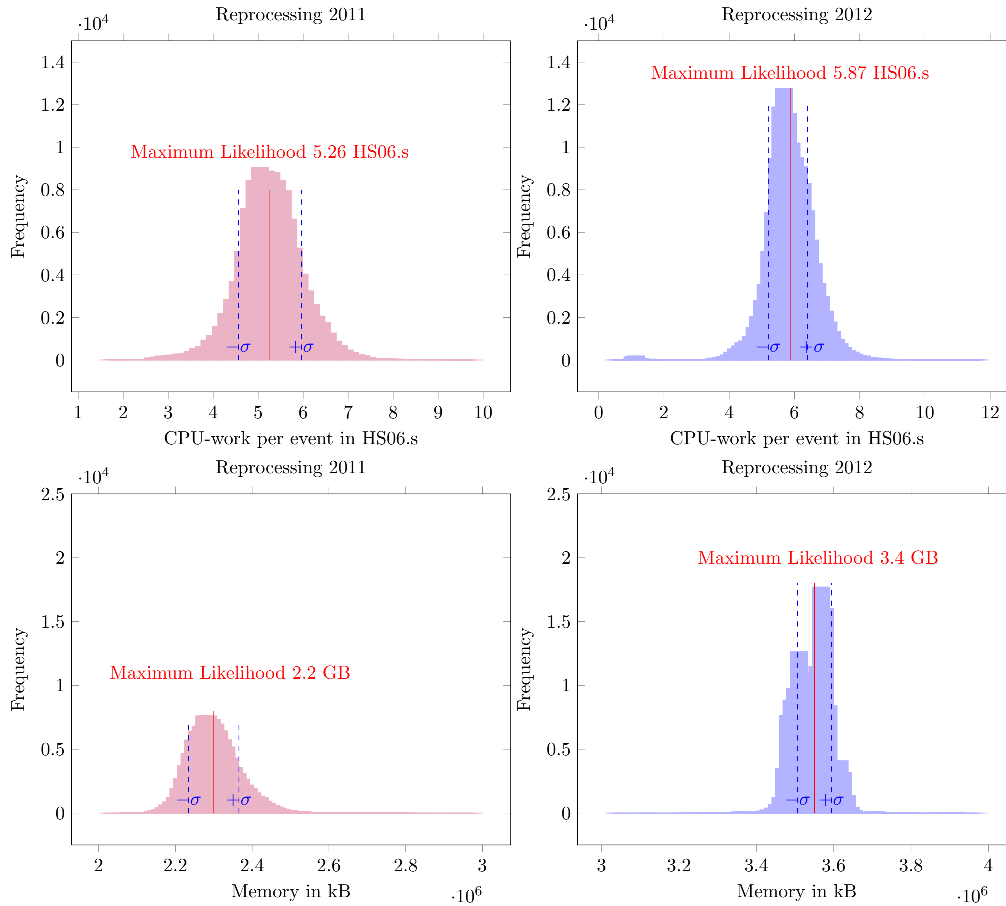


Figure 7.15: Memory requirements and generated workload of stripping jobs

of produced particles and it increases the time required to simulate the detector response. As already explained previously, more than one collision normally occurs per bunch crossing and that presents another parameter in simulation productions. Consequently, jobs simulating the same event type might show a large difference in generated workloads due to different job options.

Since some information like the applied job options cannot be easily retrieved from the databases, it is not trivial to derive good estimates from prior jobs. The generated workload does not necessarily fit a Gaussian distribution (Appendix 9.1). While Fig. 7.16 shows only the maximum likelihoods for given simulation productions, the related distributions are presented in the appendix. Even if jobs from the same production and therefore with the same job options are chosen, several peaks might occur in the overall workload. This has several reasons. The time required for generating events can significantly differ from few seconds to hours. The distribution of random numbers also matters, and users are actually allowed to choose any kind of distribution. In addition, simulating the physics of events might also cause quite different workloads. This becomes even worse due to the fact, that simulation jobs only process a minimal amount of events. If a job generates a very complex event, then it requires much more time than another job from the same production which did not produce a similar event. Since simulation jobs only process few hundreds of events, the impact of such complex events on the overall runtime is large. As a result, the underlying distribution of these runtime values is not necessarily Gaussian.

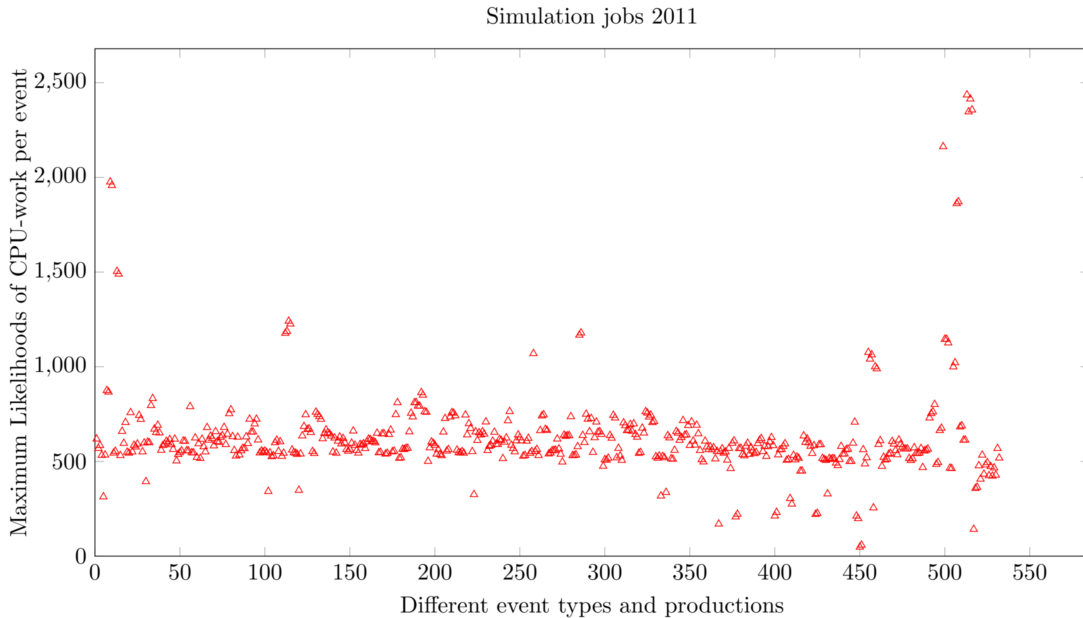


Figure 7.16: Maximum likelihoods of simulation jobs (2011) with different event types and from different productions

7.5.3 Consideration of Multiple Features

The last section focused on considering only runtime and memory requirements of prior tasks. In the following it will be analysed how meta data can be used to optimise the estimation of job requirements. Especially, physics related information like multiplicity and luminosity form the characteristic of events and have therefore a large impact on the runtime. The question is how they can be used to optimise the estimation.

Since the aim is to predict real continuous numbers, statistical tools like regression can be applied. It is relevant to reduce the problem size on a small subset of features, in order to avoid over fitting the problem. In the case of over fitting, the derived model fits very well the training data and even outliers but it will badly predict new values. Only the following features have been considered:

- File size
- Number of events
- Event Size
- HEP SPEC value of the worker node
- Average multiplicity
- Average luminosity

This represents a 6 dimensional search space where each axis correspond to one feature. The aim of linear regression (LR) is to find the line within this search space which minimizes the mean squared error [115]. All features have different units and must be normalized. This can be done by computing the z-scores which are defined as:

$$Z = \frac{x - \mu}{\sigma}, \tag{7.10}$$

where x is the observation, μ the mean and σ the variance. If data is not normalized, features with large numerical values will dominate. For instance, number of events or file

size. Their impact would lead to wrong correlations. The aim is to fit training data and derive a hypothesis from it that is used to estimate new data. With linear regression (LR) the hypothesis can be expressed as [57]:

$$h_\theta = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6, \tag{7.11}$$

where x_{1-6} are the normalized values for file size, number of events, event size, HEP SPEC value, average multiplicity and luminosity. θ_{0-6} must be defined such that the mean squared error is minimal. This can be done by minimizing the cost function via gradient descent. The cost function can be defined as [57]:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2, \tag{7.12}$$

where m is the number of training data. In order to compare results an error vector can be defined which contains the difference between predicted and real runtime values. In a next step the root mean squared error (RMSE) can be computed on this vector, which is determined by [192]:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} v_i^2}{n}}, \tag{7.13}$$

where v_i is an element of the error vector. The root mean squared error allows a fair comparison between results obtained from linear regression and maximum likelihood estimation.

Reconstruction

Linear regression has been applied on the reconstruction jobs obtained from the reprocessing productions of the year 2012 and Tab. 7.5 shows the results for different combinations of features. The first test has taken all features into account and reaches a mean squared error of 2.16. Consequently, runtime is falsely estimated by 2.16 HS06.s. This is an improvement of 22% compared to the maximum likelihood estimation (MLE), which reaches a value of 2.7 HS06.s as shown in section 7.5.2. The results also show, which features have a large impact on runtime. It appears that the average luminosity and event size have a low impact. When they are removed from the linear regression procedure the same improvement can be achieved (Test 3). Removing important features like the file size, HEP SPEC value or the number of events has a negative impact (Test 2). The estimation of memory requirements cannot be improved much. This is related to the problem, that the virtual memory value is stored in the database as previously discussed.

	CPU time per event			Memory
	Test1	Test2	Test3	Test1
File size	0.80	-	1.05	7533.51
Event size	0.19	0.80	-	-4906.07
HEP SPEC	-0.97	-	0.97	2907.18
Number of events	-1.33	-	-1.55	348.36
Average multiplicity	0.67	0.98	0.59	11926.63
Average luminosity	-0.08	-0.08	-	996.16
Offset	18.13	18.13	18.13	1763553.31
RMSE	**2.16**	**2.46**	**2.16**	**44350**
Improvement in % over MLE	**+22**	**+12**	**+22**	**+2**

Table 7.5: Linear regression results for reconstruction jobs from the year 2012

Stripping

Considering multiple features in the context of stripping jobs is more complicated since there is not a 1 to 1 relation between jobs and files. A stripping job usually parses several input files and the feature file size must represent the sum of all input files. Stripping jobs simply collect and filter out events. As a result, parameters like file size and number of events are strongly correlated with the memory requirements and runtime. Physics related parameters have minor impact as shown in Tab. 7.6. It shows, that a root mean squared error of 0.54 can be achieved which is a 25% improvement compared to the maximum likelihood estimation which reaches a RMSE of 0.74. The memory footprint of stripping jobs is strongly correlated with the file size. The larger the input file and the number of events, the larger the stream buffers and therefore the larger the memory footprint. Taking this important feature into account, can improve the estimate up to 6% compared to the maximum likelihood estimation that reaches an RMSE of 45 MB. In order to avoid over fitting the parameters average multiplicity and luminosity will be neglected in the further evaluation. They are not strongly correlated with the runtime and memory requirements.

	CPU time per event	Memory
File size	-0.19	19594
Event size	0.7	-4649
HEP SPEC	0.19	7116
Number of events	0.23	-11413
Average multiplicity	-0.08	1360
Average luminosity	0.003	1863
Offset	5.93	3547736
RMSE	**0.54**	**43073**
Improvement in % over MLE	**+25**	**+6**

Table 7.6: Linear regression results for stripping jobs from the year 2012

Simulation

It is rather difficult, to use multiple features in the context of simulation jobs. This is related to the problem, that the job options matter a lot and they cannot be easily obtained from the database. Even though the event type is known, it presents only a category to which jobs belong. Consequently, there are no numeric values apart from the runtime and memory which can be used for the linear regression.

7.5.4 Supervised Learning

The previous section has shown that estimation can be improved by taking more input features into account. Assuming that a new production is created, the question would be how to estimate the job requirements when only a small subset of training data is available. This problem corresponds to supervised learning, which intends to create a hypothesis on some provided training data. The training data contains jobs which have already finished and consequently the input and output feature vectors are known. The generated hypothesis is used to estimate the output vector of new data. They can have an arbitrary output vector due to effects, which have not be seen in the training data. The prediction model must be sufficiently general to also be applicable on unseen data. Therefore the learning algorithm has to make a set of assumptions which is also known as inductive bias [141]. In the category of unsupervised learning, data sets remain unlabelled and the algorithm tries to find patterns. This can be used in order to define clusters

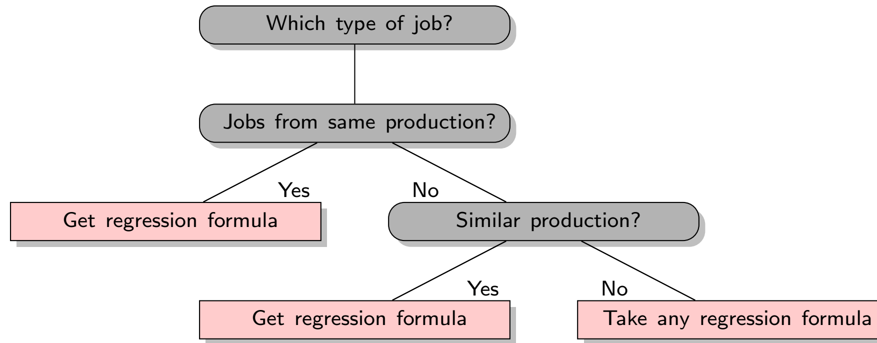


Figure 7.17: Decision tree for obtaining a regression formula

and categories. Supervised learning is more appropriate in the context of estimating job requirements.

The supervised learning algorithm has 4 steps. First, jobs from the same category must be found in order to obtain information. This can be done by a decision tree as presented in Fig. 7.17. If no similar jobs can be found, a regression formula from any other production can be obtained. The false estimate will be quite large in the beginning. As soon as a certain amount of jobs has been processed, the old formula can be discarded and a new one can be computed. In general, the error of false estimates will be large in the beginning due to the small amount of training data. However, the error will slowly improve over time, when more jobs will have been processed. The question is now, whether linear regression provides still better estimations compared to a maximum likelihood estimation. The learning algorithm has the following steps:

1. Find prior jobs from same or similar category (production ID, application version) and take formula
2. Predict requirements for the next k jobs, using either MLE or LR
3. When k jobs have finished, update formula with the new results obtained
4. Repeat step 2 and 3 until all jobs have finished

Reconstruction

In each step the error vector is extended with k values and the mean squared error is computed. This can be understood as accumulated error over time and Fig. 7.18 shows results obtained for reconstruction jobs. In the left plot k has been set to 100 and in the right one equal 1000. The shape of the curves is still the same, which means that the size of training data does not have a significant impact. Fig. 7.18 also shows, that the supervised learning algorithm with a linear regression approach can give up to 20% better estimates. The distribution of runtime per event values must be investigated in more detail, in order to understand the shape of the curve and why the error is increasing under certain circumstances. Fig. 7.19 shows the distributions sorted by the run numbers. Large squares indicate the center of these distributions, small ones present the outliers. The sum of all them yield the probability density function presented in Fig. 7.13. As it can be seen in Fig. 7.19 the center of the distributions are fluctuating, which is related to different conditions at the LHC and the detector in each run. Having such fluctuations and only a small set of training data results consequently in a large error. By the end of the year more outliers can be observed in Fig. 7.19, which is caused by performance tests of different subdetector elements. This is normally done by the end of an experiment year and requires different trigger configurations. As a result, events are recorded which differ from events recorded during the year. This is the reason why the error of the MLE

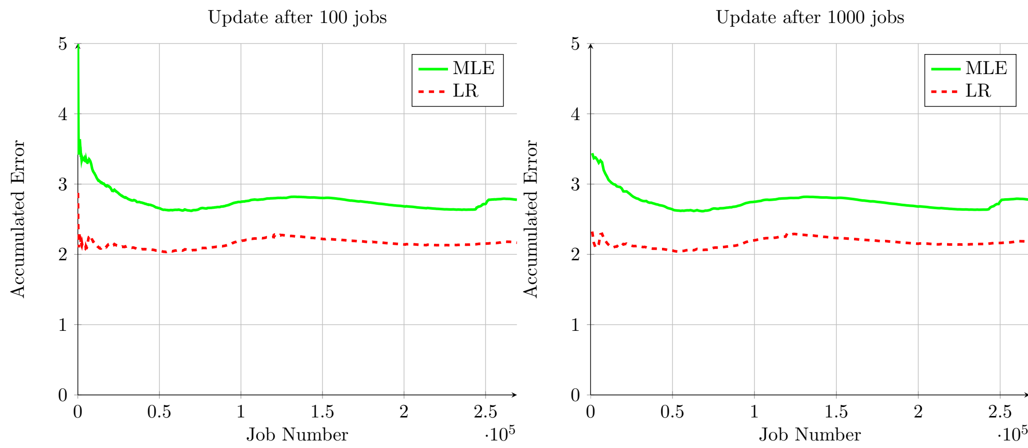


Figure 7.18: Accumulated error for the prediction of runtime per event (Reconstruction)

method increases after 250k jobs. However, the linear regression approach can estimate those outliers better.

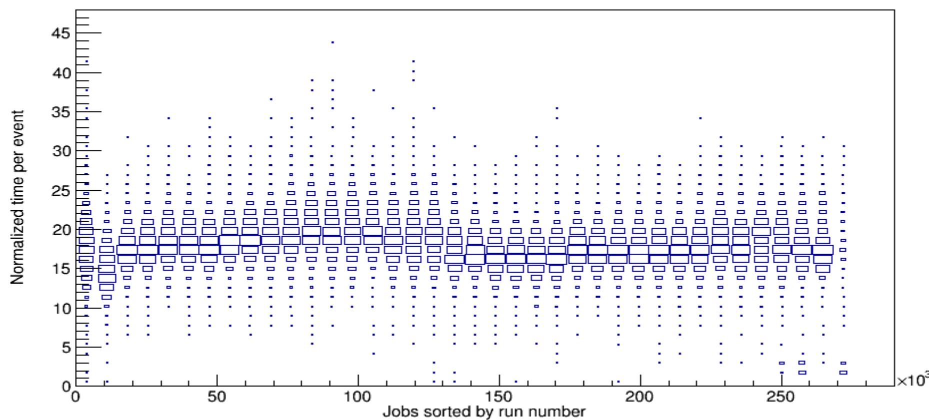


Figure 7.19: Normalized CPU Time per event of all reconstruction jobs from 2012 sorted by run number

Fig. 7.20 presents the accumulated error for the prediction of memory and it can be seen that the improvement of linear regression versus maximum likelihood estimation is minimal with about 2%. As previously explained, it is related to the problem that virtual memory is taken into account instead of the real physical memory footprint of the jobs. The standard deviation of the obtained distribution as well as difference between MLE and linear regression is small.

Stripping

Linear regression also provides better estimates in the case of stripping jobs. Since physics related parameters like average multiplicity and luminosity do not have a large impact, they have been neglected for the supervised learning algorithm. Fig. 7.21 shows the accumulated error for the estimated run time per event. As already shown in the previous section, modifying the number of training and predicted data does not influence the accumulated error much. In the left plot k has been chosen equal 100, in the right one equal 1000.

Fig. 7.22 shows the accumulated error for the estimated memory. The error is quite large in the beginning with a wrong estimate up to $6.2 \cdot 10^4$ kB. It decreases slowly until a large

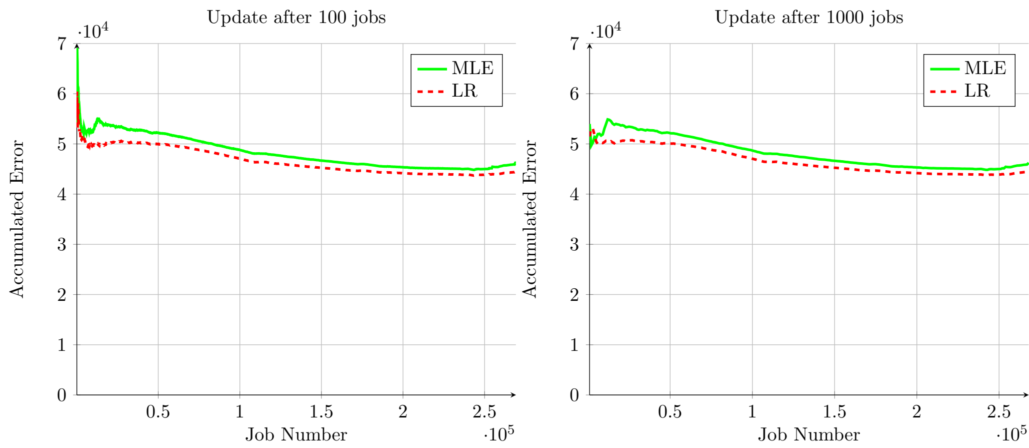


Figure 7.20: Accumulated error for the prediction of memory (Reconstruction)

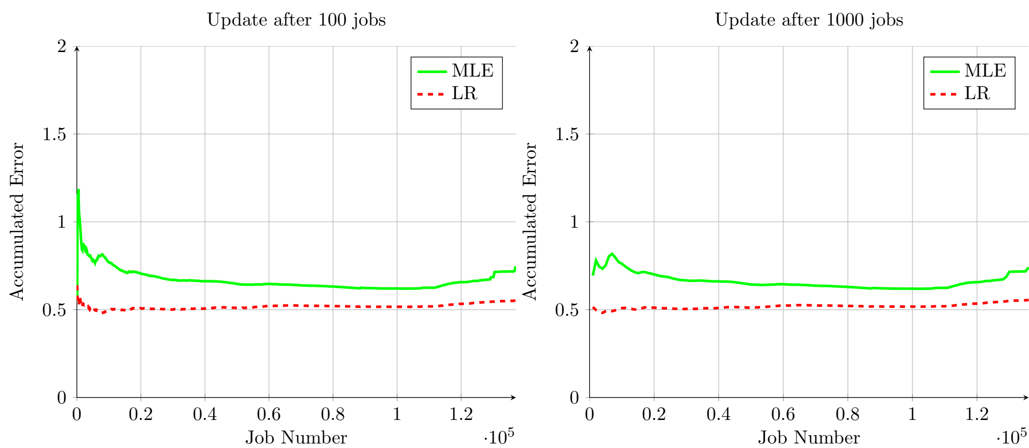


Figure 7.21: Accumulated error for the prediction of runtime per event (Stripping)

peak occurs. This is caused by outliers that cannot be correctly estimated neither by MLE nor by LR. A large majority of stripping jobs has an equivalent memory footprint of about 2.2 to 3.4 GB. But if outliers occur, they are up to two times greater than the average.

7.5.5 Summary

Job requirements are currently estimated by production managers who are responsible for creating and submitting new productions. Prediction is based on their user experience and normally some extra time is added, to make sure that jobs definitely meet the deadline. The previous sections have shown that this can be done in a more automatized way by retrieving statistics from prior jobs. Since certain features are based on a Gaussian distribution, maximum likelihood estimation is one applicable technique. It was also shown that taking more input features into account can significantly improve estimation of reconstruction and stripping jobs up to 25%. The estimation of simulation jobs is more complicated because job options play a major role and cannot be retrieved from the database. Another problem is, that virtual memory instead of physical memory is recorded and memory management depends a lot on the configurations applied at the grid site. Different operating systems or Linux kernels will handle memory pages differently. These might be the reason why no strong correlation could be obtained between input features and memory requirements.

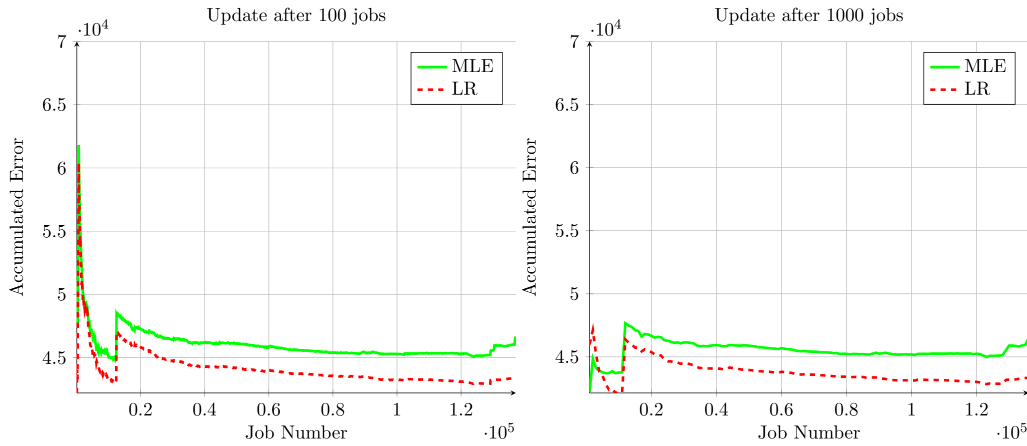


Figure 7.22: Accumulated error for the prediction of memory (Stripping)

7.6 Handling Uncertainties

The scheduling of jobs is influenced by uncertainties which are caused by false predicted memory and runtime. Instead of having a concrete value, a distribution of numbers is given, each of them having a different probability of occurrence. Even though linear regression of multiple features helps to improve estimation, there is still a certain error involved. First, this section will show in terms of a concrete example, how such uncertainties might affect scheduling decision. Results of this example have been presented in [163] and this section will mainly refer to this publication. Probabilistic backfilling will be explained and how it can be applied in this context as a further step of optimisation.

Impact of Uncertainties - An Example

Having 8 job slots and 3 jobs (simulation, reconstruction and stripping) each of them can be executed with an arbitrary number of processes. The question arises, how to choose the right combination such that the overall processing time is minimal. All possible combinations have been evaluated, as shown in Fig. 7.23. Jobs can run either one after the other, indicated by the combination (0,0) or two jobs start at a time which is presented by the outer layer of the triangle. The inner layer shows combinations when all three jobs start at a time, while the number of processes for the third job can be computed by $8-(x+y)$. Each square shows the improvement in percentage compared to the reference point (0,0) where each job runs with the maximum number of processes available. Since jobs do not scale linearly with the number of cores, the overall CPU time slightly rises. As Fig. 7.23 shows, an optimum of 18% improvement can be reached in (6,1), when 6 cores are assigned to the reconstruction, 1 to the stripping and 1 to the simulation job. The question arises, whether this optimum can be predicted with the given probability density function of prior jobs. This concrete example faces the objective of selecting jobs dependent on the increase of overall CPU time. The job which raises the overall time the least will be assigned another core. This can be expressed by:

$$\min\left(\frac{time_j(1)}{s(n)} - \frac{time_j(1)}{n}\right),$$

where $time_j(1)$ is the serial run time of job j, n the number of processes job j is using and $s(n)$ its scaling factor. The scheduler goes iteratively through the list of available cores and assigns them dependent on the evaluated minimum increase in CPU time. Runtime is therefore predicted with the probability density function of prior jobs and the maximum likelihood is taken.

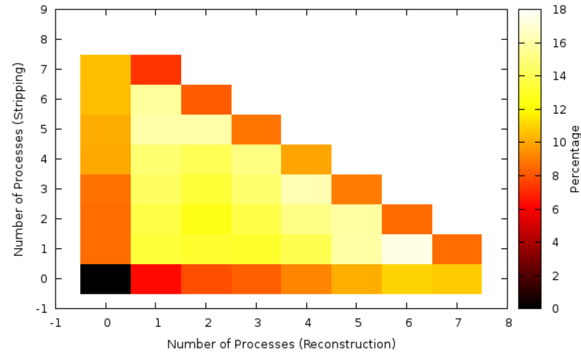


Figure 7.23: Different mixtures of three independent jobs (taken from [163])

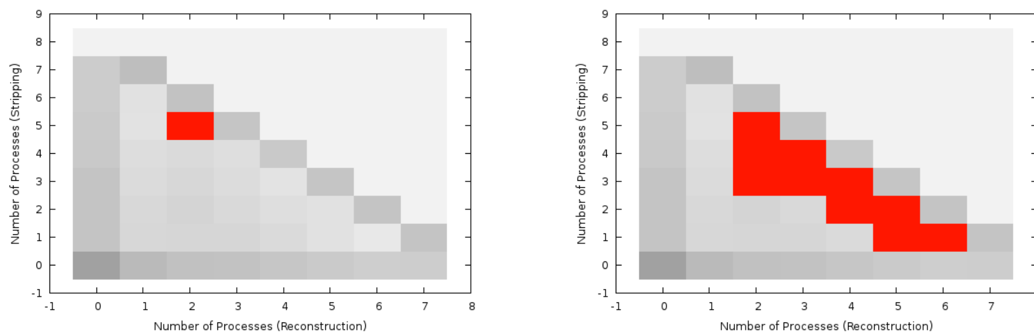
Knowing the maximum likelihoods and the scaling behaviour of the different job types, the following combination is found, presented in Fig. 7.24a. It presents the combination, which will be most likely the best. Nevertheless, the real optimum is another one and it shows, that taking the maximum likelihood as runtime estimate is a wrong assumption in this example. The question is which confidence interval the real optimum fits into. It can be expressed as:

$$time_{min} = \text{Maximum Likelihood} - x \cdot \sigma$$

and

$$time_{max} = \text{Maximum Likelihood} + x \cdot \sigma,$$

where σ is the standard deviation of the given distribution and $x \cdot \sigma$ is equal the confidence interval. Due to the variance an error is introduced which leads to a range of possible decisions while each of them has a different probability of being the global optimum. Choosing $x = 2$ results into a probability of $\sim 95\%$, that the actual runtime of the job will be in the interval. Fig. 7.24b shows the scheduling decisions, when the confidence interval 2σ is chosen. Several combinations are found and the probability is large that one of them presents the global one.



(a) Decisions found by taking into account maximum likelihoods and confidence interval of 0σ

(b) Decisions found by taking into account maximum likelihoods and confidence interval of 2σ

Figure 7.24: Decisions made by the scheduler (indicated in red) (taken from [163])

This section has just given an example on how scheduling decisions are influenced by the standard deviation of runtime distributions. The result is, the larger the variance the more decisions will be found and the more unlikely it will be to find the best one. This raises the question which mechanism must be applied to still guarantee high job throughput when jobs finish earlier or later than estimated. A backfilling method can be applied that

respects such runtime distributions instead of relying only on a concrete runtime value for each job. This will be explained in the following.

Probabilistic Backfilling

Section 7.4 has focused on how to solve the objective function in order to maximize job throughput and minimize wasted CPU cycles due to gaps and non linear speedup. This is done in an offline way, before jobs are actually started. In an online scenario two situations might happen: a job finishes either too early or too late. Consequently, the whole schedule becomes invalid because other jobs must be postponed. So the question is how to avoid gaps and refill them when they occur. Many different backfilling techniques exist, which compare predicted runtime of tasks and choose a certain one. Instead of having a concrete value, estimation is subject to an underlying distribution of several possible runtime values. [144] shows, that backfilling can be improved by respecting such kind of distributions and they call this new method probabilistic backfilling. This method will be explained in detail in this section. The basic idea is to backfill gaps when the probability of postponing the next job is small. In the context of LHCb jobs runtime distributions can be easily obtained from prior tasks. If a job finishes too early two options are possible:

- Start all consecutive tasks earlier
- Backfill gaps with another task

The first case is the easiest option since the number of computed jobs within the schedule matters and not their start and end date. Problematic is the case when a job does not meet the deadline. It is not an option to interrupt it, since this will decrease the overall throughput. Consecutive tasks will be postponed and it must be evaluated whether they will still be able to meet their deadlines or not. The technique described in [144] can be applied, which determines the probability of a job being able to finish in a certain time window.

[144] proposes to apply a uniform distribution on job runtimes whose peak is the time indicated by the user. In the context of LHCb jobs the distribution of runtime is known due to previous workloads and can be fitted by a Gaussian distribution. As described in [144] a task will be started if its probability of not postponing the next job in the queue is smaller than τ :

$$\int_{t_0}^{\infty} Pr(t_e) Pr(\exists t \in (t_0, t_e) : c_q \leq c(t) < c) \mathrm{d}t_e < \tau, \tag{7.14}$$

where $Pr(t_e)$ corresponds to the probability of a task to finish at t_e and $Pr(\exists t \in (t_0, t_e) : c_q \leq c(t) < c)$ determines the probability that at any time c processors are available. c_q is the number of processors required by the consecutive job and c the number of processes to run both jobs at a time. $Pr(t_e)$ is given by the Gaussian distribution which can be derived from prior jobs and t_e corresponds to the unknown finishing time. The formula integrates over all possible finishing times of a task as explained in [144] and can be used in order to decide whether the backfilled job will not postpone the start date of the following task.

Calculating the probability that c processors are available at a certain point, can be achieved by Dynamic Programming as explained in [144]. The following formula computes the probability that tasks $1...n$ have released c processors:

$$M_t[n][c] = M_t[n-1][c] + (M_t[n-1][c - c_n] - M_t[n-1][c]) \cdot P_t[n]. \tag{7.15}$$

It is calculated recursively, where $M_t[n-1][c]$ is equal the probability that c processors are already available without the termination of task n and $M_t[n-1][c-c_n] - M_t[n-1][c]$

Testcases	8 Cores	15 Cores	30 Cores	50 Cores
Offline Computed Optimum	2.5	2.3	4.5	4.6
Approach 1	22.8	33.1	30.2	31.6
Approach 2	11.9	15.7	13.5	15.9

Table 7.7: Comparison of different approaches for handling wrong estimates

corresponds to the likelihood that c processors become available when task n finishes [144]. The equation can be simplified as the following sum:

$$\sum_{t_e} Pr(t_e) \max_{t \in (t_0, t_e)} \{M_t[n][c_q] - M_t[n][c]\} < \tau. \tag{7.16}$$

Since the scheduling problem is subject to discrete values, the integral becomes a sum. The formula is a rough approximation to equation 7.14 and it computes the bottle neck of processor availability in each step.

Evaluation of Probabilistic Backfilling

The moldable scheduler defines the task properties such that an optimal job throughput can be achieved with respect to the available cores. The schedule constitutes the start and end time of jobs. This has been evaluated based on 4 test cases that have been presented in section 7.4. They will be reused in order to compare the offline computed optima with the real loss in throughput. Different approaches for handling wrong estimated jobs can be applied.

Approach 1: If jobs do not meet the defined deadline, they will be interrupted such that the next job is not delayed. Since the scheduler has taken the maximum likelihoods of runtime distribution into account, a large amount of jobs is underestimated. However, jobs might also finish earlier than expected which allows to move other jobs ahead of time. They get an earlier start date, but it matters that they finish before the old deadline. In the predefined cases, this approach leads to a loss of throughput of up to 33% (Tab. 7.7).

Approach 2: Instead of interrupting jobs consecutive tasks will be simply postponed. Probabilistic Backfilling can be applied, in order to estimate whether the delayed job is still able to finish at its deadline or not. Since the maximum likelihood of distributions is taken into account, the probability of finishing in time should be small. However, jobs can move ahead of time such that delays might be compensated by jobs finishing earlier. This approach is less aggressive and significantly improves loss in throughput as shown in Tab. 7.7.

Since estimation of runtime is subject to an underlying distribution it is rather difficult to generate the best scheduling decision. The larger the standard deviation of each distribution the larger the amount of solutions. Each of them might be the global optimum with a certain probability. However, knowing the distribution can be also used to reach further optimisations as it has been shown in this section.

7.7 Summary

As shown throughout this chapter scheduling of multiprocessor tasks faces several problems. As explained LHCb jobs based on the multiprocessing approach are moldable. If their degree in parallelism is not limited, the overall throughput decreases. For 30 cores a loss of up to 37% has been measured. The chapter has defined the objective function for optimising job throughput.

It has been shown that a deterministic local search method can be combined with a probabilistic meta heuristic approach. This achieves quite good scheduling decisions which are only a few percent worse than solutions found by the commercial IBM ILOG CPLEX CP Optimizer. The solutions found by the deterministic local search are used as start and restart solutions for the probabilistic meta heuristic approach.

Since scheduling requires runtime and memory estimates, a supervised learning algorithm has been developed. It has been evaluated that taking experiment specific parameters into account can drastically improve estimation up to 22% for reconstruction jobs. Estimation of memory requirements cannot be much improved since virtual memory of jobs is currently stored in the database.

The chapter concluded with the impact of wrong runtime estimates on scheduling decisions. Currently, jobs which do not meet the predefined deadline are interrupted by the resource provider. Applying the same methodology leads to a loss of throughput of up to 33%, since maximum likelihoods are taking into account during the optimisation of the objective function. Probabilistic backfilling developed by [144] has been presented. When a job finishes too late, it can be evaluated how likely it is that the successive task can still finish in time. The same steps undertaken by probabilistic backfilling algorithm can be applied in this situation. In the predefined testcases, it has lead to only a loss in throughput of up to 15.9%.

8. Conclusion

This thesis concludes with a summary of the most important results and their impacts in section 8.1 and it will give an outlook on possible future developments in section 8.2. The main research question of this work was how multi- and manycore systems can be used more efficiently at the example of the LHCb experiment. It has been shown throughout the thesis that optimisation can be applied at many levels.

8.1 Summary

Impacts of Results presented in Chapter 5: Optimisation with Non Intrusive Techniques

Research question 1 (*"What limits the software and the utilisation of computing resources?"*) described that limitations of software must be known and new technologies can be applied to improve software in a non intrusive way. This has been evaluated in this chapter.

It has been shown that many diverse techniques can be applied to reduce the memory requirements of LHCb applications in a transparent way. It has been evaluated that tools for automatic memory deduplication can achieve similar results as parallelizing the software. Applying memory compression allows to reduce the footprint in any order and a reasonable runtime can be still achieved by compressing 30% of allocated memory. These results also indicate that there is still room to improve how LHCb software allocates and uses memory.

Within the scope of the work, x32-ABI has been evaluated for HEP SPEC benchmarks, the ROOT framework and the LHCb software framework. These tests were one of the first large benchmarks executed as x32 binaries. Until now, there are many scepticism within the Linux community, whether this new platform model can be beneficial or not and as a result certain Linux distributions still do not support it. However, the results of this work have shown that such scepticism are not justified. Results have been referenced by different representatives from the Linux community and also motivated other research groups to investigate it. With increasing number of supporters, x32-ABI may become available in many more Linux distributions. The thesis has shown that x32-ABI helps to gain in runtime and memory reduction at the same time. A memory reduction of up to 25% has been achieved. Such techniques can be used in a transparent way without modifying the software and the corresponding job model.

Impacts of Results presented in Chapter 6: Optimisation with Intrusive Techniques

Research question 1 and research question 2 (*"What are the performance impacts of multi- and manycore CPUs"*) have been evaluated in this chapter. The main goal was to improve the memory footprint of LHCb software, to coordinate concurrent accesses, to understand performance impacts of hardware features and to determine software bottlenecks.

A much lower memory ratio and a larger concurrency is foreseen for future manycore systems and consequently the execution of jobs via parallel tasks becomes necessary. The thesis has shown that software can be parallelized in different ways. The parallel prototype has been optimised such that for instance the memory footprint of a parallel reconstruction job can be reduced by 50% when executed with more than 8 worker processes. Evaluations have indicated that there is plenty of room for improvement within the single threaded version of the LHCb software framework. Results have illustrated that much more memory is addressed than actually required by the jobs. In general, this should not be problematic since a process shows the same performance independently of its address space. However, many grid sites incorrectly monitor and terminate jobs based on their virtual memory.

Consequently, more work must be undertaken to limit the size of buffers, which are additionally allocated by processes but mostly not used at all. Performance counter measurements have pointed out that LHCb applications do not use CPUs in an efficient way. An IPC value (instruction per cycle) lower than one is clearly inadequate. It has been evaluated how the parallel software framework prototype scales with the number of cores. Benchmark tests have been executed on the latest manycore systems and results from different systems have been compared. It is indispensable that the corresponding job model has to allow an efficient scheduling of multi- and manycore jobs. If this is not given, the job throughput can decrease drastically due to non linear scaling of the parallel prototype. A loss of up to 37% has been measured in the case a worker node provides 30 cores.

Impacts of Results presented in Chapter 7: Optimisation at the Level of Workload Scheduling

This chapter focused on research question 3 (*"How to apply multicore job submission within the Worldwide LHC Computing Grid?"*). As stated in this research question, going from single- to multicore jobs is not an easy transition, since scheduling is subject to multiple VOs. The goal was to define a scheduler that can be part of VO's Workload Management System and that can take information about prior jobs into account.

It is still an unsolved issue, whether scheduling of multicore jobs is a grid site or VO related problem. Grid sites want to avoid resource partitioning. As a result, mechanisms must be applied that allow to run multi- and singlecore jobs on the same resources. As discussed, the main issue is that requirements indicated by the experiments are not reliable and this prevents the grid sites to increase their utilisation via backfilling.

This thesis has proposed a moldable job model, where scheduling is subject to the experiment. The aim is to better estimate workloads and to have a scheduler which defines the appropriate degree of parallelism for each job. The major advantage is that experiments can control their pilot jobs and have knowledge about requirements of different job types. Diverse optimisation algorithms have been implemented and it has been evaluated that with less than a thousand of iterations good schedules can be found which are only a few percent worse than the global optimum. In contrast to the ATLAS experiment [86], this model proposes that scheduling is done by the experiment's Workload Management System. The grid site needs only to provide an arbitrary multicore job slot. The CMS experiment also proposes an internal scheduling [119], however they do not take software

characteristics like memory requirements and scaling factors into account. These parameters are important for the proposed moldable job model.

Since the scheduler must estimate runtime and memory requirements of jobs, it has been investigated which input features of a job are correlated with them. The thesis has undergone a detailed workload analysis. A supervised learning algorithm has been implemented, which takes certain physics criteria like average luminosity, multiplicity and job specific information like file and event size into account. Tests have shown that runtime of reconstruction jobs can be improved by 22% compared to a maximum likelihood estimation. Currently, the bad runtime prediction prevents grid sites from providing multicore job slots without degrading utilisation. The thesis has proven that bad runtime estimates can be avoided. In contrast to other LHC experiments like CMS or ATLAS, the LHCb experiment supports luminosity levelling. The workload analysis, presented in section 7.5.2, has shown that due to that no tails in effects in runtime can be observed like in CMS [87].

8.2 Outlook

Future developments have to focus on how software can better profit from hardware features, like vector registers and hyperthreading. Therefore, data layouts must be changed in order to allow SIMD (single instruction multiple data) operations. Analysing cache access behaviour is another opportunity for optimisation, since this allows to improve data locality and consequently faster access to data. Work might also focus on how performance bottlenecks can be analysed in a more automatic way. The complexity of future manycore systems is rapidly growing and as shown in this thesis, software scales differently on various systems. In order to understand differences, metrics for automatic performance monitoring must be defined and evaluated. Since LHCb is running 10k jobs each day, many statistics from different CPU types can be obtained and analysed. The parallel prototype of the LHCb applications can be further improved by applying parallel writers. This avoids that the compression of files becomes a bottleneck. In addition, a combination of the multithreaded and multiprocessing approach can help to improve the scalability of software. It is foreseen to use event level parallelism, where each task is processing different events and within each task multiple threads execute algorithms in parallel.

Optimisation is a crucial task, since modifying one piece always impacts other parts. Many possibilities have been illustrated throughout this thesis and optimisation has to be applied on the whole workflow. The common idiom *'A chain is only as strong as its weakest link'* describes this very well. Assuming the case, that LHCb software would scale linearly with the number of cores, but no mechanisms are applied to better estimate job runtimes, then job throughput would be still limited by jobs failing due to underestimated runtimes. Future workloads will be much larger and it is therefore quite important, to consider the workflow as a whole.

Bibliography

[1] The KTEV Experiment . ktev.fnal.gov/public/, 1999.

[2] Welcome to the CP Violation Experiment. http://na48.web.cern.ch/NA48/Welcome.html, 2000.

[3] Grid Data Management Reliable File Transfer Services Performance. http://www.gridpp.ac.uk/papers/chep06_stewart.pdf, 2006.

[4] ELC: How much memory are applications really using? http://lwn.net/Articles/230975/, 2007.

[5] Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors . http://developer.amd.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf, 2008.

[6] LHCb File Summary Record. https://twiki.cern.ch/twiki/bin/view/LHCb/FileSummaryRecord, 2008.

[7] IBM ILOG CP Optimizer V2.3 - User's Manual. http://lyle.smu.edu/emis/docs/CP%20Optimizer/usrcpoptimizer.pdf, 2009.

[8] compcache. http://code.google.com/p/compcache/wiki/zramperf, 2011.

[9] HEP-SPEC06 Benchmark. http://w3.hepix.org/benchmarks/doku.php/, 2011.

[10] LUMI LEVELING: What, Why and How? http://www.quantumdiaries.org/2011/04/24/lumi-leveling-what-why-and-how/, 2011.

[11] POOL. https://twiki.cern.ch/twiki/bin/view/LHCb/PersistencyMigration, 2011.

[12] Power 4 The First Multi-Core, 1GHz Processor. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/, 2012.

[13] BaBar Public Information & Visitor Pages. http://www.slac.stanford.edu/BF/, 2012.

[14] European Grid Infrastructure (EGI) - Glossary, 2012. http://www.egi.eu/about/glossary/glossary_H.html.

[15] GCC, the GNU Compiler Collection, 2012. http://gcc.gnu.org/.

[16] GNU Binutils, 2012. http://www.gnu.org/software/binutils/.

[17] Jobs requiring whole nodes or multiple cores . https://twiki.cern.ch/twiki/bin/view/LCG/WMTEGMulticore, 2012.

[18] Linux @ CERN, 2012. http://linux.web.cern.ch/linux/scientific6/.

[19] Status and future plans for software technology demonstrators in CMS. http://indico.cern.ch/event/209503/contribution/5/material/slides/0.pdf, 2012.

[20] The GNU C Library (glibc), 2012. http://www.gnu.org/software/libc/.

[21] WLCG Worldwide LHC Computing Grid. http://wlcg.web.cern.ch/, 2012.

[22] LHCb's first steps in volunteer computing. https://indico.cern.ch/event/272795/session/1/contribution/8/material/slides/0.pdf, 2013.

[23] Concurrent Data Processing Frameworks. https://indico.cern.ch/event/279723/session/7/contribution/13/material/slides/0.pdf, 2013.

[24] DIRAC overview. http://diracgrid.org/files/docs/Overview/index.html?highlight=pilot, 2013.

[25] Integration of Cloud resources in the LHCb Distributed Computing . https://indico.cern.ch/event/214784/session/4/contribution/31, 2013.

[26] Intel Performance Counter Monitor - A better way to measure CPU utilization. www.intel.com/software/pcm, 2013.

[27] Multi-Core R&D Project . https://twiki.cern.ch/twiki/bin/view/LCG/MultiCoreRD, 2013.

[28] Plans to Improve the ROOT I/O Reading. https://indico.cern.ch/event/256183/session/0/contribution/3/material/slides/0.pdf, 2013. 2nd LHCb Computing workshop.

[29] Welcome to the Globus Toolkit Homepage. http://toolkit.globus.org/toolkit/, 2013.

[30] About KEK. http://legacy.kek.jp/intra-e/about/, 2014.

[31] CERN Cloud Infrastructure User Guide. http://information-technology.web.cern.ch/book/cern-private-cloud-user-guide, 2014.

[32] CERN Uses OpenStack. http://www.openstack.org/user-stories/cern/, 2014.

[33] Cgroups. https://wiki.archlinux.org/index.php/cgroups, 2014.

[34] Computing with HTCondor. http://research.cs.wisc.edu/htcondor/, 2014.

[35] Documentation. http://root.cern.ch/drupal/content/documentation, 2014.

[36] Forum on Concurrent Programming Models and Frameworks. http://concurrency.web.cern.ch/, 2014.

[37] Intel Threading Building Blocks (Intel TBB). https://software.intel.com/en-us/intel-tbb, 2014.

[38] Intel Turbo Boost Technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html, 2014.

[39] Intel Xeon Processor E7-4870 . http://ark.intel.com/products/53579, 2014.

[40] Intel Xeon Processor L5520 . http://ark.intel.com/de/products/40201/Intel-Xeon-Processor-L5520-8M-Cache-2_26-GHz-5_86-GTs-Intel-QPI, 2014.

[41] International Linear Collider. http://www.linearcollider.org/ILC, 2014.

[42] Large Hadron Collider Beauty Experiment. http://lhcb-public.web.cern.ch/lhcb-public/, 2014.

[43] LHCb Accounting. http://lhcbweb.pic.es/DIRAC/LHCb-Production/visitor/systems/accountingPlots/job, 2014.

[44] Multicore Task Force. `https://indico.cern.ch/event/296031/material/slides/0?contribId=0`, 2014.

[45] RunDB: Run 103127. `https://lbrundb.cern.ch/rundb/run/103127/`, 2014.

[46] The Grid: Software, middleware, hardware. `http://home.web.cern.ch/about/computing/grid-software-middleware-hardware`, 2014.

[47] The libunwind project. `http://www.nongnu.org/libunwind/`, 2014.

[48] The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`, 2014.

[49] The OpenMP API specification for parallel programming. `http://openmp.org/wp/`, 2014.

[50] UNICORE. `http://www.unicore.eu/`, 2014.

[51] K. Aamodt et al. The ALICE experiment at the CERN LHC. *JINST*, 3:S08002, 2008.

[52] J Albrecht, V V Gligorov, G Raven, and S Tolk. Performance of the LHCb High Level Trigger in 2012. Technical Report arXiv:1310.8544, Oct 2013. Comments: Proceedings for the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP).

[53] R Alemany-Fernandez, F Follin, and R Jacobsson. The LHCB Online Luminosity Control and Monitoring. Technical Report CERN-ACC-2013-0028, CERN, Geneva, May 2013.

[54] Federico Alessio. The LHCb Upgrade. `http://arxiv.org/abs/1310.0183`, October 2013.

[55] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[56] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[57] Andrew Ng. CS229 Lecture notes. `http://cs229.stanford.edu/notes/cs229-notes1.pdf`, 2012.

[58] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium*, pages 19–28, July 2009.

[59] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, et al. GAUDI - A software architecture and framework for building HEP data processing applications. *Comput.Phys.Commun.*, 140:45–55, 2001.

[60] J P Baud, Ph Charpentier, K Ciba, R Graciani, E Lanciotti, Z Mathe, D Remenska, and R Santana. The LHCb Data Management System. *Journal of Physics: Conference Series*, 396(3):032023, 2012.

[61] Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 10.1007/BF00289509.

[62] Mario Úbeda García, Víctor Méndez Muñoz, Federico Stagni, Baptiste Cabarrou, Nathalie Rauschmayr, Philippe Charpentier, and Joel Closier. Integration of Cloud resources in the LHCb Distributed Computing. *Journal of Physics: Conference Series*, 513(3):032099, 2014.

[63] Susmit Biswas, Diana Franklin, Alan Savage, Ryan Dixon, Timothy Sherwood, and Frederic T. Chong. Multi-execution: multicore caching for data-similar executions. *SIGARCH Comput. Archit. News*, 37(3):164–173, June 2009.

[64] Claudio Grandi (INFN Bologna). Clouds in WLCG, 2013. `https://agenda.infn.it/getFile.py/access?contribId=3&sessionId=0&resId=0&materialId=slides&confId=5900`.

[65] Pradip Bose. Power wall. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1593–1608. Springer, 2011.

[66] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2008.

[67] N. Brook, A. Bogdanchikov, A. Buckley, Joel Closier, Ulrik Egede, M. Frank, Domenico Galli, Miriam Gandelman, Vincent Garonne, Clara Gaspar, Ricardo Graciani Diaz, K. Harrison, Eric van Herwijnen, A. Khan, Sander Klous, Ivan Korolko, Genady Kuznetsov, Francoise Loverre, Umberto Marconi, Juan P. Palacios, Glen N. Patrick, Andrew Pickford, Sebastien Ponce, Vladimir Romanovski, Juan J. Saborido, Michael Schmelling, A. Soroko, Andrei Tsaregorodtsev, Vincenzo Vagnoni, and Andrew Washbrook. Dirac - distributed infrastructure with remote agent control. *CoRR*, cs.DC/0306060, 2003.

[68] R. Buyya, J. Broberg, and A.M. Goscinski. *Cloud Computing: Principles and Paradigms*. Wiley Series on Parallel and Distributed Computing. Wiley, 2010.

[69] P. Canal. ROOT I/O Improvements. *Journal of Physics: Conference Series*, 396(5):052017, 2012.

[70] Adrian Casajús, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6):062049, 2010.

[71] Adrian Casajús, Ricardo Graciani, Albert Puig, Ricardo Vázquez, and the LHCb Collaboration. The lhcb experience on the grid from the dirac accounting data. *Journal of Physics: Conference Series*, 331(7):072059, 2011.

[72] M Cattaneo. LHCb Computing Resources: 2013 re-assessment, 2014 request and 2015 forecast. Technical Report LHCb-PUB-2013-002, CERN, Geneva, Feb 2013. On behalf of the LHCb Computing Project.

[73] Marco Cattaneo. Gaudi Users Guide. `http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/GUG/GUG.pdf`, 2001.

[74] Marco Cattaneo. LHCb Computing Resources: 2014 re-assessment and 2015 request. Technical Report LHCb-PUB-2013-014. CERN-LHCb-PUB-2013-014, CERN, Geneva, Sep 2013.

[75] CERN. The Worldwide LHC Computing Grid. `http://home.web.cern.ch/about/computing/worldwide-lhc-computing-grid`, 2013.

[76] S. Chatrchyan et al. The CMS experiment at the CERN LHC. *JINST*, 3:S08004, 2008.

[77] B. Chazelle. The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation. *IEEE Trans. Comput.*, 32(8):697–707, August 1983.

[78] Su-Hui Chiang, Andrea Arpaci-Dusseau, and MaryK. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In DrorG. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 103–127. Springer Berlin Heidelberg, 2002.

[79] J. H. Christenson, J. W. Cronin, V. L. Fitch, and R. Turlay. Evidence for the 2π decay of the K_2^0 Meson. *Phys. Rev. Lett.*, 13:138–140, Jul 1964.

[80] W. Chun. *Core Python Programming*. Pearson Education, 2006.

[81] Walfredo Cirne and Francine Berman. Adaptive selection of partition size for supercomputer requests. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pages 187–207. Springer Berlin Heidelberg, 2000.

[82] Walfredo Cirne and Francine Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571 – 1601, 2002.

[83] ATLAS collaboration. The atlas experiment at the cern large hadron collider. *Journal of Instrumentation*, 3:S08003, August 2008.

[84] G. Corti, M. Cattaneo, P. Charpentier, M. Frank, P. Koppenburg, P. Mato, F. Ranjard, S. Roiser, I. Belyaev, and G. Barrand. Software for the LHCb experiment. *Nuclear Science, IEEE Transactions on*, 53(3):1323 – 1328, june 2006.

[85] G. Cortia. Overview of Monte Carlo simulation(s) in LHCb. `http://lhcb-comp.web.cern.ch/lhcb-comp/Simulation/Tutorial/01.GCorti-MCinLHCb-20091013.pdf`, 2009.

[86] D Crooks, P Calafiura, R Harrington, M Jha, T Maeno, S Purdie, H Severini, S Skipsey, V Tsulaia, R Walker, and A Washbrook. Multi-core job submission and grid resource scheduling for atlas athenamp. *Journal of Physics: Conference Series*, 396(3):032115, 2012.

[87] Samir Cury Siqueira. Event processing time prediction at the CMS Experiment of the Large Hadron Collider. Technical Report CMS-CR-2013-375. CERN-CMS-CR-2013-375, CERN, Geneva, Oct 2013.

[88] Allen B. Downey. A Model For Speedup of Parallel Programs. Technical report, Berkeley, CA, USA, 1997.

[89] Allen B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.

[90] Forti Alessandra Dr. Perez-Calero Yzquierdo, Antonio. Multicore job management in the Worldwide LHC Computing Grid. `https://indico.egi.eu/indico/contributionDisplay.py?contribId=70&sessionId=45&confId=1994`, 2014. Presented at EGI Community Forum 2014, Helsinki, Finland.

[91] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, 1989.

[92] R.M. Eijk. *Track Reconstruction in the LHCb Experiment*. Universiteit van Amsterdam [Host], 2002.

[93] The LHCb Collaboration et al. The LHCb Detector at the LHC. *Journal of Instrumentation*, 3(08):S08005, 2008.

[94] Lyndon Evans and Philip Bryant. LHC Machine. *Journal of Instrumentation*, 3(08):S08001, 2008.

[95] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *In Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996.

[96] DrorG. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - a status report. In DrorG. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2005.

[97] S Fleischmann, S Kama, R Vitillo, W Lavrijsen, and M Neumann. Experience with Intel's Many Integrated Core Architecture in ATLAS Software. Technical report, ATL-COM-SOFT-2013-120, 2013.

[98] B. Hegner for the Concurrent Gaudi Team. Running Concurrent Gaudi in Real Life: Status Update on MiniBrunel, 2013. `https://concurrency.web.cern.ch/sites/concurrency.web.cern.ch/files/ATLAS_SW_Week_MiniBrunel.pdf`.

[99] Ian Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[100] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001.

[101] Python Software Foundation. Python/c api reference manual. `http://docs.python.org/2/c-api/init.html#threads`, 2014.

[102] M Frank, C Gaspar, E v Herwijnen, B Jost, N Neufeld, and R Schwemmer. Optimization of the HLT Resource Consumption in the LHCb Experiment. *Journal of Physics: Conference Series*, 396(1):012021, 2012.

[103] R. Fruehwirth. Application of kalman filtering to track and vertex fitting. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 262(2-3):444 – 450, 1987.

[104] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, January 1979.

[105] Donvito Giacinto. Testing SLURM batch system for a grid farm: functionalities, scalability, performance and how it works with Cream-CE . `https://indico.egi.eu/indico/contributionDisplay.py?contribId=37&sessionId=46&confId=1019`, 2011. Presented at EGI Technical Forum 2012.

[106] Richard Gibbons. A historical application profiler for use by parallel schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 58–77. Springer Berlin Heidelberg, 1997.

[107] Milind Girkar, H. Peter Anvin, Hongjiu Lu, Dmitry V. Shkurko, and Vyacheslav Zakharin. *Intel Technology Journal*, Volume 16, 2012. The x32 ABI: A New Software Convention for Performance on Intel 64 Processors.

[108] V.V. Gligorov. Performance and upgrade plans of the lhcb trigger system. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 718(0):26 – 29, 2013. Proceedings of the 12th Pisa Meeting on Advanced Detectors La Biodola, Isola d'Elba, Italy, May 20 - 26, 2012.

[109] F. Gorunescu. *Data Mining: Concepts, Models and Techniques*. Intelligent Systems Reference Library. Springer Berlin Heidelberg, 2011.

[110] R Graciani Diaz. LHCb Computing Resource Usage in 2012(I). Technical Report LHCb-PUB-2012-013, CERN, Geneva, Sep 2012. On behalf of the LHCb computing project.

[111] Ricardo Graciani Diaz. LHCb Computing Resource usage in 2011. Technical Report LHCb-PUB-2012-003, CERN, Geneva, Feb 2012.

[112] A. Grama. *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003.

[113] C. Grosan and A. Abraham. *Intelligent Systems: A Modern Approach*. Intelligent Systems Reference Library. Springer Berlin Heidelberg, 2011.

[114] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[115] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[116] T Hauth, V Innocente, and D Piparo. Development and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework. *Journal of Physics: Conference Series*, 396(5):052065, 2012.

[117] B. Hegner, P. Mato, and D. Piparo. Evolving lhc data processing frameworks for efficient exploitation of new cpu architectures. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE*, pages 2003–2007, 2012.

[118] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[119] J M Hernández, D Evans, and S Foulkes. Multi-core processing and scheduling performance in cms. *Journal of Physics: Conference Series*, 396(3):032055, 2012.

[120] Ronald Hochreiter, Clemens Wiesinger, and David Wozabal. Large-scale computational finance applications on the open grid service environment. In PeterM.A. Sloot, AlfonsG. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 891–899. Springer Berlin Heidelberg, 2005.

[121] Engin Ipek, Bronis R. Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In Jose C. Cunha and PedroD. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205. Springer Berlin Heidelberg, 2005.

[122] Sverre Jarp, Alfio Lazzaro, Julien Leduc, and Andrzej Nowak. How to harness the performance potential of current multi-core processors. *Journal of Physics: Conference Series*, 331(1):012003, 2011.

[123] C D Jones, P Elmer, L Sexton-Kennedy, C Green, and A Baldooci. Multi-core aware applications in cms. *Journal of Physics: Conference Series*, 331(4):042012, 2011.

[124] Michel Jouvin. Batch Systems Review. `http://indico.cern.ch/event/274555/session/14/contribution/15/material/slides/0.pdf`, May 2014. Presented at HEPiX Spring 2014, Annecy-le-Vieux, France.

[125] Michael Kerrisk. Linux Programmer's Manual, 2012. `http://man7.org/linux/man-pages/man2/madvise.2.html`.

[126] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2012.

[127] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. Distrm: distributed resource management for on-chip many-core systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on*

Hardware/software codesign and system synthesis, CODES+ISSS '11, pages 119–128, New York, NY, USA, 2011. ACM.

[128] S. Krishnaprasad. Uses and abuses of Amdahl's law. *J. Comput. Sci. Coll.*, 17(2):288–293, December 2001.

[129] Alfio Lazzaro, Sverre Jarp, Andrzej Nowak, and Liviu Valsan. Comparison of Software Technologies for Vectorization and Parallelization. Technical Report CERN-OPEN-2014-029, CERN, Geneva, Sep 2012.

[130] Byoung-Dai Lee and Jennifer M. Schopf. Run-time prediction of parallel applications on shared environments. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 487–491, 2003.

[131] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, Inc., Boca Raton, FL, USA, 2004.

[132] Scott N. Levy, Kurt Brian Ferreira, David Fiala, Patrick G. Bridges, and Dorian Arnold. *Exploiting Content Similarity to Improve Memory Performance in Exascale Systems.* Jul 2012.

[133] F. Magoules. *Fundamentals of Grid Computing: Theory, Algorithms and Technologies.* Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series. CRC Press, 2009.

[134] Clemencic Marco. Summary of Annual Concurrency Forum Meeting. `https://indico.cern.ch/event/228370/contribution/3/material/slides/0.pdf`, February 2013. Presented at Annual Concurreny Forum, Chicago, United States.

[135] Zoltan Mathe. *Feicim: A browser and analysis tool for distributed data in particle physics.* PhD thesis, U. Coll., Dublin, May 2012. Presented 01 Jun 2012.

[136] P. Mato. Adding Concurrency to LHC Software . `http://indico.cern.ch/getFile.py/access?contribId=6&resId=0&materialId=slides&confId=184092`, 2012. Presented in the Workshop: Many-core architectures for LHCb.

[137] Pere Mato and Eoin Smith. User-friendly parallelization of GAUDI applications with Python. *Journal of Physics: Conference Series*, 219(4):042015, 2010.

[138] A Mazurov and B Couturier. Advanced modular software performance monitoring. *J. Phys.: Conf. Ser.*, 396:052054. 13 p, 2012.

[139] M.D. McCool, J. Reinders, and A.D. Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann. Elsevier/Morgan Kaufmann, 2012.

[140] M. Milano. *Constraint and Integer Programming: Toward a Unified Methodology.* Operations Research/Computer Science Interfaces Series. Springer US, 2004.

[141] Tom M. Mitchell. The Need for Biases in Learning Generalizations. Technical report, Rutgers University, New Brunswick, NJ, 1980.

[142] R. Moona. *Assembly Language Programming in GNU/Linus For IA32 Architectures.* PHI Learning, 2009.

[143] A.W. Mu'alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, Jun 2001.

[144] Avi Nissimov and Dror G. Feitelson. Probabilistic backfilling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *JSSPP*, volume 4942 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2007.

[145] E.A. Papadelis. *Characterisation and Commissioning of the LHCb VELO Detector*. 2009.

[146] S K Paterson and A Maier. Distributed data analysis in LHCb. *Journal of Physics: Conference Series*, 119(7):072026, 2008.

[147] Thomas Sven Pettersson and P Lefêvre. The Large Hadron Collider: conceptual design. Technical Report CERN-AC-95-05 LHC, CERN, Geneva, Oct 1995.

[148] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, pages 140–, Washington, DC, USA, 1998. IEEE Computer Society.

[149] V. Ramesh. *Principles of Operating Systems*. Laxmi Publications Pvt Limited, 2010.

[150] B.R. Rau and J.A. Fisher. *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*. Knowledge Representation, Learning, and Expert Systems. Springer US, 1993.

[151] N. Rauschmayr and A. Streit. Evaluation of x32-ABI in the Context of LHC Applications. *Procedia Computer Science*, 18(0):2233 – 2240, 2013. 2013 International Conference on Computational Science.

[152] N Rauschmayr and A Streit. Preparing the Gaudi framework and the DIRAC WMS for multicore job submission. *Journal of Physics: Conference Series*, 513(5):052029, 2014.

[153] Nathalie Rauschmayr. Evaluation of x32-ABI in the context of CERN applications. `http://indico.cern.ch/event/214319`, 2012. Presented at Concurrency Forum, Geneva, Switzerland.

[154] Nathalie Rauschmayr. GaudiMP - performance and KSM measurements. `https://indico.cern.ch/event/198122`, 2012. Presented at Concurrency Forum, Geneva, Switzerland.

[155] Nathalie Rauschmayr. Optimisation of Gaudi applications for multi- and many-core CPUs. `https://indico.cern.ch/event/159521/`, October 2012. Presented at 49th Analysis and Software Week, Geneva, Switzerland.

[156] Nathalie Rauschmayr. Optimizing memory consumption within GaudiMP. `https://indico.fnal.gov/contributionDisplay.py?contribId=0&sessionId=7&confId=6138`, February 2013. Presented at Annual Concurreny Forum, Chicago, United States.

[157] Nathalie Rauschmayr. Optimizing memory consumption within GaudiMP. `https://indico.cern.ch/event/201535/`, February 2013. Presented at 50th Analysis and Software Week, Geneva, Switzerland.

[158] Nathalie Rauschmayr. ROOT and x32-ABI. `https://indico.cern.ch/event/217511/contribution/26`, March 2013. Presented at ROOT Users Workshop, Saas-Fee, Switzerland.

[159] Nathalie Rauschmayr. Status of CPU concurrency R&D. `https://indico.cern.ch/event/236650/`, May 2013. Presented at LHCb Computing Workshop, Geneva, Switzerland.

[160] Nathalie Rauschmayr. Multicore Job Submission. http://www.tu-chemnitz.de/gleichstellung/isina/, April 2014. Presented at ISINA Symposium, Chemnitz, Germany.

[161] Nathalie Rauschmayr. Optimization of LHCb Applications for Multi- and Manycore Job Submission. https://indico.cern.ch/category/2254/, June 2014. Presented at CERN-IT-SDC White Area Meeting, Geneva, Switzerland.

[162] Nathalie Rauschmayr. Scheduling of Multicore Jobs. May 2014. Presented at HEPiX Spring 2014, Annecy-le-Vieux, France.

[163] Nathalie Rauschmayr and Achim Streit. Evaluating Moldability of LHCb Jobs for Multicore Job Submission. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 519–525, Sept 2013.

[164] Nathalie Rauschmayr and Achim Streit. Reducing the Memory Footprint of Parallel Applications with KSM. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 48–59. Springer Berlin Heidelberg, 2013.

[165] K.A. Robbins and S. Robbins. *UNIX Systems Programming: Communication, Concurrency, and Threads.* Prentice Hall PTR, 2003.

[166] Computing Resources Review Board (C RRB). Status of resources and financial plan. http://indico.cern.ch/getFile.py/access?contribId=26&sessionId=3&resId=0&materialId=paper&confId=128046, 2011.

[167] R.A. Rutenbar. Simulated annealing algorithms: an overview. *Circuits and Devices Magazine, IEEE*, 5(1):19–26, 1989.

[168] Gerald Sabin, Matthew Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: an iterative approach. In *Proceedings of the 12th international conference on Job scheduling strategies for parallel processing*, JSSPP'06, pages 94–114, Berlin, Heidelberg, 2007. Springer-Verlag.

[169] Erik Saule, Doruk Bozdag, and Umit V. Catalyurek. A moldable online scheduling algorithm and its application to parallel short sequence mapping. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 6253 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2010.

[170] R. R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, June 1997.

[171] Olivier Schneider. Overview of the LHCb experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 446:213 – 221, 2000.

[172] N.F. Schneidewind. *Computer, Network, Software, and Hardware Engineering with Applications.* Wiley, 2012.

[173] A. Sen and S. Srivastava. *Regression Analysis: Theory, Methods, and Applications.* Lecture Notes in Statistics. Springer, 1990.

[174] Igor Sfiligoi. Estimating job runtime for CMS analysis jobs. Technical Report CMS-CR-2013-340. CERN-CMS-CR-2013-340, CERN, Geneva, Oct 2013.

[175] J. Shalf, K. Asanovic, D. Patterson, K. Keutzer, T. Mattson, and Yelick K. The MANYCORE Revolution: Will HPC lead or follow? *SciDAC Review*, 2009.

[176] D.J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures: Third Edition.* CRC Press, 2003.

[177] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The easy - loadleveler api project. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer, 1996.

[178] E. Smith. Gaudi & parallel gaudi. `http://indico.cern.ch/getFile.py/access?resId=1&materialId=slides&contribId=56&sessionId=7&subContId=4&confId=67561`, 2009.

[179] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, pages 122–142, London, UK, UK, 1998. Springer-Verlag.

[180] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE, 2002.

[181] Rick Stevens, Paul Woodward, Tom DeFanti, and Charlie Catlett. From the i-way to the national technology grid. *Commun. ACM*, 40(11):50–60, November 1997.

[182] Graeme Stewart. HL-LHC: Software Prospects. Presented at ECFA High Luminosity LHC Experiments Workshop.

[183] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *Journal of Parallel and Distributed Computing*, 73(7):926 – 938, 2013. Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012.

[184] D. Tsafrir, Y. Etsion, and D.G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):789–803, June 2007.

[185] A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, C. Cioffi, J. Closier, R. G. Diaz, G. Kuznetsov, Y. Y. Li, R. Nandakumar, S. Paterson, R. Santinelli, A. C. Smith, M. S. Miguelez, and S. G. Jimenez. DIRAC: a community grid solution. *Journal of Physics: Conference Series*, 119(6):062048, 2008.

[186] A.B. Tucker. *Computer Science Handbook, Second Edition.* CRC Press, 2004.

[187] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, SPAA '92, pages 323–332, New York, NY, USA, 1992. ACM.

[188] A. Vajda. *Programming Many-Core Chips.* Springer, 2011.

[189] Sander van Vugt. *Red Hat Enterprise Linux 6 Administration: Real World Skills for Red Hat Administrators.* SYBEX Inc., Alameda, CA, USA, 1st edition, 2013.

[190] Innocente Vincenzo. HEP physics software applications on many-core: present and perspectives. `https://agenda.infn.it/getFile.py/access?contribId=17&sessionId=2&resId=0&materialId=slides&confId=3868`, July 2011. Presented at SuperB Workshop, Oxford, UK.

[191] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[192] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability & statistics for engineers and scientists*. Pearson Education, Upper Saddle River, 8th edition, 2007.

[193] A. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, IPPS '98, pages 542–, Washington, DC, USA, 1998. IEEE Computer Society.

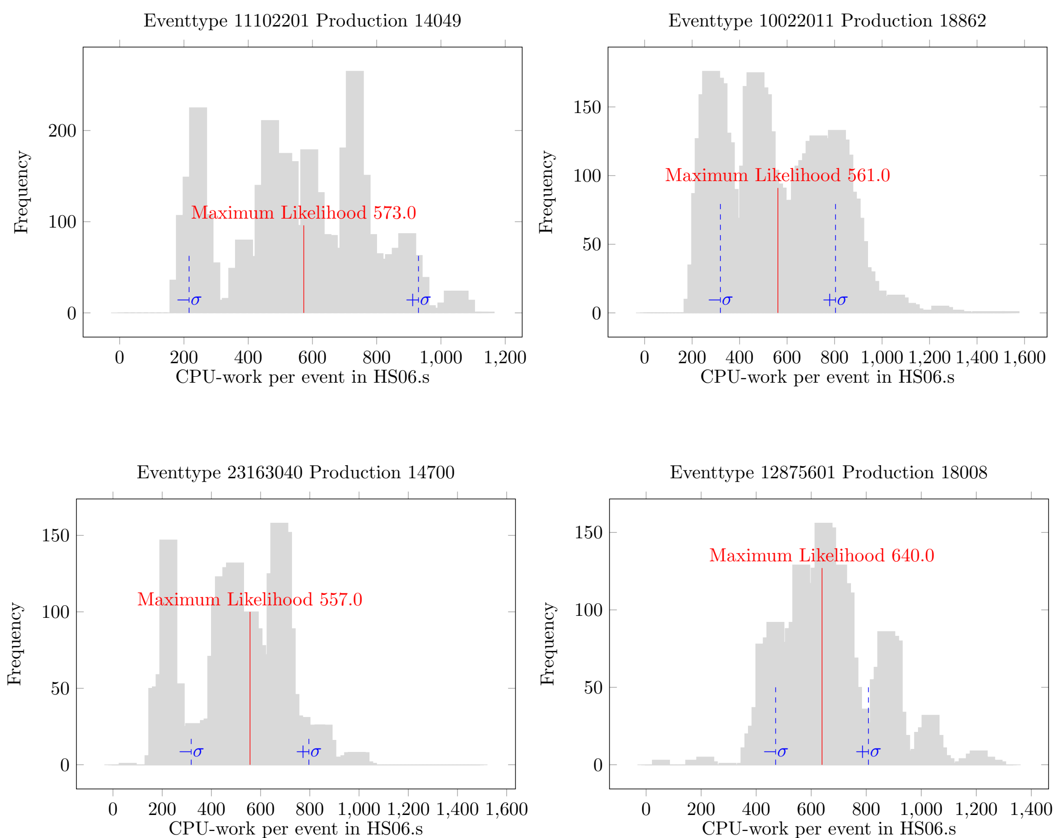
[194] Dong Ye, Joydeep Ray, Christophe Harle, and David R. Kaeli. Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture. In *IISWC*, pages 120–127. IEEE, 2006.

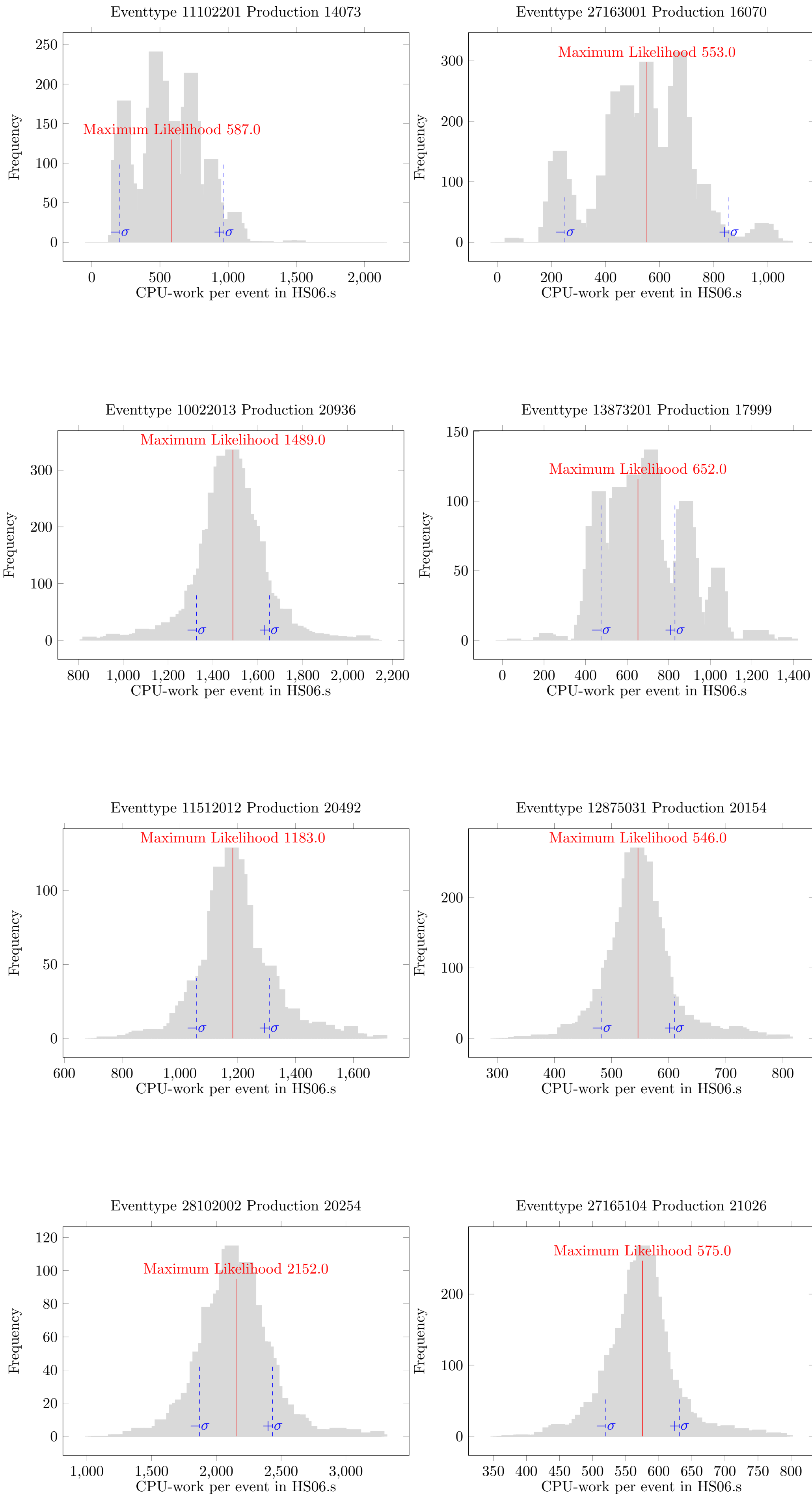
[195] O.C. Zienkiewicz, R.L. Taylor, and P. Nithiarasu. *The Finite Element Method for Fluid Dynamics*. Elsevier Science, 2013.

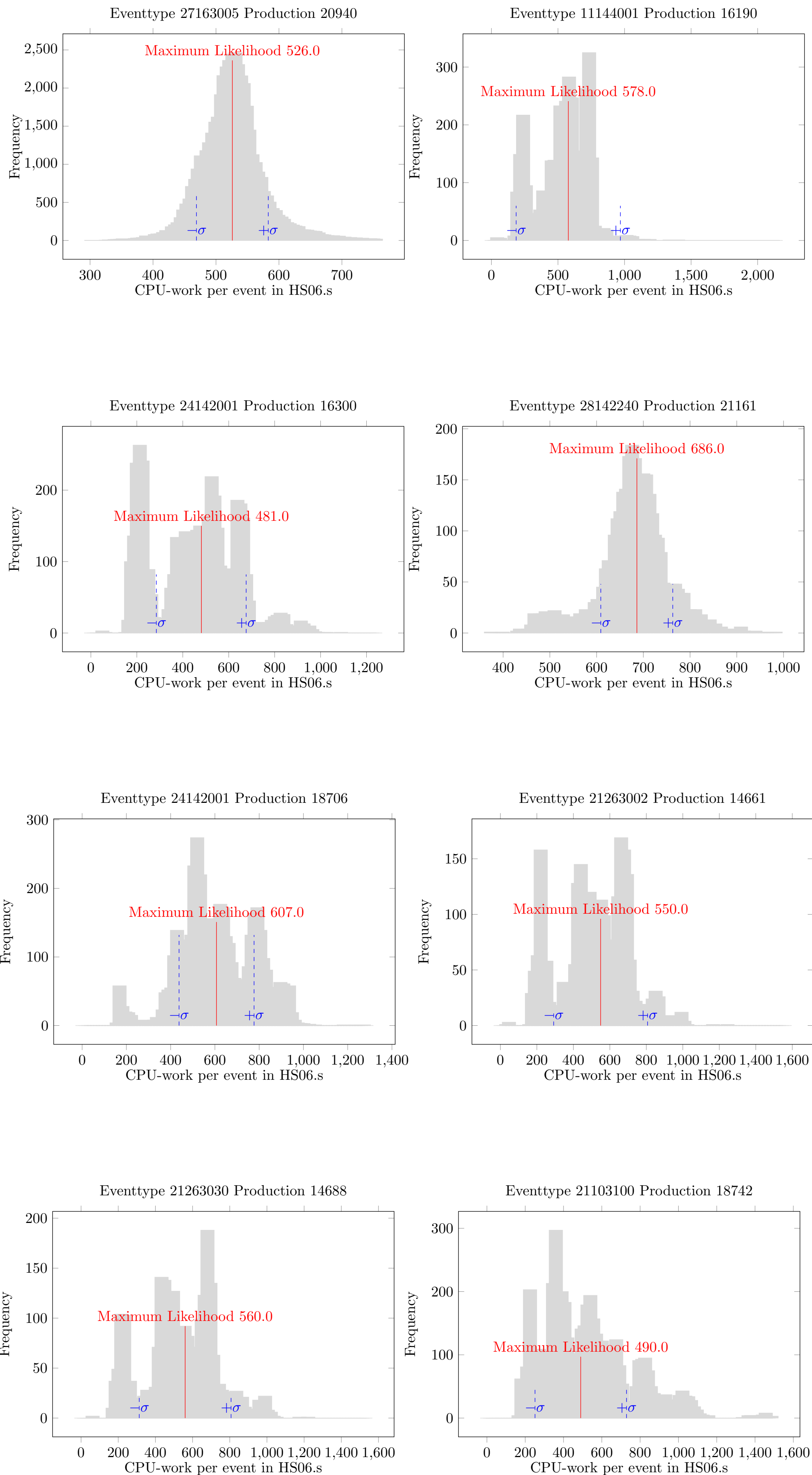
9. Appendix

9.1 Simulation Workloads MC11

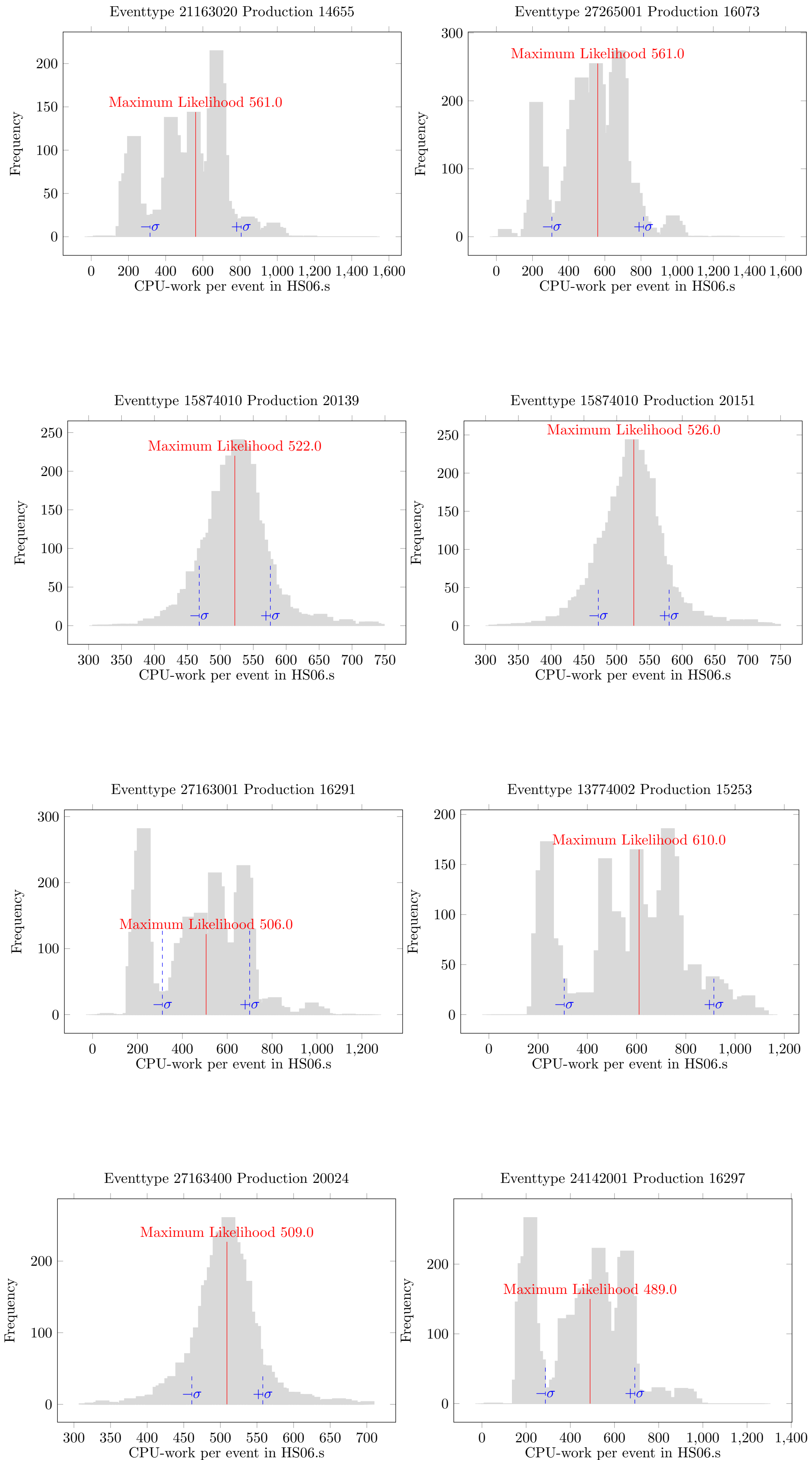
The following plots show the workload generated by simulation jobs from the year 2011 (MC11). Productions have been selected, which contained at least more than 2000 jobs. It is obvious that the generated workload does not necessarily fit a Gaussian distribution.



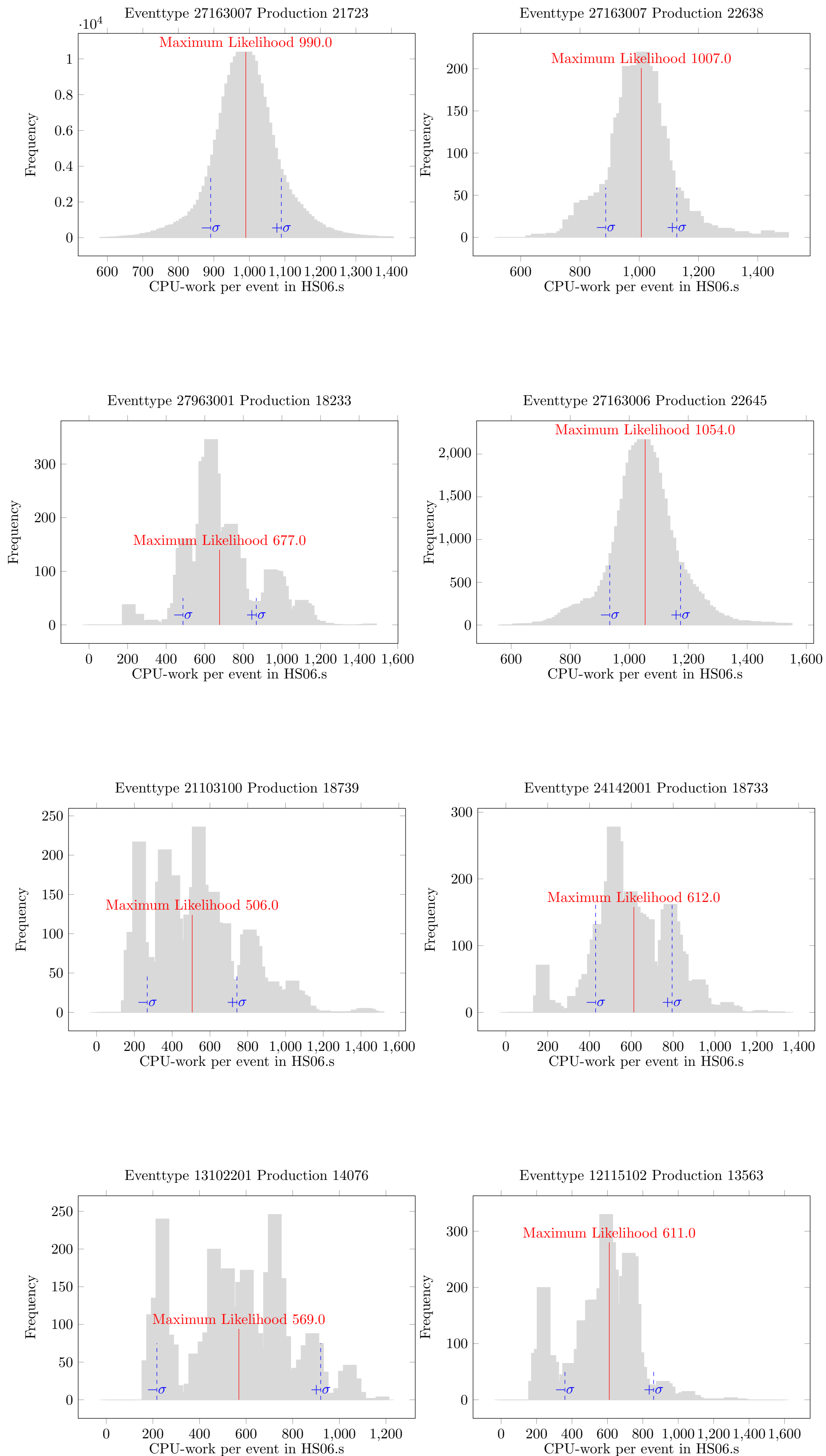

Eventtype 11102201 Production 14073
Maximum Likelihood 587.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163001 Production 16070
Maximum Likelihood 553.0
Frequency
CPU-work per event in HS06.s
Eventtype 10022013 Production 20936
Maximum Likelihood 1489.0
Frequency
CPU-work per event in HS06.s
Eventtype 13873201 Production 17999
Maximum Likelihood 652.0
Frequency
CPU-work per event in HS06.s
Eventtype 11512012 Production 20492
Maximum Likelihood 1183.0
Frequency
CPU-work per event in HS06.s
Eventtype 12875031 Production 20154
Maximum Likelihood 546.0
Frequency
CPU-work per event in HS06.s
Eventtype 28102002 Production 20254
Maximum Likelihood 2152.0
Frequency
CPU-work per event in HS06.s
Eventtype 27165104 Production 21026
Maximum Likelihood 575.0
Frequency
CPU-work per event in HS06.s

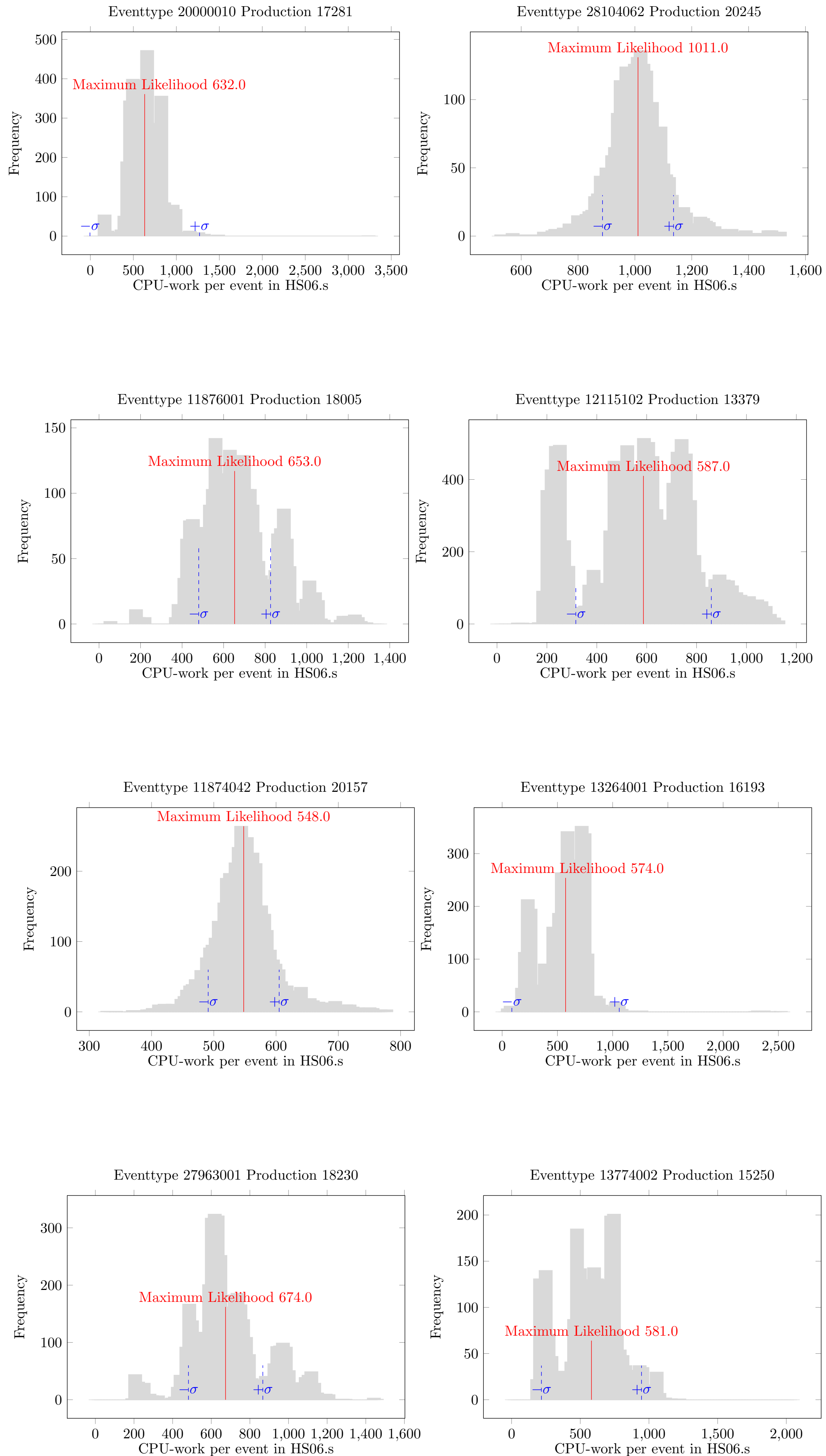


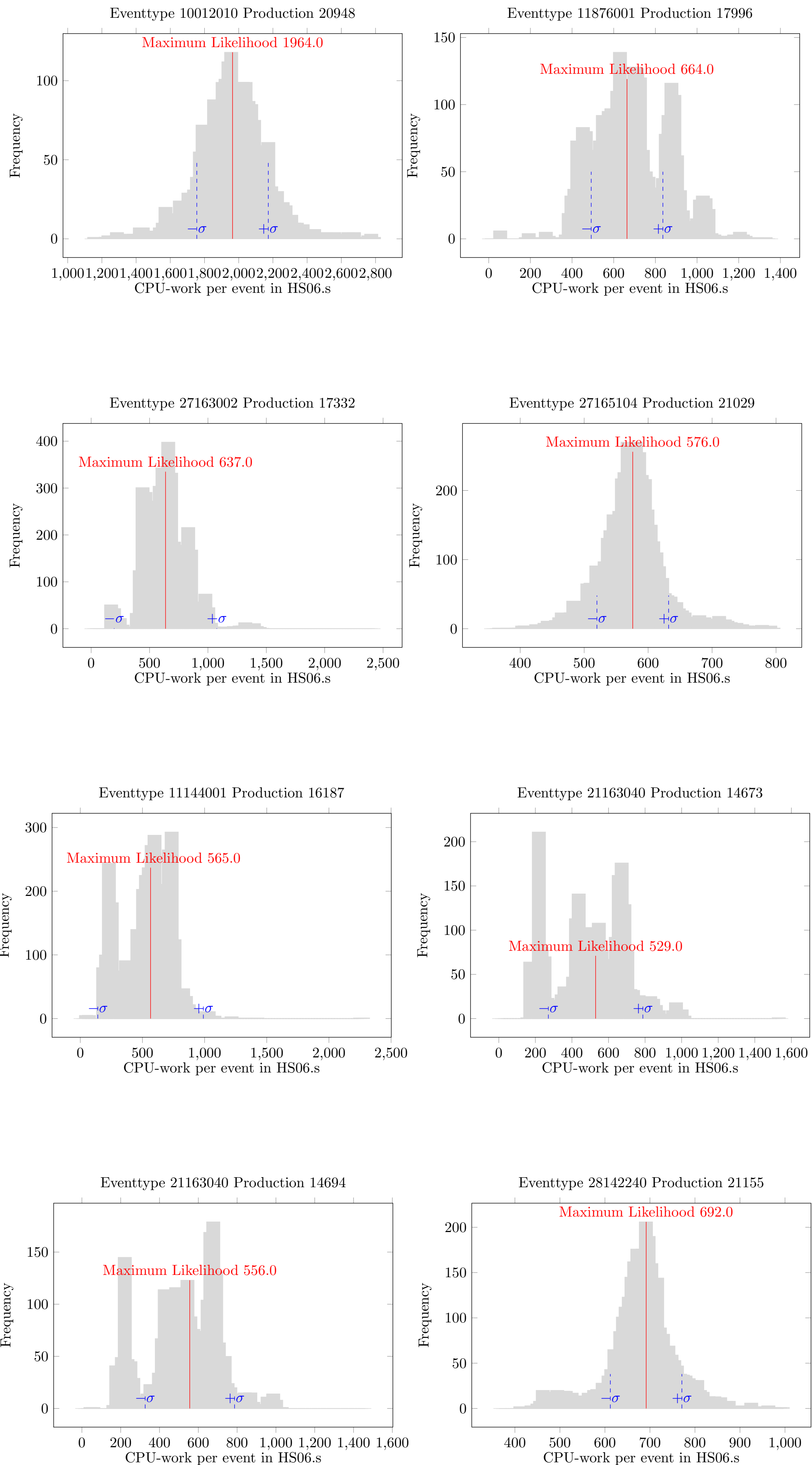
Eventtype 27163005 Production 20940
Maximum Likelihood 526.0
Frequency
CPU-work per event in HS06.s
Eventtype 11144001 Production 16190
Maximum Likelihood 578.0
Frequency
CPU-work per event in HS06.s
Eventtype 24142001 Production 16300
Maximum Likelihood 481.0
Frequency
CPU-work per event in HS06.s
Eventtype 28142240 Production 21161
Maximum Likelihood 686.0
Frequency
CPU-work per event in HS06.s
Eventtype 24142001 Production 18706
Maximum Likelihood 607.0
Frequency
CPU-work per event in HS06.s
Eventtype 21263002 Production 14661
Maximum Likelihood 550.0
Frequency
CPU-work per event in HS06.s
Eventtype 21263030 Production 14688
Maximum Likelihood 560.0
Frequency
CPU-work per event in HS06.s
Eventtype 21103100 Production 18742
Maximum Likelihood 490.0
Frequency
CPU-work per event in HS06.s

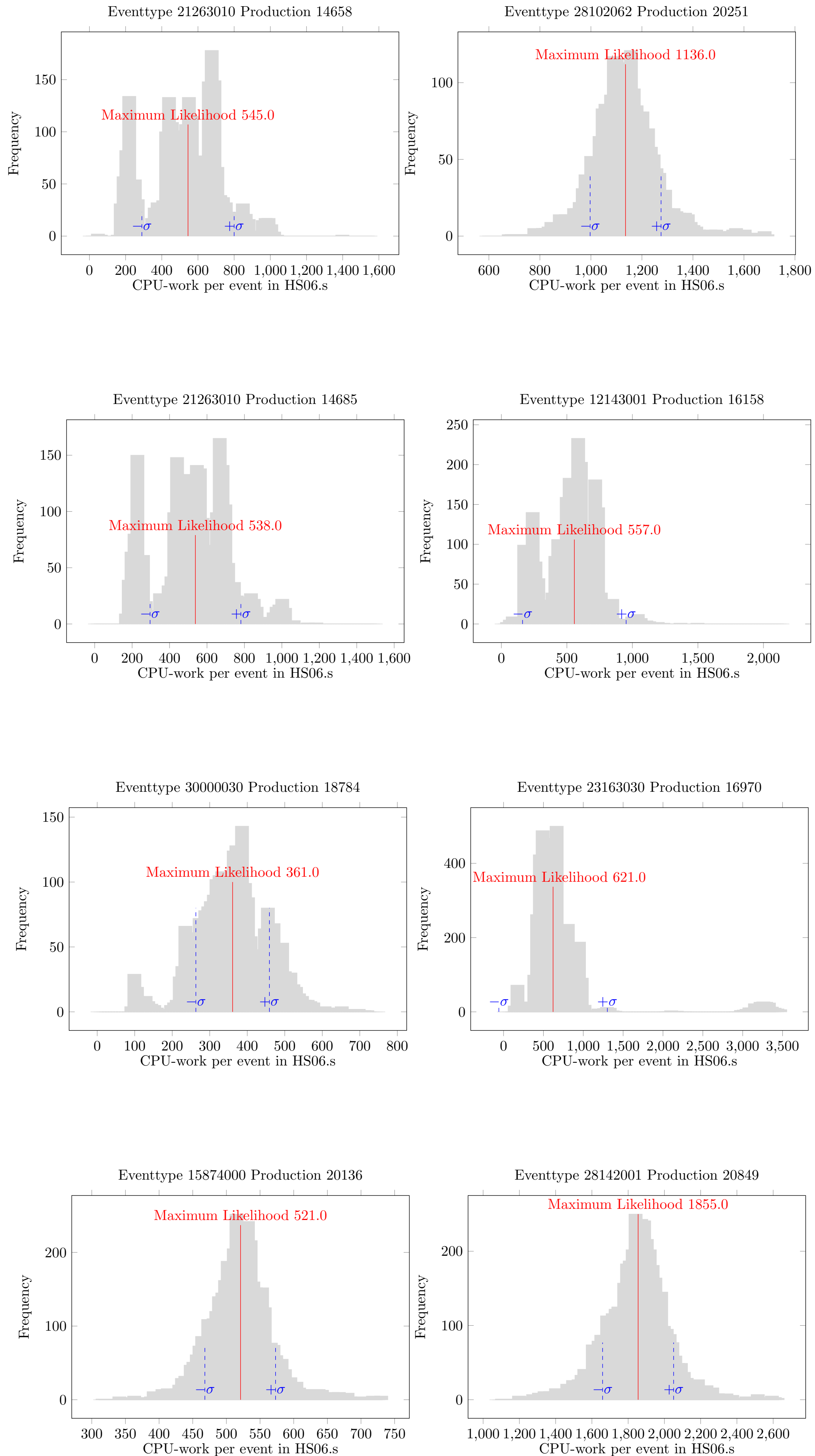


Eventtype 21163020 Production 14655
Maximum Likelihood 561.0
Frequency
CPU-work per event in HS06.s
Eventtype 27265001 Production 16073
Maximum Likelihood 561.0
Frequency
CPU-work per event in HS06.s
Eventtype 15874010 Production 20139
Maximum Likelihood 522.0
Frequency
CPU-work per event in HS06.s
Eventtype 15874010 Production 20151
Maximum Likelihood 526.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163001 Production 16291
Maximum Likelihood 506.0
Frequency
CPU-work per event in HS06.s
Eventtype 13774002 Production 15253
Maximum Likelihood 610.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163400 Production 20024
Maximum Likelihood 509.0
Frequency
CPU-work per event in HS06.s
Eventtype 24142001 Production 16297
Maximum Likelihood 489.0
Frequency
CPU-work per event in HS06.s

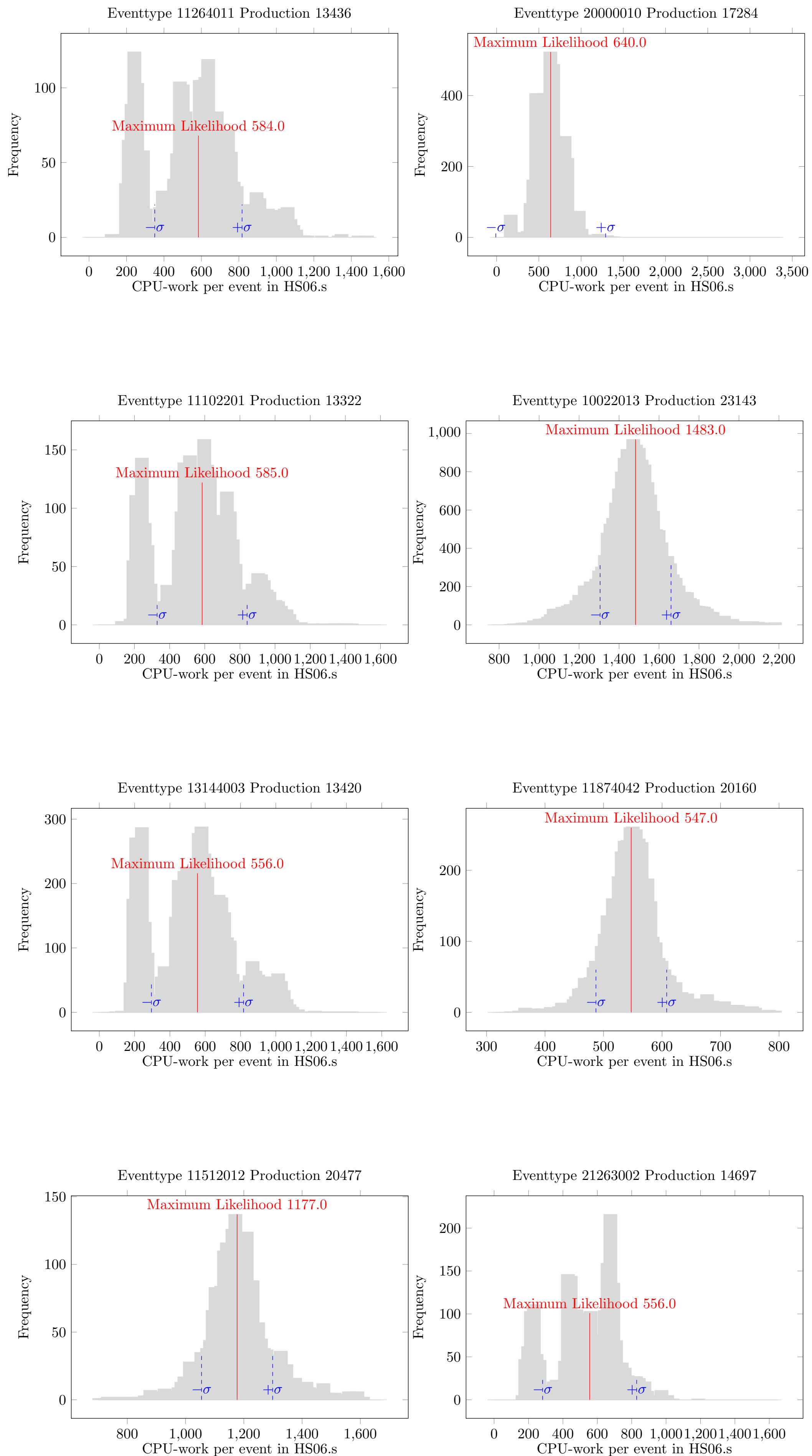

Eventtype 27163007 Production 21723
Maximum Likelihood 990.0
Eventtype 27163007 Production 22638
Maximum Likelihood 1007.0
Eventtype 27963001 Production 18233
Maximum Likelihood 677.0
Eventtype 27163006 Production 22645
Maximum Likelihood 1054.0
Eventtype 21103100 Production 18739
Maximum Likelihood 506.0
Eventtype 24142001 Production 18733
Maximum Likelihood 612.0
Eventtype 13102201 Production 14076
Maximum Likelihood 569.0
Eventtype 12115102 Production 13563
Maximum Likelihood 611.0
Frequency
CPU-work per event in HS06.s
$-\sigma$
$+\sigma$

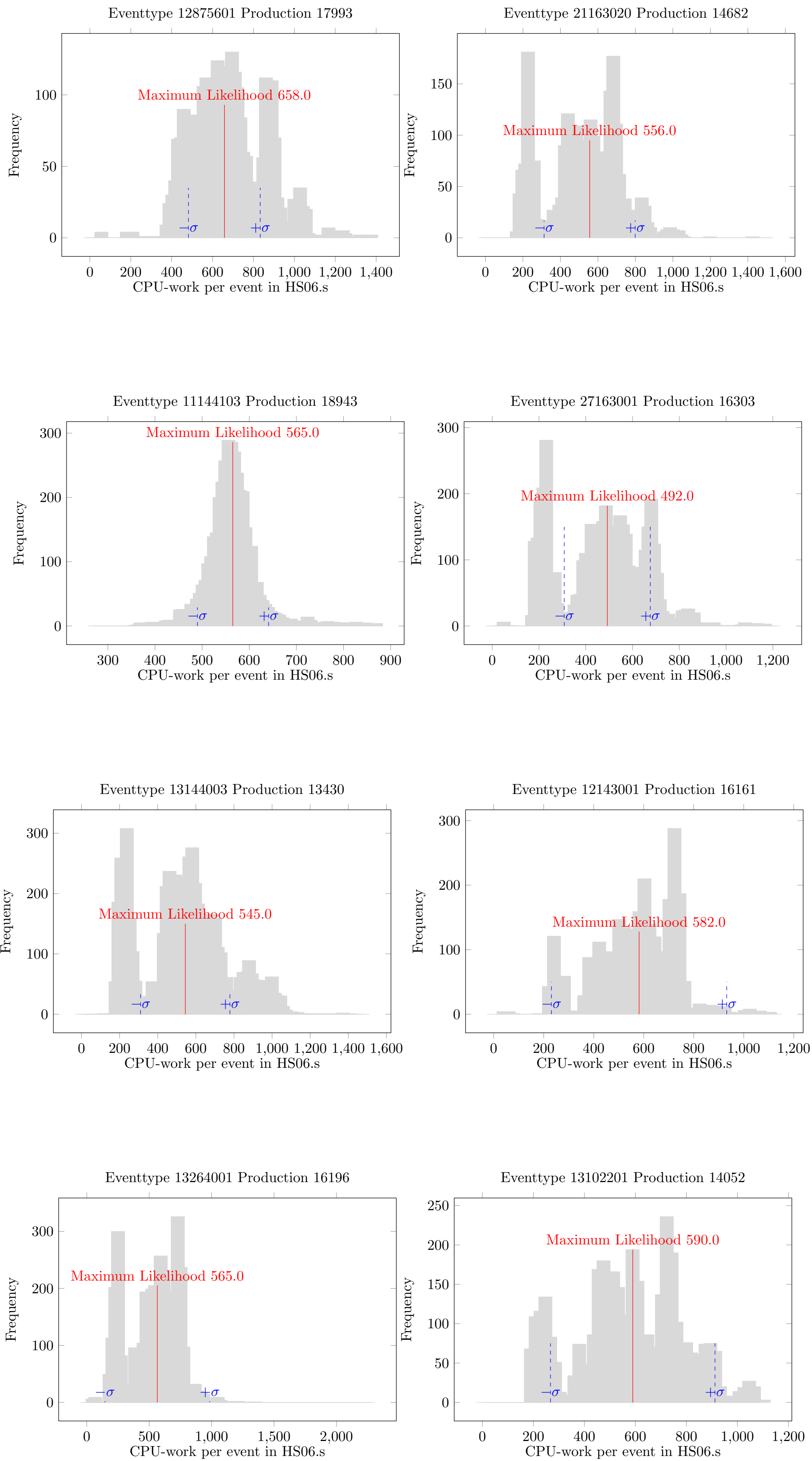

Eventtype 20000010 Production 17281
Maximum Likelihood 632.0
Frequency
CPU-work per event in HS06.s
Eventtype 28104062 Production 20245
Maximum Likelihood 1011.0
Frequency
CPU-work per event in HS06.s
Eventtype 11876001 Production 18005
Maximum Likelihood 653.0
Frequency
CPU-work per event in HS06.s
Eventtype 12115102 Production 13379
Maximum Likelihood 587.0
Frequency
CPU-work per event in HS06.s
Eventtype 11874042 Production 20157
Maximum Likelihood 548.0
Frequency
CPU-work per event in HS06.s
Eventtype 13264001 Production 16193
Maximum Likelihood 574.0
Frequency
CPU-work per event in HS06.s
Eventtype 27963001 Production 18230
Maximum Likelihood 674.0
Frequency
CPU-work per event in HS06.s
Eventtype 13774002 Production 15250
Maximum Likelihood 581.0
Frequency
CPU-work per event in HS06.s

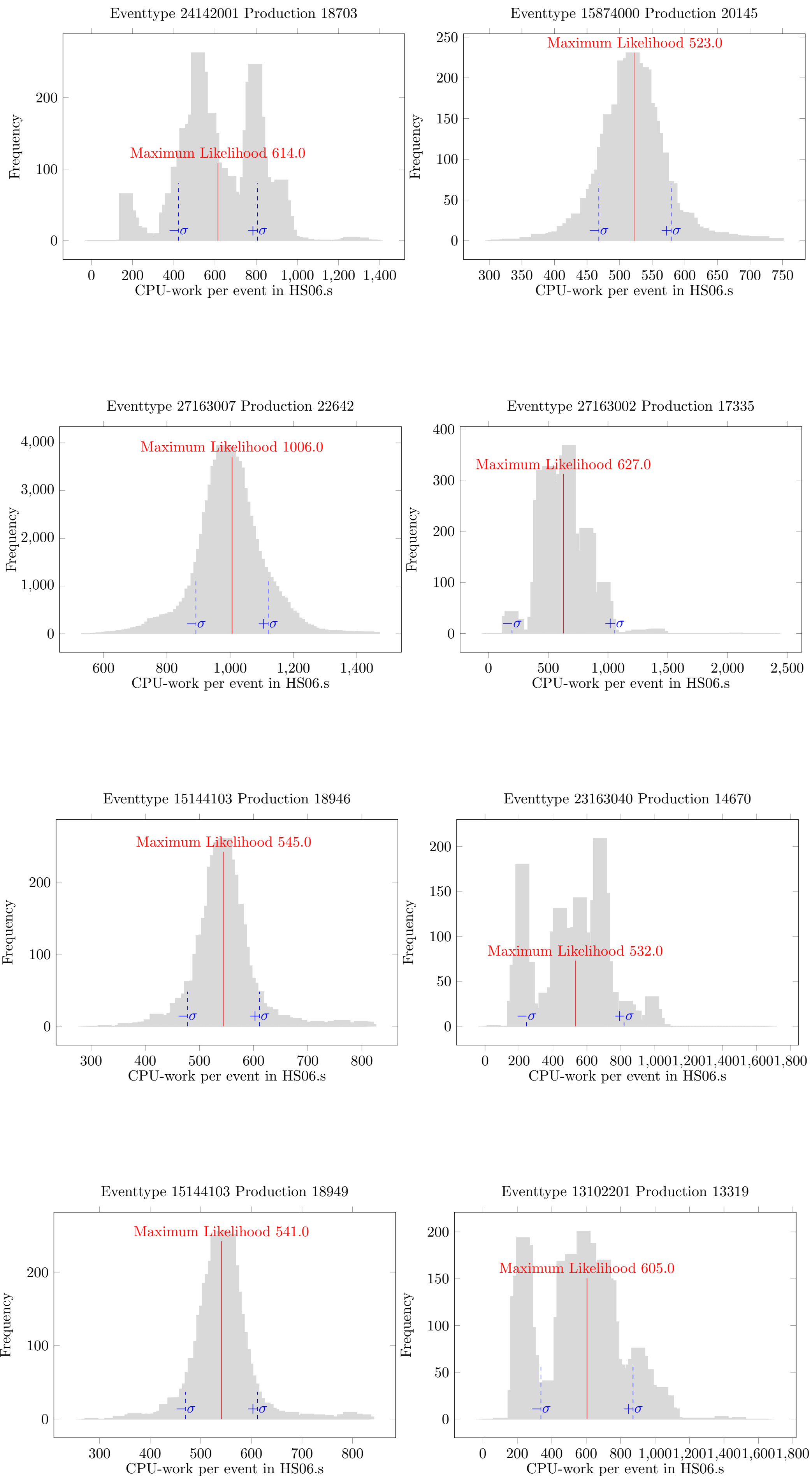

Eventtype 10012010 Production 20948
Maximum Likelihood 1964.0
Frequency
CPU-work per event in HS06.s
Eventtype 11876001 Production 17996
Maximum Likelihood 664.0
Eventtype 27163002 Production 17332
Maximum Likelihood 637.0
Eventtype 27165104 Production 21029
Maximum Likelihood 576.0
Eventtype 11144001 Production 16187
Maximum Likelihood 565.0
Eventtype 21163040 Production 14673
Maximum Likelihood 529.0
Eventtype 21163040 Production 14694
Maximum Likelihood 556.0
Eventtype 28142240 Production 21155
Maximum Likelihood 692.0

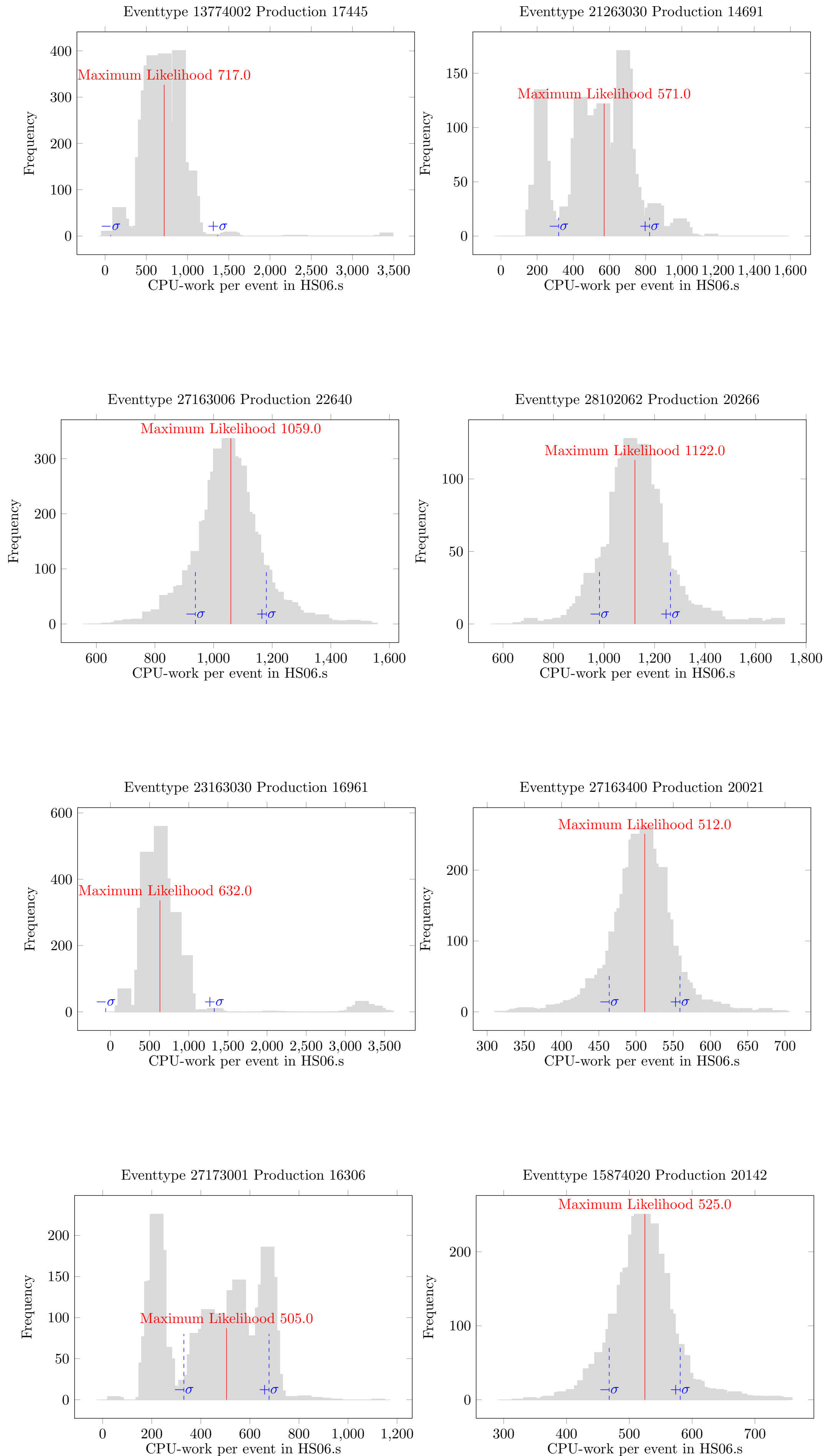


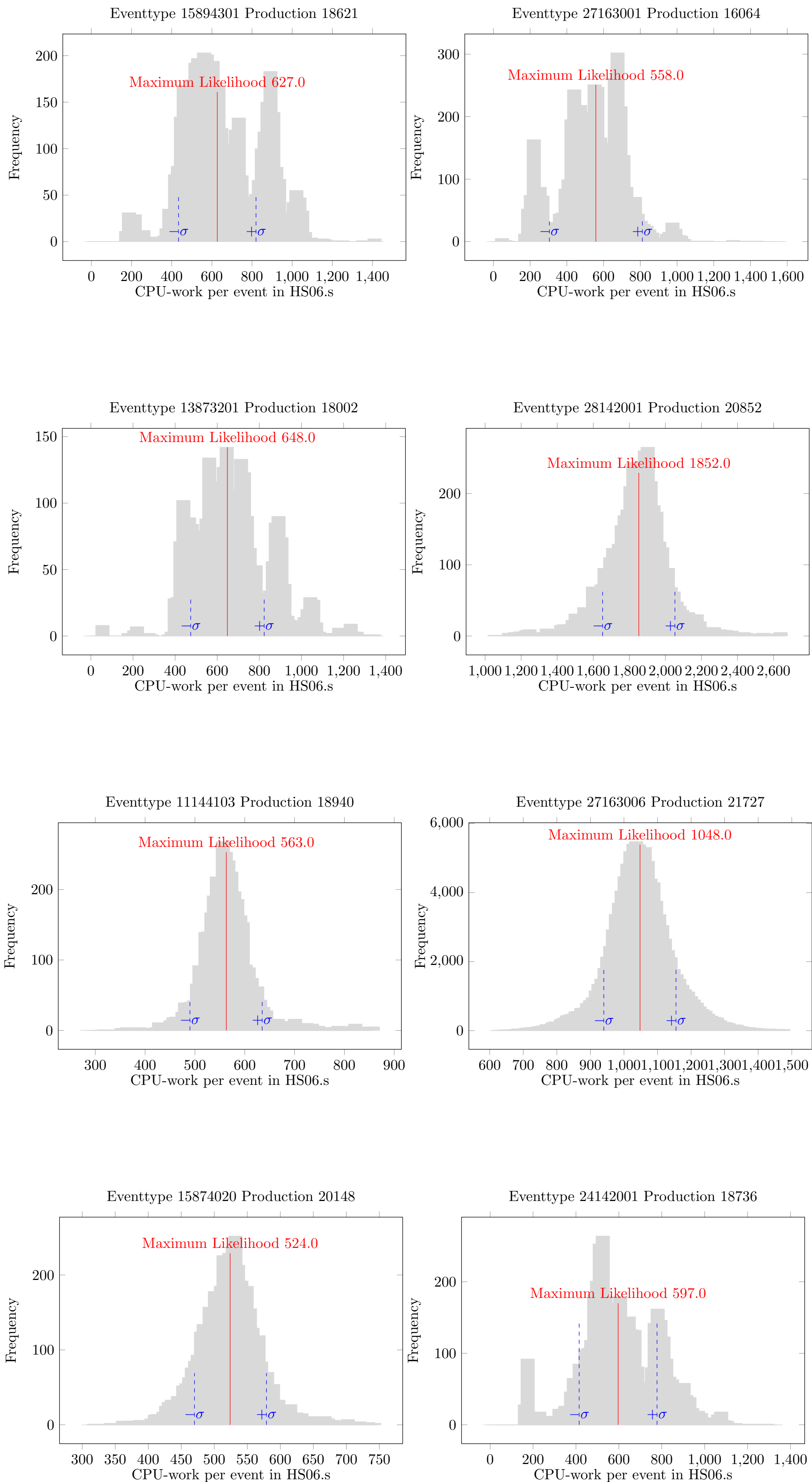
Eventtype 21263010 Production 14658
Maximum Likelihood 545.0
Eventtype 28102062 Production 20251
Maximum Likelihood 1136.0
Eventtype 21263010 Production 14685
Maximum Likelihood 538.0
Eventtype 12143001 Production 16158
Maximum Likelihood 557.0
Eventtype 30000030 Production 18784
Maximum Likelihood 361.0
Eventtype 23163030 Production 16970
Maximum Likelihood 621.0
Eventtype 15874000 Production 20136
Maximum Likelihood 521.0
Eventtype 28142001 Production 20849
Maximum Likelihood 1855.0
Frequency
CPU-work per event in HS06.s
$-\sigma$
$+\sigma$

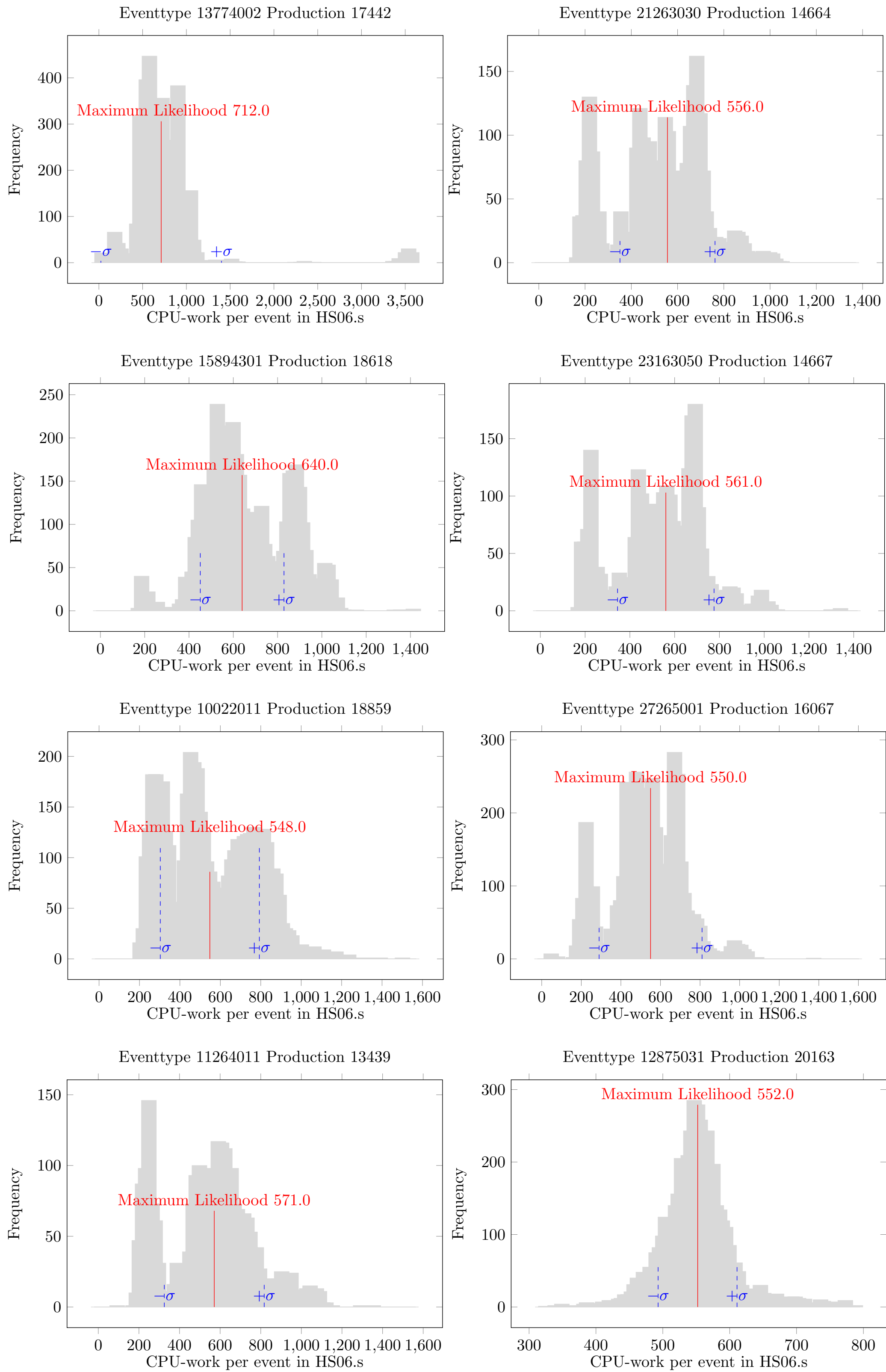

Eventtype 11264011 Production 13436
Maximum Likelihood 584.0
Frequency
CPU-work per event in HS06.s
Eventtype 20000010 Production 17284
Maximum Likelihood 640.0
Frequency
CPU-work per event in HS06.s
Eventtype 11102201 Production 13322
Maximum Likelihood 585.0
Frequency
CPU-work per event in HS06.s
Eventtype 10022013 Production 23143
Maximum Likelihood 1483.0
Frequency
CPU-work per event in HS06.s
Eventtype 13144003 Production 13420
Maximum Likelihood 556.0
Frequency
CPU-work per event in HS06.s
Eventtype 11874042 Production 20160
Maximum Likelihood 547.0
Frequency
CPU-work per event in HS06.s
Eventtype 11512012 Production 20477
Maximum Likelihood 1177.0
Frequency
CPU-work per event in HS06.s
Eventtype 21263002 Production 14697
Maximum Likelihood 556.0
Frequency
CPU-work per event in HS06.s


Eventtype 12875601 Production 17993
Maximum Likelihood 658.0
−σ
+σ
Frequency
CPU-work per event in HS06.s
Eventtype 21163020 Production 14682
Maximum Likelihood 556.0
Eventtype 11144103 Production 18943
Maximum Likelihood 565.0
Eventtype 27163001 Production 16303
Maximum Likelihood 492.0
Eventtype 13144003 Production 13430
Maximum Likelihood 545.0
Eventtype 12143001 Production 16161
Maximum Likelihood 582.0
Eventtype 13264001 Production 16196
Maximum Likelihood 565.0
Eventtype 13102201 Production 14052
Maximum Likelihood 590.0


Eventtype 24142001 Production 18703
Maximum Likelihood 614.0
Frequency
CPU-work per event in HS06.s
Eventtype 15874000 Production 20145
Maximum Likelihood 523.0
Eventtype 27163007 Production 22642
Maximum Likelihood 1006.0
Eventtype 27163002 Production 17335
Maximum Likelihood 627.0
Eventtype 15144103 Production 18946
Maximum Likelihood 545.0
Eventtype 23163040 Production 14670
Maximum Likelihood 532.0
Eventtype 15144103 Production 18949
Maximum Likelihood 541.0
Eventtype 13102201 Production 13319
Maximum Likelihood 605.0


Eventtype 13774002 Production 17445
Maximum Likelihood 717.0
Frequency
CPU-work per event in HS06.s
Eventtype 21263030 Production 14691
Maximum Likelihood 571.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163006 Production 22640
Maximum Likelihood 1059.0
Frequency
CPU-work per event in HS06.s
Eventtype 28102062 Production 20266
Maximum Likelihood 1122.0
Frequency
CPU-work per event in HS06.s
Eventtype 23163030 Production 16961
Maximum Likelihood 632.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163400 Production 20021
Maximum Likelihood 512.0
Frequency
CPU-work per event in HS06.s
Eventtype 27173001 Production 16306
Maximum Likelihood 505.0
Frequency
CPU-work per event in HS06.s
Eventtype 15874020 Production 20142
Maximum Likelihood 525.0
Frequency
CPU-work per event in HS06.s


Eventtype 15894301 Production 18621
Maximum Likelihood 627.0
Frequency
CPU-work per event in HS06.s
Eventtype 27163001 Production 16064
Maximum Likelihood 558.0
Eventtype 13873201 Production 18002
Maximum Likelihood 648.0
Eventtype 28142001 Production 20852
Maximum Likelihood 1852.0
Eventtype 11144103 Production 18940
Maximum Likelihood 563.0
Eventtype 27163006 Production 21727
Maximum Likelihood 1048.0
Eventtype 15874020 Production 20148
Maximum Likelihood 524.0
Eventtype 24142001 Production 18736
Maximum Likelihood 597.0


Eventtype 13774002 Production 17442
Maximum Likelihood 712.0
Frequency
CPU-work per event in HS06.s
Eventtype 21263030 Production 14664
Maximum Likelihood 556.0
Eventtype 15894301 Production 18618
Maximum Likelihood 640.0
Eventtype 23163050 Production 14667
Maximum Likelihood 561.0
Eventtype 10022011 Production 18859
Maximum Likelihood 548.0
Eventtype 27265001 Production 16067
Maximum Likelihood 550.0
Eventtype 11264011 Production 13439
Maximum Likelihood 571.0
Eventtype 12875031 Production 20163
Maximum Likelihood 552.0