Fair, Responsive Scheduling of Engineering Workflows on Computing Grids

Andrew Marc Burkimsher

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Doctor of Engineering.

University of York

Department of Computer Science

August 2014

Abstract

This thesis considers scheduling in the context of a grid computing system used in engineering design. Users desire responsiveness and fairness in the treatment of the workflows they submit. Submissions outstrip the available computing capacity during the work day, and the queue is only caught up on overnight and at weekends. The execution times observed span a wide range of 10^0 to 10^7 core-minutes.

The Projected Schedule Length Ratio (P-SLR) list scheduling policy is designed to use execution time estimates and the structure of the dependency graph to improve on the existing industrial FairShare policy. P-SLR aims to minimise the worst-case SLR of jobs and keep SLR fair across the space of job execution times. P-SLR is shown to equal or surpass all other evaluated policies in responsiveness and fairness across the spectra of load and networking delays. P-SLR is also dominant where execution time estimates are within an order of magnitude of the real value. Such estimates are considered achievable using user knowledge or automated profiling. Outside this range, the Shortest Remaining Time First (SRTF) policy achieved better responsiveness and fairness.

The Projected Value Remaining (PVR) policy considers the case where a curve specifying the value of a job over time is given. PVR aims to maximise total workload value, even under overload, by maximising the worst-case job value in a workload. PVR is shown to be dominant across the load and networking spectra. Where execution time estimates are coarser than the nearest power of 2, SRTF delivers higher value than PVR. SRTF is also shown to have responsiveness, fairness and value close behind P-SLR and PVR throughout the range of load and network delays considered. However, the kinds of starvation under overload incurred by SRTF would almost certainly be undesirable if implemented in a production system.

Contents

List of Tables

List of Figures

Acknowledgements

The author wishes to thank:

- Iain Bate and Leandro Soares Indrusiak for their helpful guidance and support.
- Colleagues at the industrial partner organisation for their input into understanding the research context and for the access they gave to their systems and facilities.
- The Engineering and Physical Sciences Research Council (EPSRC) for funding this research through the UK's Large-Scale Complex IT Systems (LSCITS) programme, grant number EP/F501374/1.
- The Dringhouses Belfrey Group for their humour, prayers, support and encouragement.
- My dear wife Emily Burkimsher for her patient love and encouragement throughout this EngD.

Related Publications

The author has four publications based on the work described in this thesis. The author of this thesis is the main author for all these papers.

1. Andrew Burkimsher. Dependency patterns and timing for grid workloads. In *Proceedings of the 4th York Doctoral Symposium on Computer Science*, pages 25–33, October 2011. [26] (conference acceptance rate 83%)
   This paper contributed a number of techniques for synthetic workload generation which are included in Chapter 4.

2. Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. *Future Generation Computer Systems*, 29(8):2009 – 2025, October 2013 (Online 28 December 2012). [27] (Journal Impact Factor 1.864) This paper was awarded the K. M. Stott Prize for the Best Paper in Computer Science in 2012.
   This paper contributed the survey of metrics used as a basis for Chapter 5. This paper also gave the contribution of the P-SLR scheduling policy and its evaluation, which is the basis of Sections 6.1 to 6.3 in Chapter 6.

3. Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. Scheduling HPC workflows for responsiveness and fairness with networking delays and inaccurate estimates of execution times. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par 2013)*, volume 8097 of *Lecture Notes in Computer Science*, pages 126–137. Springer Berlin Heidelberg, 2013. [28] (conference acceptance rate 26.8%).
   This paper contributed the evaluation of P-SLR in the presence of network delays and inaccurate estimates of execution times, and is the basis of Sections 6.4 and 6.4.2 in Chapter 6.

4. Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A characterisation of the workload on an engineering design grid. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 8:1–8:8, San

Diego, CA, USA, 2014. Society for Computer Simulation International. [29] (conference acceptance rate 66.7%)
This paper contributed the majority of the workload characterisation and all the synthetic workload generation algorithms presented in Chapter 4.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Engineering of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .....(candidate)

Date .....

Chapter 1

Introduction

1.1 Computing Trends

In recent years, increases in computing power have been due to greater hardware parallelism, rather than higher chip clock speeds. This is mainly due to the exponential increase in power consumption required to run processors at higher clock speeds [83]. However, there continues to be insatiable demand for computing power in a wide range of domains, from scientific computing, to e-commerce, to healthcare and engineering.

The observed pattern known as Moore's Law has continued with a steady exponential increase in the number of transistors available on processing chips [123]. To make use of all these transistors, manufacturers have created processors containing many execution cores. These range from 2-10 on general purpose CPUs [84], or more than 2500 on the specialised parallel processors on graphics cards [1, 130].

However, no single processor or graphics card can provide all the computing capacity required for large organisations. Therefore, in many High-Performance Computing (HPC) systems, many cores are linked together into clusters of computing machines [41, 42]. Yet just like the power consumption of a single processor can pose a limitation at the core level, so the power consumption of a cluster can limit its size [122]. Some of Google's data centres consume the entire output of a power station [70]. There are many situations where more capacity or redundancy is required than what is available in a single cluster. These computing clusters, spread across countries and across the world, can be linked through private networks or the internet. These groupings of clusters are a particular kind of computational architecture, termed *Grid Computing* by Kesselman and Foster [96].

Within the discipline of Computer Science, there are several inter-related fields that deal with these large-scale, networked processing systems. High Performance Computing tends to examine the hardware design and construction of computing

clusters, along with the study of parallel algorithms that are suited to such hardware. Grid computing is concerned with efficiently federating and managing the resources that make up a grid, in order to ensure that the grid operates to optimal performance in the eyes of the system administrators and grid users alike [96]. Cloud computing is a model whereby large-scale clustered computing resources are connected to the internet, and their capacity is sold as a service [164].

1.2 Aircraft Design Context

Many computing grids have been created through academic institutions pooling their existing computing resources. However, not all such grids are created like this. The author was engaged by a partner organisation who operate their own private grid. The partner organisation's primary business is the design and manufacture of aircraft, which is supported by their grid system.

Aircraft design is a complex and lengthy process. It begins with identifying a business opportunity, with specifications such as payload and range required. These specifications are passed through feasibility studies. Once these are approved, the detailed work of aircraft design begins.

There are a significant number of competing design requirements present when designing an aircraft. These can include efficiency, strength, weight, flexibility, acoustics, materials, ease of manufacture and/or maintenance, among many others.

A particularly important aspect of any aircraft design is the design of the aerodynamic properties of the wings and body of the aircraft. Traditionally, this design was performed using wind tunnel testing, though this is a time-consuming process. In the final stages of design and certification, wind tunnel tests are invaluable because of the high fidelity of the data produced. However, earlier in the design process it is desirable to more quickly iterate through large numbers of possible designs in order to converge on the most promising ones more quickly. In these early iterations, the high fidelity results of a wind tunnel are not as important as the speed of the results.

Having a quicker turnaround of aerodynamic tests enables a greater number of design iterations to take place, which in turn tends to help produce aircraft designs with more desirable aerodynamic performance characteristics. In order to meet the need for quicker turnaround times than are available from wind tunnels, a large amount of early-stage design now takes place in simulation, using advanced Computational Fluid Dynamics (CFD) software.

There are several kinds of calculations that are done using CFD, including airfoil performance in two dimensions, to various kinds of three-dimensional simulations (single airfoil, whole-wing, whole-aircraft). The CFD simulations are used to

evaluate the lift, drag and loads placed on a wing design. Further simulations take place using software to simulate the loads on the internal structure of the wing. All these simulations can be done at varying levels of complexity and fidelity depending on the software and parameters used.

1.3 Requirements of the Grid

The CFD simulations are performed using several pieces of software for different parts of the computation. In this thesis, a single, non-preemptible piece of work to be executed on one or more processors concurrently will be known as a *task*. A set of tasks with dependencies between them is known as a *workflow*. Each *job* is a submitted instance of a *workflow*. The *workload* of a cluster is a set of jobs. This follows the nomenclature of Chapin [36].

With the CFD software, there is an inherent trade-off between the computation time required to run the simulation and the fidelity of the results produced. Up to a point, the CFD algorithms can be parallelised to reduce turnaround time. This tends to mean that the computing capacity is always limited, because users always prefer high-fidelity results and short response time.

As the motivation of procuring the grid is to evaluate aerodynamic properties of designs in a shorter time than is possible with a wind tunnel, the responsiveness of the results calculated by the grid is the most important performance metric for the organisation. Improving responsiveness gives the double benefit of increased productivity and quality of the final design.

There are large numbers of users and teams using the grid to support their work. Each of these teams is under time pressure to meet the deadlines expected of them. Due to this, when the grid is heavily loaded and work must queue, there is intense interest in whether the resources of the grid are being used fairly. A regular activity of the grid administrators is to monitor and adjust the factors within their control that influence the fair treatment of work submitted.

1.4 Hypotheses

The process of prioritising or *ordering* a queue of work and assigning this work to resources or *allocating* is known as *scheduling* [36]. The primary requirement of this research as specified by the industrial partner is to achieve improved responsiveness for the work submitted relative to their existing scheduling policy, known as FairShare. FairShare prioritises work by user, based on their actual instantaneous utilisation of the cluster relative to a 'Fair Share' [93].

This thesis will investigate how to achieve the best responsiveness given the industrial design context. Responsiveness within this context is primarily driven by two factors: how well the CFD algorithms used scale with increasing parallelism and minimising waiting times for work. A large body of work already exists on how to best write CFD software to run on parallel computing hardware [161]. Achieving appropriate fairness and utilisation in conjunction with high responsiveness can only be achieved by changing the priority of work. Therefore, this thesis will investigate the development and application of appropriate scheduling policies for the workload and context of industrial design.

The value of jobs to users can vary depending on their timeliness. If this value can be quantified, this can inform the scheduling of work. This is especially pertinent in overload situations where some work has to wait. Jobs whose value is more sensitive to waiting time can be prioritised, for example. The value achieved by a scheduler can be compared to the maximum possible value achievable if every job were able to run immediately; this measure is known as the proportion of maximum value.

Two specific hypotheses will be investigated.

1.4.1 Hypothesis 1

> Using a context that reflects the industrial scenario, responsiveness and fairness can be improved over the currently implemented FairShare policy using a dynamic list scheduler that prioritises jobs and tasks using information about their dependency structure and task execution time estimates.

1.4.2 Hypothesis 2

> Using a context that reflects the industrial scenario, the proportion of maximum value can be improved over the FairShare policy by using a dynamic list scheduler that uses value curves to calculate the urgency, and hence priority, of jobs and tasks. This scheduler will take into account dependencies, and task execution time estimates in these calculations.

1.5 Thesis Structure

To understand what scheduling policies will give the most improvement, the industrial context of this Engineering Doctorate and the challenges currently faced need to be appropriately captured. Chapter 2 will examine the industrial context from several angles. Firstly, the socio-technical context of the grid will be described.

This includes the working patterns and environment of the designers who use the grid. Secondly, the grid hardware and software architecture of the organisation will be described. Thirdly, the current scheduling scheme will be described. Particular issues with this scheme that users have noted will be highlighted. A focus will be on the suitability of the current scheduling policy to address a workload containing a very wide range of execution times.

In order to analyse the industrial grid system, software tools are required. The use of these tools is of industrial as well as academic interest, to enable the industrial partner to engage in ongoing analysis and monitoring of the grid; the development of such tools being an important industrial contribution of an Engineering Doctorate. Two tools were specifically desired by the industrial partner. Firstly, a tool was needed to help users decide which cluster in the grid to submit to, given the grid and the scheduler's current state (see Chapter 2). Secondly, a suite of tools was needed to automatically determine metrics and visualise the state of the grid (see Chapter 5).

The existing literature on scheduling will be surveyed in Chapter 3 to investigate the state of the art and examine the kinds of approaches that can be applied. There will be particular focus on dynamic scheduling, as this is what a grid requires. Within dynamic scheduling methods, a focus will be placed on list scheduling, because this has been well-studied and also lends itself well to hierarchical composition, as is found in a grid computing architecture. Approaches that have been used to model and schedule workflows with dependencies will be surveyed. In addition, scheduling policies that can work with a distributed, heterogeneous hardware base with network delays, and the models that support these are also described in detail.

Any scheduling policy will naturally have to prioritise some tasks over others. A gap identified in the literature is the analysis of how schedulers prioritise work across the spectrum of execution times. A detailed understanding of the workload that the scheduler operates on is an important part of developing an effective scheduling policy. Even on a similar platform, widely different schedulers may be appropriate for different workloads. Chapter 4 will undertake a detailed characterisation of the workload run by the partner organisation. These characterisations will inform the parameters and distributions used for the generation of synthetic workloads that share the same characteristics as the industrial one.

To perform experimental simulations in an academically-sound manner, appropriate models of the grid system and the applications that run on such a system are required. Chapter 5 will describe the models used by the author to form the basis of the simulation framework. The application model represents the workload to be run in an abstract way, along with the behaviour of the users in the submission patterns of their work. For workloads where the notion of value is

considered, the value model describes how tasks have their value represented and calculated.

A model of inaccurate estimates of execution times is presented. The platform model represents the grid hardware and the middleware that manages the execution of the tasks on the system. The network model captures the delays inherent in moving data and applications between distributed sites. The scheduling model describes when and where scheduling decisions are made, and the structure (although not the particular policy) of the scheduling algorithms analysed. The models are designed to realistically represent the amount of information available to decision algorithms at each level of the grid hierarchy.

Chapter 5 also contains an evaluation of metrics to measure responsiveness and fairness in a way that best captures users' concerns. This evaluation concludes that the Schedule Length Ratio (SLR) [160] is most appropriate as, unlike other metrics, it considers the structure of dependencies in the workload.

The foundation of the contributions of this thesis is a list scheduling policy structure that calculates a projection of what the finish time would be for jobs in the queue. The projected finish time calculation uses the upward rank metric of Topcuoglu et. al. [160] which is based on execution times (known or estimated) and dependency patterns. The projected finish time is then used to calculate a metric of interest. The scheduler then prioritises the work in the queue by the chosen metric in order to build an appropriate schedule.

Chapter 6 presents and evaluates the Projected-Schedule Length Ratio (P-SLR) algorithm. This algorithm is a novel scheduler designed to minimise the worst-case SLR in the case where estimates of execution times are made available to the scheduler. P-SLR is compared against other commonly implemented scheduling policies, investigating Hypothesis 1. A key contribution of this thesis is to show that P-SLR delivers at least equal responsiveness and better fairness compared to other policies while adding a guarantee that no job will ever starve. Extensions to the original evaluation with network delays and where only inaccurate estimates of execution times are known in advance are also undertaken.

Any single heuristic metric used for scheduling will have some tradeoffs. The Projected-SLR policy will under-prioritise long-running yet urgent jobs, and over-prioritise short-running yet non-urgent jobs. Where Chapter 6 considers scheduling with task execution times, Chapter 7 considers how scheduling could be improved if users also provide information on the time-value of tasks, investigating Hypothesis 2. Specifically, if users were to specify the value delivered by the timely completion of work, as well as how this value is degraded if lateness increases.

Using this model of value, a novel list scheduling policy known as Projected Value Remaining (PVR) is developed that aims to maximise the worst-case value achieved for any jobs in a workload. An evaluation is undertaken to compare PVR

against alternative scheduling policies, including ones that also consider value in their calculations. A further contribution of this thesis is to show that PVR delivers higher workload value across the spectra of load and networking delays. It is also dominant where task execution estimate inaccuracies are within reasonable ranges, although it no longer dominates when errors are significant.

The conclusions of the thesis are discussed in Chapter 8. The summary of the evaluations will be given, detailing the success of the P-SLR and PVR policies in their respective contexts. A discussion is made of possible generalisations of these policies. Specifically, they should be easily applicable to pre-emptive online systems if the prioritisation is done for each time quantum. These policies may be also be suitable for network packet prioritisation, especially where short, latency-sensitive flows are multiplexed over the same link as larger flows that are less urgent.

Chapter 2

Background and Motivation

In the introductory chapter, it was briefly described how responsiveness and fairness are the key performance metrics that aircraft designers wish to achieve in the execution of their CFD workloads. As this research was conducted in conjunction with an industrial partner organisation, the outcome of the research should be relevant within the context of this organisation.

This chapter will describe the existing working patterns of designers, along with the grid platform architecture used by the organisation. The majority of the insights that are presented in these sections were gained through interviews with the aircraft designers and the grid system administrators. These interviews were conducted while the author was on placement at the industrial partner.

The 'FairShare' existing scheduling policy used on their grid platform will also be described. The applicability of the FairShare policy to the industrial grid as it is currently being used will be discussed. Several shortcomings will be noted, which stem from the mismatch of the industrial context to FairShare's original design aims. These shortcomings will be used to inform the subsequent direction of the research, which will be stated as a set of research problems along with a hypothesis.

In order to understand the industrial context in depth, the author developed several tools while working with the industrial partner. These tools, especially the visualisations created, were a key contribution for the partner. They have been rolled out to production use, and aid the partner in understanding and monitoring the performance of their grid system. This chapter will also include descriptions of these tools and how these tools helped in understanding the industrial problem.

2.1 Aerodynamic Aircraft Design Cycles

Aerodynamic design for aircraft is an iterative process that aims to determine the optimal outside shape of the wing and body of an aircraft [149]. This design process is

all about balancing a large number of factors [144], where improvement in one factor necessarily implies a degradation of another. The primary aim is to maximise the lift generated by a wing while minimising drag [60]. However, this design is subject to constraints - in that the wing must be sufficiently thick to hold fuel tanks and support structures to make it strong [8]. The wing must also be performant across a range of air speed values and air pressures [60]. Over-optimising for any one set of parameters may reduce its performance in others. Therefore, designers tend to work in a cyclic and iterative way - searching for a good design for the most common scenarios and then tuning the design to widen the envelope of good performance [8, 144].

Traditional aerodynamic design for aircraft is performed by crafting scale models in metal and placing these models inside a wind tunnel in order to determine aerodynamic characteristics. There are several drawbacks to using wind tunnel testing, however. First, a scale model needs to be manufactured out of metal. Because the aerodynamics are highly sensitive to the shape of the model, the models need to be machined to extreme levels of precision [118]. Any precision required in the full-size model needs to be reflected in the scale model, so if a tolerance of 1mm is desired in the full-size airframe, then a tolerance of 0.01mm would be required in a 1:100 scale model [118]. Moulding, grinding and polishing metal models to these kinds of tolerances is highly precise work, and is not amenable to economies of scale, because every iteration of a model is different [118]. At the present time, the industrial perception is that rapid prototyping techniques like 3D printing are not sufficiently precise for this kind of work.

Once the models have been manufactured a slot needs to be booked in the wind tunnel, and these slots are naturally limited. Finally, the tunnel needs to be set up and the measurements taken. This whole process is highly time-consuming, with many months between the design being finalised and the wind tunnel results being available for analysis.

For many kinds of design, particularly in the early stages where fidelity is less important than speed, designers now use CFD software to simulate the designs [149]. There are many classes of simulations that are run in CFD [47, 150], and these classes vary significantly in runtime [121]. For example, two-dimensional simulations of an airfoil might take just a few minutes to run on a small number of cores. A three-dimensional airfoil might take a few hours to run, though more complex creations of several airfoil sections joined together into a wing may take a day or more on a large number of cores.

With each of these simulations, parameters such as the angle of attack, the deployment settings of ailerons or high lift devices and the atmospheric conditions can be simulated. The larger the number of these conditions, the longer the simulations will take [46]. The highly detailed simulations, necessary in the final

stages of design and for certification, can take months to execute over hundreds of cores.

The prevailing design process is to allocate a fixed amount of time to the designers, and after this time period has elapsed, the best design found is the one selected for use. By reducing the cycle time of simulations, designers can do more iterations on each design. More iterations lead to a larger design space being explored, which tends to lead to better quality solutions in the end. These better quality solutions feed directly into the ability of the organisation to make competitive products, so reducing the cycle time of simulations is a high priority.

A requirement of the scheduling solution should be to ensure high responsiveness for jobs.

2.1.1 Wider Relevance of the problem

Due to the industrial processes within which the designers work, responsiveness of the grid workloads is key. Industrial processes across a wide range of industries are subject to the same pressure to develop high-performing solutions in a short amount of time [147]. The software packages used by the industrial partner have been employed across several industries where CFD is relevant [150]. With the recent growth in computing power, many industries including automotive [63] and integrated circuit fabrication [75] are turning to computational simulation in order to aid design space exploration.

Therefore, while it is known from this case study that responsiveness and fairness of grid workflows is critical to aircraft design, it is logical to conclude that it will also be critical wherever computational simulation is used as part of an industrial design process. The growth of computational simulation techniques in industry will mean that the importance and relevance of scheduling algorithms to support this kind of work will increase.

The data centres required for such simulation workloads are hugely expensive to build and run; with construction costs being as high as a billion dollars [153] and running costs being in the tens of millions of pounds per year [12]. Therefore, their owners want to be able to use them at peak capacity. A trend in industry is the desire to outsource the provision of computing capacity to cloud computing providers. Cloud computing came about due to virtualisation, where multiple virtual machines are run on a single physical machine in order to obtain better utilisation out of the powerful underlying hardware [164]. Idle hardware still uses a significant fraction of the power used by hardware under load, yet will be earning the cloud provider no revenue. As electrical power is the largest cost of most cloud computing providers [174], achieving high utilisation is therefore critical to their profitability [142].

However, as cloud computing becomes more prevalent and used by production services, service level agreements (SLAs) will be necessary in order to ensure that end-users receive the services they have bought [164]. Responsiveness clauses are highly likely to be part of such SLAs. Cloud computing providers will have to balance the tension of maintaining the illusion to their users that they have limitless elastic capacity, while still achieving high enough average utilisation to make their business profitable [142]. If a scheduler were available that could degrade gracefully under overload, cloud providers might be able to improve their profitability by being able to tolerate short periods of overload while still maintaining their SLAs.

2.2 Scheduling Considerations

2.2.1 High Load

One approach to reducing the cycle time would be to purchase enough computational resource such that tasks never had to queue. However, because of peaks and troughs in the submission rates of work during and outside of working hours, this may waste significant amounts of capacity (See Section 4.2 for more details). Furthermore, there is always going to be a limited budget available with which to purchase such resources. Because the fidelity of the CFD algorithms is adjustable to a degree, the designers will always be able to request more resources. CFD algorithms require phenomenal quantities of computing power [43]. It is said by designers that solving the Navier-Stokes set of equations in perfect fidelity would take longer than the age of the universe on current hardware [79].

Therefore, there will always be an insatiable appetite for more computing power and so the the grid will be run at a high level of load. This capacity limitation means that designers have to work with imprecise and lower-fidelity models in the early stages. However, because these models are designed to identify promising parts of the design space that can then be integrated with later design stages such as wind-tunnel tests, lower fidelity results are generally sufficient. However, it is natural that if extra capacity were to become available, it would be used up quickly as designers increase the fidelity of their models or explore a wider search space within each design cycle. Therefore, it is reasonable to assume that there will almost always be work queuing for the grid.

As computational capacity is the limiting factor, the grid administrators stated when interviewed that they wish to minimise overheads as much as possible. The CFD applications are implemented using a Message Passing Interface (MPI) and run over many cores simultaneously [88]. A drawback of the particular applications used is that they are not implemented with checkpointing support. Therefore, this means that tasks are run without pre-emption. Once a task is running, it either runs

to completion, or can be killed manually by an administrator. As currently configured, tasks are run without checkpointing. This means that if a task is killed, it must be re-run from the beginning if the results from its execution are still required.

A requirement of the scheduling approach is to handle scheduling non-pre-emptive tasks well at a range of load levels including full load, and degrade gracefully under overload.

2.2.2 Wide range of job duration

A key aspect of the partner's system is that there is a wide range in the duration of jobs. The small (minute-long) and large (month-long) tasks are run on the same grid infrastructure. The balance between small and large tasks also changes over time, so a scheduling policy which partitions the capacity of the grid for each kind is unlikely to be suitable.

The change in the mix of work in the queue is notable throughout the working day. The results for the smallest tasks may be required the same day so that further design cycles can take place. However, tasks that are not going to finish before the end of the working day may finish anytime before the start of the next working day without any impact on the designers' productivity. The results produced by longer jobs tend to also take longer to analyse by the designers.

The users and administrators stated that it is usually desirable, therefore, to run the smallest jobs during the day and queue the larger ones to wait overnight. However, just because a job is large does not mean it can wait indefinitely or *starve* [168]. Instead, several members of the design team expressed the desire for job response times to be 'fair', which to them meant being proportional to their execution times, which is also noted by Saule et. al. [147]. This principle of proportional fairness is formally defined and given a theoretical foundation by Wierman [168].

A requirement of the scheduling approach is to treat jobs with significantly different execution times fairly, with the aim of having response times be proportional to execution times.

2.3 Software Architecture

The CFD simulations work by breaking a volume of space down into smaller volumes. Then, the flow equations in each volume are solved and the interactions calculated between each volume and its neighbours [88]. This process happens repeatedly until a steady state is reached, which is termed *convergence*. The execution times of tasks cannot be known precisely in advance, because it is difficult to predict exactly how long the CFD algorithms will take to reach convergence.

Once a converged solution has been reached, this is transmitted to several further stages that extract pertinent measures from the solution. Each of these stages can be a different piece of software. The links between pieces of software where data is transferred are known as *dependencies*.

2.3.1 Dependencies

The complete process of executing a wind tunnel in simulation involves several stages. An example workflow is shown in Figure 2.1. The most important and time consuming task is the CFD solver, which calculates the pressure and air flow around a two- or three-dimensional model. However, this is not the only part of the process [61, 150].

The parameters of the simulation must be set up appropriately and any data files transferred to the computing cluster before the simulation can begin. Secondly, the space around the model must be divided into discrete volumes, and this process is known as meshing [43]. In some simulations, the mesh is already given as an input [150]. There are various other processes that can be run before the main solver, such as heuristics that use similar past flow solutions to 'prime' the flow field so as to achieve quicker convergence.

Once the main CFD solver has run, a variety of post-processing applications can extract information about particular features. The two main features of any aerodynamic surface are lift and drag. In addition, the presence and location of shock waves along with areas of turbulence can be extracted. Finally, the solution is usually put through visualisation software so that the designers can more quickly identify desirable or problematic features of the flow field [61, 150].

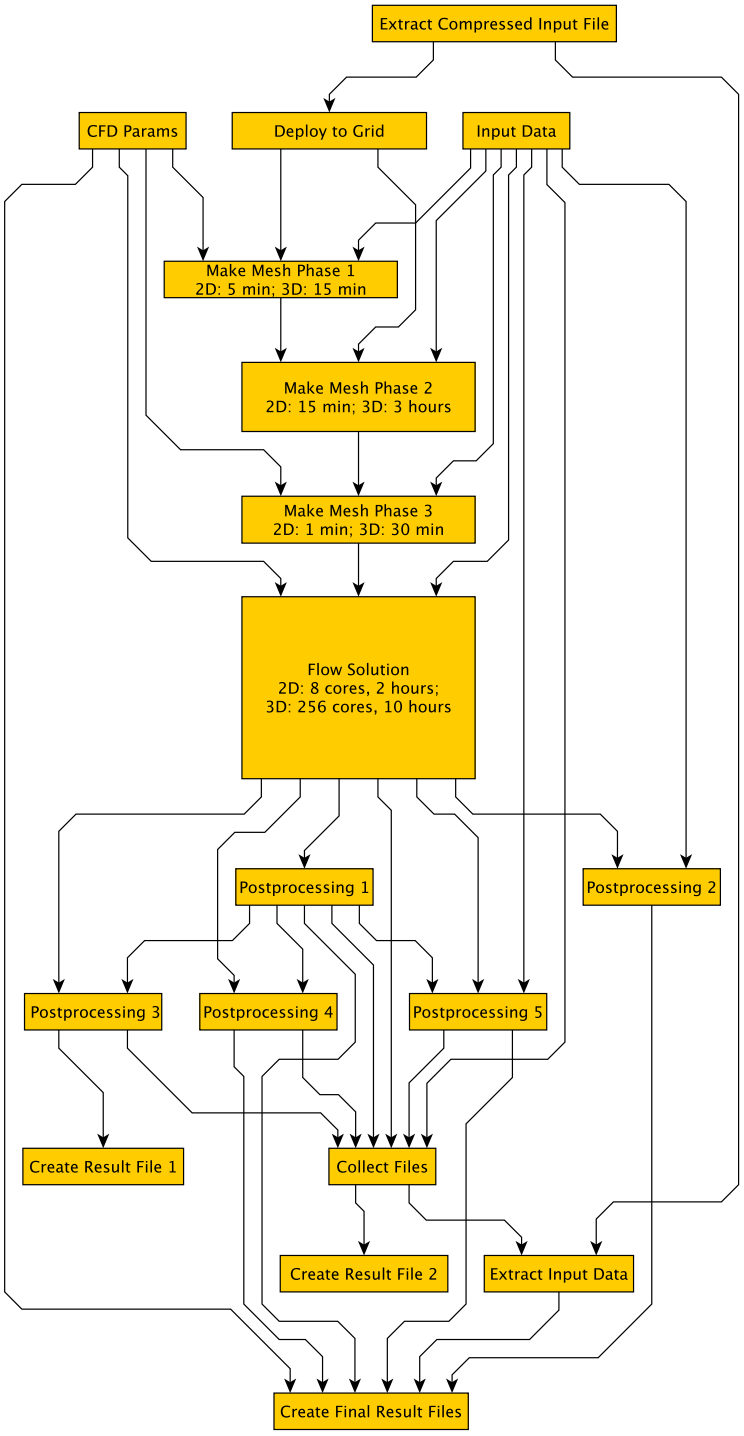
Each part of this process is run by its own application, with data files being transferred between programs in order to ensure the whole process completes. Naturally, it is not possible for the later stages of the process to execute until the earlier processes have completed and the appropriate data has been transferred.

This gives rise to the notion of dependencies - a key feature of the workload. Dependencies mean that the tasks that compose an instance of the workflow (a *job*) must be run in a given order. Where tasks are run on different clusters, any data must also be copied between these clusters before the successor task can run.

A requirement of an appropriate scheduler is that it respects the ordering of tasks necessitated by dependencies.

2.3.2 Estimates of Execution Times

Some progress has been made in the organisation to predict execution times in advance. One designer developed a tool to predict execution times. This used neural network techniques to examine the parameters supplied to the CFD software and



Note: Execution times are from single observed examples for 2D and 3D and can vary significantly.

Figure 2.1: CFD Labelled Workflow

past execution times in order to build a predictive model [103]. However, these estimates can never be truly accurate due to the fact that it is difficult to predict how quickly the algorithms will converge. Users, however, also have an idea of how long their jobs will take, especially because they tend to run a lot of similar ones at a time, and have a detailed knowledge of the size and complexity of the models they are working with. Using the predictive model and the users' own estimates, useful yet inexact estimates can be achieved.

An issue with the current grid scheduler is that it only supports a single input for an execution time estimate. This field is used as an upper bound on execution time, and if tasks exceed this upper bound, they are killed. Because the estimates are never perfect, users set this field to the maximum value possible so that their tasks are never inadvertently killed if the algorithm takes longer to converge than expected. Therefore, execution time estimates are currently not taken into account in the scheduling of work.

A final scheduling solution will need to be as resilient as possible to the effects of inaccurate estimates of execution times.

2.3.3 Bounds on Parallelism

The CFD simulations are highly amenable to parallelisation, because (at the limit), each volume of space could be assigned to its own processor [150]. However, between each step, the processors must communicate with each other before they can continue. Communication delays within the same compute server are negligible because it has one memory address space. However, they can add up to a significant delay between compute servers. These communications are especially sensitive to latency rather than bandwidth, because only a small amount of data needs to be transmitted, but it needs to be transmitted often [88].

The quantity of information to be transmitted to other compute servers at each time step represents the state of the the surface area of the volume held. As surface area grows more slowly than volume, network delays can be reduced by keeping as much volume as possible on one processor. Therefore, the communication between processors working on different volumes of space gives an upper bound on the amount of parallelism that provides a useful speedup [88].

In order to ensure a high level of processing speed, the states of all the volumes on a given machine must be held in Random Access Memory (RAM). The amount of RAM available on each server can limit how many volumes can be processed on a given machine. Therefore, for large simulations, the amount of RAM on each server within a cluster gives a lower bound on how few processors can be used.

Due to these constraints, tasks are generally sized so that they occupy most of the RAM available on a single server. This is another factor that limits pre-emptive

scheduling of tasks. There is not usually sufficient RAM in the compute servers to have more than one task resident in RAM simultaneously. Therefore, each pre-emption would require paging the entire contents of RAM to disk. As disks are orders of magnitude slower than RAM, the overhead of pre-emption is seen to be too high.

The machines that compose the grid clusters are composed of multicore processors, yet different clusters may have different numbers of cores on each multicore chip. The grid administrators indicated that it is most desirable to only have one task at a time running on a compute server. This is to minimise thrashing between applications which could negatively impact performance, along with minimising operating system overheads. Therefore, grid administrators advise users that for best performance they should always submit multicore tasks that use a multiple of the number of cores in every cluster. For example, where a grid might have clusters of dual-, quad- and hex-core processors, multicore tasks may only request cores in multiples of 12.

Parallel tasks can often have a range in the number of processors they can be split over. Where the number of processors is fixed for the duration of the task's execution, the problem of deciding how many processors to allocate to each task is known as *mouldable scheduling* [143]. A good deal of research has already been applied to this problem [147], and would at first glance seem to be applicable here.

In practice however, the restrictions on parallelism mean that there is a limited 'sweet spot' in the tradeoff between desired response time, parallel scaling, network traffic and RAM exhaustion, as noted in McCreary et. al. [120]. Users stated that they tend to have a good idea of the kinds of work they usually run and what its sweet spot is, and are able to supply an appropriate core count for tasks in advance.

A final scheduling approach will be required to consider tasks that run concurrently over multiple cores.

2.4 Hardware Architecture

To run the CFD software, the industrial partner has purchased a large amount of computational capacity. This capacity is geographically distributed and connected using WAN links. As such, it follows the architecture of a computational grid, even though it is all owned by the same organisation.

2.4.1 Unsuitability of the Cloud

Recent years have demonstrated the increasing popularity and flexibility of cloud computing. However, the grid administrators express significant reservations about the suitability of public clouds for the particular workload used. The primary concern

is that of data security, as many of the three-dimensional CFD models of aircraft and their performance results are commercial secrets key to the competitive advantage of the firm.

There are significant technical barriers to cloud adoption as well. Many jobs that are run consume vast quantities of CPU time and produce proportionally vast quantities of data. Cloud providers bill not only by compute time, but also by data transfer costs. The sheer size of the datasets used is felt to render the data transfer costs prohibitive, and the bandwidth available inadequate compared to an in-house platform. The particular hardware architectures and accelerators required by some software packages may not be available in the cloud.

As mentioned above, the CFD tasks that run across several compute servers need to have very low latency between these servers in order to achieve acceptable levels of performance. In practice, this means that tasks need to be allocated to machines physically close together - within the same rack if possible. Cloud computing providers tend to have more widely distributed networks of compute servers and do not give latency guarantees between each server. These latencies can vary widely [13]. This also means that in practice, multi-core tasks can only be assigned to a single cluster, and are limited in the amount of parallelism they can exploit by the capacity of the available clusters.

2.4.2 CPU Architectures

To gain the best performance possible, the CFD simulation, analysis and visualisation software has been extensively optimised for certain classes of hardware. Over time, however, different CPU architectures and instruction sets have been in vogue. This means that the organisation has to run several different hardware architectures to support this software. Migrating existing software to run on other architectures is perceived by the grid administrators to be too costly, mainly because of the lead times involved. Furthermore, certain architectures work with additional accelerator hardware which can provide immense speed increases for software tailored to use it such as Field-Programmable Gate Arrays (FPGAs) [7] and Graphics Processing Units (GPUs) [119].

The CPUs of these different architectures are combined into clusters. The size of these clusters in many locations can be limited by the availability of electrical power and cooling. Therefore, to attain the computational capacity required, the clusters are distributed worldwide.

The power consumption of the clusters is one of the largest costs in the operation of the grid. Therefore, it is usually uneconomic to run processors of previous generations, because their performance per watt is that much poorer. This leads to a particularity in the notion of heterogeneity in this grid. While there are several

classes of CPU architecture present in the grid, each of these architectures tends to run at or very close to the same speed.

These constraints mean that each task usually has a set of clusters that it can run on. The presence or otherwise of accelerator hardware further constrains the clusters available to it. Importantly, the execution time of a task would be similar whichever cluster is used.

A requirement of the final scheduling approach is that tasks must only be scheduled where the necessary hardware is present that is compatible for their execution.

2.5 Grid Management and Scheduling Architecture

The current grid architecture is managed using four different pieces of software to manage distinct kinds of tasks (see Table 2.1). The highest level (L4) consists of the user interface for users to create, parameterise and submit workflows. The next level down (L3) performs dependency management and initiates data transfers between clusters where necessary. Load balancing is the next lower stage (L2), where tasks are allocated to individual clusters. Management within clusters is at the lowest level (L1), where jobs are queued and scheduled onto the compute servers that make up the cluster.

The layers of software have been built up over time, starting with only L1 originally. This means that users can submit tasks at any level. Users for whom performance is particularly important often submit directly to L1 or L2. Partly, this is because there are old scripts and home-grown GUIs that have not yet been updated to make use of the higher levels. However, there are cases of undesirable interaction effects between the upper layers that result in suboptimal performance.

A particular problem for many workflows is that the dependency management takes place above the level of task scheduling, rather than integrated into it. Any tasks without dependencies are submitted by L3 to L2 and on to L1 first, because they can run immediately. However, L3 only submits subsequent tasks to the lower levels

Level	Description	Tool Used
L4	User interface for creating, parametrising and submitting workflows	ModelCenter [135]
L3	Dependency and data transfer manager	Synfiniway [62, 69]
L2	Load balancer	LSF Multicluster [171]
L1	Task Scheduler	LSF [136, 138]

Table 2.1: Grid Management Levels

once their predecessors have finished, in order to ensure that the dependencies are respected.

In the context of this work, the *multiple waits problem* is a cause of low responsiveness for jobs, and is present when a job's total pending time is greater than the time it would take for a cluster to consume all the work in the queue. It can be manifested where tasks are only added to the queue once all of their dependencies have been completed. The problem is present when the lengths of the queues on the clusters are long. By only submitting tasks to the back of the queue as their dependencies are satisfied, the whole job ends up having to wait the length of the queue multiple times before it can complete. This causes responsiveness to be far lower than if the job only had to wait the length of the queue once.

As the existing FairShare policy at L1 does not consider the structure of dependencies, it suffers from the multiple waits problem. In interviews, users expressed their frustration with this ongoing issue.

As currently set up, the scheduler in L1 is not able to make use of execution time estimates because there is no easy way for users to supply this information. Therefore on average, tasks will tend to wait in the queue for the same amount of time, whatever the scheduling policy chosen. This in turn means that responsiveness performance across the range of execution times is equivalent to that of the First In, First Out (FIFO) scheduling policy.

In order to try and suggest improvements to the industrial partner's set up, it is necessary to understand the existing scheduling policy, which is known as FairShare. FairShare is the ordering policy that operates within each cluster, as part of the LSF software that manages the grid at level L1. LSF also provides a task dispatcher (the allocation part of the list scheduler) and monitoring utilities in L1.

2.5.1 Definition of the FairShare scheduling policy

The fundamental aim of FairShare is to achieve fairness with respect to utilisation. As configured in the industrial partner, past usage is not taken into account, so it only takes into account instantaneous utilisation when calculating shares.

FairShare prioritises tasks based on a hierarchical partitioning of the cluster resources. The fundamental idea is that each department, group and user has a *share* of the cluster resources allocated to them. The priority of each task is based on how much capacity each user/group/department is currently using on the cluster, compared to their 'share'. The queue is sorted in increasing order of priority - tasks with a low numerical value for priority are run first.

The shares are organised in a tree. The root of the tree has a 100% share of the cluster. Each branch of the tree divides out this share until the leaves of the tree are reached. These leaves represent the users. The leaves of the tree do not all need to be

at the same depth. Each job in the system is assigned a path in the tree, which must be a leaf.

A formal definition of the FairShare equations is given in Table 2.2, derived from the descriptions in the LSF Fairshare documentation [136] and the original paper by Kay and Lauder [93]. The shares are defined in advance by a share tree, such as the example tree given in Table 2.3. T is the set of tasks in the workload and f is a node in the share tree. The number of cores used by each user's tasks will change over time depending on the state of the cluster. This means that the priorities of queueing tasks are dynamic, and must all be recalculated every time a scheduling decision needs to be made.

$$f_{.\texttt{parent}} = \begin{cases} \in \quad T \\ \emptyset \end{cases} \tag{2.1}$$

$$f_{.\texttt{children}} \subset T \tag{2.2}$$

$$f_{.\texttt{shares}} = \begin{cases} \sum_{s \in f.\texttt{children}} s_{.\texttt{shares}} & \text{if } |f_{.\texttt{children}}| > 0 \\ \in \mathbb{N}_{>0} & \text{otherwise} \end{cases} \tag{2.3}$$

$$f_{.\texttt{used}} = \begin{cases} \sum_{c \in f.\texttt{children}} c_{.\texttt{used}} & \text{if } |f_{.\texttt{children}}| > 0 \\ \in \mathbb{N}_{>0} & \text{otherwise} \end{cases} \tag{2.4}$$

$$f_{.\texttt{cluster\_proportion}} = \begin{cases} \frac{f_{.\texttt{shares}}}{f_{.\texttt{parent.shares}}} \times f_{.\texttt{parent.cluster\_proportion}} & \text{if } f_{.\texttt{parent}} \neq \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{2.5}$$

$$f_{.\texttt{priority}} = \frac{f_{.\texttt{used}}}{C_{\texttt{cores}} \times f_{.\texttt{cluster\_proportion}}} \tag{2.6}$$

Table 2.2: FairShare Definitions

Name	Shares	Cluster Proportion	Cores Used (example)	Priority
Root	100	1	91	0.91
Group1	60	0.6	55	0.92
User1	25	0.25	30	1.20
User2	25	0.25	20	0.80
User3	10	0.10	5	0.50
Group2	40	0.40	36	0.90
User4	22	0.22	18	0.82
User5	6	0.06	8	1.33
User6	12	0.12	10	0.83

Table 2.3: FairShare Tree Example

2.6 Critique of FairShare

FairShare was originally designed by Kay and Lauder [93] at the University of Sydney, in order to give a fair allocation of compute resources to different users. However, its design is based on a particular set of assumptions. It was designed for fairly dividing the computational resources of a single mainframe between the users of the Computing department.

2.6.1 Assumptions of workload and user characteristics

Kay and Lauder [93] stated their workload as being "almost exclusively interactive and had frequent, extreme peaks when a major assignment was due. On a typical day, there were 60-85 active terminals, mainly engaged in editing, compiling and (occasionally) running small to medium Pascal programs". This statement clearly demonstrates a relatively homogeneous workload where most tasks run for about the same amount of time. Furthermore, most tasks were run interactively. The responsiveness required of interactive tasks is on a completely different scale to that of large computational batch jobs [127]. Furthermore, there can be no queuing for interactive tasks. Instead, all interactive tasks are run concurrently, and the interactive responsiveness depends mostly on the load placed on the cluster. The FairShare policy is designed to ensure fair interactive responsiveness between users and groups.

FairShare is explicitly designed to encourage users to spread out the load they place on the machine. This is understandable for interactive work running on a machine with relatively limited resources, a 1988 VAX [93].

However, aircraft designers do not require interactive performance, and do not want to spread out their submissions of work. Instead, they want the fastest turnaround time possible. The groups for whom responsiveness is most critical submit many small jobs during the day. This tends to quickly use up their fair share, and leave subsequent jobs queueing. Because FairShare is configured to only consider instantaneous and not historical usage, then the users who submit small jobs will suffer overall. This is because all of their work will finish quickly after the end of the working day, and they will make no use of their share overnight. Overall, therefore, these users get significantly less than their fair share.

This issue highlights the difference in perception between users and administrators: while administrators care most about getting high utilisation on the cluster, users care most about the responsiveness of their tasks, irrespective of what else is running. Users wish to have fairness with respect to responsiveness, whereas the system is configured to give fairness with respect to utilisation. Instead of this

situation, a requirement of an appropriate scheduling solution is that it ensures responsiveness even with a workload that has distinct peaks in submission loads.

2.6.2 Assumption of pre-emption

Running interactive tasks concurrently on a single-CPU mainframe can only be achieved using a pre-emptive scheduler. FairShare is designed so that in the long term, the proportion of the CPU used by each user is equal to their fair share. To smooth out the fact that different users worked at different times, FairShare is designed to include a function that calculated share based on past as well as current usage. This is designed so that users who had previously been consuming more of their fair share would be given a lower priority later on. Past usage is evaluated using a decay function appropriate to the workload.

In the industrial set-up, however, FairShare is used on a non-pre-emptive multiprocessing system. Extending FairShare to a multiprocessing scenario is trivial and was addressed in the original paper by Kay and Lauder [93]. However, using FairShare in a non-pre-emptive system can lead to several undesirable effects.

An issue noted by Kay and Lauder [93] in their design for FairShare is that in moments when the computing resources are idle, a user can start work running and vastly exceed their share, because it is not competing with other tasks for resource. In a pre-emptive system, when other users start their work, the scheduler pre-empts the original user's work, and gives it only the time-slice of the computing resources appropriate to its share. However, in the industrial, non-pre-emptive system, the user's tasks will continue running until completion. This means that a user can attain a significantly higher proportion of the resources than their fair usage. Without pre-emption, the FairShare scheduler can struggle to ever re-balance the load running back to a 'fair' state.

This issue is exacerbated by the fact that in the industrial set-up, past usage is not taken into account. Instead, only the instantaneous state of the system (counted by number of cores used) is taken into account when calculating shares. This meant that even if a user were able to exceed their fair share with a large submission, once that submission had finished, their priority would be back to normal. In effect, there is no penalty for exceeding their fair share. Users who require short cycle times tend to submit tasks that require more cores, because for them responsiveness is so key. Yet because their instantaneous usage (by cores) is high, these short but highly parallel tasks are in effect de-prioritised in the queue.

Unsurprisingly, the users that were interviewed in industry were well aware of this anomalous scheduling behaviour. Users know that the time the clusters are most likely to have idle capacity is early on Monday morning. If they had a large or urgent piece of work to run, they would come in particularly early on Monday to

submit their jobs, so that their work would consume a larger-than-fair share of the cluster. Without pre-emption, this behaviour meant that this worked out well for the 'early-bird' users, but everyone else using the cluster suffered through lower responsiveness. Without fear of a scheduling penalty, users continued this behaviour. In effect, long-running tasks were prioritised when users exploited this anomaly, as the exceeding of fair share lasted only as long as the tasks were still running. This gave rise to a significant negative productivity impact on the users needing short cycle times and who submit many smaller tasks.

An opposite kind of problem can also occur because of the lack of pre-emption. There is nothing to stop a job requiring a large number of cores to be submitted by a user whose share is less than that number of cores. As long as the grid is busy, this job might never start because the user would never attain enough share to acquire the cores needed. This anomaly demonstrates that FairShare can suffer from starvation.

As FairShare was originally designed as a pre-emptive scheduler, there was no need for back-filling (where short tasks can 'jump' the queue if a large task is waiting for a large number of processors to become free [104]). However, as currently set up in the industrial cluster, no back-filling is used. For example, if there are 29 cores free on the cluster, but the task at the head of the queue requires 32, the scheduler will wait. It will not consider tasks requesting fewer cores than this limit. This set-up is due to the constraint that no execution time estimates are made available to the scheduler. To an extent, this means that CPU capacity is wasted, because cores are often left idle while waiting for enough cores to become free so the next task in the queue can start.

However, the lack of backfilling is not as much of a problem as it might seem. In clusters at the scale observed in industry, the number of cores left idle compared to the size of the cluster is small. The scale also means that there is a high turnover of tasks, meaning no task will ever wait too long for enough cores to become free.

2.6.3 Assumption of global knowledge

The FairShare policy runs assuming it knows the state of all computing resources. However, there is an independent instance of FairShare on each cluster. This leads to a problematic interaction between the load balancing software and FairShare. The load balancer, as currently set up, allocates incoming tasks to the lowest-utilised cluster, as measured by the number of tasks in the queue. To optimise the responsiveness of a user's workflows, they should each be distributed to different clusters, to consume the user's share on each. The problem is that the load balancer does not take account of the FairShare allocations of the users in its load balancing decision. Therefore, many workflows from the same user might be directed to the same cluster because it has the shortest queue. Responsiveness for this user would suffer, because all their tasks would be competing for the share of a single cluster

with a shorter queue, instead of getting their fair share across all clusters, including the busy ones.

This behaviour is further exacerbated by a policy of the load balancer that seeks to avoid task starvation. If a task has been assigned to a cluster for a fixed amount of time without having started execution, the load balancer removes it from the queue on one cluster, and assigns it to a random other cluster. If the task has not run for the delay time on that other cluster, the load balancer will move it again. Because the allocation is random, there is the chance that the load balancer will move a task back to a cluster it had previously been queuing on. The issue is that the task is then added to the back of the queue on each cluster each time it is moved. During periods where the system is under high load, some tasks from users with low share can keep being passed around from cluster to cluster and never start. This is a classic example of starvation, and is a particularly undesirable situation where high responsiveness is required.

A requirement of an industrially-suitable scheduling architecture is that it can operate at different levels, where each level has different amounts of information available. A centralised scheduler can not be assumed to be able to have global knowledge, because communicating such detailed information could saturate the network links between clusters.

2.7 FairShare Aware Load Balancing

As part of the process of workload characterisation, a tool was written by the author to extract and analyse the logs and current state from the cluster schedulers (L1). One form of analysis that was applied was to extract what each user was running on each cluster. This meant that the allocation of cores to users could be determined, and hence what each user's fair share priority would be for jobs on that cluster. A command-line tool was created for users to query the fair share priorities, so they could know which cluster would give a newly-submitted task the highest priority.

This tool proved popular with users, as it allowed them to bypass the sub-optimal allocations of the load balancer. They could either submit their tasks directly to the best cluster in L1, or specify an appropriate tag which directed L2 or L3 submissions to the desired cluster. Once the author had created the command-line version of the utility, others within the organisation integrated the code into the main submission user interface. This tool is now in daily production use and is one of the prominent industrial contributions of this Engineering Doctorate. A screenshot of this utility is shown in Figure 2.2.

```
Usage: live_hpc_fairshare_viewer.py [options]

Options:
  -h, --help            show this help message and exit
  -f FILENAME, --filename=FILENAME
                        The input file from which to extract data, if
                        previously stored. Not needed if live view desired
                        (optional)
  -u USERNAME, --username=USERNAME
                        The username for which to analyse fair share
  -g GROUP, --group=GROUP
                        The group that the username is a member of (optional)
  -q QUEUE, --queue=QUEUE
                        The relevant queue (optional, defaults to [redacted])
  -c CLUSTERS, --clusters=CLUSTERS
                        A list of clusters to compare, separate with ','
                        (optional)
  -n NUMBER_CORES, --number_cores=NUMBER_CORES
                        projected core count of a new job, helps with priority
                        calculations (optional, defaults to 0 which just
                        returns the current instanteous priority)
  -v VERBOSE, --verbose=VERBOSE
                        Print more detail on the cluster priorities (including
                        relative priority values)
[redacted]$  python live_hpc_fairshare_viewer.py -f ./test/new_extract.txt -u [redacted] -n 128
Priority order for  User  submitting a job of 128 cores
Cluster                 Group
[redacted]4 Location    Queue ID    ug_aec
[redacted]3 [redacted]  [redacted]  ug_aec
[redacted]4 [redacted]  [redacted]  ug_aer[redacted]0
[redacted]3 [redacted]  [redacted]  ug_aer[redacted]0
[redacted]4 [redacted]  [redacted]  ug_aer[redacted]1
[redacted]4 [redacted]  [redacted]  ug_aer[redacted]2
[redacted]3 [redacted]  [redacted]  ug_aer[redacted]2
[redacted]3 [redacted]  [redacted]  ug_aer[redacted]1
[redacted]3 [redacted]  [redacted]  ug_aer[redacted]3
[redacted]4 [redacted]  [redacted]  ug_aer[redacted]3
[redacted]2 [redacted]  [redacted]  ug_aec
[redacted]2 [redacted]  [redacted]  ug_aer[redacted]0
[redacted]2 [redacted]  [redacted]  ug_aer[redacted]2
[redacted]2 [redacted]  [redacted]  ug_aer[redacted]1
[redacted]2 [redacted]  [redacted]  ug_aer[redacted]3
```

Note: to protect the interests of the industrial partner, certain fields have been obscured and the list of clusters/queues/groups is truncated.

Figure 2.2: User FairShare Priority by Cluster

2.8 Summary

This chapter describes the industrial context of the research in detail. It shows how the aircraft design process is developed through design cycles. These cycles have lengths varying from minutes to months, and give rise to equivalent cycles of computational load. While this chapter described these cycles qualitatively, they will be analysed quantitatively in Chapter 4. These cycles are due to the hierarchical nature of aircraft design, a process which is followed by any other kind of engineering design. The desire of designers to have the grid be responsive and fair when executing their workflows is motivated by the organisational need for good quality solutions and a short time-to-market.

The complex nature of engineering design, and the many factors relevant to evaluating a solution mean that it is necessary to run intricate computational workflows. The software and hardware architecture as deployed by the industrial partner to run these workflows is described. The dependencies and network

transfers inherent in executing workflows pose problems for traditional scheduling policies designed around getting the best utilisation out of precious computing power.

The existing scheduling policy, known as FairShare, is shown to have been designed with assumptions that no longer hold in this industrial context. Firstly, FairShare is designed for interactive workloads where most tasks were of similar size and duration. FairShare is designed to give users incentive to spread their load around peak times by reducing the responsiveness of users with low share.

Secondly, FairShare is designed for a system with pre-emption, so that the fairness of the tree is respected at all times. It is shown that FairShare can be affected by unfair allocations in certain situations which cannot be resolved until tasks have finished, because of the lack of pre-emption.

Finally, FairShare allocates priorities while assuming it has global knowledge, although in the industrial grid each cluster ran a separate instance of FairShare. The conflict between the load balancer and the underlying FairShare algorithm can lead to sub-optimal responsiveness and fairness. This shortcoming of the load balancer was addressed by the author by implementing a tool for users to find the most appropriate cluster to submit to depending on their FairShare and hence their relative priority, rather than simply by cluster load.

This chapter further discusses the requirements of a scheduling policy for it to be suitable for industrial implementation. The policy must achieve responsiveness for non-pre-emptive jobs even under high load, and degrade gracefully when the system is overloaded. It also needs to handle jobs with a wide range of execution times fairly, with the ideal situation being where response times are proportional to execution times. Tasks must be assigned to resources that respect their hardware requirements, and in an order that respects the dependencies between tasks of the same job. The scheduler must do this whilst minimising the impact of inaccurate estimates of execution time. The scheduler must also be able to schedule work across a whole grid without any one point in the grid being able to have full knowledge of the state of the grid and the work queueing.

Chapter 3

Literature Survey

3.1 Context and complexity

The scheduling of tasks onto computing machinery is a field of study as old as computing itself [94]. Research into scheduling goes back even further in the context of large-scale project management [76]. Therefore, the literature on scheduling is large, in addition to the fact that different scheduling problems all have different priorities to try and meet. Surveying the entire body of literature on scheduling would be prohibitive. Therefore, this literature survey will mainly focus on scheduling policies already intended for use in distributed or grid computing scenarios.

In an ideal world, careful scheduling would ensure that the grid's resources are always used to full potential. However, with currently known algorithms, optimal allocation and mapping is intractable for anything more than trivial workloads. This is because the scheduling problem has been proved to be NP-Complete for all but very restricted versions [59], and even the allocation stage of scheduling is equivalent to bin-packing, which is also NP-Complete [65].

Scheduling for a multiprocessor with dependencies was proved to be NP-Complete in 1975 by Garey and Johnson and Ullman [64, 163]. Furthermore, it has been proven that no polynomial-time algorithm can deliver an optimal assignment in all cases [59]. Therefore, optimal scheduling is intractable at the scale of grid systems. This is especially the case when the further complicating factors of heterogeneity [117] and network delays [36] are also present.

Instead, heuristic policies are required. These will always have limitations, and have been proven to have upper bounds on how close they can come to an optimal schedule [3] in the general case. Many heuristic scheduling policies have been proposed (see [22, 97, 117] for surveys of policies suited to the more general problem of distributed multiprocessor scheduling). Each heuristic tends to be suited for particular platforms and workloads. Therefore, heuristics must be evaluated in

order to identify their strengths and applicability to the context in which they are employed.

The rest of this literature survey will examine heuristic approaches that have been applied to the grid scheduling problem [36]. The study of this problem is highly relevant, because grids are increasing in size and heterogeneity all the time, and the cloud computing trend is moving work away from being done on local workstations and instead placing it onto the grid architectures underlying cloud computing.

Firstly, a taxonomy of scheduling architectures will be presented and critiqued as to their relevance to the industrial context. Secondly, various kinds of information that are considered by scheduling policies in the course of producing a schedule are considered. Finally, the design of scheduling policies with specific user aims in mind is considered.

3.2 Scheduling Architectures

The scheduling involved in managing a grid involves two distinct activities. *Allocation* is where tasks are assigned to processing nodes in the network. *Ordering* is deciding, within the constraints of dependencies, what order tasks should be run on the grid and on each processing node within the grid [36]. *Scheduling* is the combination of ordering and allocation, in order to produce a *schedule*, an allocation of tasks to processors and an ordering of tasks on each processor. Ordering and allocation can happen separately or together, depending on the architecture of the scheduler. Schedulers can work in one of two ways: *static* or *dynamic* [35].

Static schedulers produce a schedule in advance, which is run on the hardware later. Static scheduling policies need to know all the work to be run in advance, along with estimates of execution times. Schedules produced statically are especially useful where the same schedule of work is run repeatedly, and examples of these can often be found in embedded systems [113].

More usually, static schedulers are run in 'batch' mode [16, 117]. This approach originated in mainframe systems where the aim was to run a set of processes overnight and to finish them all in time for the start of the next working day [41, 42]. Working in batch mode, static schedulers batch up submissions until a set volume of work or a set time is reached. The static scheduler then produces a fixed schedule of all the submissions together which is then executed on the available resources. If new work arrives while a schedule is running, it is held and added to the next batch of work.

Static schedules are usually designed to minimise the time taken to complete the execution of the whole batch, a metric known as 'makespan'. Work in a static schedule is not re-scheduled once execution has begun. Some approaches, such as the task

migration phase of Lo [114], or schedule post-processing [52, 104, 158] can seek to improve on a pre-existing static schedule before runtime begins.

Batch static schedulers may be suitable for contexts where all the work runs to the same cycle time. However, as described in Section 2.2.2, there are a variety of cycle times ranging from minutes to months required by the industrial designers who use the grid. Therefore, there is no way of finding a single cycle time that could satisfy both the responsiveness requirements of the shortest jobs while also fitting in the execution times of the largest jobs. This precludes the use of static scheduling policies.

The alternative to static scheduling is known as dynamic scheduling. Rather than group work into batches, a queue of work is maintained. As resources become free, the scheduler selects tasks from the queue and allocates them to resources. Submission of work to the queue along with the scheduler activities of selection and allocation happen continuously. Dynamic schedulers can use a wide variety of ordering algorithms to prioritise the work in the queue, and these do not necessarily depend on having execution time estimates available. Nevertheless, improving scheduling by using such estimates if they are available will be the topic of Chapter 6.

An advantage of dynamic scheduling policies is that they can be run as if they were static policies by supplying a batch of work and generating the schedule by simulating what the system would do if the jobs were really running. There is no such option for static schedulers to behave dynamically.

While the static scheduling approach is not suitable for direct application to the industrial scenario, the heuristics used to prioritise tasks can still be useful to survey. This is because it may be possible to use these prioritisation algorithms as part of the ordering process in a dynamic scheduler.

3.2.1 List Schedulers

One of the oldest classes of schedulers is that of List Scheduling [71]. List Schedulers keep the ordering and allocation activities separate, with a distinct policy for each [116, 117]. Scheduling takes place by considering each task for allocation in the order specified by the list. This continues until either the queue is empty or all the computational resources are consumed. The scheduling process is triggered again every time a task is added to the queue or a task completes (a *scheduling instant*) [72]. Pseudocode for this approach is shown in Algorithm 3.1.

The flexibility afforded by the separation of concerns between the ordering and allocation policies means that list schedulers have been tailored to many different situations. The most basic ordering policy is one that requires no re-ordering of the task queue, and instead runs tasks in First Come First Served (FCFS) order (also known as First In First Out or FIFO) [71]. Most List Schedulers in the literature use

an Earliest Start Time (EST) or Earliest Finish Time (EFT) [72, 160] allocation policy, depending on whether the platform considered is homogeneous or heterogeneous. However, determining EFTs can depend greatly on the configuration of the underlying hardware and on the model of the system considered.

List schedulers are naturally dynamic, where scheduling takes place at the same time as tasks are executed. By prioritising tasks in the queue at each scheduling event, then list scheduling is not bound to a single cycle time as static schedulers are. Instead, a prioritisation scheme could choose to run the shortest tasks first, giving them better responsiveness [125]. The list scheduling architecture is suitable for the industrial scenario, as the existing FairShare policy is a list scheduler.

Because of their structure, list schedulers can be easily chained, where the allocation of a high-level LS is to the lists of lower-level LSs, rather than to actual hardware resources. The typical network topology of datacentres and Grid systems is that of a *Thin Tree* [126]. A thin tree network has links further away from the root being faster. This reflects the high-speed interconnects available between physically proximate clusters and slower WAN links between geographically distributed datacentres. The chaining of LSs means that they can be composed into a tree structure that matches the underlying hardware platform. This aids in clarity for the scheduling policy, as well as making efficient use of the network links, as the communications flow along the platform's natural tree structure.

An issue with the list scheduling architecture when run dynamically is that if the head of the queue cannot run, the rest of the queue must wait. This can lead to lower-than-optimal utilisation, especially if there are tasks later in the queue that could run immediately. Backfilling is often proposed as a solution to this [104], but backfilling is only possible if the list scheduler is being run in a static or batch way. Therefore, the ordering policy used has a large impact on the ability of the list scheduler to deliver good performance metrics. Poor or overly-simplistic ordering policies may not reflect the prioritisation that users want.

List scheduling policies by default do not consider dependencies. Instead, they treat the queue of work as entirely independent tasks. To take dependencies into

Algorithm 3.1 Pseudocode for a typical List Scheduler

```
A is the set of all tasks
P is the set of all processors
O(A) returns an ordered list of A
for each task A(i) in the order supplied by O(A) :
    determine which processor P(j) gives either :
        Earliest Start Time
        or
        Earliest Finish Time
    assign task A(i) to processor P(j)
```

account, a system where only ready tasks can be added to the queue must be implemented. However, this can lead to the multiple waits problem if the ordering policy ignores dependencies.

An advantage that static scheduling policies have is that they know their workload in advance, and are therefore known as *clairvoyant* [91]. An issue with dynamic policies such as list scheduling is that they will always be suboptimal with respect to makespan compared to clairvoyant policies. Formally, it has been shown that no non-clairvoyant policy can achieve an average makespan shorter then $4/3$ that of a clairvoyant policy [91], where the average is across the entire space of inputs. However, in the industrial scenario it is impossible to know the workload in advance anyway, because users submit work continuously. Therefore the theoretically-better results of static clairvoyant schedulers are unattainable in the industrial scenario. Furthermore, focussing on the optimality of the schedule makespan is contrary to the desires of the users, as they care far more about the responsiveness of their jobs.

3.2.2 Generational Schedulers

Generational Schedulers (GS) work in a similar way to list schedulers. GS work by only considering a subset of the ordered list at any time. The whole subset is then allocated as if it were the entire list. This process is repeated as many times as is necessary. This is shown in Figure 3.1, reproduced from Carter et. al. [34]. As with the list scheduling architecture, the GS architecture requires an ordering and an allocation policy to be defined.

GS are essentially a kind of batch list scheduler, although they only schedule a subset of the queue with each batch, rather than the whole queue. The construction of the subset can be made in any way desired, but traditionally only selects tasks that are immediately ready to run, that is, ones that have all their dependencies satisfied [34].

A true batch list scheduler would run another batch of work once the whole of the previously scheduled batch has completed, although this is unsuitable for the industrial scenario due to there being no satisfactory batch size. GS incorporates somewhat more dynamic behaviour and instead re-runs the GS every time a running task finishes [34]. At that point, it discards the previously created schedule and adds any tasks that have not yet started back into the current batch. With this approach a task may actually be scheduled to several different processors before it actually starts execution. Yet the industrial scenario considers a geographically distributed grid. In a distributed system, the data required for tasks must be transferred to the appropriate location before the task can begin execution there. Rescheduling of the same task to different locations before it actually runs is likely to

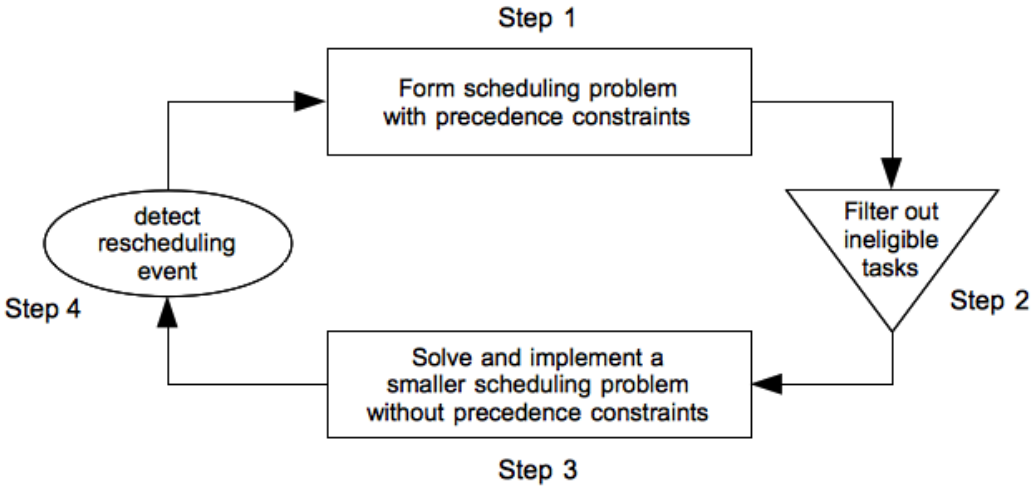


Figure 3.1: Generational Scheduler structure from Carter et. al. [34]

mean that a significant amount of network traffic is wasted, although this may only be an issue if network capacity is a limiting factor.

The Opportunistic Load Balancing (OLB) algorithm is a simple greedy generational scheduler [34, 45, 117]. This policy simply assigns each ready task in FIFO order to the processor on which it will start earliest. Topcuoglu et. al. [160] noted that this does not tend to produce schedules with optimal makespan. In addition, FIFO ordering will lead to the multiple waits problem as well as causing the responsiveness of shorter tasks to suffer relative to that of larger ones.

Liang and Jiliu [111] extend GS to consider two sets of tasks at each generation: those that have tasks still dependent on them (sources), and those that do not (sinks). The sinks are then all scheduled before those tasks which are sources. This policy could be unhelpful in a situation of high load where there are some independent tasks in the workload. These independent tasks would be prioritised over the sources, even if the sources had been waiting much longer. This could prevent the sources from starting, leading to poor responsiveness overall.

3.2.3 Task Duplication Schedulers

Given a set of tasks which must communicate and a network of finite speed, sometimes tasks will be delayed from starting due to data having to be transferred across the network. While the task is waiting, therefore, the processor it is assigned to is idle. Task duplication approaches understand this, and detect if a processor would be idle waiting for data for longer than it would take to simply run the predecessor task again [154]. Task duplication schedulers will therefore run a task twice on different processors in order to avoid the time penalty incurred by the network transfer. Task Duplication schedulers are static schedulers [14, 140, 154].

The original task duplication algorithm is CPM, developed by Colin and Chrétienne [44] for a model of unbounded processors. Pseudocode for this is shown in Algorithm 3.2. A proof is given by Colin and Chrétienne [44] that this algorithm, of polynomial complexity, produces an optimally-low makespan for task sets where communication costs are strictly smaller than computation costs. Performance improvements over the original algorithm are given in Ahmad and Kwok [2], Baruah [14] and Ranaweera and Agrawal [140].

Task duplication approaches have some serious drawbacks. These algorithms are only proved to be optimal for makespan where communication costs are less than computation costs [2]. Yet the greatest benefits they are likely to achieve would be when communication delays are greater than computation delays. Furthermore, the most difficult part of the scheduling problem comes when communication and calculation costs are roughly the same [148].

This means that these algorithms are only suitable for a portion of the possible scheduling problems. This is a significant limitation because it may not be possible to determine in advance whether or not a scheduling problem holds to the restriction that communication costs are strictly less than computation costs. This is even more true in a heterogeneous system, where average cases can be significantly different from worst cases. It is stated by Ahmad and Kwok [2] that the schedules created through task duplication algorithms tend not to be robust against differences between estimated and actual execution time, a critical flaw in real-world systems where execution times may not be known accurately in advance.

Another fundamental assumption in the task duplication algorithms is that there are no tasks from other jobs that can be run while a given task is waiting for data. That is, that they execute alone or on a very lightly loaded system. At the level of a whole grid, however, throughput is as important as response time. Duplicating tasks of the current job will cause tasks of later jobs that would otherwise have started on the idle processors to have their start times delayed. This may decrease the response time of the current job, but would also decrease overall throughput. This would cause overall response times to increase even though each job's response times may decrease. As

Algorithm 3.2 CPM Scheduling

```
A is an ordered set of all tasks
Determine the Earliest Start Time of each task without network delays
Sort A by decreasing Earliest Start Time
Determine critical paths for all a ∈ A :
    Find b, the latest − finishing predecessor of a if exists
    if b exists and is not already part of a critical path :
        Critical − link a with b
Allocate each critical path to a processor
```

high throughput and low response time for a whole workload on a grid is important, as for the reasons above, task duplication scheduling is unlikely to be suitable for scheduling at a grid level.

3.2.4 Clustering Schedulers

Clustering schedulers [19, 67, 114] are a different method to attempt to minimise the contribution of network transmissions to turnaround times. Clustering schedulers work by using the pattern of dependencies to identify clusters of communicating tasks. These clusters are allocated to the same cluster or to sets of machines that have low network costs between them. Clustering schedulers are usually static schedulers.

Lo [114] describes a graph clustering algorithm based on Max Flow/Min Cut theory where the edges in the dependency graph are weighted to represent data volumes. Chen et. al. [38] present an algorithm that combines a Generational Scheduler with a graph clustering postprocessor. Their clustering postprocessor weights nodes as to the computation time and network costs of themselves and all their predecessors. They then use an algorithm akin to greedy depth-first search to identify clusters, like that in Bittencourt et. al. [19].

A family of clustering algorithms for unbounded processors are described in Gersaoulis and Yang [67]. This family iteratively coalesces individual tasks onto processors, and pseudocode for this is shown in Algorithm 3.3. The algorithms differ in their means of efficiently identifying which pairs should be considered first, in order to most rapidly coalesce clusters. The 'DC' and 'MCP' algorithms presented identify the critical paths (longest paths through a workflow) and cluster these first. When clusters hold more than one task, the MCP algorithm uses decreasing sum of successor execution times to form an ordering. This prioritises the starting of new jobs over finishing those that have already begun.

One issue with clustering algorithms is that in their aim to minimise network traffic, they can end up clustering tasks from the same job onto too few processors. This lowers network costs but because this reduces the amount of parallelism that

Algorithm 3.3 Clustering Algorithm from Gerasoulis and Yang [67]

```
A is set of all tasks
Initialise all clusters C to only hold one task
Calculate workload makespan
For every pair (C^i, C^j) of clusters :
    if coalescing (C^i, C^j) does not increase makespan :
        Coalesce (C^i, C^j)
        Recalculate workload makespan
```

is extracted from the job, responsiveness can suffer. Lo [114] uses an 'interference' parameter on jobs that are assigned to the same processor to try to mitigate this effect.

Some batch scheduling policies have been proposed that use clustering ideals to minimise network delays. A natural way of doing this is to allocate all the tasks on the critical path of a job on to the same processing node. This algorithm is presented in Topcuoglu et. al. [160], where it is termed Critical Path On a Processor (CPOP) and also in Bittencourt [18, 19] where it is termed the Path Clustering Heuristic (PCH). These approaches work by performing depth-first searches from source tasks to sink tasks that are greedy with respect to the edge weights.

While the bulk of their analysis is related to the makespan of the workload, Bittencourt [18] considered the impact of using their PCH algorithm on fairness. They show that PCH combined with interleaving of tasks is able to deliver a fair distribution of job slowdown, a measure of responsiveness. However, their analysis only considers the scheduling of 10 workflows on 2, 10 or 25 processors. Further analysis would be needed to examine whether their approach would provide the same results with the industrial workload, with tens of thousands of workflows on a large-scale grid platform.

A problem with the approaches of Topcuoglu et. al [160] and Bittencourt [18] is that they assume tasks are all single-core, and assume networking delays occur between all cores in a system. In the industrial scenario, there are no network delays within clusters, because each cluster uses a single networked file system for disk storage. However, the approach could be extended to consider multicore tasks being assigned to the same cluster.

An assumption made by the clustering schedulers examined here is that the platforms are homogeneous in architecture, in the sense that all tasks can run on all processors. This is not the case in the industrial scenario, where some applications in a workflow are limited to a particular kind of hardware. This leads to unavoidable network delays as the data must be copied between clusters of different hardware architectures. The CPOP and PCH greedy heuristics would not be able to work directly in this situation, because it may not be possible to schedule the critical paths on a single cluster because of architectural restrictions. With a suitable extension to consider architectures, these approaches may be relevant to the industrial scenario.

Most clustering heuristics are static schedulers, which preclude them from being directly relevant to the industrial scenario. However, the clustering aspect of the problem is entirely related to the allocation phase of scheduling. Therefore, there is no reason why it could not be integrated into a fully dynamic scheduler, such as a list scheduling algorithm. Furthermore, because the volumes of data produced by the industrial workflows can be very large, minimising network transfers is an important goal. To this end, a combination of the extensions to consider multicore tasks and incompatible architectures is included in the load balancer presented in

Section 5.3. This load balancing policy keeps tasks from the same job of the same architecture together, so they will always be assigned to the same cluster.

3.2.5 Search-Based Schedulers

Due to the fact that the dependent task scheduling problem is NP-Complete, the space of possible schedules for any realistic workload is too large to explore exhaustively. There are several classes of general search-based algorithms that do well in exploring large search spaces efficiently. The reader is referred to Whitley [167] for a thorough introduction to the concepts used in search-based algorithms, and especially those of *fitness functions*, *crossover*, *mutation* and *convergence*.

Search-based algorithms work by considering a population of individuals and ranking them using a 'fitness' function. A subset of the fittest individuals are then selected and have the crossover and mutation operators applied to generate a new population. Over time, the algorithms should converge to a solution that is close to optimal.

In the context of scheduling, typically each member of the population is a static schedule. A large amount of research into search-based scheduling uses the workload makespan as the fitness [20, 21, 23, 51, 158, 166, 173]. Alternative fitness functions include the number of deadline misses [129] or the degree to which load is evenly balanced [132].

Genetic Algorithms (GAs) tend to have a substantial population and derive new populations through crossover and mutation. Each iteration takes significant compute time due to the size of the population, but few iterations are required until convergence. Each individual can have its fitness calculated independently, which means that each iteration of a GA is amenable to parallelisation. GAs have been used to perform both ordering and allocation [129, 158, 166] or just allocation [173].

Rather than evolving an entire population, the Simulated Annealing (SA) approach instead evolves a single individual [152, 158]. At each iteration, a neighbourhood of new individuals is generated through mutations of the currently selected candidate. If a new solution has a better fitness value, it is always accepted and replaces the current individual. If the fitness value is worse, it will only be accepted if it is less worse than a given 'temperature'. Over time, this temperature cools. This means that a wide search area is possible at the start, but over time the algorithm will tend towards straight hill-climbing, where only strictly better solutions are accepted. Hill-climbing through mutations is able to find good solutions, because small improvements are cumulative where the solution space is relatively smooth. Pseudocode for Simulated Annealing can be found in Algorithm 3.4.

Algorithm 3.4 Simulated Annealing Pseudocode

```
A is current state, initialised with appropriate value
B is best solution found so far
T is the initial temperature
while T > 0 or there have been recent improvements :
    Determine N through mutation of A
    if fitness (N) > fitness (B) :
        B ← N
    if fitness (N) > (fitness (A) − T) :
        A ← N
    decrement T
return B
```

Schoneveld et. al. [148] mathematically characterise the solution space for dependent task scheduling over homogeneous processors. They find that the space is self-similar, which they note is where simulated annealing is particularly good at finding optimal solutions. Furthermore, SA is known to converge well [152], because its rate of convergence or cooling can be manually tuned. A drawback of SA is that because it only evolves a single individual, it is less amenable to parallelisation than GAs. This would be a significant disadvantage as the search spaces of schedules at grid scale are very large.

Boyer and Hura [20] noted the issues with designing effective crossover and mutation operators for GA and SA algorithms and instead proposed a random search algorithm. A schedule is produced by creating a random ordering of tasks (respecting the topological ordering of dependencies) and then assigning these tasks in order using an earliest finish time allocator. The algorithm produces a large number of schedules this way and terminates either after a fixed number of iterations have elapsed or after the best schedule found has not changed for some time. Random search is highly amenable to parallelisation, as each generation of a schedule can happen completely independently. Boyer and Hura [20] claim that their approach produces similar quality solutions to SA and GAs.

The great strength of search-based algorithms is at finding solutions that are closest to optimal compared to other heuristics [23]. Furthermore, they can achieve this with with respect to a wide variety of fitness functions. However, a problem of being so close to optimal in the context of scheduling, especially when measuring optimality by makespan, is that their solutions are often 'fragile'. If tasks overrun their estimates, they are likely to cause knock-on effects that can severely disrupt the rest of the schedule and lead to behaviour that is far from optimal [54].

A weakness of search-based algorithms is that they tend to be markedly slower than heuristics used for scheduling directly [23]. The size of the search spaces are large, even for small workloads. This is because the number of possible orderings of

tasks in a queue grows with the factorial of the number of tasks. While the search algorithms do not have to consider every possible ordering, in order to find good solutions they must be able to cover a reasonable sample of the space.

An approach which can be used to reduce the time taken by search-based algorithms is to prime their populations with the output of a heuristic scheduling policy [132, 158, 173]. In effect, this uses the search-based algorithm as a schedule postprocessor. Yet with good heuristic policies, there may be diminishing returns on the usefulness of the postprocessor. The empirical research performed by Braun et. al. [21] shows that their GA typically returned schedules with makespans only 12% shorter than those created by the simple Min-Min list scheduler. This gain is actually more than offset by the execution time required to run the GA. Furthermore, the use of poor heuristics may bias the search algorithm away from more fruitful areas of the search space [173].

The most critical issue with for search-based algorithms with respect to the industrial context is that they are fundamentally static scheduling policies. This is because the individuals used for the iterative procedures are encodings of static schedules. It might be theoretically possible to use a search-based policy as a scheduler in a batch mode or as part of a generational scheduler. However, these architectures must re-run the algorithm every time a batch or a task finishes. Due to the time taken for the search algorithms to execute, this is likely to be prohibitive in terms of the computation time required [51]. As the range of task execution times precludes the use of a static or batch scheduling policy, these shortcomings mean that search-based algorithms are unlikely to be suitable for the industrial grid system.

3.2.6 Market-Based Schedulers

Most grids will tend to serve a large number of users who wish to have access to the grid's resources. However, this can lead to problems when demand for resources outstrips supply. In this situation, some jobs must either have to wait until demand reduces again, or not be executed at all (called *starvation*). Traditional schedulers handle this situation by queueing up work. They decide what to run either through statically allocated priorities or through partitioning of the underlying hardware resources.

Market-based schedulers do not schedule directly. Instead, users have budgets which they allocate to their jobs and tasks [155]. The tasks and jobs then place bids for computational resources in a virtual market [24, 53]. Agents representing resources will sell resource access to the highest bidder, in order to maximise their profit. Pseudocode for such an algorithm is shown in Algorithm 3.5.

With a properly configured market, well-understood market economics come into play [165]. In most grids, supply will be relatively stable, as upgrades happen infrequently. Therefore, pricing will be determined by demand. When demand is high, those jobs with the highest values will be executed first. This should reflect the priorities that the users desire.

Furthermore, real grids have real running costs. Although the internal market may operate in a virtual currency, it may be possible to deduce an exchange rate from this to real currency. This then lends itself to users external to the grid being able to purchase computational power on demand [6, 53]. However, a virtual currency comes with significant disadvantages such as the risks of depletion or inflation, and these need to be actively managed [24].

However, with the flexibility of a market also comes the danger of market instability [53]. This work considers a private grid, where users are nominally trustworthy agents. However, there still need to be checks and balances such that an ill-informed or inexperienced user cannot perturb the stability of the system. These checks can incur significant overhead [24, 53].

Scheduling through market forces alone is even more complex when dealing with dependent task sets. In order to achieve good turnaround times, the highest level of parallelism is desirable. Yet for valuable dependent task sets, each processor agent might wish to execute the whole set serially in order to maximise profit, even though this would be seriously detrimental to responsiveness. It is possible to use soft real-time value judgments to help decide which tasks should be run soonest, but this can impose a great deal of calculation overhead.

Algorithm 3.5 Market-Based Scheduling Pseudocode

A is the set of tasks
P is the set of processors
$B\left(A^i\right)$ is the budget of each task
$E\left(A^i\right)$ is the execution time of each task
$N\left(A^i, P^j\right)$ is the network cost for running A^i on P^j

At each time tick (centralised):
or
Continuously (decentralised):
    For all free processors P^j:
        For all tasks (centralised) A^i:
        or
        For all tasks in neighbourhood (decentralised) A^i:
            Find the highest profit task
            $$A_{\max} = \max\left(\frac{B\left(A^i\right)}{E\left(A^i\right)} - N\left(A^i, P^j\right)\right)$$
        Allocate $A_{\max}$ to P^j

A strength of markets is that they can operate in both a centralised or a decentralised way. For fault-tolerant purposes, decentralisation is extremely valuable, especially because grids are geographically distributed and can be controlled by many separate entities [53]. Removing a single point of failure is therefore technically and politically attractive to the operators of the grid. However, because of this decentralisation, market-based schedulers can generate a great deal of network traffic overhead while communicating all the bids and offers across the network.

The greatest drawback for using market-based schedulers is that the frameworks that power typical industrial Grids (GridEngine [134] and LSF [138]) are not built with the assumptions or the architecture of a market-based scheduler in mind [53]. Changing the framework that runs the grid is beyond the scope of this work as specified by the industrial partner. As the contribution of this work is designed to integrate with such existing frameworks, a market-based architecture may not be suitable.

Furthermore, pricing can only be a proxy for priority, and priority is only a proxy for ordering. The important metrics of short, fair response times for jobs and high overall throughput are not encoded into market-based scheduling policies. Instead, these can only be observed as emergent effects of the functioning of the market [24, 53]. Because of this, the author believes that tuning the market parameters in order to achieve emergent effects is likely to be more difficult than constructing a scheduling policy directly to achieve the desired performance.

3.2.7 Schedule Postprocessing

Where very large scale workloads must be scheduled, it may be desirable to produce an initial schedule with a low-complexity scheduling algorithm that can cope with such scales. However, these schedules produced may be far from optimal. There exist algorithms that require a schedule as input and use further heuristics to attempt to improve the input schedule. This gives rise to the concept of a schedule postprocessor, or what Lo [114] terms the *task migration* stage.

Backfilling is one of the most common kinds of schedule postprocessing [52, 104]. Backfilling is useful where parallel tasks are waiting for a sufficient number of cores to become free. During this time, backfilling would start a task whose execution time means that it would finish before the number of cores required become free. Dimitriadou and Karatza [52] studied the impact of inaccurate estimates on a backfilling algorithm by including the maximum amount of pessimism in the execution time estimates considered for backfilling. They found that as the quality of estimates decreased, so did the responsiveness achieved by the scheduler. They only considered estimate inaccuracies up to 30%, which is likely to be an unreasonably

tight bound following the results of Bailey Lee et. al. [107]. Mu'alem and Feitelson [124] found the opposite, however, where a small level of inaccuracy increased the flexibility of the scheduler, enabling improved responsiveness. However, Mu'alem and Feitelson [124] noted that the user estimate inaccuracies were usually worse than the estimates they considered.

While backfilling is useful in principle, it is mainly helpful where a parallel job might queue for a very long waiting for sufficient cores to become free. In the industrial scenario considered, the scale of the system means that tasks are finishing all the time, and large parallel jobs never have to wait too long to start execution. The industrial partner had disabled backfilling approaches because the inaccuracies of task execution times were large enough to cause problems, while the performance penalty of not using backfilling is minor due to the scale of the system.

Networking delays can be reduced by using a schedule postprocessor too. Wu et. al. [169] use a local search algorithm to see if moving each task to other nearby processors would achieve any decrease in the finishing time of the task. If so, then the task is re-assigned. If a global search were used, then the complexity of this algorithm would make it intractable. However, because only a local search is used, they claim that it can lead to improvements on an existing schedule in a reasonable timescale.

Maheswaran and Siegel [116] use a generational scheduling style approach as their postprocessor. They divide a workload into 'blocks', which are sets of tasks that have no dependencies between them, and only depend on tasks in previous blocks. They then use one of three list schedulers described to see if any tasks need rescheduling. They compare their postprocessed approach with a pure generational scheduler, and determine that it shows an improvement in makespan, but only of 3-4%, and they acknowledge that this may not be statistically significant.

Postprocessors can also be used to enhance a schedule according to a different metric than the original schedule was produced against. Sugavanam et. al. [158] describe an approach where the original schedules were optimised against makespan, and presented heuristics that attempted to improve the robustness of the schedule with regard to execution time inaccuracies.

The main issue with schedule postprocessing approaches for the industrial scenario is the overhead incurred relative to the gain achieved. The papers surveyed tended to give improvements of only a few percent on the workload makespan. Furthermore, other than backfilling, none of the postprocessing approaches are relevant to dynamic schedulers, which are required in the industrial scenario. It is worth considering whether a better gain would be achieved by having the aims of a schedule postprocessor (low network costs, higher utilisation etc.) encoded directly into the scheduler, rather than being added on separately. To do this, more information must be supplied to the scheduler. However, if it being supplied to the postprocessor, it must be possible to make it available to the scheduler as well.

3.3 Scheduler Input Information and Constraints

Any kind of scheduling is a kind of prioritisation. However, in order for this prioritisation to be effective, the scheduler must have some knowledge about the work it is trying to schedule. With no knowledge, the scheduler cannot perform well [41, 42]. Where constraints on the schedule are present, is is necessary to give this information to the scheduler so that an infeasible schedule is not produced.

A balance must be struck because if too much information is supplied to the scheduler it can result in increased time to create the schedule. Furthermore, the process of scheduling often involves tradeoffs, especially under situations of high or over-load. Supplying too much information can make the tradeoffs harder to manage because of the increased amount of information available. Most importantly, the scheduler needs to know enough pertinent information about the workload so that it can efficiently create schedules. These need to meet the requirements of the users and administrators of the grid along with the constraints of the workload and platform. This section will examine the kinds of information and constraints that can be supplied to a scheduler and survey scheduling policies that make use of these.

3.3.1 Execution Time Estimates

In a dynamic scheduling system, execution times cannot be known in advance with certainty, and hence estimates must be supplied. Nevertheless, providing estimates enables much better scheduling approaches to be applied. This is noted by Codd [41, 42] in some of the earliest research on scheduling, where he stated that, "When elapsed times are not available, scheduling necessarily becomes very primitive". This statement may be naïve to a degree, because modern schedulers have to balance many competing requirements and goals, which may not necessitate the knowledge of task execution.

Supplying task execution time estimates is an ongoing research problem, because users tend to only have a vague idea of the execution time of their tasks [107]. Methods analysing users' historical work submitted to the grid seem to give the most promising results, with Lazarević [105] able to give a median estimation error of 5%. However, these approaches will always have their limitations due to real submissions being highly noisy and bursty [156].

Despite execution time estimates being difficult to obtain accurately, they are used by a huge number of modern schedulers. Garey et. al. [66] describe the longest remaining time first algorithm, while they and Topcouglu [160] consider shortest remaining time first. These clearly need a knowledge of how long tasks will execute for. Tzafestas et. al. [162] weight tasks by the amount of successor work, which also requires an estimate of the time this work will take. Saule et. al. [147] use execution

time estimates along with a parallelism requirement in order to inform the number of cores allocated to malleable tasks. Cao et. al. [33] consider fuzzy time estimates represented by a trapezoidal probability distribution.

3.3.2 Parallelism/Core Requirements

At the moment a task begins to execute, the number of cores it will be started on must be known. Some kinds of parallel tasks, which include those used for CFD, can be scaled to run over a range of core counts. Some scheduling policies are able to decide on behalf of tasks how many cores to allocate at runtime, in order to optimise the packing of tasks onto cores. This is an example of what is known in the literature as mouldable scheduling [147]. The similar problem but where the number of cores can be varied during runtime is known as malleable scheduling [25].

As with most scheduling problems, these problems are NP-Complete [25] and are an active field of research [25, 56, 89, 146]. However, despite the theoretical interest in this problem, the core counts in the industrial context of this research are so closely bounded by RAM and network latency constraints that they can be considered to be fixed, as described in Chapter 2.

3.3.3 Ownership Attributes

The user who submits work to the grid can be used for prioritisation. This is especially useful where a 'fair' allocation of resources between users is necessary. For a more detailed kind of prioritisation, both the user and the group/team that they are a part of can be supplied. These, along with the core counts required are the attributes used by the FairShare scheduling policy currently in production on the industrial cluster. FairShare was already described and discussed in detail in Chapter 2.

3.3.4 Dependencies

There is a distinct spectrum within workflows relating to how parallel they can be. At one extreme, there are workflows with a single path that are strictly sequential. At the other extreme are jobs whose work can be arbitrarily divided into fragments, and whose time to execute is inversely proportional to the number of processors. These highly parallel workflows are known by the term *embarrassingly parallel*. In between these two extremes are those jobs termed semi-parallel [148]. This is where a job can be broken down into a number of tasks, some of which may be sequential and some of which may be embarrassingly parallel, but where these tasks require data to flow between them.

Dependencies are an essential feature of the workloads considered in the industrial scenario. Their intricate processing chains are made up of several pieces of software, which requires data transfer between each part of the process [117, 160]. The presence of a dependency means that successor tasks may not be scheduled until a predecessor task has completed, and, if applicable, any data transfers have been made. In order that computing resources are not wasted by scheduling a successor task before its data is ready, the scheduler must be informed of dependencies. Dependencies are represented using graphs, with a consensus in the literature that they are represented by Directed Acyclic Graphs [2, 32, 72, 120, 160].

The efficient scheduling of workflows or jobs containing such dependencies has been a subject of study since the UNIVAC computer [42, 94] and, previously, in Operations Research [76]. An important step in the study of workflow scheduling was made in the 1950s, with the development of the Critical Path Method (CPM) [94]. The Critical Path method was developed to identify those tasks that lay on the longest single path through a workflow. This longest path is termed the *critical path*. If any of the tasks on the critical path were to be delayed, the completion of the whole workflow would be delayed. The CPM also allowed the calculation of the amount that other tasks in the workflow could be late, or 'slack', without affecting the final completion time.

Dependency graphs are used in different ways by different algorithms. In order to ensure that dependencies are always satisfied before execution of tasks begins, the Levelised Min-Time heuristic groups tasks by *level* [32], as did the generational scheduling approach of Carter et. al. [34]. The level of a task is the number of tasks between the source and the considered tasks. These levels are ordered one after another, and tasks with the smallest execution times are ordered first within each level [160].

In the industrial scenario, the problem with grouping tasks by level is that jobs arrive continuously. Newly arrived jobs would add their source tasks to the first group, which is given the highest priority. This would mean that newly arrived jobs would in effect be given higher priority than those that had been waiting for longer. In a highly loaded system such as the industrial grid, this would quickly cause all jobs to starve, as new jobs would start but no jobs would be able to complete.

Instead of grouping tasks by level, the structure of the dependency graph can be used to prioritise tasks. For example, tasks can be prioritised by the counting the number of their dependencies [39, 151]. This uses the assumption that a task with a high number of dependencies is more likely to fall on the critical path, and is therefore important to execute as early as possible. Where execution time estimates are available, it is possible to explicitly calculate the critical path and schedule the tasks on the critical path first [160, 169], bearing in mind that the partial order specified by the DAG must still be respected.

Name	Refer -ence	Ordering (subject to partial order)	Ordering Parameters	Allocation
Graham's Greedy	[71]	Arbitrary (only considers ready tasks)	Dependencies satisfied	EST
"List Scheduling"	[66]	Arbitrary	-	EST
Largest Processing Time aka Min-Max	[66]	Processing Time, largest first	Task Exec Time	EST
Smallest Processing time aka Min-Min	[66]	Processing Time, smallest first	Task Exec Time	EST
Levelised Min-Time	[160]	Processing Time, smallest first (only considers ready tasks)	Task Exec Time, dependencies satisfied	EST
Multifit	[66]	Processing Time, largest first	Task Exec Time	EST and finish time less than deadline
Heaviest Node First	[151], [39]	Number of dependent tasks, largest first	Dependencies	EST
Critical Path	[151], [162], [169]	Tasks on critical path first, others arbitrary	Dependencies, Exec Time	EST
Modified Critical Path	[169]	Critical path first, rest by 'finish time' in a schedule produced by Graham's Greedy on an unbounded number of processors	Dependencies, Exec Time	EST
Most Valuable Task First	[162]	Sum of successor execution times, largest first	Dependencies, Exec Time	EST
Most Valuable Task First	[162]	Sum of the ordering ranks produced by 4 heuristics, smallest first	Orders from 4 other heuristics	EST

Table 3.1: Comparison of List Schedulers

The priorities given to tasks can also be weighted using information from the task graph. Tzafestas et. al. [162] give an algorithm that weights tasks by the total execution time of their successor tasks. Shirazi et. al. [151] use a weighting calculated from a hybrid of the critical path and the weighted total execution time. Topcuoglu et. al. [160] introduce the concept of the upward and downward ranks. These ranks give each task a weighting based on the longest path from the task to its latest sink or earliest source, respectively. The upward and downward ranks are essentially the lengths of the critical paths up to and following the considered task.

In the process of reviewing the literature, it was realised that using the upward rank as a weighting has two highly desirable properties for scheduling dependent tasks. Firstly, ordering tasks by their upward rank gives an ordering that is also topologically sorted. That means that by executing tasks in decreasing order of upward rank, all dependencies will be satisfied (if only one task were run at a time). Furthermore, tasks on the critical path will have larger upward ranks than those that are not. This is also advantageous where responsiveness is required, because ordering tasks by their upward rank will mean the critical path will be scheduled first.

An ordering policy that sorts tasks with the largest upward rank first was introduced by Topcuoglu et. al. [160], who termed it Longest Remaining Time First, or LRTF. They used it as the ordering part of their HEFT static list scheduling policy. LRTF ensures that the largest tasks are started first, which is a useful heuristic when performing bin-packing to optimise the workload makespan for static schedules. However, in a dynamic system, LRTF has some significant disadvantages. Firstly, as with grouping tasks by level, LRTF will prioritise starting newly-arrived work over finishing older jobs, penalising responsiveness and leading to starvation under periods of overload. Furthermore, LRTF runs the largest tasks first, which is the opposite of what the users in the industrial scenario require, as they need responsiveness for the smallest tasks [147]. Cao et. al. [32] describe a policy that uses the upward ranks to group tasks and then performs generational scheduling using upward ranks. This approach seems like it would doubly compound the issues with responsiveness when new tasks are started before old ones finish, however.

When running tasks on a large cluster, there may be resources free to start tasks that don't yet have their dependencies satisfied, even if their upward rank/LRTF is largest. Therefore, when operating in a grid environment, the scheduler must manage the queue and only admit tasks that have had their dependencies satisfied. By using upward ranks, though, tasks could be admitted to the head of the queue if their weighting is sufficient.

Where the scheduler tracks which tasks have had their dependencies satisfied, other policies become possible. The Shortest Remaining Time First (SRTF) policy [147] orders tasks by increasing order of upward rank. In a dynamic scheduling system,

this prioritises small tasks and tasks that have little work left to run. This is good because it should achieve high levels of responsiveness. However, under overload situations, the largest tasks may never reach the head of the queue and so would starve [10].

To try to avoid the problem of new tasks being prioritised over older ones, a policy is proposed in Hagras and Janeček [72] that orders tasks based on the job start time subtracted from the upward rank. A disadvantage with this policy is that it could not distinguish between a newly arrived small task and a large task that has been waiting for a long time. This means that small tasks are not as effectively prioritised compared to a policy like SRTF.

Hybrid weightings are also possible. Tzafestas et. al [162] suggest an approach which calculates orders using four different ordering algorithms. They determine a weighting for each task by summing the ranks of each task in all four orderings. The final order is then based on these values.

3.3.5 Scheduling with Network Delays

The clusters that make up the industrial grid (or any large-scale grid architecture) are connected by Wide Area Networks. Several kinds of network traffic place load on these WANs, including applications, input and output data and cluster state. Unfortunately, the bandwidth between geographically distributed datacenters is expensive, because of the cost of building large networks. Furthermore, the speed at which available network bandwidth is growing is slower than Moore's Law [128]. With limited inter-cluster bandwidth and an ever-increasing amount of information to be transferred, schedulers must operate very carefully to ensure that the WAN links do not become a bottleneck.

The previously discussed clustering [18, 114] and task duplication [14, 44, 82] schedulers are designed for precisely this situation. Search-based algorithms can be extended to handle network costs by including a calculation of these in how makespan is calculated [51, 166]. None of these approaches are suitable for the industrial scenario because of their static nature, however.

When scheduling a parallel job, communications between tasks on different clusters must be considered. The relative contribution of the time taken for work to execute and to transfer data can be described using the Communication to Computation Ratio (CCR) [2].

The problem of allocating tasks to processors at extreme values of CCR tends to be trivial [148]. Given homogeneous processors and negligible network transfer times, any random allocation will produce results similar to any other, and hence similar to the optimal [74]. Instead, if the time spent processing were negligible relative to the network transfers, then all jobs should be assigned to one processor in order to

avoid any network accesses. Therefore, there are distinct zones in the spectrum where sequential and parallel execution are best.

The transition point between parallel and sequential execution is investigated by Schoneveld et. al. [148] using simulated annealing to find close-to-optimal schedules across the spectrum of CCR. They found that there is a sharp transition between the zones of optimal sequential and parallel allocation. Furthermore, the time for the simulated annealing algorithm to converge on a solution is much larger at the zone of transition. The mathematical analysis of Schoneveld et. al. [148] shows what has been suspected for a long time [66], that the zone where the simplest greedy heuristic algorithms tend to produce solutions that are far from optimal is where the time spent processing and transferring data is roughly equal. To get schedulers that are closer to optimal, information must be provided about the underlying network architecture.

In the industrial scenario, network delays are substantial, and efforts to ensure that tasks from the same job run as close by as possible are useful. However, the contribution of communication delays to the response time of jobs is as yet still exceeded by the computation times. From the results of Schoneveld et. al. [148], therefore, there should be a possibility to use heuristic policies to produce schedules where networking is not the limiting factor in responsiveness.

List schedulers and generational schedulers can be extended to consider network costs by updating the Earliest Finish Time (EFT) allocator to take into account the time taken to transfer data to a task before execution [34, 160]. These papers assume a network model where there is no contention on the links, but communication and computation cannot happen at the same time. An alternative approach is taken by Huang et. al. [80], with a model that assumes that communication and computation can be concurrent, and uses the structure of the dependency graph to start transfers early where tasks not on the critical path have finished.

EFT allocation policies in list schedulers are helpful where a single list scheduler has global knowledge of the network, such as those presented by Huang et. al. [80], Topcuoglu et. al. [160] and Carter et. al. [34]. However, in the industrial grid considered, this is likely to be infeasible due to the scale and the bandwidth required to centralise all scheduling information. Instead, a hierarchical approach to list scheduling is currently used. Allocation happens at a higher level first, as tasks are load balanced between clusters. Ordering is then applied on the queues within each cluster.

3.3.6 Scheduling with Platform Heterogeneity

There can be many kinds of heterogeneity in a grid system, where machines may differ in their:

- Hardware architectures (Instruction sets, presence of FPGAs or GPUs)

- Resources on each processing node (disk, RAM, CPU core count)
- Network link speeds and topology
- Operating systems, installed software, and versions of these
- Ownership of the machine, permissions and capacity allowances

The heterogeneity problem really has two distinct aspects. The first aspect is that restrictions on the architecture, software or permissions on each cluster will mean that only a subset of the grid's resources may be available for any task to run on. This is the case in the industrial grid where some machines are provisioned with significantly more RAM or disk space than others. In this case, the allocation problem is to select an appropriate free resource from this subset bearing in mind the requirements of the task at hand. This may need to take networking delays into account, as discussed in the previous section.

The second aspect is to consider grid resources that can run the same tasks, but where the processors run at different speeds. Many researchers have considered this problem [49, 51, 117, 160], and a summary of algorithms for scheduling with heterogeneity is given in Table 3.2. It is also essentially an allocation problem. Information that the allocator may require includes the platform speeds and load. Under situations of heavy load, the allocator must make the tradeoff between assigning tasks to highly-loaded but fast clusters vs. more lightly loaded but slower clusters. In the dynamic schedulers surveyed, this tradeoff is universally managed by selecting the resource that will give the EFT for tasks [160]. The differences between policies relate to how much information they require to calculate this EFT.

An approach considered in Dhodi et. al. [51] is to schedule the tasks with the highest estimated execution time to the fastest processors. Where dependencies are present, it may also be advantageous to assign those tasks on the critical path to the fastest processors, following the CPOP policy given in Topcuoglu et. al. [160]. Due to the subtleties of processor architecture and design, the execution speed of tasks may not be entirely driven by processor clock speed. The Sufferage policy is suggested by Maheswaran et. al. [117] by comparing the execution speeds of tasks between the faster and slower resources. The tasks that would suffer the most by having to run on the slower resources are allocated to the fastest resources.

The issue with these approaches is that they add significant complexity to the allocation problem. As discussed in Chapter 2, the industrial partner has found that it is uneconomical to run processors that are any less than state-of-the-art due to their power consumption requirements. Therefore, the added complexity of considering heterogeneity as part of the allocation mechanism would not be worthwhile.

Name	Reference	Ordering (subject to partial order)	Ordering Parameters	Allocation
Critical Path On a Processor (CPOP)	[160]	Critical path first, then sum of predecessor and successor execution times for others, largest first	Dependencies, Exec Time	Insertion-Based EFT
Heterogeneous EFT	[160]	Sum of successor execution times (estimated), largest first	Dependencies, Exec Time	Insertion-Based EFT
Longest Dynamic Critical Path	[49]	Sum of successor execution times (exact where possible, otherwise estimated), largest first	Dependencies, Exec Time, Processor speeds	Insertion-Based EFT
Sufferage	[117]	Difference in execution time between best and next-best processor, largest first	Dependencies, Exec Time, Processor speeds	EFT

Table 3.2: Heterogeneous List Schedulers

3.4 Scheduling for User-Level Aims

Traditional scheduling policies have been primarily designed to serve the needs of grid administrators. Static schedulers seek to minimise the makespan of a workload. In effect, this means that they maximise the utilisation of the grid resources during a period of time. If these policies were applied in a dynamic scheduling scenario, their aim would be to continuously maximise utilisation.

In the traditional days of mainframes, computing time was seen as highly precious in relation to the time of the users. However, in recent years, the price of computing time has fallen sufficiently so that users' waiting time cannot be considered as merely an incidental cost. Instead, in the industrial scenario context that the work is placed in, the time of the highly skilled users needs to also be considered as highly valuable.

The users' perspective on scheduling tends not to concentrate on utilisation, because it is irrelevant to them how busy or otherwise the clusters are. Instead, as described in Chapter 2, users care about responsiveness, fairness and the value returned by the execution of their jobs.

3.4.1 Scheduling for Responsiveness

A key requirement of users is that their jobs are returned quickly. Furthermore, as noted in Chapter 2, their productivity can be reduced as waiting times increase. This

is because the number of iterations they can perform on a single design is reduced. An important observation is made by Saule et. al. [147] that:

> "A desired property of a scheduler is to avoid starvation while ensuring overall good response time." [147]

One way of measuring the responsiveness of jobs in a dynamically scheduled system is by the stretch metric [15]. Stretch is the actual response time of a job relative to what the response time would have been had the system been empty. Muthukrishnan et. al. [125] show that the Shortest Remaining Time First (SRTF) dynamic scheduling policy is good for optimising average stretch. This result is validated by Bansal and Pruhs [10], who show that SRTF is also good for optimising average flow. Flow [15] is a measure of job throughput of the system.

A problem of SRTF, however, is that it is not starvation-free. As noted in Bansal and Pruhs [10]:

> "[SRTF] will not starve jobs until the system is near peak capacity."

The trouble with this assertion is that for the industrial scenario considered, the grid almost always operates at or near peak capacity - it is almost always *saturated*. The survey by Bansal and Pruhs [10] considers several variants of SRTF, including shortest total time first (STTF) and shortest elapsed time first (SETF). SETF seems like a poor choice of scheduler, because under high load, new jobs would always have priority to start before old jobs have completed. This would likely lead to a very high level of job interleaving, which in turn leads the response times of all jobs to be large.

Saule et. al. [147] gave an alternative algorithm to minimise average stretch, known as Deadline Based Online Scheduling (DBOS). DBOS takes the critical path of the job and assigns a deadline of the CP extended by a fixed percentage of the CP. This means that job deadlines are weighted by their execution time. This is the same model adopted by Ghazzawi et. al. [68], although they apply the model to the admission control problem rather than scheduling itself. This approach may also struggle where there are changes in the level of load, because the percentage of the CP with which to adjust the deadline may need to adjust over time in response to load levels. Furthermore, DBOS is a batch scheduler, rendering it unsuitable for situations where the variation in execution times is large.

3.4.2 Scheduling for Fairness

While no user wishes to be kept waiting for too long, users also tend to want to perceive that their jobs are treated fairly. A particularly unfair situation can occur for workloads where there is a range of job sizes. Saule et. al. [147] note that:

> "Since the flow time metric does not take the size of the tasks into account, objective functions that utilize this metric tend to create schedules in which small tasks spend as much time in the system as the large tasks. This results in small tasks waiting in the system queue longer than the large tasks, hence introduces unfairness against small tasks."

Previous work on fair scheduling for workflows with dependencies has been performed for static [175] and batch [77] schedulers. These are ineffective when there is a wide variation in runtimes [27, 40, 57] because the response times required of the smallest tasks (hours) are orders of magnitude smaller than the execution times of the largest tasks (months), so no batch size will suit both. Effective prioritisation by the scheduler is required to keep the system responsive for the smallest tasks but avoid starvation for the largest ones. An ideal situation expressed by the users in the industrial scenario would be that all jobs in the system would have a waiting time proportionate to their execution time [147, 168].

This definition is refined by Zhao and Sakellariou [175] and defines fairness as all workflows having a similar value of the slowdown metric. They define slowdown as the response time of the job executing on the cluster alone divided by the actual response time when other jobs are also present. This is similar to the reciprocal of the SLR metric used by Topcuoglu et. al. [160] but would differ on clusters with a small number of processors where a job could use more than the available number of processors, as SLR considers jobs run on an unbounded cluster.

Zhao and Sakellariou [175] present a static scheduling policy that runs the job with the smallest slowdown first. They calculate the downward rank and critical path of each job by running each workflow alone on the system first. Jobs that consume a larger share of the cluster are likely to suffer more (and hence have a smaller/worse slowdown value) by having to share their capacity with other jobs. The approach by Zhao and Sakellariou [175] will therefore tend to prioritise the largest jobs first. As their approach is a static scheduling policy, this is useful to ensure the largest jobs are started first when the desire is to minimise makespan. Furthermore, the task graphs of all jobs are merged into a single DAG graphs. This makes it possible to find the longest critical paths of any job, and schedule those first.

In a dynamic system, when the desire is to maximise fairness, it would seem to make sense to use the slowdown metric to calculate how 'late' jobs are dynamically, and use this for scheduling. Furthermore, it is not possible in a dynamic system to merge all the DAGs of separate jobs into one large job, as this would be equivalent to batching. Instead, a method would be necessary to weight tasks within each DAG on a common scale so they can be scheduled without having to merge the graphs. These intuitions for dynamic systems form part of the basis for the novel scheduling policy that is presented in Chapter 6.

A further issue with the approach of Zhao and Sakellariou [175] is that their algorithm is only tested on 2-10 jobs and 20-500 tasks in a workload, and they note it may struggle to scale above this. The task execution times were also sampled from a uniform distribution, which is not what was observed in the industrial scenario. They also run each workflow alone on the cluster first in order to measure its execution time. Devoting an entire cluster or the whole grid to execute a single job at a time is clearly infeasible in the industrial scenario as throughput would be unacceptably low. Furthermore, as the industrial jobs only need to be run once to deliver their results, there would then be no point in running them again. While some of their intuitions are helpful, their policy as presented would be clearly unsuitable for an online grid scheduler as is required.

Arpaci-Dusseau [9] uses multi-level feedback queues (MLFQs) to try to fairly balance response times. Their policy does not assume that execution times are known in advance, but instead moves tasks between queues by monitoring their elapsed time. However, this is only possible because they assume a pre-emptive execution model, which is not available in the industrial framework considered.

3.4.3 Scheduling for Value

Approaches for scheduling for responsiveness and fairness use execution time estimates to define their metrics. However, the value to users of different jobs may not be perfectly related to computation time. Instead, some scheduling policies consider a model where users supply a value parameter along with their job [37, 86]. The aim of the scheduler is to maximise the value returned by the system, rather than simply metrics relating to responsiveness or fairness. Lee et. al. [106] used integer programming-inspired heuristics in a static approach to maximise the value of tasks returned, where tasks required the use of several kinds of resources.

Dynamic scheduling policies cannot predict load in advance. They are therefore likely to encounter at least some periods of overload, where work arrives faster than it can be processed. Many scheduling policies designed to provide responsiveness, especially SRTF, can suffer from starvation under overload [10]. The Earliest Deadline First (EDF) policy is also designed to give good responsiveness with respect to deadlines. However, as a system moves into overload, missed deadlines can compound and reduce throughput dramatically [31].

A particular strength of value-based scheduling is in overload situations [37]. This is because having a notion of value means that the least valuable tasks can be postponed or discarded [24, 31]. Without this knowledge, then arbitrary tasks may be discarded or be starved of resources. This approach is particularly relevant for the industrial context, therefore, because of the common periods of transient overload.

Locke lays much of the groundwork of value scheduling in his PhD thesis [115]. In it, he shows that in a dynamic system, ordering tasks by decreasing value density is optimal in situations where the workload is schedulable by EDF. However, this proof will not necessarily hold for overloaded systems, as EDF exhibits rapid performance degradation under overload [31]. An algorithm equivalent to Locke's value density termed Highest Density First is given by Bansal and Pruhs [10], although they conclude that SRTF was better in real-world situations.

The value returned by tasks need not be static. Irwin et. al. [86] consider a model whereby the values of tasks decays linearly with waiting time. This means that values will eventually turn negative, leading to tasks with extended waiting times not just having zero value, but applying a penalty. It is possible to use a market-based system to actually perform the scheduling [24, 86]. Locke [115] considers a model where value could vary over time, including to increase. This relates to real-time systems, where it is often just as bad for a task's results to arrive early as arrive late [30]. However, in the industrial context there is no penalty to tasks arriving early.

Burns et. al. [30] consider a system where multiple alternative pieces of software can provide results. These different pieces of software have a tradeoff between the precision or *utility* of their results and their computational resource requirements. They give a detailed framework of how to develop a mathematically-sound assignment of values to processes and their alternatives, although they do not discuss where the notion of value should be derived from in the first place. Furthermore, they do not give a scheduling policy.

A shortcoming of the value policies surveyed is that while the values of jobs may vary with time, the execution times of tasks are not necessarily well-known in advance. Users may also only have a weak understanding of how much value should be assigned to different pieces of work. Furthermore, most research on value-based work considers independent tasks, whereas in the industrial scenario, value is only realised at the completion of an entire workflow. Chen and Muhlethaler [37] consider dependencies with a static value-scheduling algorithm akin to a generational scheduler that groups dependent tasks by level and then prioritises by the number of successor tasks.

3.5 Summary

This literature survey examines the state of the art in scheduling research and examines the approaches that have been applied to workflow scheduling on grid computing architectures. Solving the grid scheduling problem optimally is known to

be intractable (NP-Complete), especially at the scales at which production grid systems operate.

A variety of scheduler architectures are surveyed. Although static scheduling structures are not applicable to the dynamic nature of grid scheduling, they are nevertheless surveyed as the policies they consider can give further insight into the development of dynamic policies. Static or batch architectures included the Generational, Clustering and Search-based approaches. List schedulers were described and their merits in terms of flexibility and scalability were highlighted. However, they need to be configured with ordering and allocation policies that are suited to the workload and platform. Market-based schedulers were surveyed, although the network overheads these incur in practice renders them less appropriate to the industrial scenario. Furthermore, the fact that desirable schedule attributes are not tuneable but emergent, along with the architecture being alien to the existing grid management systems makes a market-based system less promising an avenue for development.

Real platforms have constraints, and schedulers must model these constraints accordingly. Dependencies are a critical part of grid workloads, and respecting them is an essential role of the scheduler. By using dependency information, better schedules can be created, across the scheduling architectures. Execution time and parallelism requirements help the scheduler achieve better prioritisation and matching of tasks to available hardware. The difficulties in accurately estimating execution times even with state-of-the-art techniques is also noted.

Networking delays are inherent to any distributed system, and several ways of modelling these and taking them into account by scheduling policies were surveyed. Heterogeneity in the underlying platform is also a complicating factor in scheduling. Many scheduling policies were designed to manage heterogeneity where different processors ran at different speeds. This is less important for the industrial scenario, because the pressure to reduce power consumption leads to continual upgrades to use homogeneous state of the art processors.

Scheduling policies designed with respect to the concerns of users rather than those of the system owners were surveyed. While some papers propose policies for scheduling for fairness and responsiveness, these are static policies or could suffer from starvation under overload. Considering the industrial scenario, there seems to be a need for a dynamic scheduling policy that handles dependent workflows and delivers responsiveness and fairness to users, even in the case of overload. This problem is the motivation behind Hypothesis 1 described in Chapter 1. Specifying the value of jobs can help in the tradeoffs necessary under overload, and investigating this is the motivation behind Hypothesis 2 in Chapter 1.

Two insights in the literature are likely to be suitable in designing such a policy. Firstly, the use of the upward ranks [160] helps to inform prioritisation of tasks within

the same workflow. Secondly, the fairness approach of Zhao and Sakellariou [175] by running tasks that are the most late first would seem like a logical approach to apply to dynamic scheduling when the desire is to minimise lateness. Both of these insights as well as the ability to deliver responsiveness and fairness form part of the ordering stage of scheduling. Therefore, this thesis will concentrate on ordering policies that can be deployed as part of a list scheduler.

Chapter 4

Workload Characterisation

Good performance of a scheduler depends not only on the scheduling policy used, but also on the workload it is given to schedule. A scheduler may be ideal for one workload, but completely ill-suited to another. Proper evaluation of scheduling policies requires appropriate workload characterisations [57]. These characterisations can be used to create a wide variety of synthetic workloads with which policies can be evaluated in simulation. The industrial partner made 30 months of logs available for analysis, up to the end of August 2012. This chapter will characterise the workload placed on the partner's grid based on these logs.

Many different aspects can be considered when characterising a workload. This chapter will characterise the industrial partner's workload primarily from the angle of time, and the constraints and demands posed on the timing of workload execution.

To model the flow of jobs through a system, the patterns of arrivals of work will be characterised along with patterns of grid utilisation. These patterns will be investigated at the human timescales of days, weeks and years. The relationship between arrival and utilisation rates can illuminate times of overload when management of the queue is most critical.

The size of jobs, in terms of the amount of core-time they take to run, will also be characterised. Differing distributions of job sizes will pose different challenges to scheduling policies. Normal distributions [86] may be suited to a FIFO policy where most tasks wait for roughly the same amount of time.

A particular feature of engineering workloads is the structure of dependencies between tasks. These have been little investigated previously. The structure of dependencies constrains the possible orderings a scheduler can apply to task sets, because some tasks must finish first so that their results can be consumed by their successors.

This chapter will briefly summarise work where previous workloads have been characterised. A detailed characterisation of the workload is undertaken, based on log files obtained from the partner, including submission patterns (Section 4.2) and

execution volumes (Section 4.3). The structure of dependencies within the workload is also presented (Section 4.4). In some figures, the scales on the axes have been obscured to protect the interests of the industrial partner, although this does not influence the trends and distributions observed. Algorithms to generate synthetic workloads matching the observed distributions and structures are presented along with the appropriate characterisations.

4.1 Related Work

There is a large volume of literature on workload characterisation. Many surveys are concerned with utilisation patterns on web services, where some recent examples include those by Poggi et. al. [139] and Ren et. al. [141]. Web services tend to run well below their maximum possible utilisation, so as to have the capacity to scale up when peaks in traffic arrive. An overall utilisation of just 6% is noted by Poggi et. al. [139], 5-10% in Kavulya et. al. [92], while an average of 50%, rising to 70% at peak was observed in Ren et. al. [141]. Feitelson and Nitzberg [57] noted utilisations that varied between 40% and 80% depending on the time of day. Utilisations this low pose little challenge to a scheduler, as anything submitted can just run immediately, so any scheduling policy will achieve acceptable results. In research-oriented grids higher utilisation of between 90 and 100% has been observed [40, 172]. With utilisation this high, the implication is that jobs usually queue for some time (or *pend*) before being executed. As soon as jobs are queuing, the scheduling policy which manages the queue becomes important.

Patterns in arrival times over working days and weeks were noted in Chiang and Vernon [40], You and Zhang [172] and Feitelson and Nitzberg [57]. It comes as no surprise that the peak of job submissions appear during normal working hours. This feature of academic grids is in contrast to the web workloads, where peaks appear before and after usual working hours [141]. Weekends are also naturally quieter. Chiang and Vernon [40] found a roughly even distribution of jobs between the working days, whereas You and Zhang [172] found a high peak on Mondays, which decreased during the week. If the scheduler is aware of or responsive to these patterns, a scheduler might be able to make different decisions depending on whether it is the middle of the day or night.

The management of the queue is especially important where there is a wide variation in runtimes. Chiang and Vernon [40] and Feitelson and Nitzberg [57] observed that many workloads have a large number of small jobs that contribute only a small fraction of the load. Conversely, only the small proportion of large jobs contribute the bulk of the load. Effective prioritisation by the scheduler is required to keep the system responsive for the smallest tasks but also to avoid starvation for the largest ones.

Much work has been performed on how to schedule in the presence of dependencies between tasks on a grid [160] and how they can be modelled using Directed Acyclic Graphs [117]. However, the difficulty of scheduling DAGs on a grid depends significantly on the structure of dependencies within such graphs and the opportunity or otherwise of extracting parallelism. Little work seems to be have been performed on characterising the dependency structures within grid workloads. Examples of structured graph topologies are analysed in the literature [32, 73, 99, 131, 140, 160] but all of these looked at the internal structure of algorithms, rather than the composition of applications into workflows. In this chapter, dependency graphs from the industrial workload will be presented and characterised using several graph-theoretic metrics. A means of generating random graphs with a degree distribution matching that observed in industry will be presented.

4.2 Working Pattern of Designers

The users who place the vast majority of the load on the grid are the designers. These people work in a way which could be considered reasonably typical for an engineering group. The staff in the design team follow many natural rhythms in their work. This section will describe the rhythms found in the submission and execution of work on the partner's industrial grid. The figures presented in this section are based on data from log files spanning two and a half years. The author developed software in Python to parse the logs and then analyse and produce the charts shown in this chapter.

4.2.1 Submission Cycles

The simplest rhythm is the natural circadian cycle of working hours and mealtimes. Figure 4.1 shows the number of tasks submitted per 15-minute block throughout the day.

The highest rate of submission is during working hours (08:00-17:00). The pattern observed fits with similar observations of such daily rhythms [58, 105, 109, 139]. It is natural that the lowest level of work submission is overnight, when most workers are sleeping. There is a steady baseline level of work submissions even when no-one is at work, as the result of automated scripts. Unlike the workload of Poggi et. al. [139], however, because this is an industrial grid, working hours only occur on 5 days per week, not every day. On Saturdays and Sundays, only the baseline level of submissions take place (see Figure 4.2).

The working hours are not perfectly defined because the groups using the grid work in different time zones. However, the bulk of the work from the grid comes

from a single geographical region, where time zone differences are less than two hours. The peaks in job submissions are seen when the largest number of users are simultaneously at work, during the morning and afternoon. There is a distinct drop in the middle of the day when workers break for lunch.

The results returned by the jobs with the smallest execution times (minutes up to a few hours) can usually be analysed by the users within the same day, if they arrive in time. It is important to note that users will only start analysing returned results if there is sufficient time for them to do so before the end of the working day. If not, they tend to leave this analysis until the next day.

The design cycle for many designers is not so quick as to be able to get the results back the same day, and these designers follow a daily design cycle instead. They tend to expect results to be ready when they arrive at work in the morning, and they then analyse results and work on new designs during the day. Before they leave in the evening, they submit their revised designs to have their performance analysed overnight.

The rate of job arrival per hour can be normalised to a probability mass function (pmf) by dividing the counts by the total number of jobs submitted. This function can then be used to reproduce the pattern of load observed by adjusting the inter-arrival

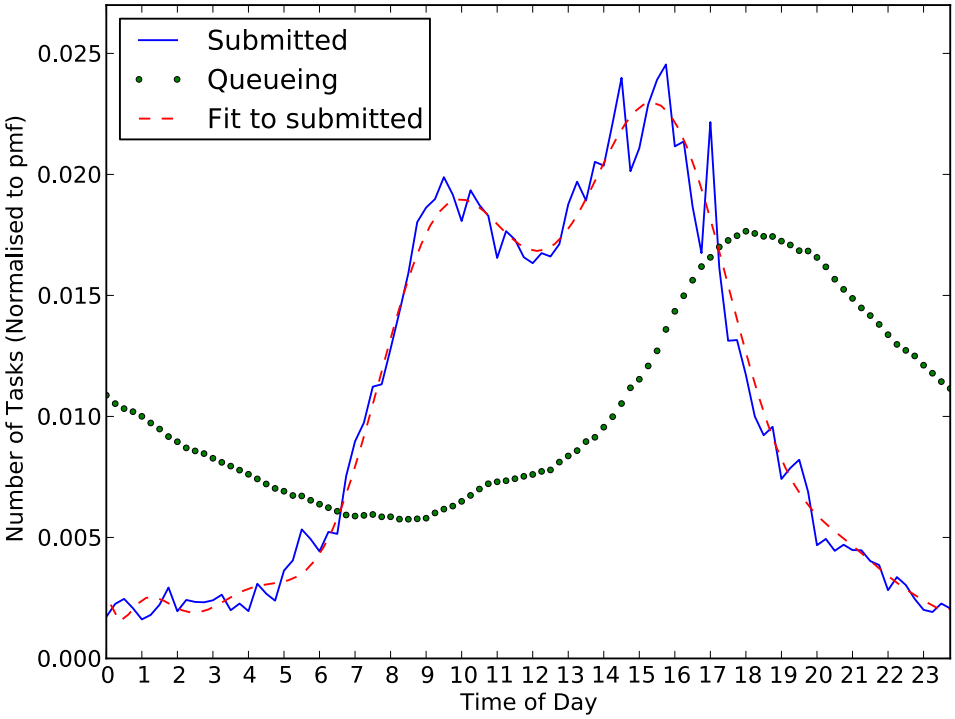


Figure 4.1: Daily Submissions and Queueing

#samples	200
$y =$	$0 \leq y < 24$
c	$3.41 \cdot 10^{-3}$
x	$-1.29 \cdot 10^{-2}$
x^2	$3.06 \cdot 10^{-2}$
x^3	$-3.13 \cdot 10^{-2}$
x^4	$1.73 \cdot 10^{-2}$
x^5	$-5.82 \cdot 10^{-3}$
x^6	$1.26 \cdot 10^{-3}$
x^7	$-1.85 \cdot 10^{-4}$
x^8	$1.88 \cdot 10^{-5}$
x^9	$-1.34 \cdot 10^{-6}$
x^{10}	$6.62 \cdot 10^{-8}$
x^{11}	$-2.24 \cdot 10^{-9}$
x^{12}	$4.93 \cdot 10^{-11}$
x^{13}	$-6.38 \cdot 10^{-13}$
x^{14}	$3.67 \cdot 10^{-15}$

(as shown in Figure 4.1)

(a) Daily

Mon	0.167
Tue	0.191
Wed	0.192
Thu	0.198
Fri	0.187
Sat	0.042
Sun	0.012

(b) Weekly

Table 4.1: Probability Mass Functions for submission rates

times of tasks using Algorithm 4.2. Table 4.1b gives the pmf values for arrivals on each day of the week. There were too many samples made on the time of day to give the pmf value for each time, so instead, the parameters of a polynomial function fitted using the least-squares method are given in 4.1a. Because the pmf is a mass function, not a density function, the number of samples matters, and is shown in the table.

Achieving responsiveness to serve these cycle times is paramount, because jobs that are 'late' affect the productivity of designers. The stages of aircraft design usually have fixed time budgets that the designers have to work to. The quality of a design is usually determined by the number of iterations the designers can perform within the given time frame.

However, in some ways there will never be enough computing power, because if there were, designers would run earlier stage simulations in higher fidelity. Whenever there is cluster capacity available, it tends to be used as much as possible. Most of the time, more work is submitted during working hours than can be processed immediately. Instead, work queues up during the day and this queue is drawn down overnight (Figure 4.1). This build up of work also happens over the scale of a week (Figure 4.2), where the queue length increases during the week, and is drawn down again at the weekend. From this, it can be seen that the grid spends a significant proportion of its time in a saturated utilisation state.

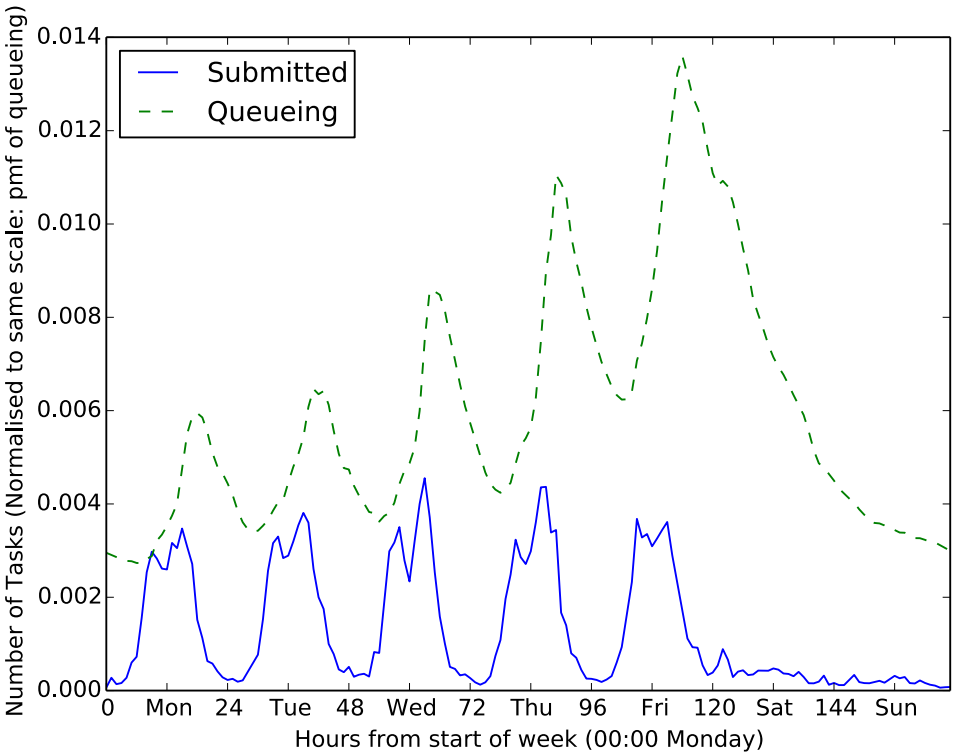


Figure 4.2: Weekly Submissions and Queuing

4.2.1.1 Submission Cycle Generation

These distinct patterns of variation in submissions pose significant challenges to schedulers, especially when the load is high enough to lead to periods of platform saturation. In order to properly evaluate scheduling policies, it is necessary to generate multiple representative workloads that follow these patterns.

The load placed by a workload on a platform can only ever be defined with relation to the platform. However, it is desirable to be able to adjust the loading factor independently of the workload and platform. This can be achieved by adjusting the inter-arrival times of jobs. This approach is advantageous, because it allows schedulers to be evaluated with the same workload at different loading levels. The algorithm for calculating the arrival times of each job for a given platform and workload is given in Algorithm 4.1. It works by calculating what the next arrival time would be if the current job could be perfectly parallelised across the whole grid, and increasing or decreasing the arrival time based on the desired load factor.

The algorithm in 4.1 is limited to giving a constant-load arrival rate, a poor reflection of the patterns observed. Algorithm 4.2 extends Algorithm 4.1 to set arrival times for every job in a workload by using probability mass functions (pmf)

Algorithm 4.1 Pseudocode to define job arrival time with desired load factor

Symbol	Parameter
c	Number of processing cores in the system
l	Desired loading factor (full load = 1)
j	Array of all jobs in workload
$j^i_{\texttt{exec}}$	Total load (core-seconds) of Job j^i
$j^i_{\texttt{sub}}$	Submit time of Job j^i

$$\begin{aligned} j^1_{\texttt{sub}} &= 0 \\ j^{i+1}_{\texttt{sub}} &= j^i_{\texttt{sub}} + \frac{j^i_{\texttt{exec}}}{c \cdot l} \end{aligned}$$

for the time of day and day of week. This new time point is then scaled on the load level desired and the pmf of the daily and weekly load distributions.

While this example assumes a pmf with a sample for each hour or each day, any probability mass function is possible, particularly where higher resolution is required. For example, the pmf given for the arrival times over the day defined in Table 4.1 and shown in Figure 4.2.1 actually uses 200 samples over the course of the day. In the case of this pmf, therefore, $p_d(h) = \left(200 \cdot \text{pmf}_{\texttt{day}}(h)\right)^{-1}$.

4.2.2 Grid Utilisation Cycles

Careful scheduling is especially necessary when the grid is under transient periods of overload (when the arrival rate of work exceeds the ability of the grid to process this work), and when the grid is operating at its maximum realistic capacity. This section investigates the utilisation of the grid over the course of a day and week. This is done by showing the distributions of utilisation for each hour or day encountered in the logs. In calculating the utilisation, only the fraction of time used by any task running within that hour or day was counted.

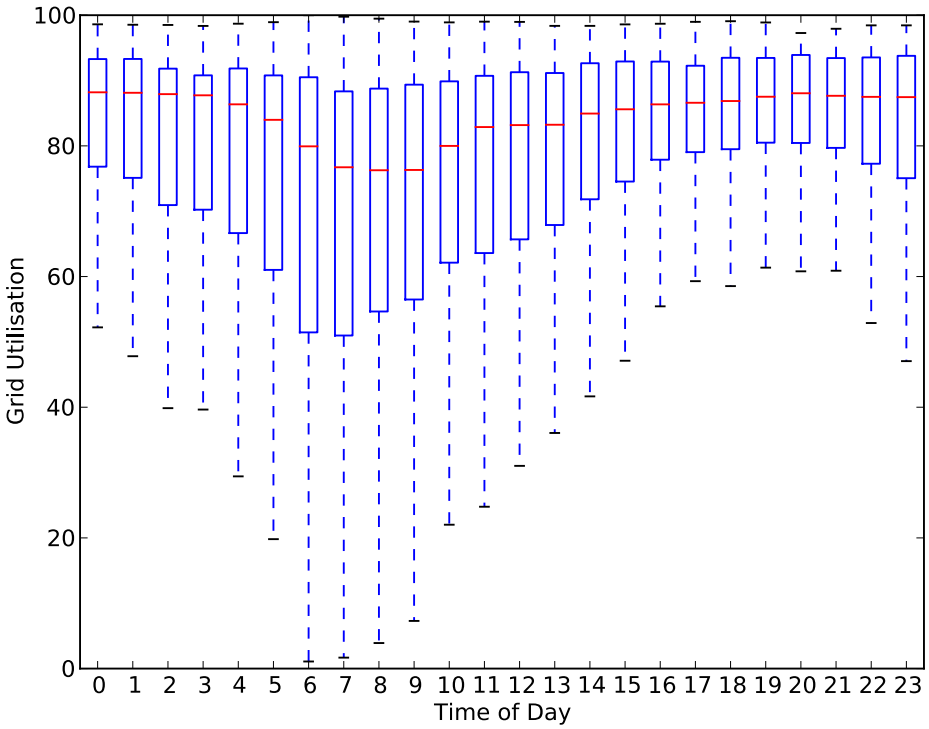
Figure 4.3 shows the distribution of utilisation of the cluster cores by time of day. It should be noted that this chart shows the utilisation of all cores in the grid, including those of specialised architectures. This means that above utilisations of about 80%, some work will be almost certainly be queueing, because it is limited as to which cluster or architecture it can run on.

Furthermore, a large fraction of the tasks on the cluster run on a number of cores simultaneously. When one of these tasks reaches the head of the task queue, their current scheduling policy waits until sufficient cores are free before starting the task, due to the absence of backfilling. These factors mean that actual full utilisation of the

Algorithm 4.2 To generate submission patterns by changing inter-arrival times

Symbol	Parameter
c	Number of processing cores in the system
l	Desired loading factor (full load = 1)
j	Array of all jobs in workload
$j^i_{\texttt{exec}}$	Total load (core-seconds) of Job j^i
$j^i_{\texttt{sub}}$	Submit time of Job j^i
$\text{pmf}_{\texttt{day}}(h)$	Probability mass function of arrivals over a day (by hour), such as in Table 4.1a
$\text{pmf}_{\texttt{week}}(d)$	Probability mass function of arrivals over a week (by day), such as in Table 4.1b

```
set_arrival_times (c, l, j, num_samples) :
id, binid, last_fill, last_sub, time_increase = 0
for day in 0..6 :
  for smp in 0..num_samples :
    min_bin[id] = 60 · 24 · 7 · pmf_day(day) · pmf_week(smp)
    id = id + 1
for j^i in j :
  newmins = j^i_exec / (c · l)
  binfill = last_fill + newmins
  if binfill < min_bin[binid] :
    last_fill = binfill
    time_increase = 60 / num_samples · newmins / min_bin[binid]
  else :
    acc = 0
    while binfill ≥ min_bin[binid] :
      acc = acc + 60 / num_samples · (min_bin[binid] − last_fill) / min_bin[binid]
      last_fill = 0
      binfill = binfill − min_bin[binid]
      binid = (binid + 1) mod id
    last_fill = binfill
    time_increase = acc + 60 / num_samples · binfill / min_bin[binid]
  last_sub = last_sub + time_increase
  j^i_submit = last_sub
```



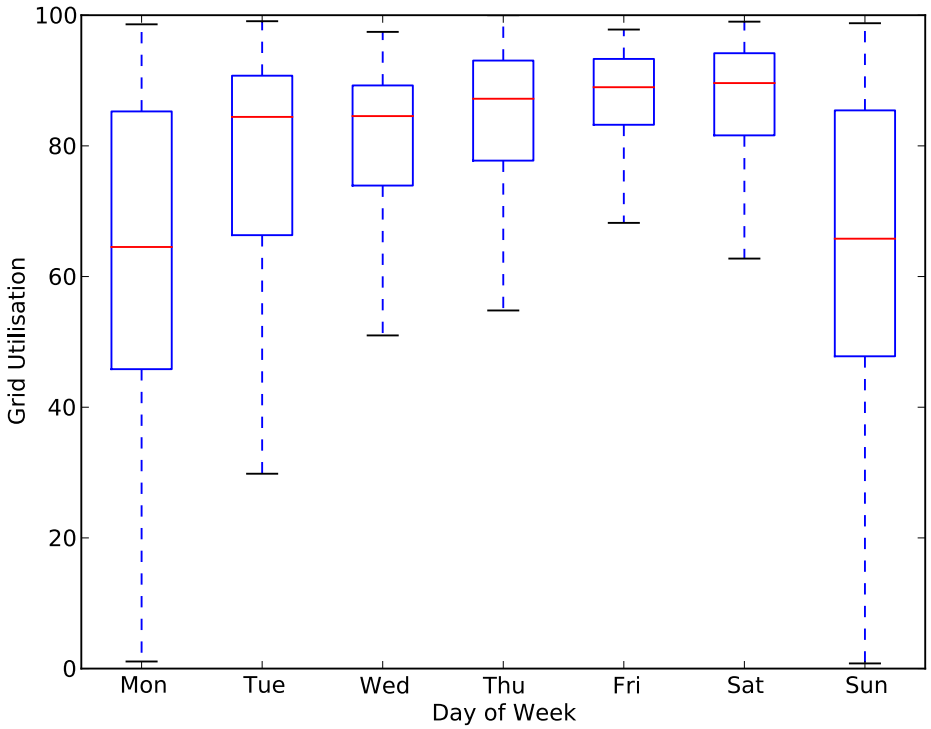
Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 4.3: Daily Utilisation

grid is almost unachievable and that tasks will be queuing well below 100% utilisation on some clusters.

There is significant variation, because of the large number of days that were sampled. However, the variance decreases at the end of the day, and shows how cluster utilisation rises to saturation at the end of every working day. The work submitted each day is only caught up on overnight, reinforcing the impression from Figure 4.1. The lowest point of utilisation tends to be around the time people arrive at work in the morning, when as much as possible has been caught up on overnight. This is in distinct contrast to the web workloads observed by Li et. al. [109], who saw peak utilisations of around 30%, or Kavulya et. al. [92] with a utilisation of 10%.

This cycle of work queueing up and only being caught up on when the staff are not present is manifest on a weekly basis as well (Figure 4.4). Only Sunday and Monday have median utilisations much below saturation point. During the week, the average utilisation increases as more work is submitted during each working day than can be processed by the next day (corroborated by the average queue length in Figure 4.2). Monday has somewhat lower average utilisation because the most likely times for the grid to have any idle time is before the staff arrive on Monday morning.



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 4.4: Weekly Utilisation

While most of the work is caught up on during the week, there are also seasonal trends in the workload submitted. Figure 4.5 shows the number of tasks submitted in each week of the year. The most striking feature of this is the impact the summer holidays have on the number of tasks submitted. The last week of the year - between Christmas and New Year - also has very few tasks submitted. Although very few jobs are submitted during these periods, they are also one of the busiest times for the grid. Many of the designers run their largest, most detailed jobs over the holidays, when the long computation time doesn't have an impact on their productivity, because they are away anyway.

As reported by users, and as reflected in the figure, it can take some time to analyse the results of these larger jobs, which is why it takes time for the usage to rise again after the summer holiday. However, this is also because it takes a long time for the grid to draw down the large tasks that are submitted before the holidays. When pending times are significant because there is still a large amount of queued work on the grid after the holiday, they are less likely to submit all but the most important work. This pattern does not manifest itself during single weeks, though, where a very similar volume of work is submitted on each working day.

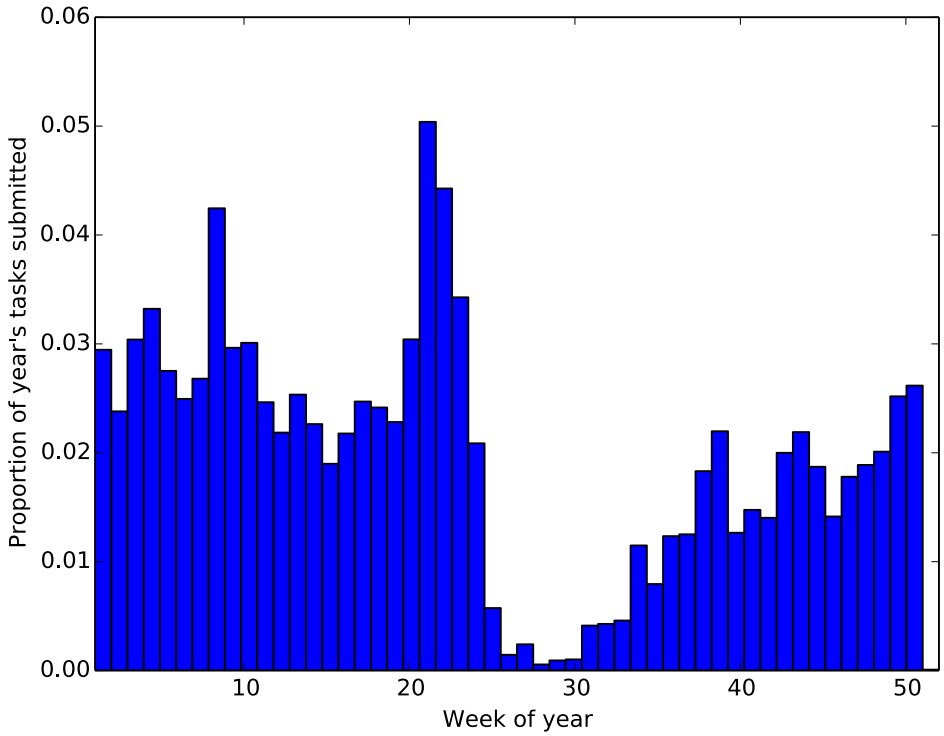


Figure 4.5: Annual patterns

Another feature of the view across the year is the wide range of number of tasks submitted per week. These are subject to business cycles, as projects come and go between different teams of designers.

The layering of these cycles gives rise to recurrent peaks in the arrival rate of work, which the grid only just manages to catch up on before the next wave of work arrives. These peaks follow daily, weekly and annual cycles in addition to the cycles imposed by the flow of business projects.

In such a sizeable grid, tasks will be arriving and finishing at a fairly high rate. Figure 4.6 shows the probability of having to wait longer than a certain number of minutes for a task to arrive or finish. Because of the high variability in arrival rates, sometimes the arrival rates are very high. This is why the the probability of having to wait a long time for the next arrival of a job is low, and is lower than the probability of waiting for a job to finish at lower timescales. Above about 120 minutes, the daily, weekly and seasonal cycles means there is more variability in the arrival rate, giving a higher probability of waiting longer for the next job to arrive than finish,

The finish rate of jobs is more constant, which is why the probability of a job finishing in a given time is lower under about 120 minutes. However, the probability of finish is still remarkably high, and it follows a power law between the points with

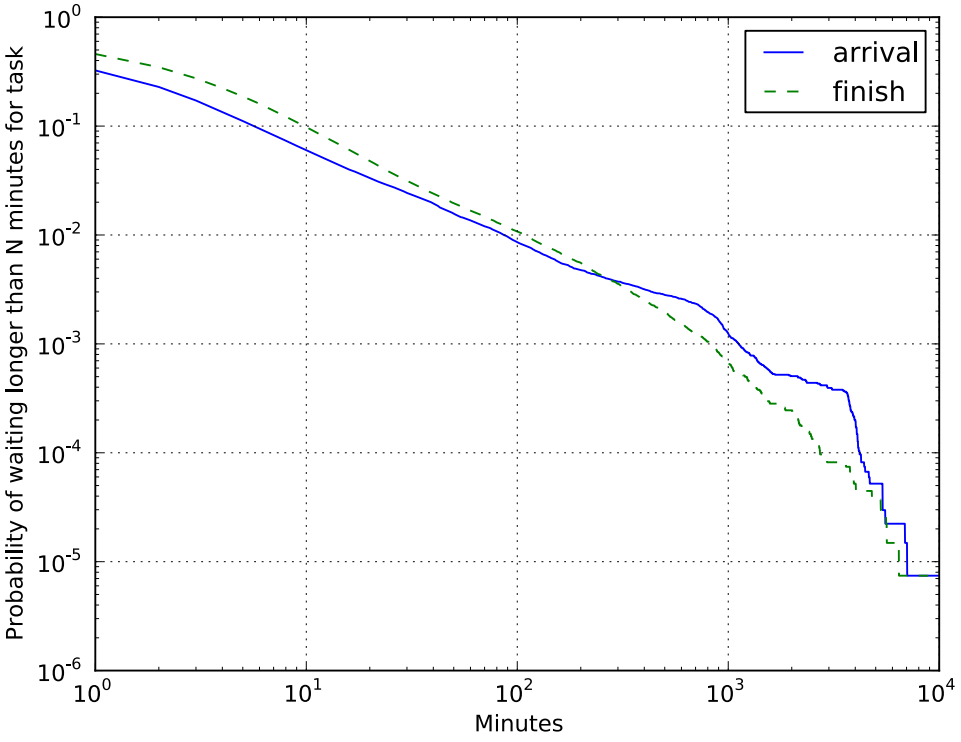


Figure 4.6: Inter-arrival & inter-finish time probabilities

a 10% chance of waiting longer than 10 minutes for the next job to finish, to 0.1% for 1000 minutes. Beyond 1000 minutes, the lines become aliased because of the very few occurrences of there ever being wait periods this long.

In summary, the findings of this section are that work is submitted in daily, weekly and seasonal cycles. During working hours, work is submitted faster than it can be processed, and queue length increases accordingly, only being drawn down outside of working hours. The grid therefore spends a great deal of its time in a saturated state. Ideally, a scheduler could take into account these patterns in order to better optimise for small tasks during the day and longer tasks at night or at the weekend. Due to the large volumes of work passing through, the inter-arrival and inter-finish times of work are low. This suggests that the current setup of not using a pre-emptive scheduling policy is not a hindrance, because something is always about to finish.

4.3 Workload Composition

Engineering designs are made by hierarchically decomposing the problem into small parts, and then composing the completed designs until a final, complete design is reached. Early stage designs require low-fidelity and so need only a small amount of

computation time for each CFD simulation. However, these are iterated over quickly (up to several iterations per day) and so there are a large number of these small tasks. As designs progress, the models considered get more complicated and require improved fidelity. This naturally requires more compute time for simulations. The largest jobs used for certification of an entire aircraft in high fidelity are very compute-intensive, and may need to execute over many months.

The hierarchical composition of the design process suggests a workload that follows exponentially-distributed patterns. Notably, this is in contrast to previous research that has suggested alternative distributions of work found in large-scale grid systems [92], who observed a log-normal distribution. The characterisation in this section reinforces this suggestion.

4.3.1 Volume

Figure 4.7 shows the execution times of all the jobs in the 2.5-year workload, sorted by execution time. Where some jobs run over many cores, the execution time is given multiplied by the number of cores used. Therefore, the size of jobs is measured in core-minutes. The striking feature of the graph is the straightness of the line, when the job size is plotted on a logarithmic scale. This suggests that the distribution of job

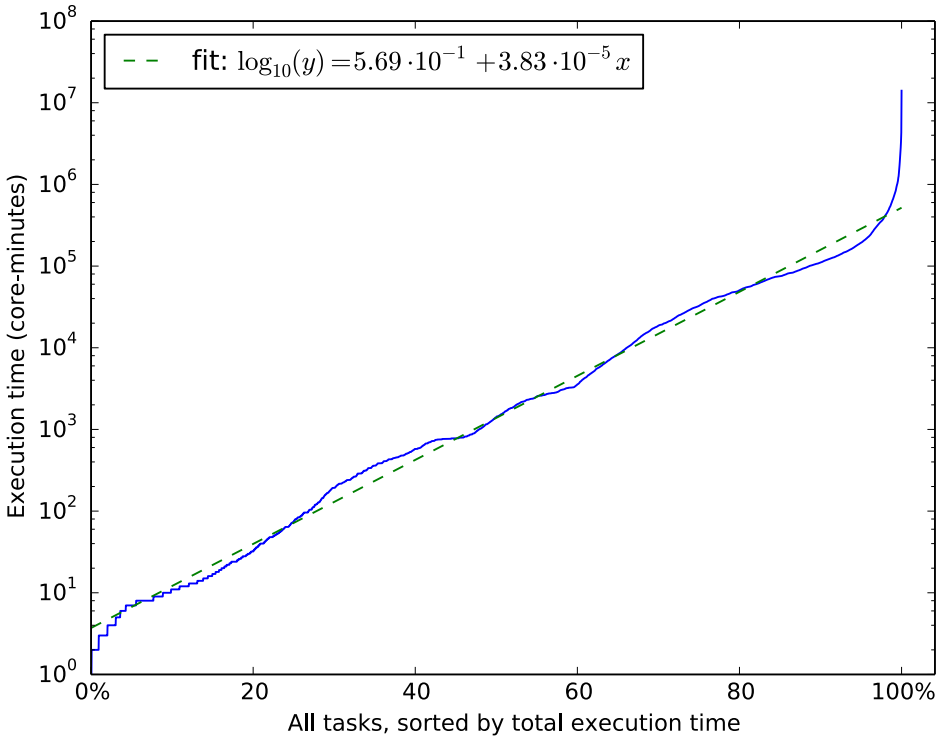


Figure 4.7: Job Volume Distribution

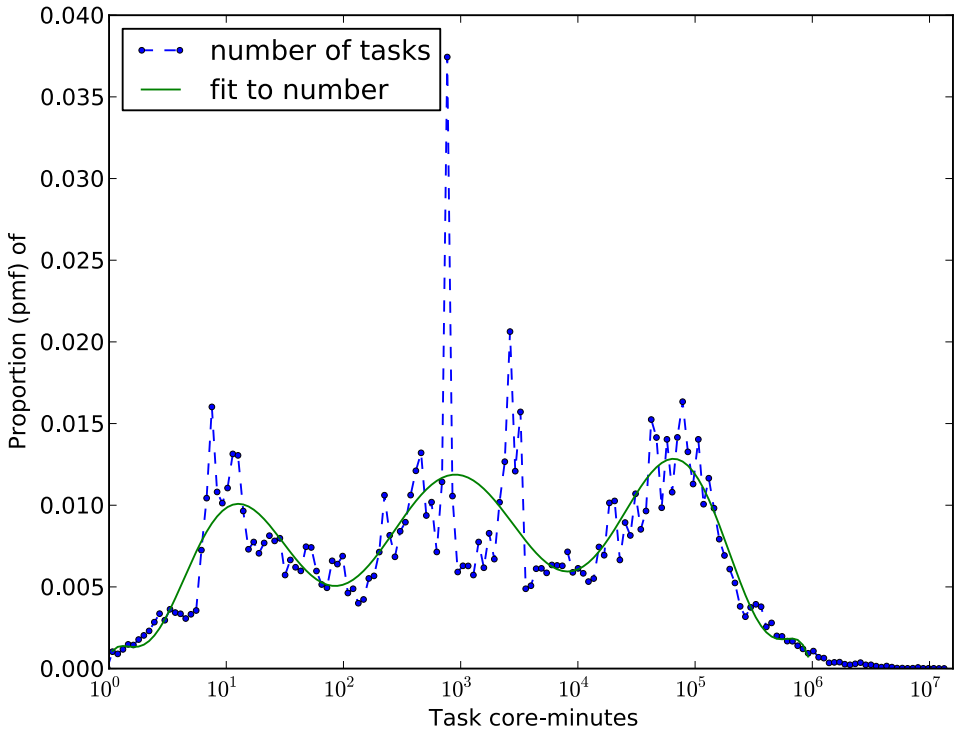
execution times follows a log-uniform distribution, at least between 10^1 and 10^5 core-minutes. This suggests that there are a roughly similar number of large and small tasks, with the median task execution time being approximately 1000 core-minutes.

The gradient of the slope is steeper below about 10 core-minutes of computing time. This is likely because it is usually not worthwhile for users to submit such small jobs to the grid, when they could easily be run on a local PC. Some small jobs are still run on the grid, however, when memory or transient disk space requirements are greater than those available on a desk PC. Logging, system maintenance or data transfer tasks may also be present in these small jobs. The aliasing present in the curve at the low end is due to the logs only recording times to the nearest minute. The flattening of the slope in the middle of the curve indicates a particular peak of jobs around 10^3 core-minutes. This is likely to show the peak of jobs submitted where the results are needed within the same day for fast iteration.

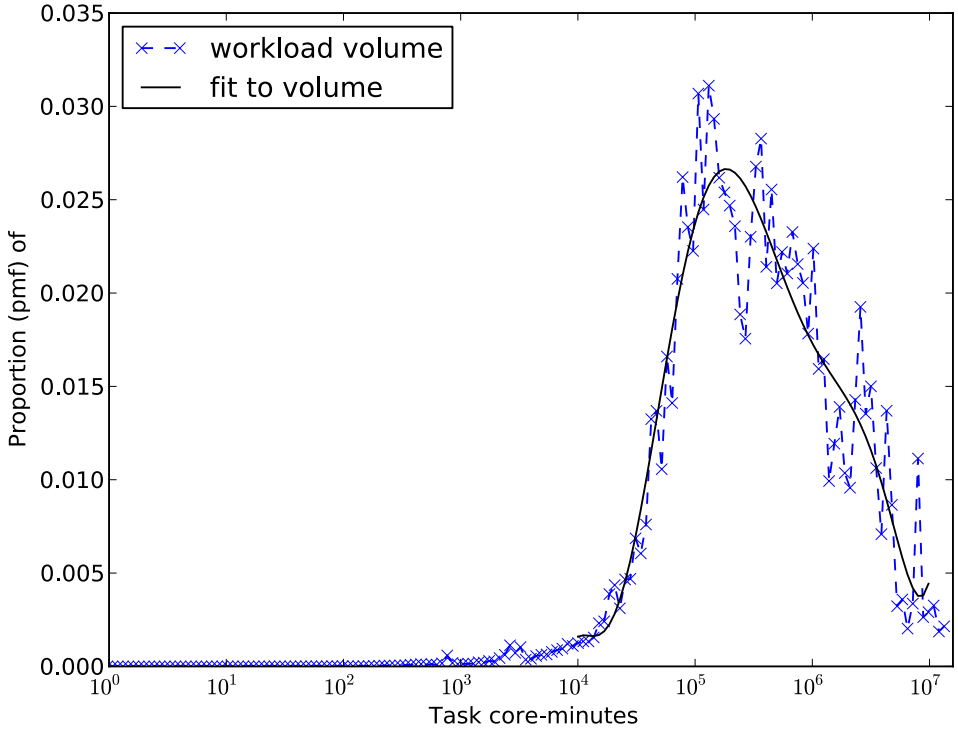
An alternative view of this data is through a logarithmic histogram of the tasks' execution times, shown in Figure 4.8a. Here, the uniform nature of the distribution is still apparent, at least between 10^1 and 10^5 core-minutes. In this view of the data, three distinct peaks of work are apparent. The first peak, centred on 10^1 core-minutes is likely to correspond to small tasks used for system maintenance or data transfer. The second peak, at around 10^3 core minutes, or 16 core-hours, corresponds to the tasks submitted where results are required during the same working day. If 64 cores were allocated to a job of this size, the computation time would be 15 minutes. The final peak, at around 10^5 core-minutes or 70 core-days, corresponds to the tasks that need to be returned overnight. If 128 cores were dedicated to this job, about 13 hours would be required.

An important feature of the distribution is the small number of jobs that are very large. These are the jobs that are run in order to put a high-quality airframe model through rigorous testing, which goes towards the certification of an aircraft. Within the logs that were analysed, there were 28 tasks that took over 10 core-years of CPU time (10^7 core-minutes) to complete. Even with 128 cores allocated to them, these jobs would take over 2 months to complete execution. These jobs are not jobs that have overrun in error, because their sheer size means that they would have been closely monitored by system administrators, and have had specific approval given to run.

The fact that the workload has a similar number of small and large jobs could distract from the fact that the larger jobs represent a much larger fraction of the load placed on the cluster. Figure 4.8b also shows the proportion of the workload volume placed on the cluster by job size. While the majority of jobs in terms of numbers execute in less than 10^4 core-minutes, this figure shows that their contribution to the load is small. The bulk of the load comes from jobs between $10^{4.5}$ and $10^{6.5}$ core-minutes. This poses further challenges to schedulers, because of the risk of the shorter jobs, which require higher responsiveness, having to queue behind the large jobs.



(a) Distribution of number of tasks



(b) Distribution of task volume

Figure 4.8: Workload Volume Distributions

To be able to reproduce these distributions, polynomial curves have been fit to the distributions observed in Figures 4.8a and 4.8b. The parameters of these curves are given in Table 4.2.

$p(\log_{10}(y)) =$	number (pmf)	volume (pmf)
#samples	200	200
degree	$0 \leq \log_{10}(y) \leq 6$	$4 \leq \log_{10}(y) \leq 7$
c	$8.04 \cdot 10^{-4}$	$-5.81 \cdot 10^{2}$
x	$1.41 \cdot 10^{-2}$	$7.82 \cdot 10^{2}$
x^2	$-1.14 \cdot 10^{-1}$	$-4.47 \cdot 10^{2}$
x^3	$3.71 \cdot 10^{-1}$	$1.41 \cdot 10^{2}$
x^4	$-5.06 \cdot 10^{-1}$	$-2.64 \cdot 10^{1}$
x^5	$3.60 \cdot 10^{-1}$	$2.94 \cdot 10^{0}$
x^6	$-1.49 \cdot 10^{-1}$	$-1.81 \cdot 10^{-1}$
x^7	$3.69 \cdot 10^{-2}$	$4.72 \cdot 10^{-3}$
x^8	$-5.42 \cdot 10^{-3}$	-
x^9	$4.35 \cdot 10^{-4}$	-
x^{10}	$-1.47 \cdot 10^{-5}$	-

Table 4.2: Job Number and Volume Curve Fit Parameters

4.3.1.1 Volume Distribution Generation

Generating workloads with job execution times that conform to a realistic distribution is crucial when evaluating the effectiveness of scheduling policies to apply to a grid for a given organisation. This is especially the case where the workload has such a wide variation of execution times as the one observed here.

In Algorithm 4.3, a method of creating workloads sampled from log-uniform distributions is presented. The specified parameters represent those found in the industrial workload. The expression uniform (a, b, k) represents a function returning k random samples from the uniform distribution $[a, b)$.

Algorithm 4.3 Task Execution Time Generation

$$\begin{aligned} \text{base\_samples}[1..n] &= \text{uniform}\left(0, 1.34 \cdot 10^{4}, n\right) \\ j^{i}_{\texttt{exec}} &= 10^{\left(3.83 \cdot 10^{-5} \cdot \text{base\_samples}[i] + 56.9\right)} \end{aligned}$$

4.3.2 Multi-Core Tasks

A feature of intensive simulation workloads are tasks that must execute over a number of cores. Particularly in the case of this workload, multi-core tasks must execute on the number of cores specified simultaneously. This is due to the structure of the particular CFD flow solvers used. The volume of space to be simulated is broken up into segments using a mesh. Each point in the mesh has a calculation performed for each time step, and then the results of that point are cascaded to all its neighbouring points. A large number of time steps are usually needed to achieve either convergence, for steady-state solutions, or a time-varying field, for solutions where the study of turbulence is important.

It is important to note that these multi-core tasks are considered by the grid system to be single tasks, as one piece of software executes, just over multiple cores. This is in contrast to the dependencies between potentially different pieces of software, described in Section 4.4, which join tasks together to form jobs.

The exact number of cores used for a task is flexible before the task has started, and is informed by several constraints. For larger tasks, there is often a minimum number of cores required due to memory requirements. Large tasks often need more Random Access Memory (RAM) than is available on any single computing node. Exceeding

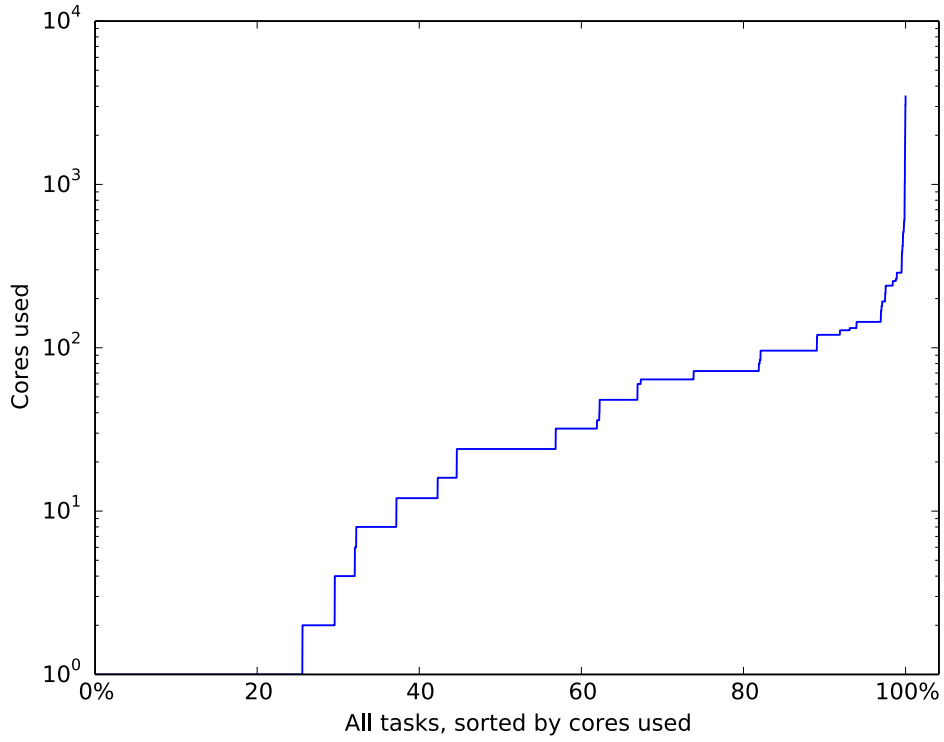


Figure 4.9: Workload task count by cores used

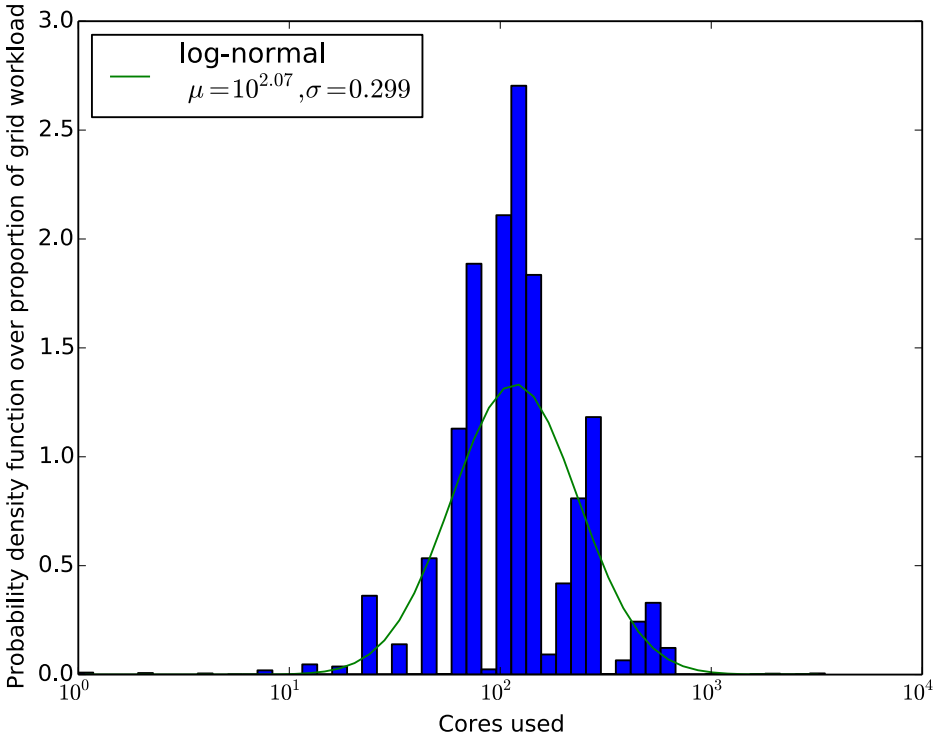


Figure 4.10: Workload volume by cores used

the available RAM and moving into swap space on disk is highly detrimental to the performance of tasks. Additionally, increased parallelism leads to the whole task executing in a shorter period of time, because the work is divided between more cores.

Although the workloads scale reasonably well in adding more cores, there are softer constraints on the maximum number of cores used. As the number of cores increases, so does the network bandwidth required to synchronise the points at each time step in the CFD solver. Even though the network bandwidth internal to the clusters is large, the accumulation of small delays mean that diminishing returns are available from further parallelism above several hundred cores per task. Furthermore, tasks requiring a larger number of cores can take longer to start executing, because it takes longer for sufficient cores to become free.

Users tend to have a good idea about the kinds of jobs they submit, however, and can choose the number of cores to assign to a task at submission time. The distribution of cores per task is shown in Figure 4.9. This shows that the majority of tasks use less than 100 cores. Around a quarter of tasks run on only a single core, as well. The step-function nature of the distribution shows that not all possible numbers of cores are used. Instead, users are instructed to select a number of cores that is a

multiple of the number of cores in the servers available. This enables their tasks to use complete multi-core servers to work on, with the aim of reducing memory capacity conflicts between tasks sharing the same compute server and to minimise network communications between servers, where possible. The administrators also suggest that bin-packing tasks onto the clusters is easier when tasks all have set blocks of core sizes, especially in, for example, powers of 2.

Although the single-core tasks are a quarter of the number of tasks submitted, these tasks place very little load on the cluster. Unsurprisingly, the tasks that place more load on the cluster are those that are assigned more cores to execute with. The load placed on the cluster by tasks with a given number of cores is shown in figure 4.10. This roughly approximates a log-normal distribution with a mean of 100 cores. As before, this shows the number of cores used to be rounded off to an appropriate multiple of the number of cores per server.

The most highly-parallel jobs here do not actually contribute most of the load to the cluster, even though figure 4.8a shows that the largest jobs do contribute most of the load. This means that at least some of the largest jobs are not run on the largest number of cores available. This is likely due to several factors. Firstly, the largest jobs are also some of the least urgent, and so users do not mind waiting a long time. As previously mentioned, the inefficiencies inherent in scaling to larger levels of parallelism may also mean that some of these large jobs do not actually benefit all that much from further parallelism. In fact, they may take up more of the grid's resources at one time (disadvantaging other users), without much of a net gain for the user who submitted the job. Furthermore, in order to achieve good packing of jobs to clusters, jobs of the same size are preferred.

4.3.3 Groups

The industrial partner is naturally organised into many different departments, which are organised into groups. When users submit jobs, their work is tagged with their name and the group they are a part of. Some users are part of multiple groups, and submit according to what work they are doing. Figure 4.11 shows the distribution of work volumes by the groups that compose the organisation. Similar to the previous figures, the distribution shows a straight line distribution on a log scale. This shows that the group volumes submitted follow a log-uniform distribution for almost all the groups. There are a few large groups that break this trend, as can be seen from the uptick at the top of the line. These groups are those that submit the jobs for the large-scale aircraft certification activities.

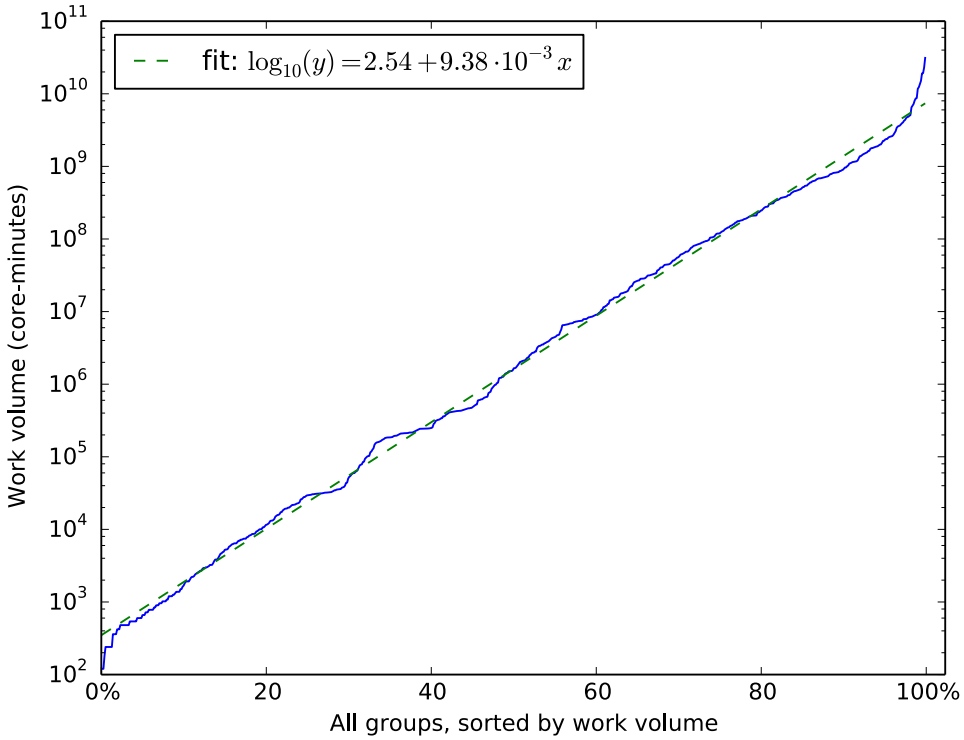


Figure 4.11: Workload by groups

This section has shown that task volumes fairly closely follow a log-uniform distribution in their execution times, which can be described by a log-linear trend through a sorted list of their execution times. As has been found by previous studies [40, 57], there are a smaller number of large jobs, but they contribute a very large share of the workload. This is to be expected from such a log-uniform distribution of task execution times. The volume of work by cores required is found to follow a log-normal distribution with a mean of 100 cores. This reinforces the observation that the 25% of tasks that are single-core contribute little to the workload volume.

4.4 Dependency Structures

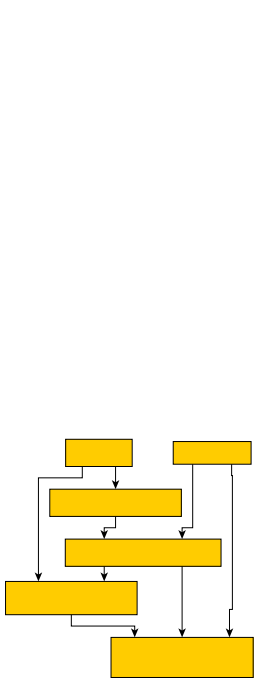
The structure of dependencies is a key aspect of characterising the engineering workload, where each task run on the grid is part of a job and a higher-level workflow. Dependencies have been widely modelled in previous work by assuming that they are Directed Acyclic Graphs (DAGs) [117, 160]. However, simply stating that the dependencies are DAGs gives no further information about the internal structure of these graphs. Some method must be used to create structure when generating workloads.

Previous work has considered graphs that represent the data flow internal to particular algorithms. Kwok and Ahmad [99] give several patterns that follow common algorithmic structures, including linear chains, fork-join and diamond. Ranaweera and Agrawal [140] mentioned using DAGs following the 'Cholesky decomposition' and 'Gaussian elimination' algorithms. Topcuoglu et. al. [160] considered 'Fast Fourier Transformations' as well as the unstructured shape of a molecular dynamics application. Olteanu and Marin [131] give a survey of graph structures and the parameters used to generate them, but without giving the generation algorithms. These graphs used inside algorithms tend to be highly structured and can be generated by repeating or nesting fixed structures. Some distributed algorithms run on the grid contain algorithms such as these, so algorithms to generate these structures are given in Section 4.4.1.

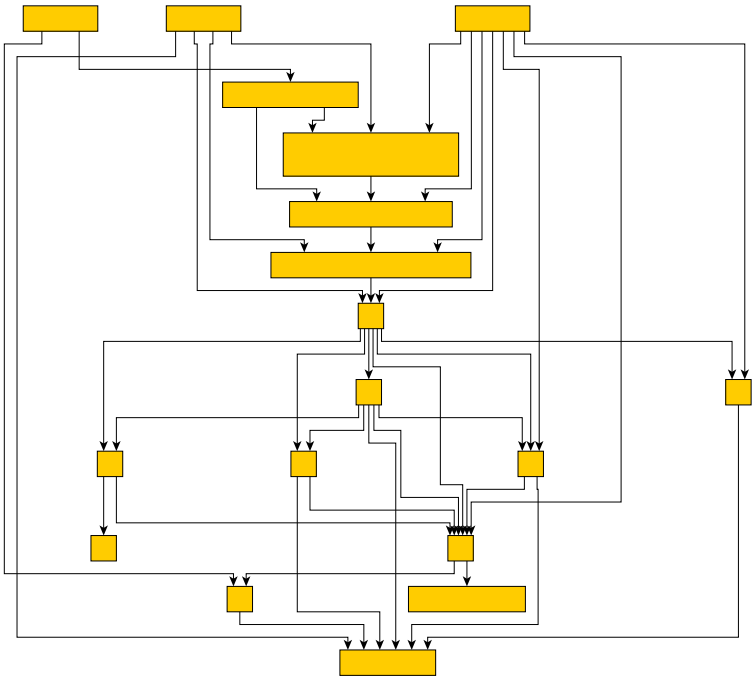
While the algorithms working inside individual pieces of software give highly structured graphs, workflows are composed of many pieces of software. A challenge in analysing dependency patterns is that the grid manager did not include dependencies in its log files. However, the submission software employed by the users does store the structure of the workflows. Although it was not possible therefore to make statistical generalisations of the frequency of dependency patterns within the workflows, common structures can be described.

Figure 4.12 shows three workflow structures obtained from the workflow submission tool. The three examples represent the least complex, the average and the most complex workflow examples that were found when analysing the submission dependency patterns qualitatively. Table 4.3 gives several graph-theoretic metrics applied to the industrial workflow patterns. The most pertinent feature of these graphs is that they have lower levels of structure, and instead have more in common with graphs where the edges are placed randomly. However, the distribution of node degrees is not uniform, as would be expected with a truly random graph. Instead, the degrees of nodes follow an exponential distribution.

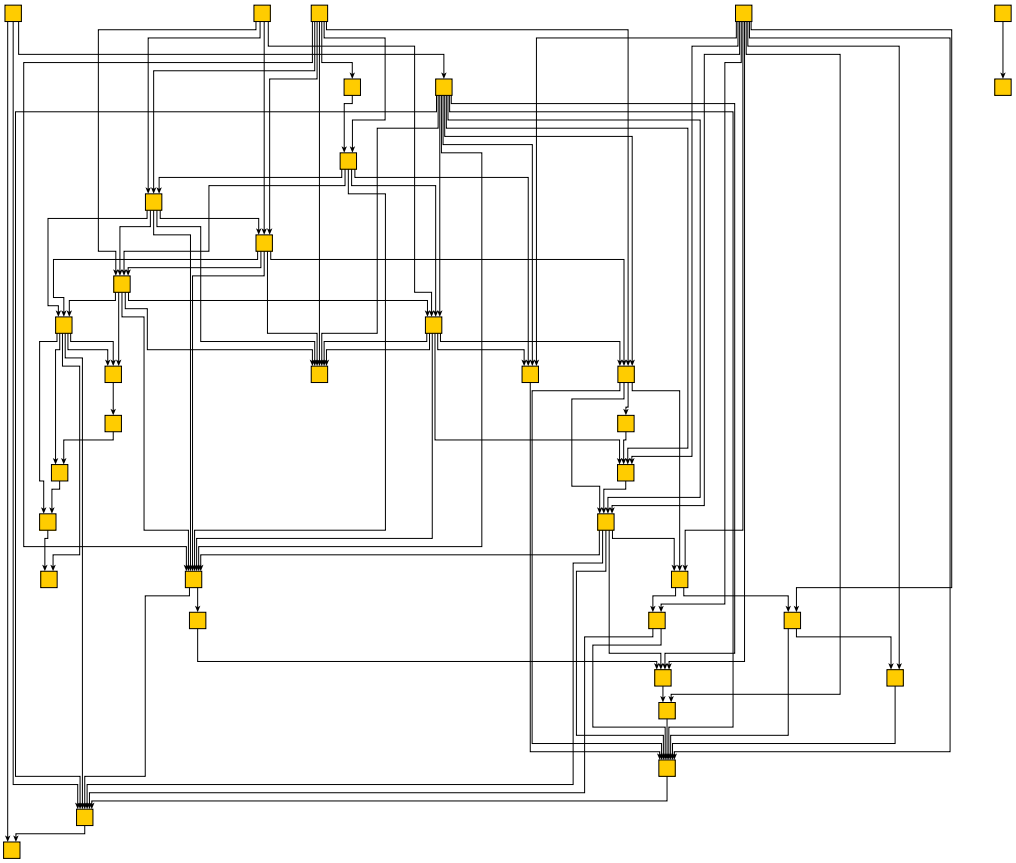
The common Erdős–Rényi graph generation algorithm [55] generates graphs by having all possible edges present with a given, fixed probability. Section 4.4.2 shows how the Erdős–Rényi algorithm gives node degree distributions that do not match that observed in the industrial dependency graphs. Therefore a new algorithm is presented (Algorithm 4.8) to generate random graphs that respects the exponential distribution of degrees found in the industrial graphs.



(a) Simple



(b) Average



(c) Complex

Figure 4.12: Dependency Patterns

	Simple	Average	Complex	Erdős–Rényi	Exponential Degree
Nodes	6	18	36	36	36
Edges	8	39	98	98	98
Edge Density	26.6%	12.7%	7.78%	7.78%	7.78%
Sources	2	3	5	5	5
Sinks	1	3	5	5	5
In-degree μ	-	-	2.69	1.33	2.72
In-degree σ	-	-	2.15	1.31	2.37
Out-degree μ	-	-	2.64	1.33	2.72
Out-degree σ	-	-	2.61	1.2	2.61

Table 4.3: Dependency Graph Metrics

4.4.1 Structured Graphs

4.4.1.1 Linear Pattern

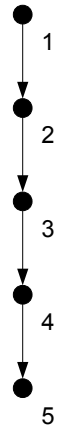
The most basic DAG dependency pattern is that of linear dependencies. This is when there is a single chain of purely sequential tasks with dependencies between them, as shown in Figure 4.13a. However, this pattern could well be considered unrealistic for a grid workload. This is because grids tend to perform best on parallel workloads, so it is highly unlikely that a substantial part of any real grid workload would be composed of linear dependent chains of work. Nevertheless, if it were, an appropriate scheduling policy could be a pipeline arrangement. The pseudocode to set up dependencies like this is shown in Algorithm 4.4.

4.4.1.2 Fork-Join Pattern

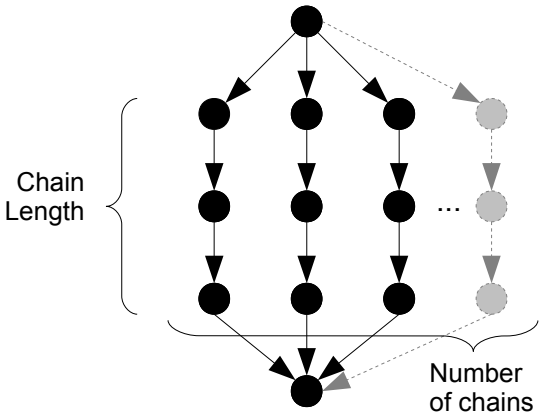
Many workloads are parallelised by applying the same sequence of operations to different chunks of data [99]. Each chain is one following the linear dependencies pattern. These chains are spawned by an initial setup task and the results are collected by a final task. This is inspired by the MIMD (Multiple Instruction Multiple Data) parallelism pattern. A diagram showing this arrangement is shown in Figure 4.13b. Pseudocode for generating such a configuration is shown in Algorithm 4.5.

4.4.1.3 Diamond Pattern

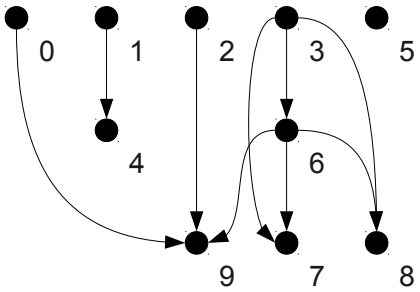
The diamond pattern (called mean value analysis by Kwok and Ahmad [99]) as shown in Figure 4.13c. This is similar to the fork-join model, but where the fork stage does not take place all at once, but requires several stages to perform. It could also be considered like a binary tree branching out to the maximum width, and then condensing down again to collect up the data. Pseudocode for defining these dependencies is given in Algorithm 4.6.

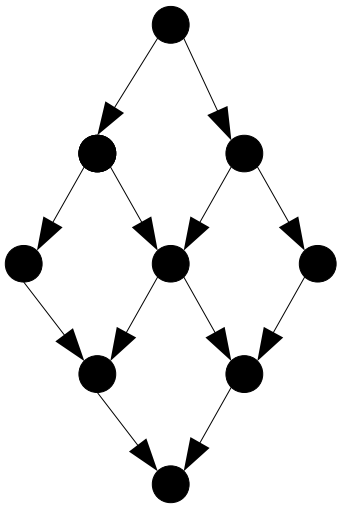


(a) Linear Dependencies

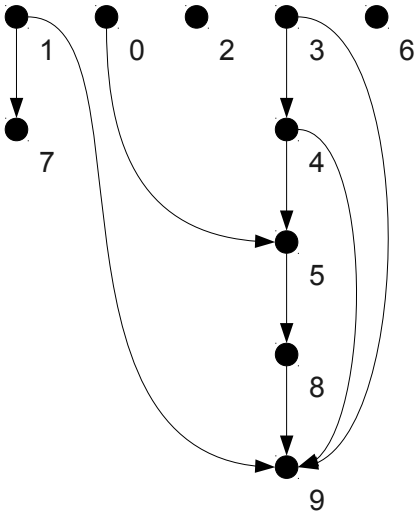


(b) Fork-Join





(c) Diamond



(d) Random (Erdős–Rényi) ($T = 10, P = 0.3$)

Figure 4.13: Dependency DAG shapes

Algorithm 4.4 Pseudocode for the Linear Dependencies pattern

```
N is the number of tasks
task[1].dependencies = {}
for taskid in 2..N :
    task[taskid].dependencies = {task[taskid − 1]}
```

Algorithm 4.5 Pseudocode for the Fork-Join pattern

```
fork_join_pattern(num_chains, chain_length) :
all_tasks = empty list of tasks
inner_matrix = matrix of tasks (num_chains by chain length)
for x in 1..num_chains :
    for y in 2..chain_length :
        inner_matrix[x, y].dependencies.add(inner_matrix[x, y − 1])
        all_tasks.add(inner_matrix[x, y])
    inner_matrix[x, 1].dependencies.add(initial_task)
    final_task.dependencies.add(inner_matrix[x, chain_length])
all_tasks.add(initial_task)
all_tasks.add(final_task)
return all_tasks
```

Algorithm 4.6 Pseudocode for the Diamond pattern

```
d = diamond_edge_length
task_matrix = matrix of tasks (d by d)
for x in 1..d :
    for y in 2..d :
        if x > 1 :
            task_matrix[x, y].dependencies.add(task_matrix[x − 1, y])
        if y > 1 :
            task_matrix[x, y].dependencies.add(task_matrix[x, y − 1])
```

4.4.2 Random Graphs

These strictly regular structures do not completely represent what was observed in the industrial workflows. The fork-join model of computation tends to happen inside each multi-core task, rather than at the job level. Some chaining is present, but it is not perfectly linear. The key tasks with large computation times tend to take a proportionately large number of inputs and have their output consumed by a proportionately large number of successors. Overall, industrial job dependency graphs have less structure than these fully-structured graphs. This section will present methods for generating DAGs with elements of randomness.

4.4.2.1 Erdős–Rényi (Probabilistic Edge Presence)

A common way of generating DAGs with random structure is to use the Erdős–Rényi [55] model to create random graphs, with an algorithm to do so given by Tobita and Kasahara [159]. In this method, each possible edge in a graph is present with a given probability (Algorithm 4.7). Two sample task graphs are shown in Figure 4.13d.

This algorithm has the advantage that the shape of the dependency graph can vary significantly, and given enough samples should provide a wide variety of shapes with which to exercise a scheduler. However, there is a strong likelihood when low probabilities are used that the dependency graph for each job can have disconnected sections. At the other extreme, this model approximates the Linear Dependencies model (if transitive dependencies are removed). For all these reasons, this method is only really suited to probability values in the middle of the probability range.

The Erdős–Rényi model of generating random graphs has a further shortcoming, because it tends to produce only a narrow spread of in-degree and out-degree over the nodes in the graph. Figure 4.14 shows the in- and out-degree distribution for nodes in a random graph with the same number of nodes and edges. The distribution for the complex industrial pattern (top left) is noticeably more dispersed than that generated by the Erdős–Rényi model, under the same conditions. The Erdős–Rényi model specifically has a very low likelihood of nodes with a large in- or out-degree. In the industrial workloads, a relatively large number of postprocessing tasks consume

Algorithm 4.7 Pseudocode for the Erdős–Rényi pattern

```
n = number of tasks
p = dependency probability
for taskid in 1..n :
    for y in 2..d :
        for possible_task_id in taskid..n :
            if p ≤ random() :
                tasks[taskid].dependencies.add(tasks[possible_task_id])
```

the output of the flow solution, meaning that the task of the flow solution has a large out-degree. The final task that collects up the results to be visualised or returned to the user tend to use the outputs of the postprocessing stages, and tends to have a high in-degree. The flow solution can also have a high in-degree from all the inputs and pre-processing stages.

4.4.2.2 Nodes with Exponential Degree Distribution

Because the Erdős–Rényi model has these shortcomings, this work presents a new method of generating random graphs. Algorithm 4.8 is designed to gives greater connectivity to some nodes, representing a higher level of structure than a purely random graph, and this more closely parallels the structures observed in industry. This method uses the UUnifast algorithm from Bini and Butazzo [17] to generate a logarithmic distribution on the in- and out-degree for the nodes, and then creates random dependency connections that satisfy these distributions. Because UUnifast gives real values for a distribution which are not applicable when generating node degrees, an integer method is given in Algorithm 4.9.

Once the node distributions have been created, it is necessary to form edges of ordered pairs of tasks that ensure the degree distributions are respected. The tasks

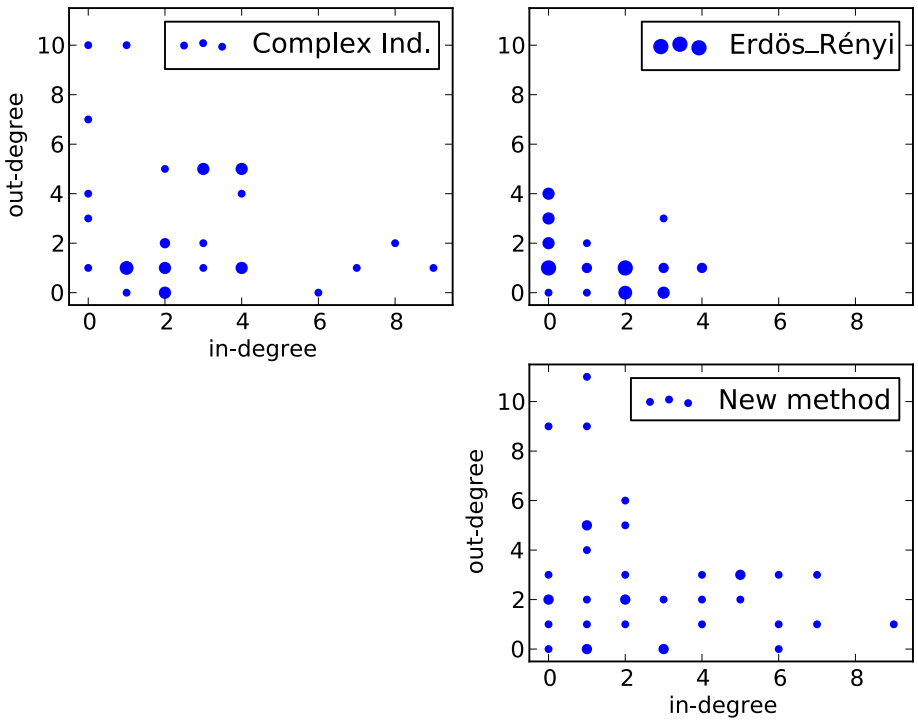


Figure 4.14: In- and out-degree distribution

with the largest in- and out-degrees are assigned dependencies first to try to ensure the distribution of random edges is feasible. However, the random generation can give edge distributions that are impossible to satisfy. Therefore, iteration is used in Algorithm 4.8 to discard impossible solutions and retry creating a new graph with new distributions.

The distribution created using the new method and the UUnifastInteger node degree distribution is shown in the lower right of Figure 4.14. In this case, the new method gives a greater spread of node degrees than the Erdős–Rényi algorithm, similar to that found in the real-world example.

Algorithm 4.8 Python code for random graph generation with high spread of in- and out-degrees

```
def GraphGen(n, e) :
found_outer = False
while not found_outer :
 ins_by_node = sorted(UUnifInt(n - 1, e) + [0])
 outs_by_node = sorted(UUnifInt(n - 1, e) + [0])[:: -1]
 found_inner = False
 itercount = 0
 while (not found_inner) and (itercount <= 40) :
  itercount+ = 1
  found_inner = True
  o = {x : outs_by_node[x] for x in range(n)}
  i = {x : ins_by_node[x] for x in range(n)}
  edges = []
  while len(edges) < e :
   K = max([x[0] for x in i.items() if x[1] > 0])
   P = [x[0] for x in o.items() if x[1] > 0 and x[0] < K]
   D = [(u, K) for u in P if (u, K) not in edges]
   if len(D) == 0 :
    found_inner = False
    break
   else :
    new_edge = random.choice(D)
    o[new_edge[0]]- = 1
    i[new_edge[1]]- = 1
    edges.append(new_edge)
 found_outer = not len(edges) < e :
return edges
```

Algorithm 4.9 UUnifastInteger

```
UUnifastInteger(samples, sum_of_samples):
vectU = []
sumU = sum_of_samples
for i in 0..(samples − 1):
 nextSumU = round(sumU · (random())^(1/(samples−i)))
 vectU.append(sumU) − nextSumU
 sumU = nextSumU
vectU.append(SumU)
return vectU
```

4.5 Summary

In this chapter, the characterisation of the industrial workload is described, highlighting the importance of responsiveness of grid tasks for the productivity of users and the organisation. The daily, weekly and yearly submission patterns are observed. The vast majority of the workload is submitted during working hours, and it is done at a rate faster than the grid can process immediately. The grid therefore spends a fair amount of its time operating at saturation, with work queueing. The queues are drawn down outside of working hours, such as overnight and at weekends. Any suitable scheduling policy for these must be able to deal with extended periods of time where the cluster is overloaded, and prioritise effectively the work that requires immediate responsiveness against that which can wait until a quieter time.

To generate workloads that conform to these arrival patterns, an algorithm is proposed which is parametrisable by a desired average load factor and daily and weekly load cycles. This algorithm works by adjusting inter-arrival times, and so can be applied to existing workloads, keeping other aspects of the workloads the same.

The task execution times are shown to follow a log-uniform distribution, meaning that the largest jobs place the vast majority of the load on the cluster. Small tasks, measured by both execution time and the number of cores required are shown to place little load on the cluster. An algorithm is given to generate such a distribution of task execution times. The volumes of tasks by the numbers of cores used are shown to be log-normally distributed, with most of the load on the cluster being placed by tasks with a mean of 100 cores.

With workloads having such a wide range of execution times, responsiveness may suffer if an inappropriate policy were used. This is because it is highly undesirable to ever have the shortest jobs queueing behind the largest jobs. This range also shows that is important to consider how schedulers prioritise work across the whole range.

Patterns of dependencies between tasks are surveyed, and are found to have a particular node degree distribution where some nodes are very highly connected,

given the size of the graph. It is shown that neither structured nor completely random graphs adequately capture this graph structure. A new method is proposed that generates graphs with given node degree distributions.

Scheduling policies suited to dynamic workloads need to be able to execute multiple workloads with dependencies at the same time. The dependency handling algorithms also need to be sufficiently low-complexity so that they are scalable to the industrial workloads encountered.

The parameters of this workload are likely to be common to engineering design workloads, where the appetite for computational capacity is large, and a hierarchical decomposition of work is followed. Evaluation of scheduling policies using these kinds of workloads is essential in order to be able to provide appropriate scheduling. To make use of synthetic workloads with attributes following these distributions and patterns, appropriate models need to be developed. These models can then be used to simulate a grid system. These simulations can allow a wide variety of schedulers to be evaluated for their suitability for the given workloads. The next chapter will describe and justify the models and simulation framework created by the author for the purposes of such scheduler evaluation.

Chapter 5

Experimental Platform, Metrics and Method

The hypotheses of this thesis given in Chapter 1 require the application of scheduling policies in a context that reflects the industrial scenario. The first stage in investigating this is to gain a deep understanding of the industrial scenario. The socio-technical context and user requirements of the grid system are described in Chapter 2 along with the issues of the current FairShare scheduling system. Having understood the user requirements of the industrial problem, it is also necessary to understand the technical aspects of the problem through a characterisation of the workload. The workload is characterised in Chapter 4, and methods were given to generate synthetic workloads matching the trends observed. Chapter 3 examines the state of the art in grid scheduling and surveys a wide variety of scheduling policies that can be applied to the grid scheduling problem.

In the remainder of this thesis, scheduling policies will be evaluated using simulation. This is because changes to scheduling algorithms are logistically difficult, if not impossible, to study in the field. An entire grid is unlikely to be made available to a researcher simply for experimentation because the cost of operating a production grid is large. Evaluation performed on small-scale systems may also lead to results that are not reflective of the behaviour that would be observed at the large scale of the production grid.

Therefore, this chapter will present a set of models that abstract the pertinent features of the industrial grid. These models will be defined in a way so that they are amenable to implementation in a software simulation. A key requirement of this software simulation is that it reflects the scale seen in the production systems. The models should therefore represent a level of abstraction such that simulation times are tractable, even with grid-scale workloads.

The models of the grid system will be composed of three fundamental sets of models. The *Application Model* will represent the workflows run on the grid and their

requirements. The *Platform Model* will describe the grid hardware infrastructure. The *Scheduling Model* describes how and when scheduling decisions are made. The scheduling model represents a hierarchical list scheduling architecture that is modular so that many possible list scheduling policies can be implemented.

In order to fairly compare scheduling policies in order to investigate the research hypotheses, it is necessary to specify the metrics with which different policies will be evaluated. A wide variety of metrics have been used in the literature for evaluating the performance of scheduling policies. A survey of these metrics will be performed, including a formal definition of each. The metrics will then be evaluated as to their capacity for insight that they can bring to given scheduling situations.

The models that the simulation experiments will be based on require many parameters to be specified. For the experimental evaluations, four 'profiles' or sets of these parameters were used. In order to be able to reproduce the simulations, all the parameters used as part of these simulations are noted in Section 5.7, with a summary of these in Table 5.4.

5.1 Application Model

The application model is a means of formally defining the work that the system must execute. This work follows the nomenclature of Chapin [36]. A single, non-preemptible piece of work to be executed on one or more identical processors/cores concurrently will be known as a *task*, denoted T^i. A set of tasks with dependencies between each other are grouped into a *job*, denoted J^k. Tasks in one job may only depend on other tasks in the same job. A set of jobs will be known as a workload W.

The dependencies between tasks inside a job will take the form of a Directed Acyclic Graph (DAG), following the usual construction for HPC workflows [117, 160]. The structure of the DAGs will reflect those discussed in Chapter 4.

Tasks and jobs have several parameters that will be defined below. This work considers a discrete-time model, with all events taking place on time ticks $\tau \in \mathbb{N}^0$. However, the following parameters and the metrics in Section 5.5 could equally be calculated for a continuous model of time.

- Task actual execution time : $T^i_{\texttt{exec}} \in \mathbb{N}^\star$
- Task cores required : $T^i_{\texttt{cores}} \in \mathbb{N}^\star$
- Task start time: $T^i_{\texttt{start}} \in \mathbb{N}^0$
- Task finish time: $T^i_{\texttt{finish}} = T^i_{\texttt{start}} + T^i_{\texttt{exec}}$
- Task dependents/successors: $T^i_{\texttt{succ}}$

- Task upward rank: $\forall\, T^j \in T^i_{\texttt{succ}} : T^i_{\texttt{R}} = T^i_{\texttt{exec}} + \max(T^j_{\texttt{R}})$
- Job submission time (not necessarily the same as start time): $J^k_{\texttt{submit}} \in \mathbb{N}^0$
- Job start time: $\forall\, T^i \in J^k : J^k_{\texttt{start}} = \min\left(T^i_{\texttt{start}}\right)$
- Job finish time: $\forall\, T^i \in J^k : J^k_{\texttt{finish}} = \max\left(T^i_{\texttt{finish}}\right)$
- Job response time: $J^k_{\texttt{response}} = J^k_{\texttt{finish}} - J^k_{\texttt{submit}}$
- Job total execution time: $\forall\, T^i \in J^k : J^k_{\texttt{exec}} = \sum\left(T^i_{\texttt{exec}} \times T^i_{\texttt{cores}}\right)$
- Job critical path: $\forall\, T^i \in J^k : J^k_{\texttt{CP}} = \max\left(T^i_R\right)$

A job is considered to be *in flight* during the interval $\left[J^k_{\texttt{start}}, J^k_{\texttt{finish}}\right)$, which means between the instant its first task starts until its last task has finished. The critical path (CP) time $J^k_{\texttt{CP}}$ of a job is the longest path through the DAG of the dependencies [100], and defines the minimum job execution time even if the number of processors were unbounded.

5.2 Platform Model

The resources in the grid are grouped into homogeneous clusters, each containing a number of cores. These are connected in a tree structure [126], with a router at each node and a cluster at each leaf. A running task consumes all the cores that it runs on - there is no sharing of cores between executing tasks. Multi-core tasks are only run inside a single cluster, and it is assumed that there is at least one cluster in the grid able to provide sufficient cores to satisfy the most highly parallel of the multi-core tasks.

5.2.1 Heterogeneity

Tasks are assumed to take the same amount of time to run, whatever cluster they run on. This is because of what was observed in industry, where all the clusters of the same architecture use the same processors. This being due to the fact that running processors that were any less than state-of-the-art consume too much electrical power to be cost-effective.

Although the execution speed of the processors is taken to be homogeneous, a coarser-grained model of heterogeneity is included. The model serves to partition the grid into those clusters which a task can and cannot run on. This is done using two sets of attributes. Attributes are stored as key/value pairs and can take one of two types.

Architectural attributes define things such as the CPU instruction set architecture or the presence or otherwise of accelerator hardware. These are defined per cluster and

must match exactly for a task to be able to run. For example, tasks that are compiled for the 'x86' architecture can only run on clusters containing that hardware.

Quantity attributes define what capacity of resources like RAM and scratch space is available to each core in a cluster. In the industrial system, some machines of the same architecture are configured with more RAM or disk than others, to better suit different kinds of workload. Tasks can run on machines whose quantity attributes are greater than or equal to their requirements. For example, tasks with low memory requirements may be able to run on all the x86 clusters, but tasks with high memory requirements may only be able to run on the x86 clusters with large memory capacities.

A set of architectural and quantity attributes are combined into a *Kind*. Tasks and clusters are each assigned a Kind. Each router is assigned a set of kinds representing all the clusters below it in the tree. Each job contains a set of kinds representing the requirements of all the tasks inside it.

5.2.2 Network Model

By far the most common network model is of a completely connected cluster of processors [100], known as a *clique* in graph theory terms. In this model, network transfers between processors do not interfere with each other. Therefore, each transfer can be modelled simply by the time it will take the transfer to complete. Such networks can also be represented by a tree, where each higher link has sufficient bandwidth to serve all those links below it simultaneously. This model, known as a *Fat Tree,* has been taken as the typical network topology for High-Performance Computing since it was first described by Leiserson [108]. The internet is a network that principally follows this architecture [11].

Some schedulers using this model define a standard network bandwidth, and then use the expected data volume between tasks to determine an estimated time of transfer [38]. Others simply expect to be supplied with the transfer times between tasks as weighted edges on the DAG that defines the dependencies within a job [100].

Some schedulers extend the clique network model so that rather than having a fixed speed for the whole network, each link of the clique has its own available bandwidth. Using this scheme, it is possible to model any unloaded arbitrary network simply by setting the clique bandwidths appropriately. Schedulers using this model include those in Carter et. al. and Topcuoglu et. al. [34, 160]. However, this model fails to take into account the fact that on arbitrary networks with multiple simultaneous transfers, some transfers are bound to contend for links in the network. When this occurs, the bandwidth available must be divided between the tasks that are sharing the link, making each transfer slower.

Accurately modelling network interference is difficult, however, because accurate knowledge of the whole network topology and link speed is necessary, and this may not even be known for the entirety of a grid. Scheduling algorithms that include a consideration of interference in their network models are presented by Kermia and Sorel [95] and Dhodi et. al. [51]. A makespan calculation that includes network transfer times is used by Wang et. al. [166] as their fitness function in their search-based algorithm. Their network model has network transfers 'lock' the links that they are using for the duration of the transfer, a concept known as *Maximum Interference*. While this is unduly restrictive, as in reality links can be shared, this is more realistic than most network cost algorithms which just take into account the time or the quantity of data transferred. Interference is much more likely over the geographic links, therefore it is better for schedulers to schedule communicating tasks as close together as possible and to avoid using the geographic links as much as possible.

It can be argued that the internet and clique models of bandwidth distribution are not accurate for the networks that interconnect Grids. Instead, these networks are also tree structured, but have the highest bandwidth connections between those processors that are closest together, the highest bandwidth of all being between processor cores on the same silicon die. Links that connect geographically disparate datacenters are the slowest links involved in the network, because they are the most expensive to construct and have the longest latencies. This kind of structure is known as a *Thin Tree* [126]. Such a network architecture gives another reason that communicating tasks are best placed as close together as possible in order to achieve optimal performance. Using a tree structure can reasonably approximate a real network's structure, because all fully connected networks possess a spanning tree [48].

This work will use a Thin Tree network model, in order to provide an acceptable model to investigate the effects of network delays on scheduling while adding minimal computational overhead. Network delays are only considered between tasks running on different clusters, as these links are likely to represent long-distance geographical links in reality. Within a cluster, network delays are assumed to be small enough to be negligible.

Some tasks may be of the same architecture and could be scheduled on the same cluster, where these delays would not be manifested. However, tasks with different architectures may never be able to run on the same cluster, and hence have an unavoidable network delay. Any unavoidable network delays are taken into account when determining the critical path of a job.

The network speed is calculated by using the fact that the network is tree structured. Therefore, any two clusters will share a common parent node somewhere in the tree. The number of nodes in between a cluster and the common parent is

measured in levels. With a higher number of levels between two nodes, the transfer speed will decrease exponentially. The speed equation takes a parameter p to vary how much slower the higher levels of the network become. Contention between different transfers using the same link is not considered in order to keep simulation time to acceptable levels.

$$N_{\texttt{latency}} = (\texttt{max\_levels\_to\_common}\,(C_1, C_2))^p \tag{5.1}$$

To find the data volume to transfer, the communication to computation ratio parameter (CCR) is used [2], along with the execution time of the task $T^i_{\texttt{exec}}$, as shown in Equation 5.3. Data volume is calculated in this way so that network delays can be varied by adjusting the CCR independent of the workload or platform.

$$CCR = \frac{T^i_{\texttt{data}}}{T^i_{\texttt{exec}}} \tag{5.2}$$

$$T^i_{\texttt{data}} = T^i_{\texttt{exec}} \times CCR \tag{5.3}$$

The time taken to transfer data between two tasks is determined by dividing the data volume required by the speed of the network between them.

$$T^i_{\texttt{net\_delay}} = T^i_{\texttt{data}} \times N_{\texttt{latency}} \tag{5.4}$$

5.3 Hierarchical Scheduling Model

As discussed in Chapter 3, list schedulers are a promising candidate for the online non-pre-emptive scheduling required in the industrial scenario. List scheduling will therefore be the model on which the research in this thesis is based. In a simple case, a single list scheduler for a whole grid would suffice. A single queue would prioritise all incoming work, and a single allocator would send work to the various clusters that make up the grid as and when resources became free.

In a large-scale grid such as that observed in industry, a single scheduler is likely to be a performance and reliability bottleneck. This is because grids have a distributed nature and are hence subject to network costs and limitations in bandwidth between their component clusters. This means that a single scheduler may not be able to have highly detailed and up-to-date information about the state of the whole grid, as simply communicating this information to a central node would swamp the available bandwidth.

This thesis therefore considers a hierarchical scheduling model: a tree of list schedulers. Nodes in the tree are referred to as *routers*, and the leaves of the tree are the clusters that make up the computational resources of a grid. Each cluster itself

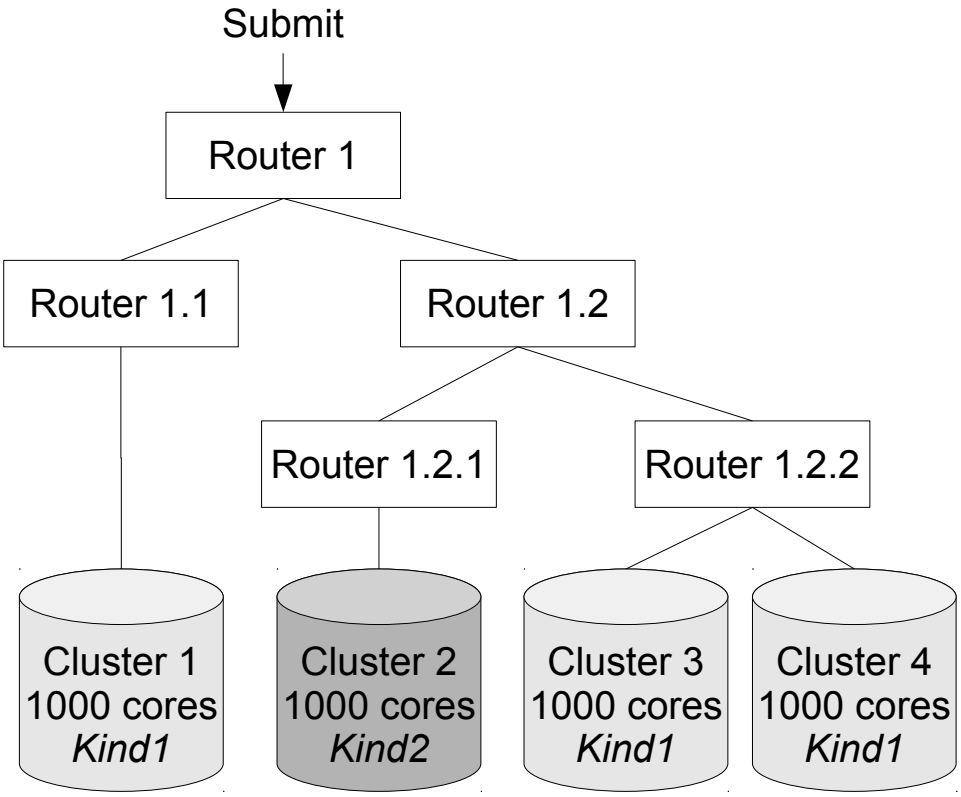


Figure 5.1: Thin Tree Network Diagram

contains a list scheduler. An example of a simple platform following this architecture is shown in Figure 5.1.

As jobs arrive in the system they are first given to the root node of the network tree, which is always a router. For each job, its list of required kinds is compared to that of the routers below the current one. If any routers match all the job's kinds, the job is immediately assigned to one of them (using a load balancing approach if multiple routers match). If no sub-router matches all the kinds, then the job is broken up into groups of tasks that all share the same kind. These groups of tasks are then assigned together, using load balancing, down the tree until they reach a cluster. Groups of tasks from the same job with the same kind are always assigned to the same cluster. This follows the inspiration of the clustering schedulers surveyed in Chapter 3 which seek to avoid unnecessary network transfers.

Tasks will only spend time queuing once they have been allocated to a cluster, as the cascading down the tree following the load balancing policy takes place instantaneously on submission. This is to reflect the way the industrial grid manager LSF works in reality [85]. LSF allocates instantaneously in order to start initial data copies to the clusters early, with the hope that the network transfers will complete before tasks reach the head of the cluster queue.

Load balancing is essential in real grids to ensure all the clusters are used to their maximum capacity. With the aim of the scheduling system to maximise responsiveness, it is desirable to minimise the waiting time across all clusters. Therefore, good load balancing in each router should try to keep the queue length the same across all clusters below it. Therefore, as clusters have different capacities, more work should naturally be sent to the larger clusters in proportion to their capacity.

The load balancing considered in this thesis essentially allocates work to the cluster where it expected to queue for the least amount of time. The algorithm calculates the expected queue length by taking the amount of work (in core-seconds) in each cluster's queue (C_Q) and dividing it by the number of processing resources that cluster contains ($C_{\texttt{cores}}$), as shown in Equation 5.5. The jobs are assigned to the cluster with the smallest expected queue length. These statistics are considered suitably high-level that they could be obtained by routers in a grid without imposing an undue overhead on performance or network bandwidth. Where the load balancing takes place between routers, each router offers the best performing value of any the clusters beneath it.

$$\forall T^i \in C_Q : load\_balancing\_factor = \frac{T^i_{\texttt{exec}} \times T^i_{\texttt{cores}}}{C_{\texttt{cores}}} \tag{5.5}$$

Inside a cluster, tasks queue until they reach the head of the queue, under a chosen ordering policy. Once they are selected to run, allocation is simply done to processors as they become free (an Earliest Start Time allocation), because of the lack of network costs inside clusters. Within a homogeneous cluster, this is equivalent to an Earliest Finish Time allocation [160].

Although routers and clusters both implement list schedulers, the ordering policy of the load balancer and the allocation policy of the cluster are trivial. This architecture effectively gives the result that allocation is done first through the load balancing in the routers, and then ordering is applied when the tasks are on the clusters. This is the reverse of most list scheduling architectures, and yet is suitable for the architecture of grids where perfect knowledge of the whole cluster is impractical to achieve due to the slow network links between clusters.

The key part of this model is the ordering policy applied on each cluster, because it is reasonable to assume that at this level, a great deal more information about the state of the cluster and the work to be performed can be analysed. It is also where the jobs will actually spend their time queuing and hence where prioritisation is applicable. The cluster ordering policy will therefore be the one that will most affect the ability of the grid to achieve good QoS. Measuring the ability of a scheduler to achieve good QoS requires appropriate metrics, which the next sections will describe and evaluate.

5.4 Industrial Metrics

It is natural that grid administrators monitor the performance of their grid. To do this, they use several classes of metrics. The distinctive patterns of submission and execution of work over a week mean that the administrators choose to collect most metrics over the period of a week. For the purposes of calculating metrics, a week starts at 06:00 every Monday. This is when the cluster is perceived to be quietest, because little work is submitted outside of working hours. A more detailed characterisation of these patterns is found in Chapter 4. Work that is submitted in a given week is given the notation W_S^i and the work that completes W_C^i. The set of tasks that spend any time executing during the week is W_E^i and the set of tasks that pend is W_P^i. The start and end of a week-long interval is given by $W^i_{\texttt{start}}$ and $W^i_{\texttt{finish}}$ respectively.

$$W_S^i = \forall T^i \in J^k \wedge \forall J^k \in W : \; W^i_{\texttt{start}} < J^k_{\texttt{submit}} \leq W^i_{\texttt{finish}} \tag{5.6}$$

$$W_C^i = \forall T^i \in J^k \wedge \forall J^k \in W : \; W^i_{\texttt{start}} < J^k_{\texttt{finish}} \leq W^i_{\texttt{finish}} \tag{5.7}$$

$$W_E^i = \forall T^i : \; T^i_{\texttt{start}} < W^i_{\texttt{finish}} \wedge T^i_{\texttt{finish}} > W^i_{\texttt{start}} \tag{5.8}$$

$$W_P^i = \forall T^i : \; T^i_{\texttt{submit}} < W^i_{\texttt{finish}} \wedge T^i_{\texttt{start}} > W^i_{\texttt{start}} \tag{5.9}$$

Originally, these weekly metrics had been compiled and visualised manually, using spreadsheet software. This was a time-consuming process, however, so much so that reports were only produced approximately every quarter. This was often too late to troubleshoot problems on the grid, and so they were only really used to analyse trends. As part of the industrial placement, the author further developed the analysis software so that the weekly metrics could be compiled and visualised automatically. A screen shot of the 'dashboard' interface created is shown in Figure 5.2.

The primary concern of grid administrators is that the grid is being well-utilised. The number of jobs submitted per week is monitored, to identify trends in the rising demand for computing power. These trends help inform forecasts of when new capacity will need to be added.

$$W^i_{\texttt{submitted}} = \left| W_S^i \right| \tag{5.10}$$

The most important metric for utilisation is the number of CPU-days consumed each week (Equation 5.11). This can be compared to the maximum available to get

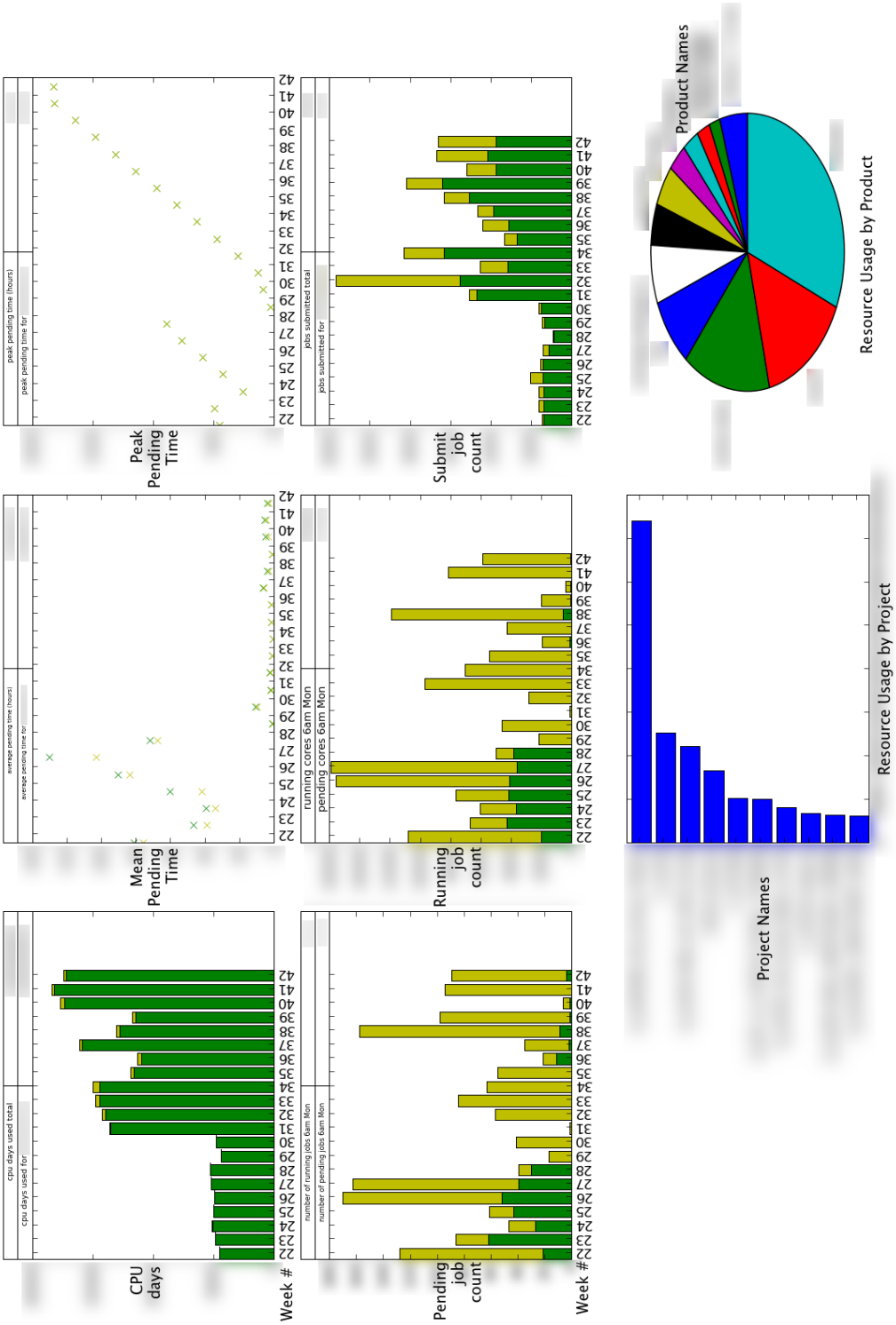


Figure 5.2: Dashboard of Industrial Metrics Screen Shot

a percentage figure of utilisation (Equation 5.12). To calculate the number of CPU days used the $W^i_{\texttt{cpu\_time}}$ must be multiplied by a value appropriate to the time-base resolution used for defining task execution times.

$$W^i_{\texttt{cpu\_time}} = \sum_{W^i_E} \left(\min \left(T^i_{\texttt{finish}}, W^i_{\texttt{finish}} \right) - \max \left(T^i_{\texttt{start}}, W^i_{\texttt{start}} \right) \right) \times T^i_{\texttt{cores}} \tag{5.11}$$

$$C_{\texttt{utilisation}} = \frac{W^i_{\texttt{cpu\_time}}}{C_{\texttt{cores}} \cdot (\texttt{seconds/minutes/hours})\,\texttt{in\_week}} \tag{5.12}$$

Filters are applied to break down these figures by project and group. A project represents the design of a particular airframe, whereas a group represents the aerodynamics or loads teams. Utilisation broken down by project and by group is used to determine whether the fair share tree is correctly configured for the current workload. Where some projects are deemed more urgent, the users and groups working on those projects may have their fair share raised.

However, the fair share assigned to a user or a group only indirectly affects response times. If the cause of low responsiveness is that a user or a group is running above their fair share, then it is possible to reduce response times by increasing the share. However, as detailed in Chapter 2, poor response times can occur even when users and groups are operating well within their share.

The administrators measure responsiveness through three metrics. The most commonly used one is the average pending time of all tasks (Equation 5.13), which is the time between the task being submitted and it starting execution. The problem with this measure is it poorly captures user requirements. Users have a wide variation in the design cycle lengths they work to. For some users a pending time of an hour would be unacceptable, whereas for others several days pending time may not even be noticeable. As there are so many tasks submitted, using the mean can mask very poor response times.

$$\forall T^i \in W^i_P : \; W^i_{\texttt{mean\_pending}} = \frac{\sum_{W^i_P} T^i_{\texttt{start}} - T^i_{\texttt{submit}}}{\left| W^i_P \right|} \tag{5.13}$$

$$\forall T^i \in W^i_P : \; W^i_{\texttt{max\_pending}} = \max \left(T^i_{\texttt{start}} - T^i_{\texttt{submit}} \right) \tag{5.14}$$

To give some more insight into starving tasks, the worst case pending time is also measured (Equation 5.14). The difficulty is that the single worst-case pending time often corresponds to a task that has been submitted with erroneous parameters, and so will never be able to run. These have to be cancelled manually by the administrators, but it often takes a large amount of pending time before they are seen to be a problem, especially given the long cycle times of some groups.

The final measure of response times used is a count of the number of tasks still pending and running at the start of the week (Equations 5.15 and 5.16). If there is still work waiting then the administrators take this to mean that the cluster is overloaded - because it has not caught up on the previous week's work. However, this metric fails to take account of the varying cycle times of designers, many of whom need their tasks to be returned far more quickly than by the next week.

$$\forall T^i : \; W^i_{\texttt{Mon\_pending}} = \left| T^i_{\texttt{submit}} < W^i_{\texttt{finish}} < T^i_{\texttt{start}} \right| \tag{5.15}$$

$$\forall T^i : \; W^i_{\texttt{Mon\_running}} = \left| T^i_{\texttt{start}} < W^i_{\texttt{finish}} < T^i_{\texttt{finish}} \right| \tag{5.16}$$

Fairness is measured by comparing the currently running utilisation of groups compared to their fair share. As discussed in Section 2.6.1, this reflects the interests of the administrators who are concerned about utilisation, rather then the users, who care about responsiveness.

5.5 Metrics

The previous section introduced the metrics used by the industrial partner. Different metrics are more or less relevant to different stakeholders. The metrics relevant to the system administrators correspond to those related to utilisation, and the metrics that represent the users' point of view correspond to the responsiveness and fairness metrics. This section presents a survey of the literature of which other metrics have been applied to the evaluation of scheduling policies. A further category of metrics not seen in the industrial scenario was noted as being commonly used in the literature, and these were the relative metrics. Relative metrics compare schedulers by counting the number of 'best' schedules (by another metric) over a number of scenarios in a problem space.

All these metrics are considered here specifically within the context of the industrial scenario outlined above, which is the dynamic or online scheduling of jobs onto a fixed, distributed grid platform. However, the metrics are not limited to being used in such circumstances, and most should be able to provide insight into both static and dynamic scheduling approaches. A summary of the applicability of each metric is presented in Table 5.1.

Metric		Utilisation	Responsiveness	Fairness
Workload Makespan		•		
Flow		•		
Average Utilisation		•		
Peak In-Flight		•		
Cumulative Completion		•	•	
Average or Worst Case	Speedup		•	
	Stretch		•	
	Schedule Length Ratio		•	
Standard Deviation	Speedup			•
	Stretch			•
	Schedule Length Ratio			•
Gini Coefficient				•

Table 5.1: Insight given by selected metrics

5.5.1 Utilisation Metrics

Utilisation metrics measure how much of a platform's maximum potential is actually being used. Achieving a high throughput of work is contingent on achieving good utilisation. Wherever possible, it is desirable to avoid having idle resources if there is ever work queuing.

Workload Makespan

The classic metric used to compare schedulers is the workload makespan (Equation 5.17), which is widely referenced in the literature [3, 22, 71, 97, 100, 120]. This is defined by the time at which all the work in the workload has completed.

$$\forall J^k \in W : W_{\texttt{makespan}} = \max\left(J^k_{\texttt{finish}}\right) \tag{5.17}$$

While some papers use only this metric for comparing schedulers [34, 38], it is insufficient for measuring the responsiveness or fairness in a schedule. This is because, in the simulation of a dynamic system, the workload makespan may be mostly determined by the last few jobs in the workload to arrive. What it can help to measure, on the other hand, is utilisation, as a component of the Flow or Average Utilisation metrics. Because it requires the workload to complete execution, the workload makespan metric only really applies to the evaluation of static scheduling problems.

Flow

A measure of throughput is simply to count the number of tasks or jobs completed over the workload makespan. This is known in the literature as flow [15] (Equation 5.18).

$$Flow = \frac{|W|}{W_{\texttt{makespan}}} \tag{5.18}$$

Flow does not attempt to account for the differing sizes of work, so a platform may be able to achieve wildly different values of flow depending on the makeup of the workload. This renders it less useful for comparing schedulers across different workloads.

In a dynamic system, it may not be possible to measure the makespan of a workload, because work is continually arriving. In this case, flow can be defined as the number of jobs to finish in a given time interval $(\tau_{\texttt{start}}, \tau_{\texttt{finish}}]$ (Equation 5.19).

$$\forall J^k \in W \wedge \tau_{\texttt{start}} < J^k_{\texttt{finish}} \leq \tau_{\texttt{finish}} : Dynamic\ Flow = \frac{|J^k|}{\tau_{\texttt{finish}} - \tau_{\texttt{start}}} \tag{5.19}$$

Average Utilisation

A further metric can be derived from the workload makespan, known as average utilisation [87] or efficiency [160]. This is defined as the proportion of the possible execution time determined by the workload makespan that was actually consumed. The number of processing units in the grid is denoted $G_{\texttt{cores}}$.

$$\forall J^k \in W : Average\ Utilisation = \frac{\sum J^k_{\texttt{exec}}}{W_{\texttt{makespan}} \times G_{\texttt{cores}}} \tag{5.20}$$

This metric can also be extended to dynamic systems by calculating the CPU time used between two points in time, as is given in Equations 5.11 and 5.12. Interval utilisation is useful because weekly or daily average utilisation values can be monitored.

Peak In-Flight Count

As mentioned in Section 5.1, a job can be considered in-flight between when the first task of that job starts execution and the last task of that job finishes. Here a novel metric is proposed (Equation 5.21), known as peak in-flight count, that gives the maximum number of jobs in flight at any given time.

$$\forall t \in [0, W_{\texttt{makespan}}], \forall J^k \in W \wedge J^k_{\texttt{start}} \leq t < J^k_{\texttt{finish}} : \textit{Peak In Flight} = \max\left(\left|J^k\right|\right) \tag{5.21}$$

This can be used to determine how much the scheduler has interleaved the jobs in the workload. High peaks may indicate scheduling problems where some jobs are starving for resources. The peak in-flight count can also reveal the effect of network delays. An abnormally high peak in-flight count might indicate that the scheduler is starting work on new jobs because all the current in-flight jobs are blocked waiting for network transfers to complete. This may point to using an alternative scheduler that is better suited to avoiding network bottlenecks.

5.5.2 Responsiveness Metrics

Responsiveness metrics compare how a scheduler is able to keep job latency low. There will always be a minimum time that a job will take to execute, and this is determined by its critical path. However, the time spent queueing or on network transfers will impact the responsiveness of a job. Responsiveness metrics can be a tool for measuring how well the scheduler is able to cope under periods of heavy load. The metrics of Speedup, Stretch and SLR are defined for each job in a workload. Therefore, the average value of these metrics for all the jobs in a workload can be used to provide a single value to compare scheduler performance. It can also be useful to compare the worst-case performance of the responsiveness metrics, because it is the users whose jobs are experiencing worst-case performance that will be the ones to complain, especially if the worst-case is significantly different to the average.

Cumulative Completion

A metric that rewards early completion of work, and hence good average responsiveness, is proposed by Braun et. al. [22]. Whereas the utilisation metrics only derive value from the time the workload was completed, this gives some insight into the way this was achieved. This metric calculates the sum of completed *job* execution times at each time tick in the execution (Equation 5.22). Because it is assumed that only a completed job is useful to a user, it can only count the completed tasks' execution times once the whole job is finished.

$$\forall J^k \in W : \textit{Cumulative Completion} = \sum J^k_{\texttt{exec}} \times \left(1 + W_{\texttt{makespan}} - J^k_{\texttt{finish}}\right) \tag{5.22}$$

The cumulative completion metric values work being completed early on in the schedule, by cumulating the values of completed jobs at each subsequent tick. If the workload makespans between schedules are different, the values of cumulative completion are not directly comparable. Therefore, where cumulative completion values need to be compared, the cumulative completion value should be calculated with the workload makespan value of the longest schedule.

This metric also gives partial insight into utilisation, because schedulers that achieve higher utilisation and higher throughput will cause more jobs to finish sooner, and hence raise the Cumulative Completion value. A shortcoming of this metric is that it is most suited to static schedules, because the finishing of the workloads is all relative to their makespan. However, it can be extended to the dynamic case by only sampling jobs that arrived in a given duration.

Speedup

A common metric to measure responsiveness is known as Speedup [160] (Equation 5.23). It is defined as how much faster each job was able to run compared to if it had been run on a single processor.

$$J^k_{\texttt{speedup}} = \frac{J^k_{\texttt{exec}}}{J^k_{\texttt{response}}} \tag{5.23}$$

This can be useful to see how much parallelism the scheduler has been able to extract from the job. However, in most HPC and grid systems, jobs are usually designed to be highly parallel in order to take the fullest advantage of the grid platform and because it would take vastly too long on a single processor. Therefore, while a speedup above 1 may intuitively sound desirable, speedup values may only be considered acceptable at a much larger value. Furthermore, it has no notion of comparing the actual speedup to the maximum possible speedup, when dependencies are present, because it does not take into account the critical path.

Stretch

Stretch is the reciprocal value to speedup, as described by Bender et. al. [15] (Equation 5.24).

$$J^k_{\texttt{stretch}} = \frac{J^k_{\texttt{response}}}{J^k_{\texttt{exec}}} \tag{5.24}$$

The stretch metric is useful because it removes the effect that jobs of different sizes have on their execution times. It shows the 'retardation' of jobs due to the scheduling and load of the system. However, it may be somewhat misleading because the minimum execution time of a job is not necessarily correlated to its total

execution time. This is because the parallelism available in two jobs with the same total execution time can be different due to differences in the core count of tasks or the structure of dependencies (see an examination of this issue in Section 5.6.3).

Schedule Length Ratio

To counteract the problem of the stretch metric not taking into account the minimum execution time of a job, Topcuoglu et. al. [160] use the concept of Schedule Length Ratio (SLR) (Equation 5.25). It is also known as *slowdown* in some papers [98] and by other names [140], although other papers define slowdown somewhat differently [175]. SLR is a similar metric to stretch, but is defined relative to the critical path rather than the total execution time. This is because the shortest execution time of a job on a highly parallel platform is determined by the length of its critical path.

$$J^k_{\texttt{SLR}} = \frac{J^k_{\texttt{response}}}{J^k_{\texttt{CP}}} \tag{5.25}$$

Of the three responsiveness metrics, SLR is the most representative of the performance of the scheduler alone. This is because it is simply a comparison between the actual and ideal response times [98]. SLR is independent of the total execution time or the parallelism available in the job.

The ideal value for SLR is represented as 1, where the actual response time is equal to the ideal response time. This ideal value may be impossible to achieve in a finite grid. Furthermore, network delays that are not present on the critical path, but are still introduced by the scheduling decisions made, may contribute to raising the SLR value above 1.

To obtain a single value for the performance of the scheduler over a whole workload, the mean or worst-case SLR values for the whole workload can be used. These metrics are particularly useful in the case of system overload, where some SLR values must increase over a value of 1.

5.5.3 Fairness Metrics

It is possible to achieve a kind of perfect fairness in a naïve way by only running a single job at a time. However, this will almost certainly mean that utilisation and throughput over the whole grid are unacceptably low. This means that there can be a tradeoff in a non-pre-emptive system between fairness and utilisation.

There may be an underlying assumption that by raising utilisation, responsiveness is maximised, and hence fairness will be near optimal as well. This assumption seems implicit in the many grid scheduling policies that seek to optimise for the smallest workload makespan. This assumption may hold when the task/job execution times are tightly clustered or follow a normal distribution. However, it

breaks down when a distribution with a very wide spread of execution times is encountered, such as that described in Chapter 4. In such a situation, even if there is high utilisation, this may be where all the largest jobs are running, and the smallest jobs experience very poor responsiveness [98].

The importance of measuring fairness can be illustrated with the following example. A First In First Out scheduler might introduce a relatively constant delay to all jobs that come through the system. However, this would penalise the SLR of jobs with a short critical path far more than that for jobs with a long critical path [147, 168]. This is likely to be perceived by users as an unfair situation. Furthermore, this may be particularly undesirable because short jobs may well also be the ones for which responsiveness is the most important, as was observed in the industrial case study. Wierman [168] identifies this aspect of fairness and named it *proportional fairness*. Wierman also considers the notion of *temporal fairness*, which is the preservation of the order that work arrived in, although this is less important to the industrial context.

For all these reasons, metrics are needed to quantify the level of fairness, to ensure that the tradeoff between high utilisation and responsiveness is managed appropriately. The average values of the Speedup, Stretch and SLR metrics can be used to gauge the responsiveness a scheduler is able to achieve with a given workload. By comparing the spread of values relative to the means, fairness metrics can be developed.

A fairness metric is considered by Klusáček [98] that is based on the sum of squared deviations from the mean slowdown. While this is a useful starting point, this metric is not normalised. This value is also a stepping stone to the calculation of the standard deviation, a more usual measure of spread within populations. A small value for the standard deviation of the responsiveness metrics is likely to be considered fair if the desire is to treat each job equally.

Theoretical backgrounds to fairness metrics are given by Lan [102] and Wierman [168]. Wierman, however, only presents a metric that combines proportional and temporal fairness, which is not applicable to the industrial context. Lan et. al. [102] consider a number of generalised fairness metrics, although when it is desirable to have the fairness values on a normalised scale, these simply represent the Gini Coefficient.

Gini Coefficient The Gini Coefficient (GC) is a measure of the inequality of resources allocated to a given population [112]. It has been widely applied to the distribution of wealth in societies. In the context of this thesis, however, it is the allocation of responsiveness to jobs by the scheduler. The GC takes a value between 0 and 1, where 0 indicates a completely fair distribution where every member has an

equal share, and 1 is completely unfair, where a single member has all of the resources.

$$GC(W) = \frac{2 \times \sum_{k=1}^{|W|} \left(k \times J_{\text{SLR}}^{k}\right)}{|W| \times \sum_{k=1}^{|W|} \left(J_{\text{SLR}}^{k}\right)} - \frac{|W|+1}{|W|} \tag{5.26}$$

5.5.4 Relative Metrics

Schedulers can be compared by counting the number of 'best' schedules each scheduler achieves over a set of problems. To decide which schedule is best, an existing metric such as the workload makespan is used [2, 120]. The 'best' scheduler is then considered to be the one that had the highest number of wins over the problem space [19].

These approaches are known as relative metrics. Relative metrics can often be useful for real-world scheduling problems, because finding the optimal schedule is computationally intractable. A simple count may not be able to show how much better the best scheduler is. Where a numerical value for relative performance is desired instead of a count, it is common to compare the metric(s) for the considered scheduler against some accepted 'baseline' scheduler [120]. This allows analysis showing that the alternative scheduler is X% better. While relative metrics may help in the end decision of which the best scheduler is, they do not provide any greater insight into the schedules produced than the underlying metrics that they are based on. Therefore, they will not be evaluated here.

5.6 Metric Evaluation

Metrics are used to provide insight into schedules. The different classes of metrics defined above provide different kinds of insight. This section will apply the metrics to three example schedules that contain known scheduling issues. The ability of the metrics to identify the issues involved will be evaluated. The examples show the importance of being able to measure issues of utilisation, responsiveness and fairness, respectively. The examples contained in this section are deliberately small so that they can be completely described briefly, yet still demonstrate the presence of the scheduling issues. For the purposes of simplicity, all the jobs are given arrival times of $\tau = 0$. Nevertheless, they are designed to be viewed as dynamic scheduling problems, as the issues of responsiveness and fairness are less relevant to static scheduling problems. The discussion in this section will attempt to identify the metrics that provide the best insight into these scheduling issues.

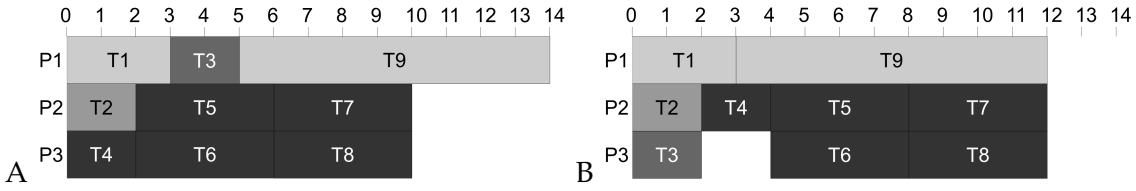
5.6.1 Low Utilisation Issue

If the packing of tasks on to processors is not sufficiently dense, then low utilisation of the processors will result. Graham's classic paper concerning scheduling anomalies [71] contains an example of contrasting schedules. The workload given by Graham [71] is presented in Table 5.3b and is intended to run on three processors. Two schedules (A and B) of the same workload are given in Figure 5.3a. Schedule A is Graham's workload scheduled with an anomaly that increases makespan, whereas schedule B is a schedule without the anomaly (see 5.3a). Metrics for these two different schedules are presented in Tables 5.3d and 5.3c.

The significant feature of the workload is that the critical path of J^1 is long enough that it defines the minimum workload makespan (Table 5.3d). In schedule A, the whole workload makespan is extended because T^3 delays the execution of T^2. The flow and average utilisation metrics depend on the workload makespan. Because the workload is the same but its makespan in schedule A is longer than in schedule B, then the flow and average utilisation metrics are lower for schedule B. The peak in-flight count metric remains the same, although Schedule B only has a single duration of the peak between τ_0 and τ_2, whereas schedule A has two periods of time at the peak value, $\tau_0 - \tau_2$ and $\tau_3 - \tau_5$. The utilisation metrics are useful here, because they show that schedule B contains less wasted capacity in the schedule, and hence makes more efficient use of the resources.

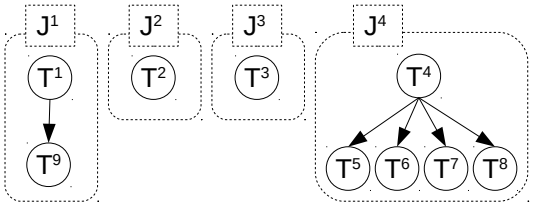
All the responsiveness metrics except cumulative completion show an improvement from schedule A to schedule B (Table 5.3d), because three jobs finish earlier and only one job finishes later. The cumulative completion metric rewards the early finish of the larger J^4 in schedule A. This is because the cumulative completion metric rewards jobs that finish earlier, and the movement of empty scheduling space to the end of a schedule. If a new job arrived only after this space had passed, the capacity represented by the empty space would have been wasted. This stands in contrast to the average utilisation metric, which would suggest that the lower average utilisation is better, but does not take into account where in the schedule this low utilisation phase appears.

A high value for cumulative completion may be valuable, but it does not indicate how fairly the jobs in the workload are being treated. The fairness metrics (Table 5.3d) also show that schedule B is an improvement over schedule A. This is because J^1 and J^3 complete sooner, and hence closer to their critical path time. The finish time of J^4 is extended, but as this is one of the larger jobs, the increase is less when taken as proportional to its execution and critical path time. This means that the variation as a proportion of the job responsiveness metrics for each job is lower (Table 5.3c), giving a lower standard deviation of these metrics which defines an increase in fairness.



(a) Gantt Chart

Job	Task	Dependencies	$T_{\texttt{exec}}$
J^1	T^1	-	3
J^1	T^9	T^1	9
J^2	T^2	-	2
J^3	T^3	-	2
J^4	T^4	-	2
J^4	T^5	T^4	4
J^4	T^6	T^4	4
J^4	T^7	T^4	4
J^4	T^8	T^4	4



(b) Graham's Workload [71]

Metric	J^1		J^2		J^3		J^4	
	A	B	A	B	A	B	A	B
Stretch	1.17	1	1	1	2.5	1	0.56	0.67
SLR	1.17	1	1	1	2.5	1	1.67	2.0
Speedup	0.86	1	1	1	0.4	1	1.8	1.5

(c) Job Metrics

Metric	A	B	B/A
			Utilisation
Workload Makespan	14	**12**	0.86
Average Utilisation (%)	81.0	**94.4**	1.17
Flow (jobs/tick)	0.29	**0.33**	1.17
Peak in-flight	3	3	1.00
			Responsiveness
Mean Stretch	1.31	**0.91**	0.70
Worst-case Stretch	2.50	**1**	0.40
Mean SLR	1.59	**1.25**	0.78
Worst-case SLR	2.50	**2.00**	0.80
Mean Speedup	1.02	**1.13**	1.10
Worst-case Speedup	0.40	**1**	2.50
Cumulative Completion	**148**	142	0.96
			Fairness
Std. Dev. Stretch	0.84	**0.17**	0.20
Std. Dev. SLR	0.67	**0.5**	0.74
Std. Dev. Speedup	0.58	**0.25**	0.42
Gini Coefficient SLR	0.197	**0.150**	0.76

(d) Workload Metrics

Figure 5.3: Low Utilisation Issue Example

From this example, it can be seen that utilisation metrics are important, because they can reveal inefficiency in how the platform is being used. The responsiveness metrics for each job show how smaller jobs are proportionally affected more than larger ones when they are subjected to delay. This is further revealed in the fairness metrics, which show an improvement in fairness (lower variation) for schedule B compared to schedule A.

5.6.2 Multiple Waits Issue

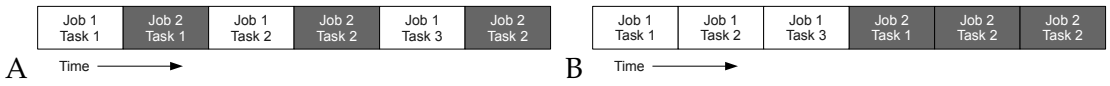
The multiple waits problem is exhibited when there is too great an interleaving of jobs in a system, leading to low responsiveness even though utilisation is high. Table 5.4b gives a workload with dependencies that will be run on a single processor. Figure 5.4a shows two possible schedules of this workload, and as the two jobs arrive at the same time, the one with the lower index begins first. The dependencies of two jobs are shown in Table 5.4b.

A trivial example of the multiple waits problem is shown in Figure 5.4a. Schedule A shows a high interleaving of the two jobs, as could have been scheduled by a list scheduler using FIFO ordering over *tasks* (e.g. the scheduler given in Section 6.2.2). Schedule B, on the other hand, shows the two jobs executed in sequence. This could have been created using a list scheduler using a FIFO ordering over jobs instead of tasks (e.g. scheduler from Section 6.2.3). The most pertinent feature of this example is that J^1 completes execution significantly earlier under schedule B than under schedule A, while J^2 completes execution at the same time (Figure 5.4a).

The utilisation metrics that depend on the makespan are the same, because the workload makespan is the same (Table 5.4d). Only the utilisation metric of peak in-flight count shows a difference between these two schedules. The peak of 2 in schedule B suggests that there is greater than desirable interleaving of work, because the peak in-flight count is greater than the processor count.

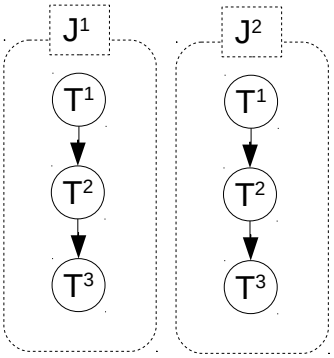
The earlier completion of J^1 in schedule A means that a higher cumulative completion value is achieved (Table 5.4d). The average stretch, speedup and SLR metrics also favour schedule A, also because J^1 finished earlier. It is important to note that in schedule A, the stretch metric has a value of 1, whereas the SLR metric has a value of 1.5. This is because stretch is defined relative to execution on a single processor, which matches this situation. Having a stretch value of 1 may seem to indicate that there is no further improvement that can be made. However, the dependency structure in J^1 shows that there is parallelism that has not been exploited in this example. The SLR metric reveals the potential for a lower response time if there were more processors available.

The fairness metrics of the standard deviation of stretch and speedup indicate instead that Schedule A is to be preferred (Table 5.4d), because the two jobs finish



(a) Gantt Chart

Job	Task	Dependencies	$T_{\texttt{exec}}$
J^1	T^1	-	1
J^1	T^2	T^1	1
J^1	T^3	T^1	1
J^2	T^1	-	1
J^2	T^2	T^1	1
J^2	T^3	T^2	1



(b) Workload

Metric	A (J^1)	A (J^2)	B (J^1)	B (J^2)
Stretch	1.66	2.00	**1.00**	2.00
SLR	2.5	2.0	**1.5**	2.0
Speedup	0.6	0.5	**1**	0.5

(c) Job Metrics

Metric	A	B	B/A
			Utilisation
Workload Makespan	6	6	1
Average Utilisation (%)	100	100	1
Flow (jobs/tick)	0.30	0.30	1
Peak in-flight	2	**1**	0.50
			Responsiveness
Mean Stretch	1.83	**1.50**	0.82
Worst-Case Stretch	2.00	2.00	1
Mean SLR	2.25	**1.75**	0.78
Worst-case SLR	2.50	**2.00**	1
Mean Speedup	0.55	**0.75**	1.36
Worst-case Speedup	0.50	0.50	1
Cumulative Completion	9	**15**	1.67
			Fairness
Std. Dev. Stretch	**0.24**	0.70	2.94
Std. Dev. SLR	0.35	0.35	1
Std. Dev. Speedup	**0.07**	0.35	5
Gini Coefficient SLR	**0.056**	0.071	1.29

(d) Workload Metrics

Figure 5.4: Multiple Waits Issue Example

closer in time. While this seems more fair, this should be considered in light of the decrease in responsiveness. Interestingly, the standard deviation of SLR does not change, indicating that when dependencies are taken into account, the schedules are equally fair.

This example demonstrates that responsiveness metrics of mean SLR, stretch and speedup are important, because they reveal how quickly each job is getting through the system. The cumulative completion metric also reveals the benefit of having some jobs finish earlier, even if the workload makespan is the same. The peak in-flight count is also shown to be useful, because it reveals where there is excessive interleaving of jobs. The responsiveness metrics also show that while a schedule may seem fairer, it may also be less responsive, and both sets of metrics should be considered if a tradeoff is to be made between them.

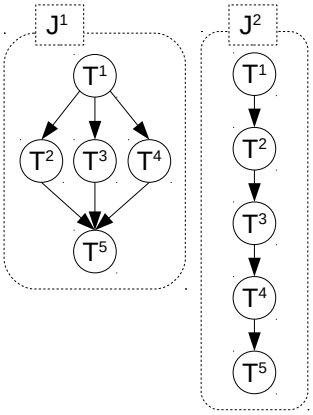
5.6.3 Advantages of SLR over Stretch/Speedup

A further example schedule is given in Figure 5.5 that is intended to highlight the advantage gained by using the SLR metric over the stretch or speedup metrics on tasksets containing dependencies. The schedule shown in Figure 5.5b contains two jobs, with identical numbers of tasks and identical execution times (of tasks and of the whole job), as defined in Table 5.5a. The only thing that differs between the jobs is their dependency structure, and hence the length of their critical path. The critical path length of J^1 is 3, whereas for J^2 the critical path length is 5. When these two jobs are scheduled onto a single processor each, the SLR metric reveals that this is optimal for J^2, because of its dependency structure, and yet it is suboptimal for J^1, because J^1 has further opportunities for parallelism (see Table 5.5c). Nevertheless, the stretch and speedup metrics cannot distinguish between the scheduler's performance, because both jobs have the same total execution time.

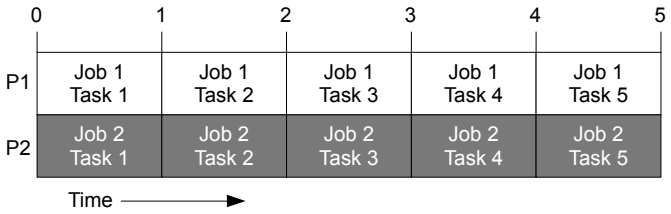
5.6.4 Metric Evaluation Summary

This section has shown, using the examples of three issues, that the measurement of utilisation, responsiveness and fairness is important. Scheduling issues can occur in each of these categories. Where dependencies are concerned, it has been shown that taking the critical path of jobs into account (as the SLR metric does) is essential, as there is otherwise a loss of insight. Having defined and evaluated a number of metrics, these will be applied to the evaluation of a number of scheduling policies running over synthetic and industrial workloads.

Job	Task	Dependencies	T_{exec}
J^1	T^1	-	1
J^1	T^2	T^1	1
J^1	T^3	T^1	1
J^1	T^4	T^1	1
J^1	T^5	T^2, T^3, T^4	1
J^2	T^1	-	1
J^2	T^2	T^1	1
J^2	T^3	T^2	1
J^2	T^4	T^3	1
J^2	T^5	T^4	1



(a) Workload



(b) Gantt Chart

Metric	Job 1	Job 2
Stretch	1	1
SLR	**1.66**	1
Speedup	1	1

(c) Responsiveness Metrics

Figure 5.5: SLR Advantages Example

5.7 Experimental Simulation Method

In order to fairly evaluate scheduling algorithms in order to satisfy the hypotheses from Chapter 1, they need to be compared in the same context. The models presented in Sections 5.1 to 5.3 in this chapter can be used to create a wide range of contexts.

Four experimental contexts or *profiles* were used to support the investigation of the hypotheses. These profiles were designed to give the best balance of execution speed and experimental coverage. Each profile corresponds to a particular experimental investigation. Many parameters are necessary to generate these profiles, and this section will describe and justify the numbers selected for these profiles. A summary of the profiles and their parameters are described in Table 5.4.

Profiles 1-3 are used to investigate Hypothesis 1 from Chapter 1, which concerns the responsiveness and fairness of scheduling policies. Profile 1 is used to investigate these attributes across the spectrum of load, whereas profile 2 is used to investigate the attributes across the spectra of network delays and inaccurate execution times. Profile 3 simulates the industrial platform and executes a workload matching that observed in the industrial log files analysed in Chapter 4. Profile 4 is used to investigate Hypothesis 2 from Chapter 1, which concerns the value returned to users by scheduling policies. For each set of parameters in the profile, 30 workloads were generated and used for evaluation.

5.7.1 Synthetic Workload

5.7.1.1 Workload Volume

The most basic decision to be made regarding the workloads is to decide on the number of tasks and jobs to be run. In the workload characterisation in Chapter 4.2, it was observed that there were distinctive arrival patterns over days and weeks due to the working hours of users. In order to fairly simulate these patterns, simulations must run over a sufficiently long time period so that many iterations of these week-long patterns are present. This is to avoid only a part of the pattern being present in the simulation, and therefore have simulation results that would not represent a true long-running system.

For these reasons, a simulation duration of at least one year was selected as being a sufficiently long time to include many week patterns and to minimise simulation end effects. The simulator developed was able to handle the real workload of the cluster on a platform that represented the scale of real grid, as represented in Profile 3. The actual number of tasks and the volume of the industrial workload is not disclosed for reasons of commercial sensitivity.

Although it is possible to simulate the single instance of the industrial workload running alone, many more simulations are necessary to evaluate larger parameter

spaces. Resource constraints of CPU time and RAM meant that running simulations at full industrial scale was intractable when a wide space of parameter values was considered. Therefore, the number of jobs and the size of the grid (number of cores) was scaled down in proportion so that the run-time of the simulation would still be a year or more at full (100%) load. Simulations where the load factor was lower would run the same workload over a longer duration. This is due to the method of adjusting load by changing the inter-arrival times of jobs described in Section 4.2.

Profiles 1 and 2 had workloads containing 10,000 tasks, which corresponded to 1,000-10,000 jobs depending on the dependency patterns used. Profile 4 had 12,500 jobs which gave 100,000 - 125,000 tasks for each workload. Profiles 1, 2 and 4 all had a total workload volume of 10^{10} core-minutes, which is roughly equivalent to two years' worth of work for a 10,000 core grid. This ensures more than a year of submission patterns will be observed even in the overload scenarios.

5.7.1.2 Execution Time Distributions

It is important to reflect the distribution of task execution times observed in the industrial scenario in the synthetic workloads generated. For profiles 1, 2, and 4, execution times were sampled from the distribution described in Figure 4.7 according to Algorithm 4.3 given in Chapter 4. The UUnifast-Discard approach from Davis and Burns [50] was used to ensure that the distribution of execution times for tasks and jobs was as desired as well as giving a specified total workload volume.

5.7.1.3 Arrival Patterns

Profile 1 uses Algorithm 4.1 that gives constant arrival rates of workload. While this is useful to investigate the impact of changing load, it poorly reflects the peaks and troughs of industrial submission cycles. Therefore, Profiles 2 and 4 use Algorithm 4.2 to introduce daily and weekly cycles of load to workloads.

5.7.1.4 DAG Shapes

Evaluating schedulers with a variety of dependency graph structures is essential. Profiles 1 and 2 used workloads following a selection of DAG shapes with the structures described in Section 4.4.1. The parameters for this selection and to generate these shapes is given in Table 5.2. A record of dependencies was not stored in the industrial logs so each task from the logs was considered to be independent in Profile 3. In the workload made for Profile 3, tasks from the same user that were submitted at the same instant were grouped into the same job, though without dependencies. Profile 4 consisted of jobs with random DAGs with exponential degree distribution, generated using Algorithm 4.8.

	Uniform Independent	Log-Uniform Independent	Erdős–Rényi	Fork/Join	Chained Fork/Join
Task core count distribution	random selection from (1,5,10,15,20)				
Number of workloads	30				
Workload Execution Time	10^{10}				
Proportion of Kind1:Kind2	80:20				
CCR	0.2				
Distribution of Job Execution times	Uniform	Log-Uniform	Log-Uniform	Log-Uniform	Log-Uniform
Distribution of Task Execution Times	-	-	Log-Uniform	Log-Uniform	Log-Uniform
Number of Jobs	10,000	10,000	1,000	1,000	1,000
Tasks per Job (uniform distrib.)	1	1	1-20	-	-
Dependency Probability	-	-	0.3	-	-
Fork/Join Width	-	-		1-10	1-5
Fork/Join Length	-	-	-	3-15	3-15
Chains of Fork/Join segments	-	-	-	-	1-3

Table 5.2: Parameters used in workload generation

5.7.1.5 Fair Shares

For evaluating the FairShare scheduler, a share tree is required that follows the share tree architecture discussed in Section 2.5.1). Profile 1 used a share tree of 5 equal shares and where each job was randomly placed into one of the 5 shares. Profile 3 used the real industrial share tree. For profiles 2 and 4, a richer share tree was used (shown in Table 5.3) where the shares would not all be equally balanced and so give a more comprehensive test of the FairShare scheduler. Jobs in the workloads of profile 2 and 4 were assigned to each of the share leaf paths randomly with a chance proportionate to the share of each 'user'.

5.7.1.6 Load

The level of load on the platform can be measured by the percentage rate at which work is arriving compared to the maximum rate at which this work can be processed. Comparing schedulers at a range of loads is essential, because of the variation experienced in grid and cloud scenarios. A load ratio for a workload can only ever be defined with relation to a platform, yet it is desirable to be able to adjust the load ratio independently of the workload and platform. This can be achieved by adjusting the inter-arrival times of jobs, following the algorithm described in Algorithm 4.2.

Share Path	Shares	Share Path	Shares
/root	1	/root/group4/kara	4
/root/group1	1	/root/group4/lana	5
/root/group2	1	/root/group4/mera	6
/root/group3	1	/root/group4/nora	3
/root/group4	2	/root/group4/olga	4
/root/group5	1	/root/group4/petra	5
/root/group1/anna	10	/root/group4/qia	3
/root/group1/becca	10	/root/group4/rana	4
/root/group1/cara	5	/root/group4/sara	2
/root/group2/dana	10	/root/group5/tana	1
/root/group2/ella	10	/root/group5/ulla	1
/root/group2/fara	10	/root/group5/viva	1
/root/group3/gemma	1	/root/group5/wanda	1
/root/group3/hanna	2	/root/group5/xandra	1
/root/group3/ida	3	/root/group5/yena	1
/root/group4/jenna	3	/root/group5/zara	3

Table 5.3: Synthetic Share Tree Used for Simulations

When investigating load, an upper limit on overload was needed. A value of 120% overload was chosen as an upper limit as that would imply that at least one sixth of the work submitted (by volume) would be unable to run. Over an extended period such as a year, this is likely to be unacceptable to users. If there were continually this much overload, it is likely that a better solution, such as admission control, user education or hardware upgrades be implemented to reduce overload.

Profile 1 investigated a range of load between 80 and 120%, in increments of 10%, to examine how gracefully performance degraded under the different policies as the threshold of overload is passed. Where the daily and weekly peaks of work are considered, there may be short periods of overload even when the average load is well below saturation. Therefore Profile 3 considered load between 20 and 120%.

Profile 2 only considered an overload situation at 120% so that it would be necessary for some jobs to wait and therefore the ability of the schedulers to keep responsiveness and fairness high can be compared.

5.7.1.7 CCR

Networking delays are considered according to the model of Section 5.2.2 whenever there needs to be communication between clusters. Profile 1 uses a CCR value of 0.2 so that network delays are present, but are relatively small compared to the computation costs. Due to there being no dependencies available in the industrial workload, the CCR is irrelevant for profile 3, so it was set to 0. Profile 2 and 4 are used to investigate

the impact of network delays across the range of CCR, using ranges of 0.0 to 1.5 and 0.0 to 2.4 respectively.

5.7.1.8 Inaccurate Estimates of Execution Times

It is usually impossible, except in rigorously studied Real-Time Systems, to have precise estimates of how long work will run for [105, 107, 156]. In the experimental set up considered in this thesis, it is assumed that the person who submits work or an automated job profiler provides an estimate of execution time, which is helpful but imperfect. In simulation, however, the exact execution times are known in advance. Therefore, it is necessary to introduce inaccuracies into the model. In this work, two possible ways are considered to convert exact execution times ($e_{\texttt{orig}}$) into inaccurate estimates:

Normal Error This creates an estimate by sampling a normal distribution, shown in Equation 5.27, with a parameter N to vary the standard deviation, and hence the inaccuracy, of the estimate.

$$e_{\texttt{est}} = \left\lceil \texttt{normal}\left(\mu = e_{\texttt{orig}},\ \sigma = e_{\texttt{orig}} \times \frac{N}{100}\right)\right\rceil \tag{5.27}$$

Logarithmic Rounding This form of inaccuracy (Equation 5.28) reflects the expertise of users in being able to classify jobs by whether they will take minutes, hours or days. However, this classification may be most precise users can be. Execution times are essentially rounded to the nearest power of M.

$$e_{\texttt{est}} = M^{\lceil \log_M(e_{\texttt{orig}})\rceil} \tag{5.28}$$

Profile 1 does not consider inaccurate estimates, as it is used to examine the influence of load. Profile 2 considers the extremes of inaccurate estimates. Therefore, the standard deviation of the normal distribution used to introduce error ranges from 0 to 10^8% of the original value. The log rounding uses values of M from 1 to 10^7. These very large values are used so that the most extreme rounding essentially gives every task the same predicted execution time. In profile 4, the investigation of value is intended to consider realistic rather than extreme scenarios. Where log-rounding inaccuracies were considered, a range of 1 to 20 was considered. Normally-distributed inaccuracies used the mean of the execution time and a standard deviation of up to 200% of that execution time.

5.7.2 Synthetic Platform

Each profile used a slightly different platform in order to ensure the workload considered ran for a period of a year or greater yet was not so large that CPU or RAM resources on the simulation machines where exhausted. All platforms follow a common architecture, with several clusters connected with a thin tree network. Each cluster is homogeneous, but there were two kinds of cluster architecture, termed *Kind1* and *Kind2*. In profiles 1 and 2, the proportion of the workload that requires the *Kind2* architecture is deliberately lower than the share of the grid, in order that the network is the bottleneck when running tasks on the *Kind2* cluster.

The platform used in profile 1 has four clusters, each with 1,000 cores. Three clusters are of *Kind1* and one is of *Kind2*. These were connected by a single router. The simulated grid used for Profile 2 also has four clusters, but these clusters had 400 cores each. One cluster was Kind2 and the other three were Kind1. The platform in profile 2 is connected by several routers, following the architecture shown in Figure 5.1. Profile 4 specified a grid of three clusters, two of 2000 cores each of Kind1 and one of 1000 cores of Kind2. These were connected through a single router. Profile 4 was designed to most closely correspond to a scaled-down version of the real industrial grid architecture.

5.8 Summary

This chapter describes the application, platform and scheduling models that can represent the industrial context in a way that is amenable to simulation. The application model represents the work to be run on the grid, which consists of tasks with dependencies between them, which are grouped into jobs. The formal structure of the workloads has also been described in this chapter whereas Chapter 4 characterises the industrial workload and gave means of generating workloads following the observed patterns.

The platform model is also given, describing how the processing resources are grouped into clusters and connected using a tree-structured network of routers. Heterogeneity is modelled such that tasks run on a subset of clusters, depending on resource requirements, but have the same execution time wherever they are run. The network architecture of a thin tree is described, along with a low-complexity model to determine network latencies between tasks executing on different clusters.

The hierarchical list scheduling model is described, specifying the way jobs and tasks are cascaded down through the network. The allocation policies used for load balancing between clusters and on the clusters themselves are described. The ordering policies on the routers are in a sense irrelevant because jobs and tasks are pushed down to the clusters instantaneously on arrival, although they were

Profile	1	2	3	4
Experiment to evaluate	Responsiveness and fairness against load	Responsiveness and fairness against network delays; inaccurate estimates of execution times	Responsiveness and fairness, industrial workload	Value against load, network delays, inaccurate estimates of execution times
For investigation Of Hypothesis	1	1	1	2
Tasks/workload	10,000	10,000	>100,000	100,000 - 125,000
Jobs/workload	1,000 – 10,000	1,000 – 10,000	>100,000	12,500
Workload total execution time (minutes)	10^{10}	10^{10}	-	10^{10}
Execution time distribution	Uniform and sampled from industrial log-uniform	Sampled from industrial log-uniform	industrial workload	Sampled from industrial log-uniform
Arrival Patterns	Constant	Constant and day/week pattern	industrial workload	Constant and day/week pattern
DAG Shapes	Selection, See Table 5.2	Selection, See Table 5.2	None	Random Exponential Degree
FairShare Tree	5 equal shares	Synthetic (Table 5.3)	Industrial tree	Synthetic (Table 5.3)
Load %	80, 90, 100, 110, 120	120	Industrial	10, 20, 40, 60, 80, 90, 100, 110, 120
CCR	0.2	0.0 – 1.5	0.0	0.0 – 2.4
Kind1:Kind2 ratio	80:20	80:20	n/a	80:20
Inaccurate estimates	None	Normal σ=0-10^8 Log base: 1-10^7	None	Normal σ=0-200 Log base: 1-20
Platform	4000 cores: 3000 Kind1, 1000 Kind2	1600 cores: 1200 Kind1, 400 Kind2	Industrial	5000 cores: 4000 Kind1, 1000 Kind2
Network Topology	Single router	Hierarchical (Figure 5.1)	Hierarchical, industrial topology	Single router

Table 5.4: Experimental Profiles

specified to be FIFO. The ordering policies for tasks on clusters were not specified, because these will form the main area of research. Various different ordering policies for the clusters will be evaluated in following chapters.

So that fair evaluations can be made, a survey of metrics is undertaken. The collection of metrics is essential for the owners and operators of grid and cloud platforms to ensure good utilisation of their platforms and quality of service for their users. Furthermore, the perspective of the users necessitates the evaluation of metrics that deal with quality of service. A number of metrics are presented, grouped into those dealing with Utilisation, Responsiveness and Fairness. It is shown that while utilisation metrics have traditionally been used to evaluate scheduling policies, they are less suitable in a dynamically-scheduled system such as a grid or a cloud. Instead, responsiveness and fairness metrics are better able to show how a scheduling policy is managing the resources under its control in order to maximise the benefit to users.

The metrics are evaluated as to their ability to give insight into scheduling issues. The Schedule Length Ratio (SLR) metric of Topcuoglu et. al. [160] is shown to be particularly useful for workloads with dependencies, because it uses the critical path of the job as the performance benchmark to compare against. This allows it to more fairly compare the responsiveness of jobs with critical paths of different lengths. The next chapter will present and evaluate a novel scheduling policy designed to achieve good responsiveness and fairness with respect to SLR.

To validate that these models are appropriately representative, the industrial logs were used to create a workload suitable for use in the simulator. The simulator was configured to represent the same platform and scheduling policy as the industrial grid. The simulated set up gave responsiveness and fairness metrics for each job that were at most 10% different from those observed in reality, and most were even closer. The differences are likely due to the absence of dependencies in the logs, as well as the limitations of the network model and the fair-share awareness improvements the simulator used in the load-balancing policy.

Four experimental profiles are defined, each suited to evaluating a particular aspect of the research hypotheses. These profiles specify the parameters used with the algorithms of Chapter 4 to generate several classes of synthetic workloads. The profiles also define the simulated grid platforms these workloads would be executed on, as well as the limits of the spectra of load, network delays and inaccurate execution times that scheduling policies are to be evaluated across.

Having defined the experimental approach, Chapter 6 will define and evaluate a scheduling policy designed to achieve high responsiveness and fairness, even in the presence of overload. Chapter 7 considers the application of value measures to jobs in a workload, and defines and evaluates scheduling policies designed to optimise the value returned.

Chapter 6

Scheduling using SLR

In Chapter 5, the models and metrics required to evaluate scheduling policies and grids were discussed. It is suggested that the most informative metric for measuring responsiveness is the *Schedule Length Ratio* (SLR) [160], when applied to each job in a workload. This chapter presents a novel ordering policy, termed Projected Schedule Length Ratio, or *P-SLR*. P-SLR is designed to achieve responsiveness and fairness while retaining a guarantee that no job will ever starve.

This chapter then considers ordering policies that can form a basis for comparison for P-SLR. Issues with these existing policies are discussed, which include their shortcomings with respect to responsiveness, fairness or the absence of starvation.

P-SLR is then evaluated using a variety of different means to show that it meets its aims. Firstly, it is compared against baseline schedulers for responsiveness and fairness using a range of different synthetic workloads and load ratios, along with an industrial workload. The effects of adding network delays and inaccurate estimates are also investigated.

Responsiveness will be measured using the median value of the worst-case SLRs observed in each trial. The worst-case SLR is used because of the desire for high responsiveness to be achieved for all users. Using the median value instead of the mean will prevent any truly pathological cases from biasing the results. Fairness will be measured using the median of the Gini Coefficients [112] calculated for the SLRs in each trial. Statistical significance will be tested using a repeated measures t-test because the workloads are the same, meaning the job SLRs can be directly compared. The threshold for statistical significance is set at the 5% confidence interval ($p = 0.05$).

6.1 The Projected-Schedule Length Ratio Policy

Having considered the benefit realised by using the SLR metric to measure scheduling performance, a novel ordering algorithm called Projected-Schedule Length Ratio (P-

SLR) is now presented. The P-SLR ordering policy takes the concept of upward rank, and uses it to give a projection of when the job would finish if the considered task were run immediately. This projection of the job finish time is used to calculate a projection of what the job's SLR metric would be, which is used as the basis of the ordering policy.

The nominal intent of the P-SLR orderer is that as the load of the system rises (especially into a state of overload), all jobs should 'suffer' equally. At a scheduling instant, the upward rank of every task is used to predict what the SLR of the job would be if this task were executed immediately. (Algorithm 6.1). The task where the P-SLR is largest (is most 'late') is run first. This is distinct from the approach used by Hirales-Carbajal et. al. [77] which uses the downward rank (looks backward) to calculate a partial value for SLR based on the tasks that have already completed.

The advantage of using the SLR metric is that small jobs can 'jump' the queue to run quickly because their SLRs are more sensitive to the same waiting time. However, eventually, even large jobs will run because their projected SLR will rise as they wait, just more slowly than for small jobs. This is designed to have all jobs experience a waiting time proportional to the length of their critical path, a desirable attribute for fairness and responsiveness [147].

P-SLR is starvation-free because the projected SLR rises for all jobs as they wait, which means all jobs will eventually run, as long as overloads are transient. This follows a similar line of reasoning as Salles and Barria [145]. In the case of extreme overload where the work queue continually grows unboundedly, the waiting time term (the second part of the equation in Algorithm 6.1) comes to dominate, reverting the ordering to that of FIFO, thus avoiding starvation in all cases.

A particular factor of note that is shown in Algorithm 6.1 is that the predicted finish time is incremented by 1. Two jobs of differing sizes could be submitted at the same scheduling instant. Without this increment, both jobs' projected SLR would be 1, and hence the choice between them would be arbitrary. By adding a lateness penalty to every calculation, the projected SLR is able to distinguish between short and long jobs that arrive at the same time, and prefer running the shorter one first. This improves responsiveness overall, because the SLRs of small jobs are most sensitive to waiting time. This is the opposite behaviour to the static scheduler proposed in Zhao and Sakellariou [175], which prefers running the larger job first, which is good for lowering workload makespan but causes the responsiveness of the smallest task to suffer.

Another factor to note is that tasks not on the critical path for a job may have a small upward rank, even though they may be ready early on. This can mean that the projected SLR for these tasks is less than 1. However, the result of this is that they are prioritised lower than the tasks on the critical path; a desirable attribute [175].

Algorithm 6.1 Projected SLR ordering algorithm

$$\texttt{projected\_slr}(T^i, J^k, \texttt{curr\_time}, Q) =$$

$$\frac{\left(T_R^i + \texttt{curr\_time} + 1\right) - J^k_{\texttt{arrive}}}{J^k_{\texttt{CP}}} + \left\lfloor \frac{\texttt{curr\_time} - J^k_{\texttt{arrive}}}{\forall J^n \in Q : \max\left(J^n_{\texttt{CP}}\right)} \right\rfloor^2$$

6.1.1 Algorithmic Complexity of P-SLR

For the calculation of the complexity, several terms can be defined. The number of jobs in a workload can be referred to as j with t being the number of tasks. The number of tasks in the queue Q at a given moment is termed l. Defining these formally from the previous definitions in Section 5.1 gives:

$$\begin{aligned} j &= |W| \\ t &= \sum_{J^k \in W} \left|J^k\right| \\ l &= |Q| \end{aligned}$$

The P-SLR function needs to be calculated for each task in the queue at each scheduling instant. The upward ranks of each task can be pre-calculated when a job is submitted, so these can be assumed to be known and not have to be re-calculated each time the scheduler is run. A scheduling instant is triggered each time a job arrives or a task finishes. In the worst case where each job has only a single task and where no start or finish times ever coincide, this would mean that there are $2t$ scheduling instants for a given workload.

At each scheduling instant, the scheduler must evaluate the P-SLR for each task in the queue and sort the queue by this value. Assuming the upward ranks are known, the calculation of the P-SLR for each task can be performed in a constant amount of time. This would mean that the number of operations to be performed at each instant would be $l + l \log l$, assuming a sorting algorithm of $O(n \log n)$ worst-case complexity is used.

The number of operations run by P-SLR in a dynamic system would therefore on average be $2t(l + l \log l)$. However, a further worst-case can be imagined if P-SLR were applied to a scenario where it starts with all the work to do already in the queue (as is usual in static scheduling). In this case, the length of the queue l would be equal to the number of tasks t. This would mean that the worst-case complexity is $2t(t + t \log t)$, or $2t^2 + 2t^2 \log t$, which simplifies to $O\left(t^2 \log t\right)$ using Big O notation.

6.2 Alternative Scheduling Policies

This section will consider a set of ordering policies that can be applied on each cluster, within the models defined previously.

6.2.1 Random

The random ordering policy randomly chooses from the set of ready tasks which should be the next task to run. This policy is useful as it can provide a baseline against which the performance of other ordering policies can be compared, because it operates with no information about the workload. For any ordering policy to be worth using, it must demonstrate that it produces significantly better schedules than the random scheduler. Although in the short-term, the random scheduling policy could suffer from starvation, it is statistically improbable that a job could starve forever.

6.2.2 FIFO Task

The FIFO Task orderer is another simple ordering policy, albeit one that is widely used. Jobs are decomposed into their component tasks. As tasks become ready, they are placed into a FIFO queue. Tasks are removed from the head of the queue and allocated to the grid as resources become free. Any FIFO queues are starvation-free, because while ever the cluster is executing work, jobs will rise to the head of the queue and be executed in the order they arrived.

6.2.3 FIFO Job

This is a slight modification to the FIFO Task ordering policy, designed to avoid the multiple waits problem. Ready tasks in the queue are ordered first by the order in which their respective jobs were submitted, then by the order in which they became ready. FIFO Job is starvation-free in the same way as FIFO Task, because it is based on a FIFO queue.

6.2.4 Fair Share

The FairShare policy is described in detail in Chapter 2. It aims to achieve fairness with respect to utilisation according to a share tree [93, 133, 137]. The issues this causes for the industrial partner were discussed in Chapter 2.6.

6.2.5 Longest and Shortest Remaining Time

The Longest Remaining Time First (*LRTF*) and Shortest Remaining Time First (*SRTF*) ordering policies use concept of *Upward Rank* [160]. Upward Rank is defined for

each task, and is the length of the critical path that remains to be completed after the task has executed. LRTF and SRTF sort the list of tasks by decreasing and increasing Upward Rank, respectively. These policies can suffer from starvation under overload, because the shortest (LRTF) or longest (SRTF) tasks may never reach the head of the queue. The highly regarded HEFT scheduler uses the LRTF ordering policy [160].

6.3 Evaluation of P-SLR for Responsiveness and Fairness

6.3.1 Experimental Hypotheses for Responsiveness, Fairness and Utilisation and Testing Approach

The new P-SLR policy needs to be evaluated in order to compare its performance to the other policies given. This section will give three experimental hypotheses that will be investigated, along with the ways in which the hypotheses will be tested. These experimental hypotheses will be examined using the synthetic simulation parameters and platform of Profiles 1 and 2 from Section 5.7. They will also be investigated using the real industrial workload and simulated platform of Profile 3 (Section 5.7).

- Experimental Hypothesis A: The P-SLR orderer gives schedules with a higher degree of fairness than alternative policies. i.e. it does not particularly favour small or large jobs, but achieves the same responsiveness across the range of job total execution times.

The distribution of SLR throughout the workload is used to measure fairness. Three classes of prioritisation relative to execution time are described, as shown in Figure 6.1. Class 1 is where longer jobs are prioritised over short jobs, with Class 2 being the opposite case. Class 3 is where there is equal prioritisation with respect to responsiveness across the range of job run-times.

The common scheduling policy First In First Out (FIFO) falls into Class 1 because on average, each job will wait in the queue for the same length of time [147]. This waiting time is proportionately larger relative to execution time for smaller tasks, penalising the SLR of short-running jobs. This pattern is true for any policy not considering execution times [147]. The Longest Remaining Time First (LRTF) scheduler also falls into Class 1. The Shortest Remaining Time First (SRTF) scheduler is of Class 2. The P-SLR scheduler is deliberately designed to exhibit Class 3 behaviour.

To measure fairness, the standard deviation of the SLRs for each workload will be used. These will be displayed graphically as a box plot, to show the relative measures. Statistical significance will be tested using a repeated measures t-test. It is useful to use the repeated measures test, because the workload and load ratio are the

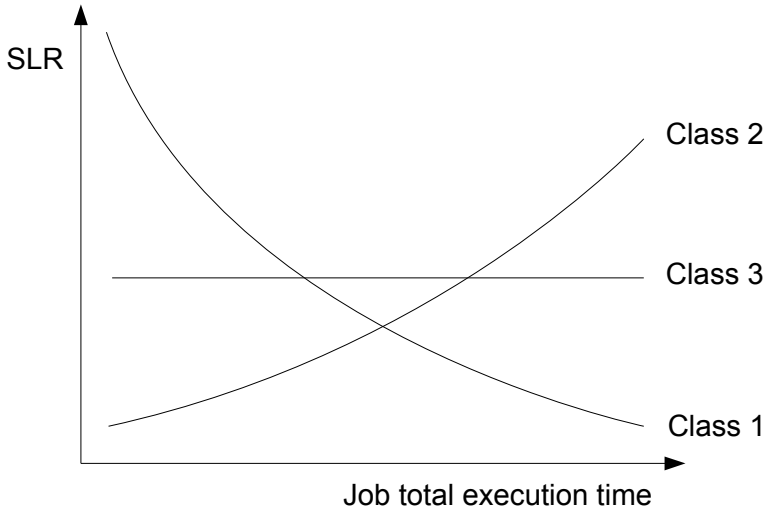


Figure 6.1: Classes of prioritisation by execution time

same, and only the ordering policy has changed; this means that pairs of values can be compared. The threshold for statistical significance is set at the 5% confidence interval. The null hypothesis A will be that the P-SLR ordering policy gives values of SLR Standard Deviation indistinguishable from the alternative scheduler.

However, it is useful to visualise how the different ordering policies achieve fairness across the spectrum of job execution times. This will be achieved by plotting the worst-case SLR value by the decile of job execution time. This will make it possible to see which schedulers effectively prioritise large or small jobs, or achieve a fair balance of SLRs across the range of job execution times. At low load ratios, it is possible that no jobs would prioritised over others because anything can run immediately, and hence the plot would be uninformative. Therefore, the worst-case SLRs by decile of execution times will be plotted at 120% load ratio.

- Experimental Hypothesis B: The P-SLR orderer gives schedules with a higher degree of responsiveness than alternative policies

As outlined above, responsiveness is best measured using the SLR metric for each job in a workload. The responsiveness of the P-SLR orderer will be evaluated by examining the worst-case SLR for each workload when run with each scheduler. This will be evaluated for statistical significance also using the repeated measures t-test. For further insight, the worst-case SLR metric will be plotted against load ratio to see how the different ordering policies cope as load increases. The worst-case SLR is a better metric of responsiveness than mean SLR, because the mean could mask poor performance on a small subset of jobs, even though that poor performance may be critical to users.

At each step of load ratio, the worst-case SLRs of each workload will be recorded for each ordering policy. To see if P-SLR is the most responsive, the percentage of

cases in which P-SLR dominates the other ordering policies will be calculated. To check whether this dominance is statistically significant, the repeated measures t-test will also be used. The null hypothesis B is that the P-SLR orderer gives no significant improvement in worst-case SLR values.

- Experimental Hypothesis C: The P-SLR orderer does not give a significantly different rate of utilisation over alternative policies

Utilisation metrics that use the makespan are not ideal for measuring a dynamic system, because the makespan will tend to be most influenced by the last few tasks to arrive. Average utilisation may be poor if most of the cluster is idle while the last task finishes. However, if an ordering policy gave significantly lower utilisation than others, it may not be as desirable because it cannot make good use of the cluster. Average utilisation values given by each ordering policy will be plotted in a box-plot to see the range of values. The null hypothesis C is that there is no statistically significant difference between the average utilisation values of the other orderers and the P-SLR orderer.

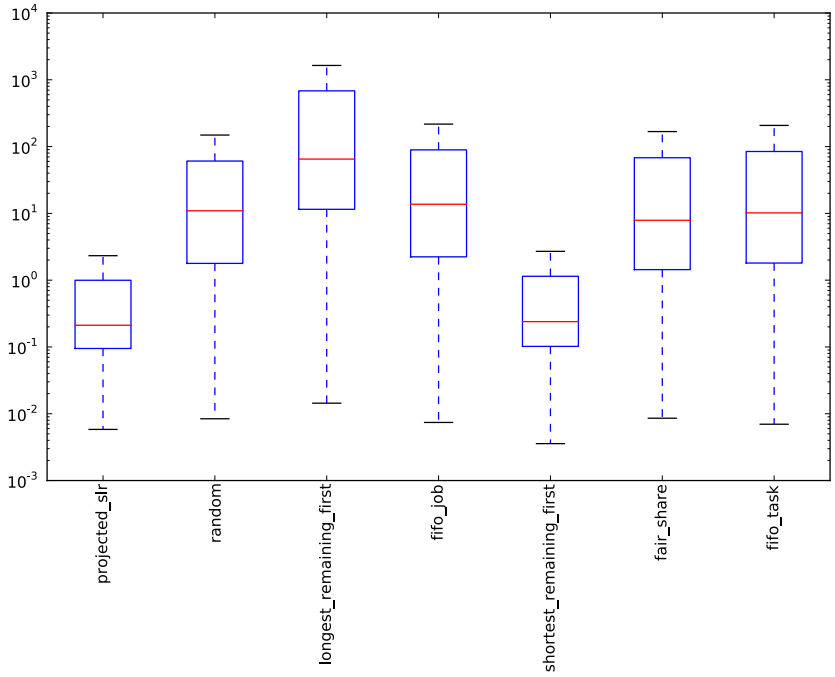
Further metrics will not be analysed in as much detail, but will still be plotted for completeness. The cumulative completion metric, using a standard makespan for all schedules, will be plotted on a box-plot. This will be to see how quickly the schedulers are able to finish the work that has arrived. The peak in-flight count will also be plotted on the box-plot, to see how much interleaving of jobs is made to happen by the ordering policies.

6.3.2 Scheduler Evaluation (Synthetics)

To give confidence in the investigation the performance of the scheduling policies, a large number of synthetic workloads were generated, according to the parameters in Table 5.2. There were 5 kinds of workload with 30 individual workloads each, evaluated for 5 load ratios, each using one of 7 ordering policies. This gave 5250 individual schedules produced.

6.3.2.1 Fairness

Standard Deviation of SLR To evaluate the fairness of the ordering policies, the standard deviation of the SLR values is calculated for each schedule produced. These values are displayed in a box-plot, shown in Figure 6.2. The null hypothesis A states that the P-SLR orderer would produce standard deviations of SLR indistinguishable from the other ordering policies. This was tested for statistical significance using a repeated measures t-test, $p = 0.05$. The null hypothesis A is refuted by the t-test giving a p value lower than 0.05 for all orderers except the SRTF policy. The reason for this is clear from Figure 6.2, that the distribution of standard deviations of SLR



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 6.2: Standard Deviation of SLR by ordering policy

is similar between the P-SLR and SRTF. To further examine this result, the effect of scheduling policies on different sized tasks will be examined.

Mean SLR by Decile Figure 6.3 shows the mean SLR by decile of job execution times at 120% load ratio. This high level of load was chosen so that some tasks must wait, and then the schedulers can be compared by which kinds of tasks are made to wait.

The LRTF orderer prioritises the longest jobs the most, with the lowest decile score for the largest tasks, and penalises the smallest tasks most, with the highest mean SLR score for the smallest tasks. This is exactly what would be expected of the ordering policy.

The Random, Fair Share and FIFO ordering policies all follow a similar profile. This is due to the fact that in these orderers, all tasks will wait in the queue for approximately the same amount of time. Naturally, this penalises the SLR of the shorter tasks more than the larger ones.

The SRTF orderer follows the opposite pattern, prioritising the smallest tasks the most and penalising the largest tasks. Across most of the workload space, the SRTF orderer gives the lowest mean SLR value. However, this crosses over for the 10th decile (the largest jobs), where the highest mean SLR is produced by SRTF. However, because the largest tasks are so large, they are much less sensitive to delays than

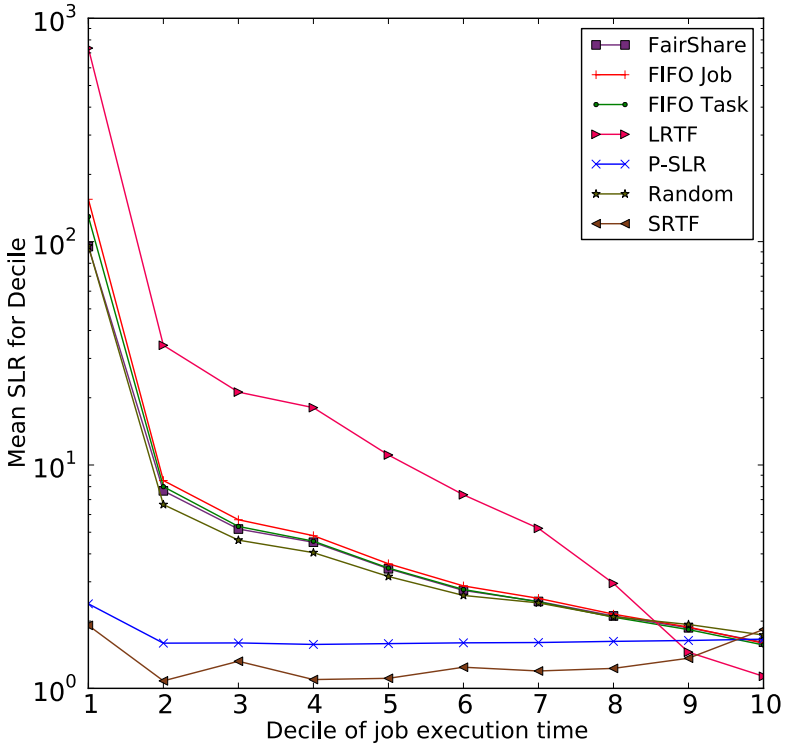


Figure 6.3: Mean SLR by decile of job execution times, 120% load ratio

shorter tasks. In the simulations, the workloads were allowed to run to completion after jobs had finished arriving, which meant that every job would eventually finish. In reality, in an overloaded system, this may not be the case. Because the SRTF scheduler is not starvation-free, the worst-case for the largest jobs may be much worse in reality.

The P-SLR orderer, as intended in its design, shows no bias in terms of SLR across the range of execution times. Because the largest jobs are guaranteed to run, this will have an impact on all of the smaller jobs in the system. However, this penalty is shared out equally across the workload.

The uptick in mean SLR seen in the first decile can be attributed to small jobs arriving when no resources are free in the cluster. Even the delay until the next instant when some resources become free can therefore cause SLR to increase significantly. As the size of a cluster increases, however, this uptick would be less pronounced for a similar workload, because the expected delay until some processors become free will decrease.

For the fairness metrics, it can be seen that the P-SLR ordering policy provides a statistically significant (repeated measures t-test, $p = 0.05$) improvement in fairness over the LRTF, Fair Share, Random and FIFO-based ordering policies. The P-SLR

orderer also delivers a statistically insignificant difference in fairness from the SRTF ordering policy, even while P-SLR offers a guarantee that no job will ever starve.

6.3.2.2 Responsiveness

Worst-Case SLR The responsiveness achieved by each ordering policy can be measured by the worst-case SLR for each schedule. The null hypothesis B for responsiveness states that worst-case SLR values produced by the P-SLR ordering policy are indistinguishable from those produced by alternative policies. The distributions of worst-case SLR values for each schedule are shown in Figure 6.4. Statistical significance between the distributions is evaluated using the repeated measures t-test, $p = 0.05$. As with the fairness hypothesis A, the null hypothesis B is rejected for P-SLR compared to all the ordering policies except SRTF. P-SLR and SRTF give significantly better responsiveness than the other scheduling policies, and are statistically indistinguishable from each other. These ordering policies achieve low worst-case values because they prioritise or give equal treatment to the smaller jobs in the workload, as shown in the previous section. Because the smallest jobs are also the most sensitive to delays, reducing their SLR value is key to achieving the best responsiveness possible.

Median values of Worst-Case SLR The ordering policies can also be compared as to how their ability to achieve responsiveness as the load ratio is increased. This is plotted in Figure 6.5. Throughout the range of load, the longest remaining first orderer has the worst worst-case SLR. This is to be expected, because it prioritises the largest tasks, and the smallest tasks' SLRs suffer proportionately more when they are delayed. At the other end of the scale, SRTF and P-SLR show similar values for the lowest worst-case.

It is not surprising to observe that the random orderer achieves better or similar mean worst-case SLR values across the spectrum of load ratio when compared to the Fair Share and FIFO-based orderers. This is because shorter tasks must always wait the whole length of the queue in FIFO-based ordering policies. This will penalise small tasks heavily, and lead to a high worst-case SLR. The random scheduler, on the other hand, allows some small tasks to 'jump the queue', and hence lower the likelihood that they will have to wait the full duration of the queue before being executed.

It is possible to use the worst-case SLR metric to calculate a relative metric of dominance. Dominance is the number of schedules where the worst-case SLR achieved by P-SLR is less than or equal to that achieved by the alternative orderer. The values of the dominance metric across the load ratio spectrum are shown in Table 6.1, where values in bold indicate a lack of statistically significant difference

(repeated measures t-test, $p = 0.05$) between P-SLR and the alternative policies. As with the other metrics, it is found that P-SLR dominates all the other ordering policies except SRTF. In the case of SRTF, the null hypothesis B cannot be refuted and therefore P-SLR is statistically indistinguishable, across the load ratio spectrum.

Mean Values of SLR The dominance metric can also be applied to the mean SLR metric, another responsiveness measure (Table 6.2, values in bold again indicate a lack of statistically significant difference, repeated measures t-test, $p = 0.05$). As previously observed with the other metrics, P-SLR dominates all the orderers except SRTF. However, at higher load ratios, the null hypothesis B is again refuted, showing that there is a significant difference between the performance of P-SLR and SRTF. This is because SRTF achieves better performance for small tasks, which make up the majority of the workload considered. This brings the mean down, and has SRTF dominate P-SLR for mean SLR under high load.

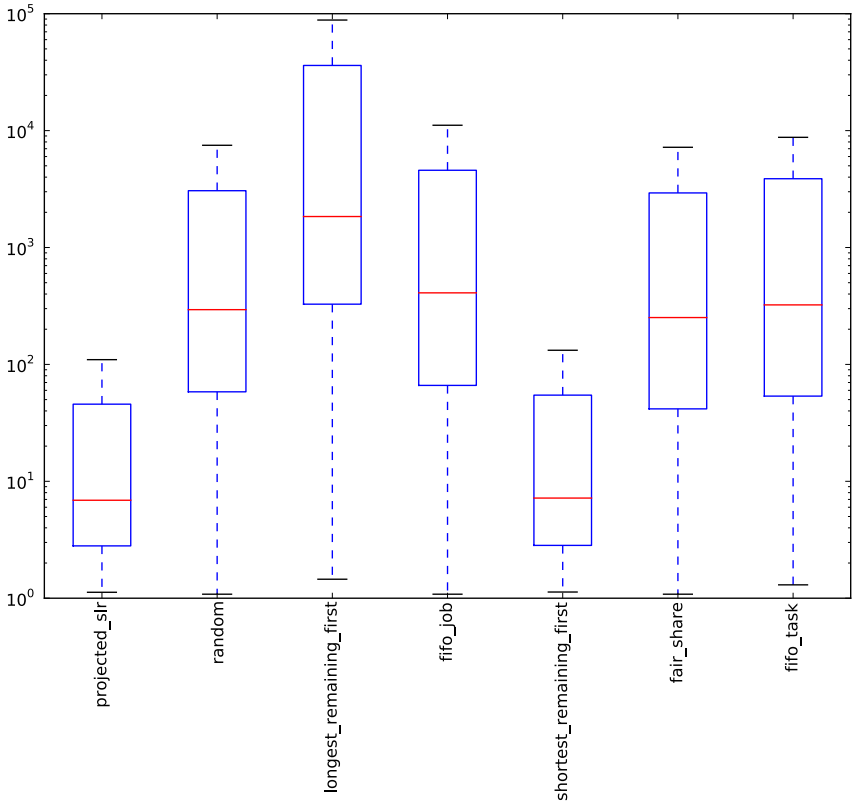
The responsiveness measures, therefore, show that the P-SLR orderer gives more responsive schedules than the Random, LRTF, Fair Share and FIFO-based orderers, by dominating their mean and worst-case SLR values across the load spectrum. The P-SLR achieves worst-case SLR results statistically indistinguishable from the SRTF order, although the SRTF orderer achieves significantly better mean SLR results at high load (repeated measures t-test, $p = 0.05$).

6.3.2.3 Utilisation

Average Utilisation Figure 6.6 shows the average utilisation across the different orderers. In this experiment, the null hypothesis C is rejected for all other ordering policies. Statistically, P-SLR has a higher average utilisation than SRTF and a lower utilisation than all other schedulers. However, although this produces a statistically significant result (repeated measures t-test, $p = 0.05$) because of the large sample size, it can be argued that the difference is small, as can be seen from the size of the boxes in Figure 6.6.

% Dominated by Projected-SLR	**Load %**				
Policy	80	90	100	110	120
Longest Remaining Time First	96	100	100	100	100
Shortest Remaining Time First	**54**	**47**	**56**	**58**	**57**
Random	87	93	100	93.3	100
FIFO Task	87	98	100	100	100
FIFO Job	92	94	100	100	100
Fair Share	87	95	98.6	100	99.3

Table 6.1: Dominance of Projected-SLR orderer over Worst-Case SLRs



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 6.4: Worst-Case SLR by ordering policy

% Dominated by Projected-SLR	**Load %**				
Policy	80	90	100	110	120
Longest Remaining First	97	100	100	100	100
Shortest Remaining First	**59**	**54**	44	26	21
Random	88	97	100	100	100
FIFO Task	91	98.3	100	100	100
FIFO Job	94	97	100	100	100
Fair Share	92	98.6	100	100	99.3

Table 6.2: Dominance of Projected-SLR orderer over mean SLRs

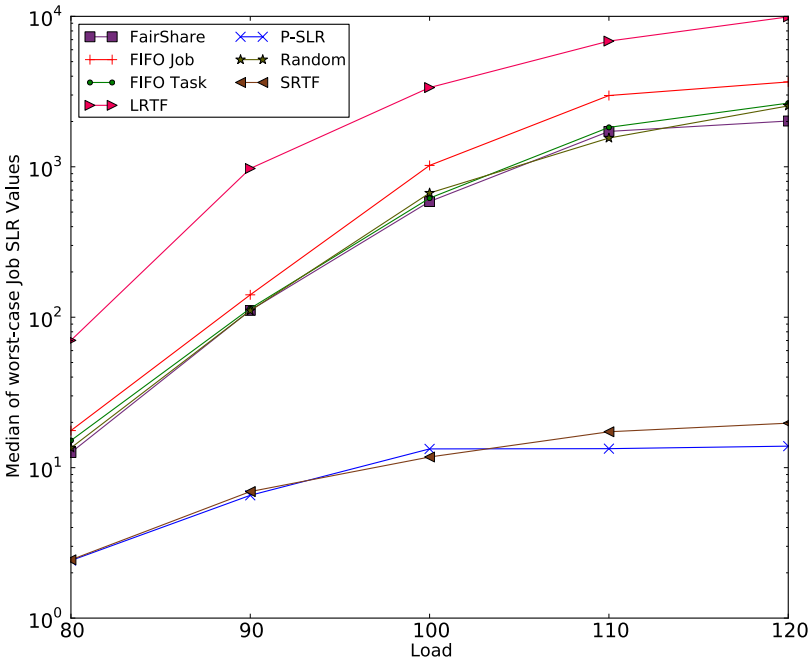
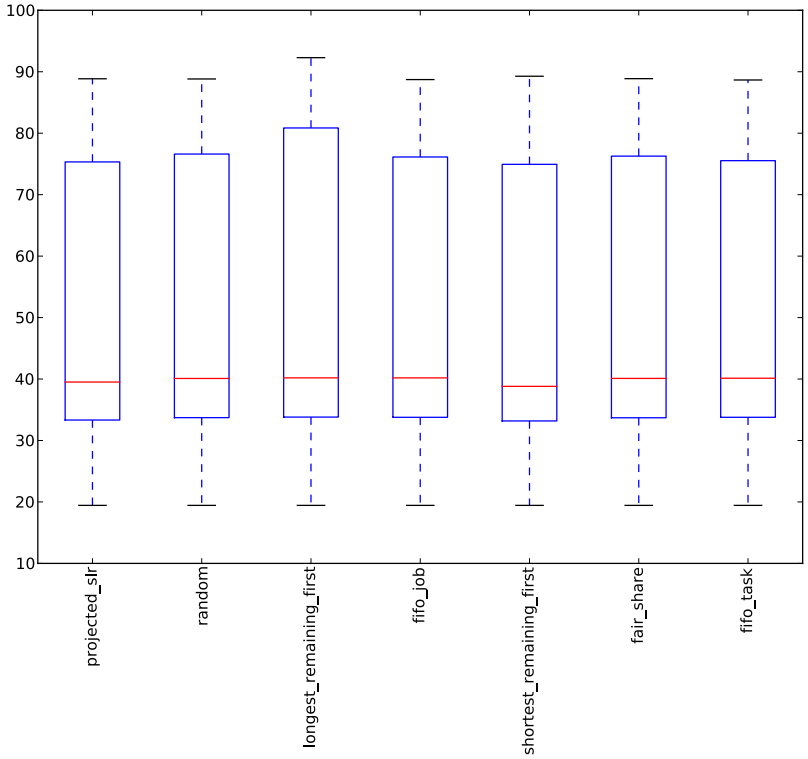


Figure 6.5: Median worst-case SLR by load ratio



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 6.6: Average Utilisation by Ordering Policy

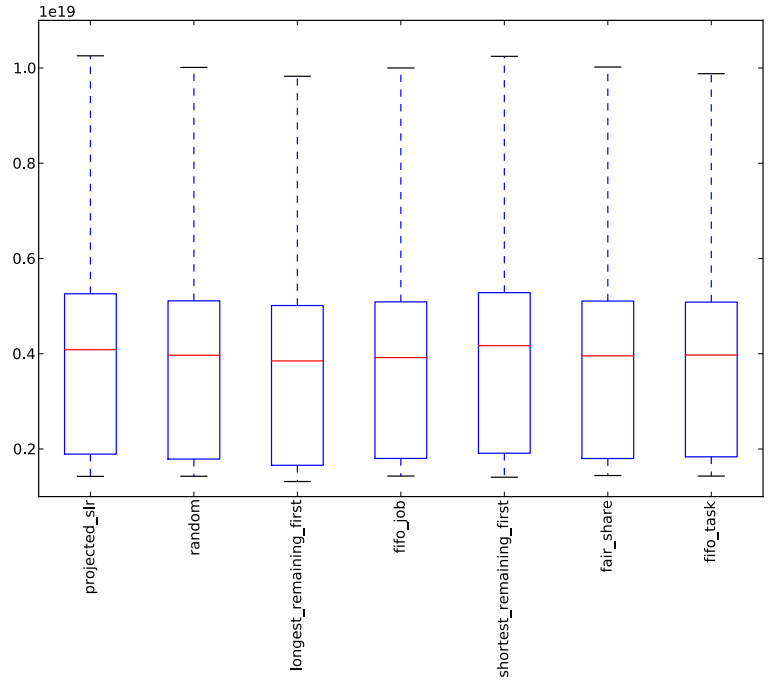
Furthermore, utilisation is calculated based on the workload makespan. In this simulation, where the workload is left to run to completion, the workload makespan is likely to be decided by a single large job that arrives late in the schedule. This is corroborated by the low median values for utilisation shown. These are only low over the whole makespan, because the makespan is significantly extended by large jobs running at the end of the schedule.

Therefore it is concluded that although the utilisation achieved by the P-SLR scheduler is statistically significantly lower than for the orderers other than SRTF, it is not of a magnitude that is cause for concern. Utilisation can be considered to be effectively equal over the orderers, because the differences between them are so small.

Cumulative Completion A box-plot showing the cumulative completion values for the different scheduling policies is shown in Figure 6.7. Although the plots look fairly similar, the P-SLR orderer is statistically significantly (repeated measures t-test, $p = 0.05$) better than all other orderers except SRTF. This is because cumulative completion is linked to responsiveness. If more tasks finish earlier in the schedule, then the schedule will be more responsive, and the cumulative completion metric will be higher. Because the P-SLR and SRTF metrics are indistinguishable in their responsiveness, it would follow that they are indistinguishable in their cumulative completion.

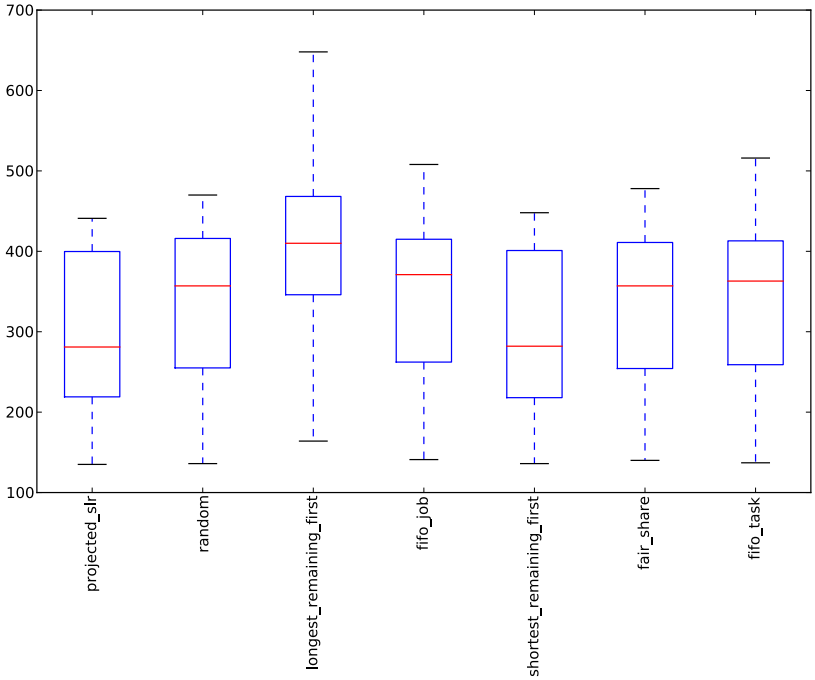
Peak In-Flight The results for the peak in-flight metric are shown in Figure 6.8. Similarly to the results found for the other metrics, P-SLR has a statistically significantly (repeated measures t-test, $p = 0.05$) lower peak in-flight for all other ordering policies except SRTF. This is also to be expected due to the responsiveness findings, because a high level of responsiveness will mean that more tasks are finishing more quickly, and hence there will be fewer in-flight.

Another interesting feature to notice is to consider the peak in-flight count of the LRTF orderer. From the cores per task presented in the workload parameters above (Table 5.2), it can be seen that the average number of cores per task is expected to be just over 10. The median value for peak in-flight jobs given for the LRTF orderer is just over 400. Therefore, at the point in the schedule of peak in-flight, there are more jobs in-flight than there are possible to be servicing at once, given that the platform consists of 4000 cores. This finding reinforces the responsiveness metrics that show the LRTF ordering giving poor responsiveness. LRTF starts a lot of jobs quickly, but takes a long time to finish them, as is shown by the high peak in-flight and the lower responsiveness achieved.



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 6.7: Cumulative Completion by Ordering Policy



Red line at median, box shows interquartile range (IQR), whiskers are at most extreme value within lower/upper quartile -/+ 1.5 IQR [81].

Figure 6.8: Peak In-Flight by Ordering Policy

6.3.2.4 Evaluation Summary

In this evaluation of policies using synthetic workloads, it is shown that the P-SLR ordering policy has significantly improved fairness and responsiveness when compared to the Random, LRTF, Fair Share, FIFO Task and FIFO Job policies. The P-SLR policy produces fairness and responsiveness results that are statistically indistinguishable from the SRTF ordering policy. However, it can be argued that the P-SLR ordering policy is a better choice for a production policy, because it is starvation-free. P-SLR guarantees that all jobs and tasks will eventually run, however large they are. Using SRTF, on the other hand, may lead to the largest jobs starving for resources indefinitely in a system in overload, where the arrival rate of work continually exceeds the ability for the system to service this work.

6.3.3 Scheduler Evaluation (Industrial)

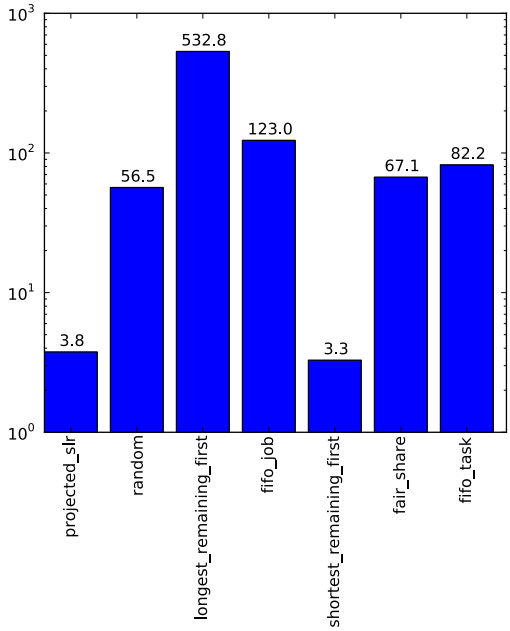
In this section, the performance of the P-SLR ordering policy will be evaluated using a single workload derived from the logs obtained in the industrial case study. The platform used for these experiments reflects the industrial platform in the number, size and connectivity of the clusters, as described in Profile 3 in Section 5.7. Therefore, the results for the FairShare policy shown here reflect the values seen in the production system as the FairShare tree used is the same.

6.3.3.1 Fairness

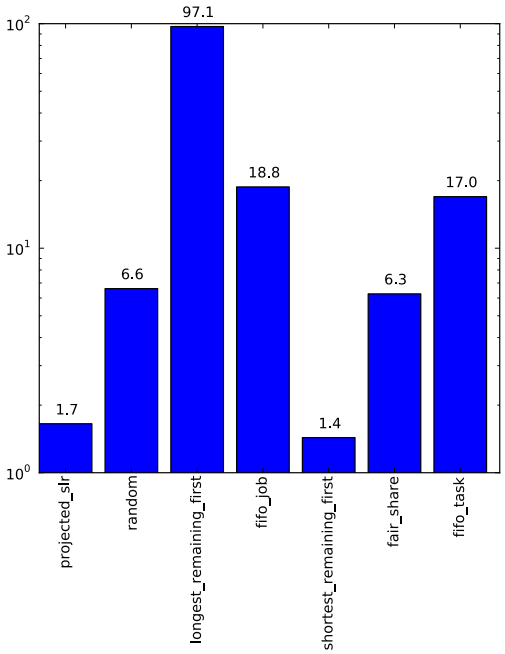
Standard Deviation of SLR The fairness of the schedules produced using the industrial workload are shown in Figure 6.9a, as measured by the standard deviation of their SLR values. It is clear that P-SLR and SRTF show dramatically higher levels of fairness compared to the alternative policies. P-SLR shows a slightly higher standard deviation of SLR, though this difference is small compared to the differences with any of the alternative policies. Furthermore, the benefit of having a guarantee that a schedule will always be starvation-free (given by P-SLR) is likely to outweigh the slight decrease in fairness over SRTF. When also considering SLR across the range of execution times (Figure 6.10), the slightly lower degree of fairness can be explained by the P-SLR policy having a slightly higher average SLR across the whole workload. The average case suffers slightly to guarantee a better worst-case. Because the shorter jobs are more sensitive to an increase in SLR than the large jobs, this would amplify their differences and hence give a higher standard deviation.

What is worth noting is the strong performance, in terms of fairness, of the random policy. It is slightly fairer than the currently used Fair Share policy and much better than the FIFO and LRTF policies. It can be argued that random ordering

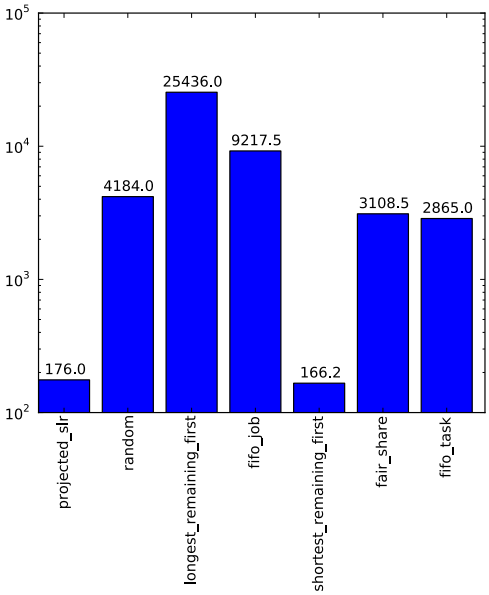
could give fair results, but that were equally poor in responsiveness, although this is not the case for reasons outlined below.



(a) Standard Deviation of SLR



(b) Mean SLR



(c) Worst-Case SLR

Figure 6.9: Functions of metrics over SLR by ordering policy (Industrial Workload)

Mean SLR over decile of execution time The pattern for responsiveness when using the industrial workload parallels the patterns seen using the synthetic workloads. The SRTF policy achieves the highest mean responsiveness, although the results given by P-SLR are closely competitive (Figure 6.9b). Across the deciles of execution time (Figure 6.10), SRTF consistently outperforms P-SLR, albeit slightly. Interestingly, even for the highest deciles, the mean SLR values are equivalent for SRTF and P-SLR. This is likely due to several reasons. Firstly, the largest jobs are so large that even with a pending time of weeks, when their execution times are in the order of months, their SLR value may still be low. Secondly, load balancing between the clusters may direct shorter jobs to alternative clusters when there are large jobs pending on a given cluster, which may mitigate the likelihood of starvation for the large pending jobs. Thirdly, in the industrial scenario, it is likely that simply through natural variation in the submission rates of work, there will be occasions where the clusters are not fully loaded and therefore the longest jobs can start. Even though these occasions may happen only every few months, this will hardly affect the SLR of the largest jobs, that themselves run for a few months.

It could be argued therefore that SRTF is the most appropriate policy for achieving high responsiveness and fairness for most users most of the time. However, the clusters tend to get busier over time, and procuring a new cluster is a lengthy process. This will lead to it becoming ever more likely that the largest jobs will starve. Furthermore, there are genuine organisational needs of the data, and by using P-SLR, the wait time for these largest jobs will be bounded, which is helpful for organisational planning for when the data is ready. Therefore, the guarantee of non-starvation offered by P-SLR is valuable, and the impact of the slightly higher mean SLR across the workload is so small as to be very likely to be acceptable (especially noting the logarithmic scale on the y-axis of Figure 6.10).

As expected, the LRTF policy gives the poorest responsiveness of any policy, because it intentionally penalises the shortest jobs to the advantage of the longer jobs. Both of the FIFO policies suffer for responsiveness because of the wide range in execution times between the shortest and longest jobs. The SLR value of a minutes-long job will naturally be very high if it is waiting in the queue behind a month-long job. The Random policy improves slightly on this, because the shortest jobs do have some chance of getting in before the largest jobs.

Most interesting is that the Fair Share policy seems to be working favourably compared to Random across most of the space of execution times. This suggests that the organisation have crafted their Fair Share table mostly correctly. However, it is poorer than Random for the shortest jobs, which tend to be those whose responsiveness is most highly prized. However, no matter what Fair Share tree is are used, it is not possible for Fair Share to be competitive with P-SLR and SRTF as Fair Share does not take into account any information about execution times.

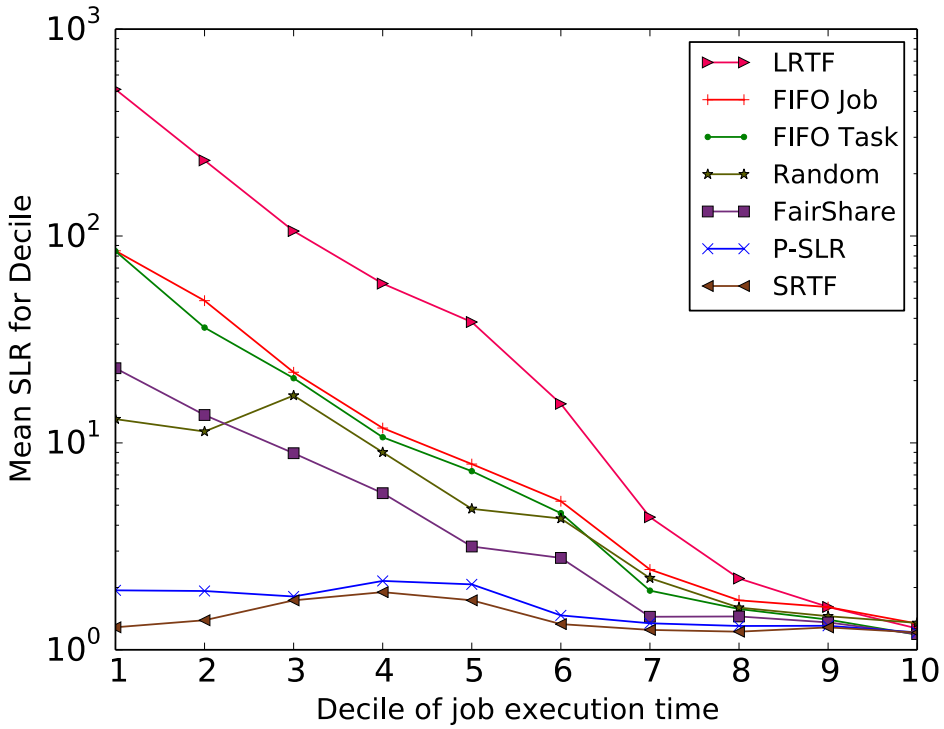


Figure 6.10: Mean SLR for decile of job execution time (Industrial workload)

6.3.3.2 Responsiveness

Worst-Case SLR During the observed period of the industrial logs, there are repeated periods of overload. The worst-case SLR results are a useful measure to distinguish how well the policies were able to keep up with responsiveness even under such overload as experienced in a real system. As can be seen from Figure 6.10, the worst-case SLRs tend to be found for the smallest tasks, as these tasks are the most sensitive to changes in pending time.

In Figure 6.9c for worst-case SLR, once again P-SLR and SRTF show values of comparable magnitude, although SRTF is again slightly ahead for the industrial workload. This is due to its aim in prioritising the shortest jobs that have the most sensitive SLR measurements. LRTF returns the poorest worst-case responsiveness, for much the same reason. FIFO Job does surprisingly poorly, as it is has poorer worst-case responsiveness than Random and FIFO Task. It is to be expected that the worst case for Random would be poor, because of some unlucky short task that has to wait a very long time. In this particular workload, the multiple waits problem does not cause FIFO Task to be poorer than FIFO Job because the workload as obtained from the logs does not contain dependencies.

6.3.3.3 Utilisation

The results for the utilisation metrics were so close as to be unhelpful to display graphically, and so have instead been presented in Table 6.3. The Average Utilisation values were identical because the Workload Makespan values were also identical. This is because of a single long-running job arriving just before the end of the sampling period of jobs, and which kept on running long after everything else in the sample had completed. This is also the reason the average utilisation seems so low - it is not that the clusters were actually that quiet in reality, instead it is because of a significant period where in the simulation, only the last single long-running task is left executing. However, both the Average Utilisation and the Cumulative Completion values demonstrate that P-SLR is able to keep utilisation as high as the alternative policies, even while increasing fairness and responsiveness.

The Peak In-Flight values are not identical but none have a huge degree of difference. SRTF has the lowest peak, which is not surprising because it will tend to get short jobs out of the way quickly. The multiple waits problem is not manifested here because of the absence of dependencies in the logs.

6.3.3.4 Industrial Evaluation Summary

The results from the industrial evaluation corroborate those from the synthetic workloads. When the P-SLR ordering policy is applied to the workload derived from the trace of an industrial HPC, it gives fairness, responsiveness and utilisation results comparable to that of the best alternative policy, SRTF. However, it does this while still providing a starvation-free guarantee. It is seen in Figure 6.10 that P-SLR can achieve responsiveness across the range of execution times, just as SRTF can.

Policy	Average Utilisation	Cumulative Completion	Peak In-Flight
P-SLR	58.64	7.685×10^{18}	489
Random	58.64	7.686×10^{18}	515
LRTF	58.64	7.688×10^{18}	490
FIFO Job	58.64	7.686×10^{18}	543
SRTF	58.64	7.684×10^{18}	439
Fair Share	58.64	7.687×10^{18}	441
FIFO Task	58.64	7.688×10^{18}	407

Table 6.3: Utilisation Metrics (Industrial Workload)

6.4 Evaluation of P-SLR with Networking Delays and Inaccurate Estimates of Execution times

6.4.1 Experimental Hypotheses and Approach for Network Delays and Inaccurate Estimates of Execution Times

The evaluation in this section will seek to investigate two experimental hypotheses.

> Experimental Hypothesis D: Projected-SLR delivers better responsiveness and fairness than schedulers which do not use execution time estimates, even when the estimate inaccuracy is significant. P-SLR is competitive with scheduling policies that do make use of execution time estimates.
>
> Experimental Hypothesis E: Projected-SLR delivers competitive responsiveness and fairness metrics independent of communication to computation ratios.

These hypotheses will be investigated using Profile 2 from Section 5.7 and using the workload mix defined in Table 5.2.

6.4.2 Inaccurate Execution Times

For all the results with inaccurate execution times, the Random, FIFO Task, FIFO Job and FairShare policies are not affected by the inaccurate estimates, because they do not make use of these estimates. Therefore, their results are equal across the spectra of inaccurate estimates.

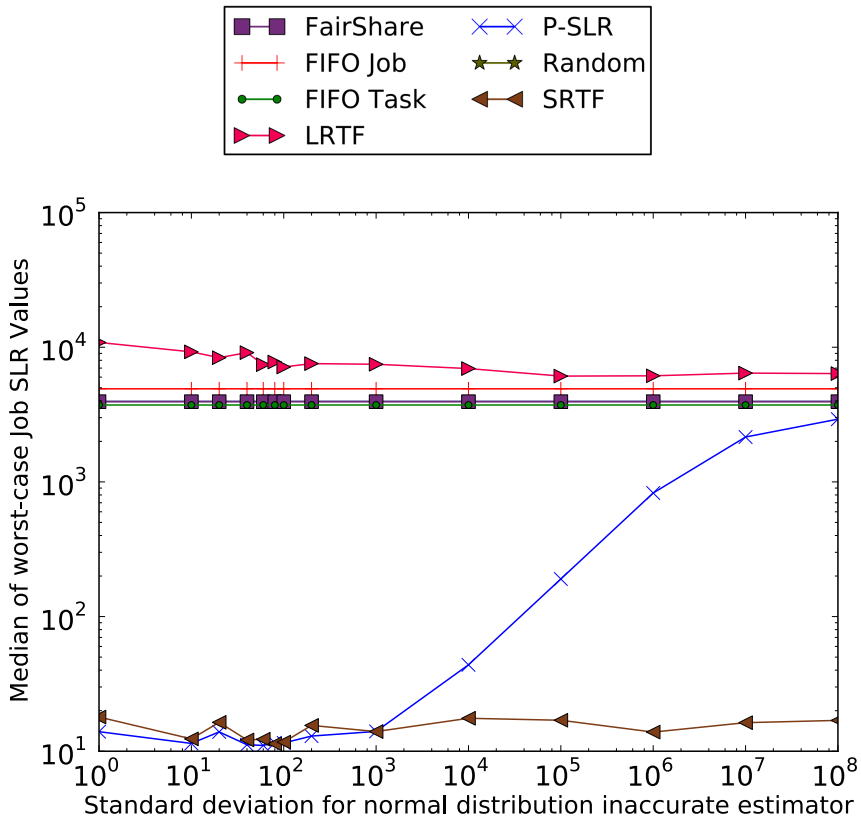
6.4.2.1 Responsiveness

With normally distributed inaccuracies (defined in Section 5.7.1.8, results in Figure 6.11a), the P-SLR policy dominates by having the lowest worst-case SLR values until the standard deviation is 1000% of the value of the exact time. It is reasonable to assume that virtually all real-world estimates will have ranges less than 1000%.

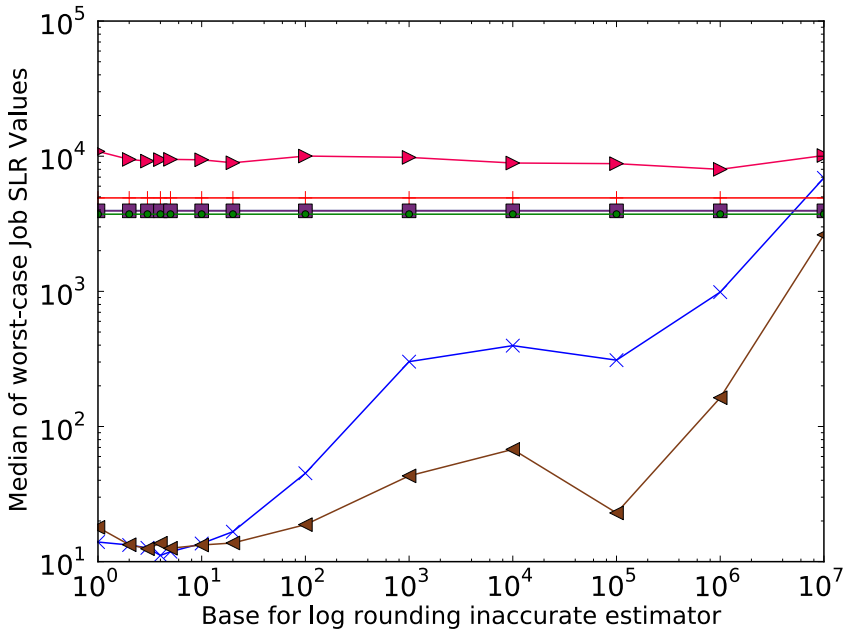
The difference between P-SLR and SRTF in this range is not statistically significant (repeated measures t-test, $p = 0.05$), which shows the strength of the P-SLR policy as it adds the guarantee of non-starvation. The divergence after 1000% is due to this guarantee because SRTF is letting the largest tasks starve. The largest tasks have SLRs which are least sensitive to waiting time, keeping the worst-case SLR fairly low.

Once the estimation error gets sufficiently large, the estimates become effectively random. Therefore, the worst-case SLR of the P-SLR orderer rises to similar levels as the schedulers that do not make use of execution time estimates.

Similar results are apparent where estimates are log-rounded (defined in Section 5.7.1.8, results in Figure 6.11b). Where execution times are rounded to the nearest



(a) Normally distributed inaccuracies



(b) Log rounding inaccuracies

Figure 6.11: Responsiveness

power of 10 or below, P-SLR dominates the worst-case SLR values, although it is not statistically distinguishable from SRTF (repeated measures t-test, $p = 0.05$). Still, it is to be expected that users could give a good indication of their job taking closer to 1, 10, 100, etc., minutes.

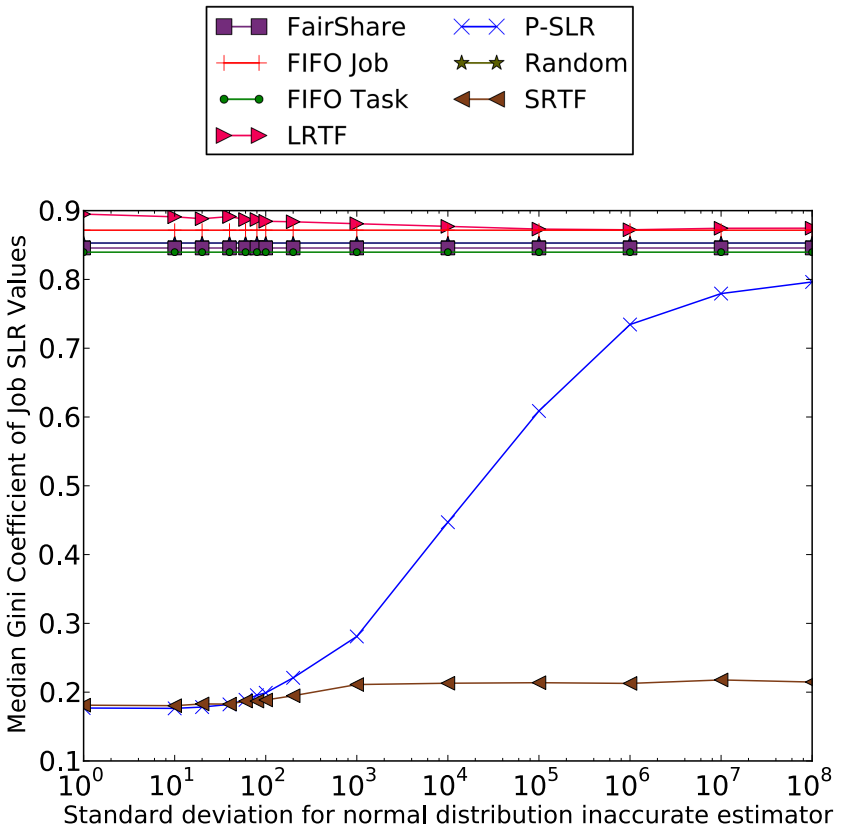
As the estimates get yet more coarse above a base of 10, SRTF provides better worst-case responsiveness than P-SLR. This is to be expected, because inaccurate estimates move the behaviour of schedulers closer to Class 1 behaviour. As P-SLR with accurate estimates exhibits Class 3 behaviour, any perturbation to this will make it tend towards Class 1 behaviour. Whereas for SRTF, because it shows Class 2 behaviour, perturbations will initially make its behaviour more like Class 3, although eventually it too will exhibit Class 1.

The LRTF orderer, as expected, shows poorer worst-case responsiveness than any of the policies that do not consider execution time. This is because it makes the smallest tasks starve, and these tasks are the ones whose SLR is most sensitive to waiting time. LRTF is useful, though, because it gives an upper bound on how poor responsiveness can get because it shows the most extreme Class 1 style behaviour.

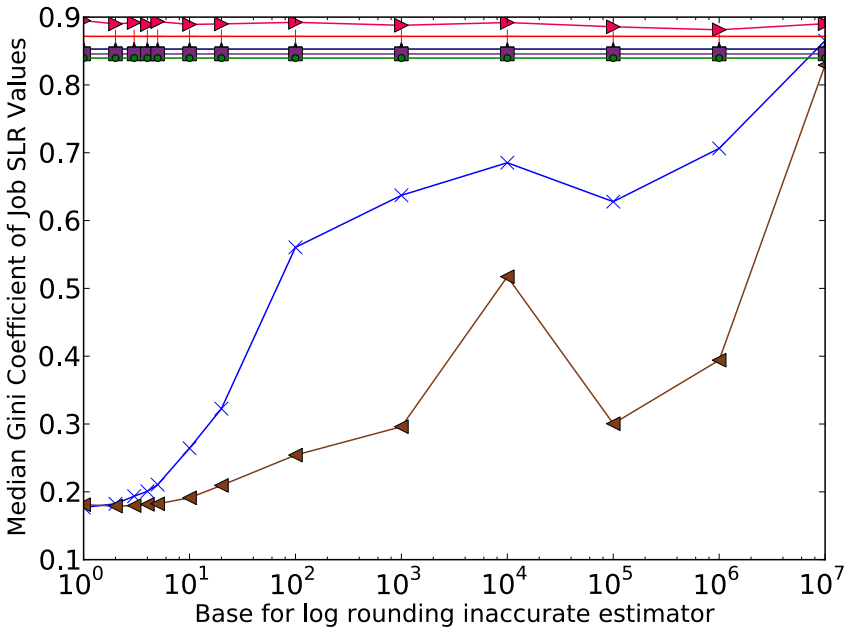
These results show that up to a threshold value of 10^3 for log-rounding M or normal standard deviation μ, the P-SLR and SRTF policies have statistically insignificant (repeated measures t-test, $p = 0.05$) differences in responsiveness. Responsiveness for P-SLR approaches that of the schedulers that do not include execution time estimates when for either normal standard deviation percentage and log rounding the error is around 10^7. These values are far above the maximum levels of inaccuracy of around 100% found by Bailey Lee et. al. [107]. This would suggest that in reality, the P-SLR scheduler could be considered most favourable for practical scheduling, because it gives a guarantee of non-starvation, unlike SRTF, and leads to an improvement in responsiveness performance over that of schedulers that do not consider execution time estimates.

6.4.2.2 Fairness

As with the results for responsiveness, the fairness results for normally distributed error (Figure 6.12a) are dominated by P-SLR at the lowest values, although they are also statistically indistinguishable (repeated measures t-test, $p = 0.05$) from SRTF up to a threshold of $\mu = 100\%$. This is to be expected, as P-SLR is designed to show Class 3 behaviour, which emphasises fairness. Above this threshold, P-SLR exhibits progressively more Class 1-like behaviour, with the smallest jobs suffering most as their execution time estimates begin to overlap with larger jobs. SRTF causes the largest jobs to starve, but because their SLRs are less sensitive to waiting time, the SLR distribution remains closer to Class-3.



(a) Normally distributed inaccuracies



(b) Log rounding inaccuracies

Figure 6.12: Fairness

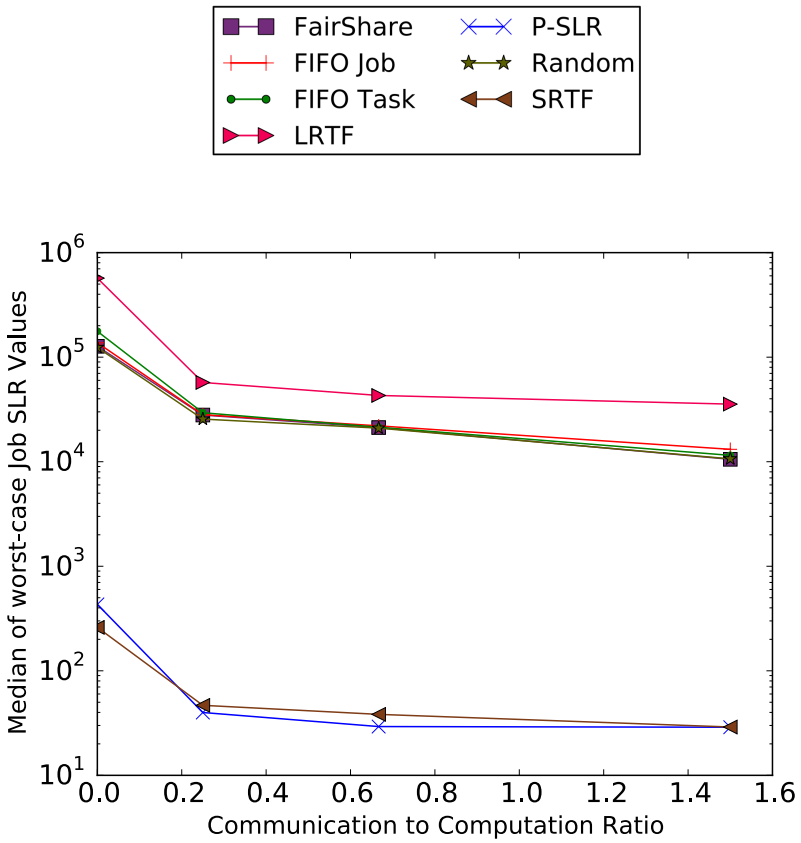
The normally-distributed inaccurate estimator is not able to introduce sufficient error below a standard deviation percentage of 10^8 to cause significant impact on the fairness of the SRTF policy. If the estimation errors are normally distributed, therefore, SRTF may provide better fairness than P-SLR when the standard deviation of the errors is above 100% of the exact time.

With the log rounding estimator (Figure 6.12b), other than the case where there is no inaccuracy, the SRTF orderer is statistically significantly more fair, according to the Gini Coefficients, than for P-SLR. As before, this is due to the SRTF causing the largest jobs to starve, but this not having a large effect on those jobs' SLR values. P-SLR immediately starts to exhibit Class 3 behaviour in the presence of inaccurate estimates, whereas SRTF moves from Class 2, then to Class 3, before eventually showing Class 1 at a rounding power of $M = 10^7$.

The LRTF policy shows the worst-case bound on unfairness, as it is the most extreme example of Class 1 behaviour. The bound on how unfair it makes things improve as estimates get worse, because it is not as able to achieve the worst case.

The fairness results show that for small inaccuracies in execution time estimates, P-SLR and SRTF show similar results. However, for larger inaccuracies, SRTF gives fairer results as it shows more of a Class 3 behaviour profile. However, this is once again due to the largest jobs being starved of resources, and hence a tradeoff must be made between higher fairness in the presence of inaccuracies, as provided by SRTF, or a guarantee of non-starvation, as provided by P-SLR.

Hypothesis D stated that P-SLR would deliver better responsiveness and fairness than schedulers that do not use execution times, even when the estimate accuracy is significant. This has been shown to be the case (Figures 6.12a and 6.12b), with better responsiveness and fairness when the standard deviation inaccuracy percentage μ is less than 10^7 and when the log rounding base M is less than 10^8, all extremely high levels of inaccuracy. P-SLR has been shown to be competitive with SRTF in responsiveness up to a threshold inaccuracy of 10 times the value of the original estimate. In fairness, P-SLR is competitive at small inaccuracies, but SRTF dominates above this, refuting a part of Hypothesis D. It is then a tradeoff for a grid owner to decide whether, if estimates of execution time have large inaccuracies, absolute fairness (SRTF) or an absence of starvation (P-SLR) is more important. These results are likely to be highly relevant to the industrial partner, as by implementing one of these policies they would be able to dramatically improve the responsiveness of the shorter jobs that are run without having a negative impact on the responsiveness of the larger jobs. This is the case even when the only execution time estimates available are those given by users, which can be fairly inaccurate.



(a) Responsiveness

(b) Fairness

Figure 6.13: Network Delays

6.4.3 Networking Delays

6.4.3.1 Responsiveness

A pronounced feature (Figure 6.13a) is that there is an increase in worst-case responsiveness when network costs become present. This is because the CPU resources are no longer the single bottleneck and network costs come into play.

Throughout the range of network delays examined, P-SLR and SRTF showed similar levels of responsiveness. SRTF is slightly better when there were no network delays, but P-SLR is slightly better when there were delays present. However, P-SLR and SRTF were not statistically significantly different (repeated measures t-test, $p = 0.05$).

The LRTF policy again shows the worst case bound of responsiveness because it tends to starve the smallest tasks.

6.4.3.2 Fairness

The results in Figure 6.13b also show greater fairness in the presence of network delays, because of the improvements in overall responsiveness. However, in this case, P-SLR is statistically significantly (repeated measures t-test, $p = 0.05$) more fair than SRTF throughout the range of CCR. This is because although their worst case values are similar (Figure 6.13a), P-SLR shows more Class 3 behaviour, giving a better balance of SLR values overall. All the schedulers other than SRTF are very unfair across the space of network delays when compared to P-SLR.

Hypothesis E considered whether P-SLR delivers competitive responsiveness and fairness across the range of communication to computation ratios. P-SLR dominated all schedulers other than SRTF in responsiveness, although it is statistically indistinguishable from SRTF. In fairness, it dominated all other schedulers. Therefore, it is considered that Hypothesis E has been demonstrated to be true. Across the space of networking delays examined, P-SLR provides equal or better responsiveness and better fairness than the best alternative scheduler, SRTF, but does so while in addition providing a guarantee that no job will ever starve.

6.5 Summary

6.5.1 Summary of Results

An evaluation is made of the ordering part of list scheduling policies designed to run at the cluster level within a hierarchical grid scheduling scheme. The Projected-Schedule Length Ratio (P-SLR) policy is then developed with the aim of achieving high responsiveness and fairness even under periods of overload, without significantly impacting the cluster utilisation.

Evaluation of these scheduling policies is then performed in simulation. Firstly, using synthetic heterogeneous workloads with a logarithmic distribution of execution times and a selection of dependency patterns. This evaluation took place using a simulated grid comprising a number of heterogeneous clusters with networking delays between them. Secondly, by using a workload extracted from a trace of the industrial grid running over a simulated platform designed to reflect the configuration of the production grid.

The P-SLR scheduler is found to give more responsive and fairer schedules than the Random, Longest Remaining Time First, Fair Share, FIFO Task and FIFO Job ordering policies; without having a major impact on utilisation. The P-SLR orderer achieves responsiveness and fairness performance that is statistically indistinguishable from the Shortest Remaining Time First ordering policy, even though P-SLR is guaranteed to be starvation-free, while SRTF is not.

The Average, Worst-Case and Standard Deviation of Schedule Length Ratio metrics are shown to provide a suitable level of insight for evaluating quality of service from a users' perspective. This is because these capture the users' concerns about responsiveness and fairness while taking into account the structure of dependencies for workloads that contain them. It is proposed that the Projected-SLR policy is a suitable candidate for production use as a scheduler for HPC systems, due to its ability to achieve good responsiveness, fairness and utilisation and to degrade gracefully under periods of overload.

The responsiveness and fairness performance of the P-SLR scheduler is found to be robust to network delays. P-SLR provides equal or better responsiveness (measured by worst-case SLR) and better fairness (measured by the Gini Coefficient of SLRs) in the presence of network delays than the best alternative scheduler, SRTF, but does so while in addition providing a guarantee that no job will ever starve.

The responsiveness performance of P-SLR is found to be robust below a certain threshold of execution time inaccuracy. This threshold is 10 times the original execution time of the task. Above this threshold, SRTF is able to provide better responsiveness. P-SLR is not able to give the best fairness compared to SRTF once significant estimation inaccuracies were present. This shows a strength of the SRTF policy, because SRTF is better at keeping SLRs low for small tasks whose SLRs are more sensitive to longer waiting times when inaccurate estimates are substantial. However, P-SLR still dominated all other alternative policies, showing that where estimates of execution time are available, it can make good use of them, even where the inaccuracies are large.

The simulator used was designed to be sufficiently performant to run the industrial-scale workloads in a reasonable amount of time. Using an Intel Core i7-860 processor each individual simulation case took about 2-5 minutes to run in 2

GB RAM. The full-scale industrial workload took about 8-10 minutes to run using 8-9 GB RAM. These results are for execution on a single core.

6.5.2 Possible Extensions and Applications of P-SLR

The maximum P-SLR of tasks in the queue is likely to be a useful metric for dynamic queue monitoring. P-SLR was originally designed with the idea that it would be compatible with an admission controller. An admission controller could monitor this maximum P-SLR, and cease to admit tasks to the grid if it became too high. This could be done either at a global level, or limited to specific users.

Alternatively, the maximum P-SLR at a given moment could be a useful tool for load balancing, where tasks are assigned to the cluster with the lowest maximum. This may help ensure that responsiveness as well as load is evenly distributed between clusters in a grid. Evaluating admission and allocation policies that are enabled by the calculation and monitoring of P-SLR would be a natural course of future work.

This maximum value of P-SLR could also be used in grid systems that support expansion into the cloud under situations of acute overload. If the maximum, mean or median P-SLR of tasks in the queue passed some threshold value, it may indicate that the grid is overloaded and more resources should be acquired to keep responsiveness to acceptable levels. Once demand fell, these cloud resources could be released if the observed P-SLR fell below a certain threshold again. By setting the thresholds appropriately, desired responsiveness could be preserved while minimising the use of cloud computing, which naturally adds cost over and above the basic cost of running a grid.

The use of P-SLR as a threshold value could also be used in computing cluster environments where energy is a concern. If P-SLR fell below a threshold value, idle machines could be shut down in order to save power, while keeping responsiveness to acceptable levels.

Future work based on P-SLR could be to introduce a weighting factor to better handle situations where, for example, small jobs need even higher responsiveness than large ones relative to their execution time. P-SLR values could also be weighted using user or group information, to intentionally prioritise the work of some users over others.

The usefulness of P-SLR is not just for grid systems, as it is a general policy that could be applied to any system where there is a queue of work to be performed. In the simplest case where all tasks were the same length and there were no dependencies, then it would operate exactly like First In First Out (*FIFO*). In fact, for tasks of the same execution time, they will be treated in FIFO order. The intuition behind P-SLR

is that tasks of different execution times should be treated differently, and small tasks that require low latency should be treated accordingly.

For this reason, P-SLR may also be suited to scheduling data flows in network routers that require QoS, where small data packages that require low latency could be prioritised more effectively relative to large transfers than if FIFO were used. Nevertheless, because the policy is starvation-free, all flows would make progress towards completion. Where data flows are broken up into non-pre-emptive packets, each packet could be represented by a task with a dependency on the previous packet. This would allow the upward rank of each packet to be represented by the amount of data left in the flow.

Scheduling problems outside the domain of computing may also benefit from using the P-SLR policy. In operations management, overruns are a fact of life. In conjunction with the Critical Path Method (*CPM*) [94], monitoring the values of P-SLR could help project managers prioritise tasks between different projects, in order to minimise average lateness of project completion. P-SLR could also help estimate how late projects may be, or for tasks with a P-SLR of less than 1, how much slack these tasks have.

In a pre-emptive system, P-SLR would still be applicable, although the calculation of the P-SLRs for the task queue would need to be performed for each scheduling quantum or some other fixed interval rather than at task arrival or completion. Tasks could only pre-empt already running ones once their P-SLR is greater than that already executing. In a system such as this, some measure of hysteresis may be desirable to prevent thrashing.

Using P-SLR could also be useful in systems that have interactive features that require low latency and low computation coupled with long-running compute-intensive jobs. Using P-SLR would allow these priorities to be determined dynamically rather than having to manually prioritise different processes.

Chapter 7

Scheduling using Value

7.1 Background

As detailed in Chapter 2, users run workflows to support their work. They require responsiveness in these workflows otherwise their productivity suffers. The previous chapter assumes all tasks simply require the best responsiveness possible, subject to a fair allocation between users. The previous chapter also considered the starvation of tasks to be the extreme end of unfairness.

This is fine for a system that is underloaded or where there are transient overloads. The definition of transient is important, however. If the 'transient' overloads last significantly longer than the time the shortest tasks execute for, it may well be that there is no way to satisfy the requirement that all tasks must eventually run within a reasonable amount of time [24]. Therefore, the situation will arise where it is preferable to follow the principle of "survival of the fittest"; to let some jobs starve in order that the majority are able to run to completion.

This idea informs the notion of jobs having 'value' to users. In realistic systems, there will be some jobs that are genuinely more important than others. For instance, the work some users do may simply be more valuable to the organisation. Alternatively, at a given moment in time, the timely completion of a given user's work is on the critical path of a project, and must be given priority over other tasks not on the critical path.

The approach of the previous chapter assumes that eventually, all work submitted must be executed. Intentionally, the P-SLR scheduler is designed to be compatible with an admission controller. Research into admission control for grid scheduling is an active field of research [24, 68, 78, 170]. An admission controller would most likely be configured with a threshold [78] to cease the admission of new work to the queue if the SLR of jobs in the queue rose above this threshold.

However, even with an admission controller, the future state of the system can never be known precisely. There may be situations in which users wish to submit

work speculatively, to use slack capacity overnight if available, for example. However, the results of such a job would not be critical to the ongoing work of the user or their project. The value of such a job would be qualitatively less than ones that are essential for user progress or project completion.

Under periods of high load, the P-SLR scheduler endeavours to keep the SLRs of all tasks rising evenly, as discussed in Chapter 6.3. For the largest tasks, this may mean having to wait a long time measured in working timescales, even if it is short in proportion to its execution time. While on average this may be desirable, there may be situations where there are long-running jobs that are urgent. Alternatively, there may be more speculative jobs that only run for a short time, but could wait until a night or a weekend to be run, as they are less urgent.

Therefore, it is desirable to include the ability for a scheduler to intentionally starve some jobs under periods of overload. Every job submitted will have a length of time after which even if results are produced, they are irrelevant to the user. This is especially the case in this industrial context where CFD results must be returned more quickly than real wind tunnel tests, otherwise the users may as well just use the wind tunnel instead.

This chapter will describe an approach to satisfy these kinds of concerns. In order to do this, there needs to be a mechanism to indicate to the scheduler the value and responsiveness requirements of jobs. Determining the allocation of value and value budgets to users and to jobs is, in general, a stakeholder issue and should be left for the managers in a given organisation to decide. This is because value depends on organisational priorities which naturally change and develop over time. Ascertaining and agreeing on these can be difficult, which is why the fairness of the P-SLR scheduler overall may be attractive.

If the value of work can be indicated to the scheduler, the scheduler will be better equipped to deal with the kinds of tradeoffs required during periods of overload. The scheduler may then be able to prioritise and ensure responsiveness for urgent work under periods of overload without the need for an additional admission controller. With an appropriate model of value, the scheduler could even consider how the value of completed jobs changes with time, and use this to inform the decision on what must be run immediately, and what can wait until a quieter period, such as over lunchtime, overnight or at a weekend.

7.1.1 FairShare and Urgency

The existing industrial FairShare set up is configured to give higher priority to more urgent jobs, by giving users with urgent tasks more ‘share’ of the cluster with which to run their work. However, this confuses urgency requirements, which are a measure of time, with share requirements, a measure of space. Giving a user a higher share

can sometimes have the effect of giving them higher priority. However, this does not always occur. A user may also wish to submit two jobs at the same time, identical other than in their urgency requirements. If they each would consume the whole of the user's FairShare, then the FairShare scheduler might run the less urgent one first, because it has no way of distinguishing between them. Due to the non-pre-emptive nature of the industrial system, the urgent task may then never run until its results are worthless, having had to queue behind the less urgent task.

It could be argued that the user should just be given a greater share to just run their jobs in parallel, but this will naturally penalise other users of the cluster. Adjusting priorities like this based on perceived urgency requires frequent, manual adjustment of the share tree. This is the industrial status quo, but it incurs considerable maintenance overhead. Therefore, a means is needed of defining the value of a job to the scheduler, because the urgency and the execution time of jobs may not always be proportionally-related [168] as is assumed by the P-SLR scheduler.

7.1.2 Work Related to Value Scheduling

As the value of the results of a job to a user may change over time, this leads to the concept of value curves. Lai [101] showed that using value curves instead of fixed values for tasks gives greater market efficiency in the long run. A value curve is a function of the value of the job to the user depending on the completion time of the job [37, 86, 90].

Irwin et. al. [86] consider a model whereby the values of tasks decays linearly with waiting time. Jensen et. al [90] propose a model where the decrease in value has a linear followed by an exponential phase. Alternatively, value curves have been proposed that can rise and fall, as in real-time systems then early completion of work can be as bad as late completion [30, 37]. In the industrial scenario earlier completion is always valued, so any value curves can be assumed to be non-increasing. However, they may not simply be linear or exponential. This is because there may not be any difference in value between two times in the middle of the night when the users are not at work. Instead, a richer model of an arbitrary yet non-increasing function is required to fully capture user requirements along with the impact of working hours.

Once value curves have been defined, it is the job of the scheduler to seek to maximise the returned value over the whole workload [37, 110]. It is also the job of the scheduler to starve tasks that are too late or too low-value to be useful during any periods that the system is under overload [37]. Value curves enable users to submit low-value jobs speculatively. This is because if there is spare capacity, it can be used by these low-value jobs, but if not, these jobs will expire. With these

low-value jobs having expired, they will not consume capacity later, potentially during a busier period when capacity is at an even higher premium.

The nature of scheduling by using value naturally lends itself to economic or market-based scheduling architectures [101]. However, as discussed in Chapter 3.2.6, these are unsuitable for the industrial grid context because a change in scheduling architecture would be required as well as a change in scheduling policy. Markets also need to be carefully tuned as desirable scheduling decisions are the result of emergent effects, rather than the direct action of a heuristic [24].

7.1.3 Chapter Structure

This chapter considers approaches designed to maximise value that are based on a list scheduling architecture. Section 7.2 will deal with creating a model of value suitable for capturing industrial concerns while also amenable to running in simulation. Section 7.3 will discuss the best ways of measuring value. Section 7.4 will define several existing list scheduling policies and propose a novel approach termed Projected Value Remaining that could be applied to the problem. Section 7.6 will present the findings of the evaluation.

7.2 Model of Value

The notion of value in this chapter is assumed to be derived from a pseudo-economic model. There is time pressure bearing on computational results at two levels. The higher level pressure comes from the whole design process of an aircraft. Aircraft design balances a large number of tradeoffs with no completely perfect solution. If the designers can explore more of the huge design space, they are more likely to a find a design that achieves the precise balance in the tradeoffs desired. As tiny improvements in aerodynamic performance can save large amounts of fuel and hence money for airlines, there is intense interest in finding the best solution possible. Yet aircraft design is also subject to the pressures of the marketplace, where speed to market is also very important. A good design is a critical part of a new aircraft and has real tangible financial benefits to the manufacturer. These high-level measurements of value can inform the assignment of how much value the design department has to play with, and can be subdivided appropriately between the different teams.

At the lower level, designers produce their designs through iterative refinements in their day-to-day work. Computational simulation is an essential part of this iterative process. Aircraft designers are highly skilled professionals and hence are also well-compensated. Their time is valuable and must be put to the best use possible by the organisation. It is very costly to prevent a designer from progressing

in their work by keeping them waiting for their simulation results, at least during working hours.

The value curves considered in this chapter are meant to appropriately capture this economic stakeholder context. Using an economic model allows different stakeholder perspectives to be encoded into the same numeric scale. For example, the rate of curve decrease may be very different inside and outside working hours. Alternatively, different users may have different total values assigned to their jobs depending on whether they are an intern or the head of department, for example.

7.2.1 Value Curve Definition

This section will describe a way of defining value functions that change with responsiveness. Different users and groups require very different responsiveness characteristics from their work, as discussed in Chapter 2. Yet individual users are likely to have classes of similar jobs that all need similar responsiveness.

Therefore, the model of value considered in this chapter assumes that value curves are independent of jobs, and represent a particular profile of desired responsiveness. These curves are defined as a sequence of points and are applied to a job at a moment in time in order to calculate its value. These curves then need to be appropriately scaled when applied to each job to reflect differences in jobs' size and value.

Figure 7.1 shows the template used to define value curves. Every job is assigned a maximum value $V_{\texttt{max}}$ that it can return to the user. A value curve is defined as a piecewise function with three subdomains, punctuated by an initial ($D_{\texttt{initial}}$) and a final ($D_{\texttt{final}}$) deadline, as defined in Algorithm 7.1. Before the initial deadline, the value returned is always the maximum $V_{\texttt{max}}$. Between the initial and final deadlines, the value is calculated using a sequence of points which reach 0 at $D_{\texttt{final}}$. Once the final deadline has been reached, the value either remains 0 or a negative penalty of $V_{\texttt{max}}$ is applied to reflect the loss in user productivity.

One of the key requirements in the industrial scenario is responsiveness. Finishing work earlier will always be of the same or higher value to users than finishing work later. Therefore, unlike some models of value, this work will assume that the value curve between the initial and final deadline is non-increasing. The points that define

Algorithm 7.1 Value Calculation

$\texttt{Value}\left(J^k, \texttt{SLR}\right) =$

$$\begin{cases} J^k_{\texttt{VMax}} & \texttt{if SLR} \leq D_{\texttt{initial}} \\ 0 \; or \; -J^k_{\texttt{VMax}} & \texttt{if SLR} \geq D_{\texttt{final}} \\ J^k_{\texttt{VMax}} \times \texttt{factor}\left(J^k_{C^i}, \texttt{SLR}\right) & \texttt{if } D_{\texttt{initial}} < \texttt{SLR} < D_{\texttt{final}} \end{cases}$$

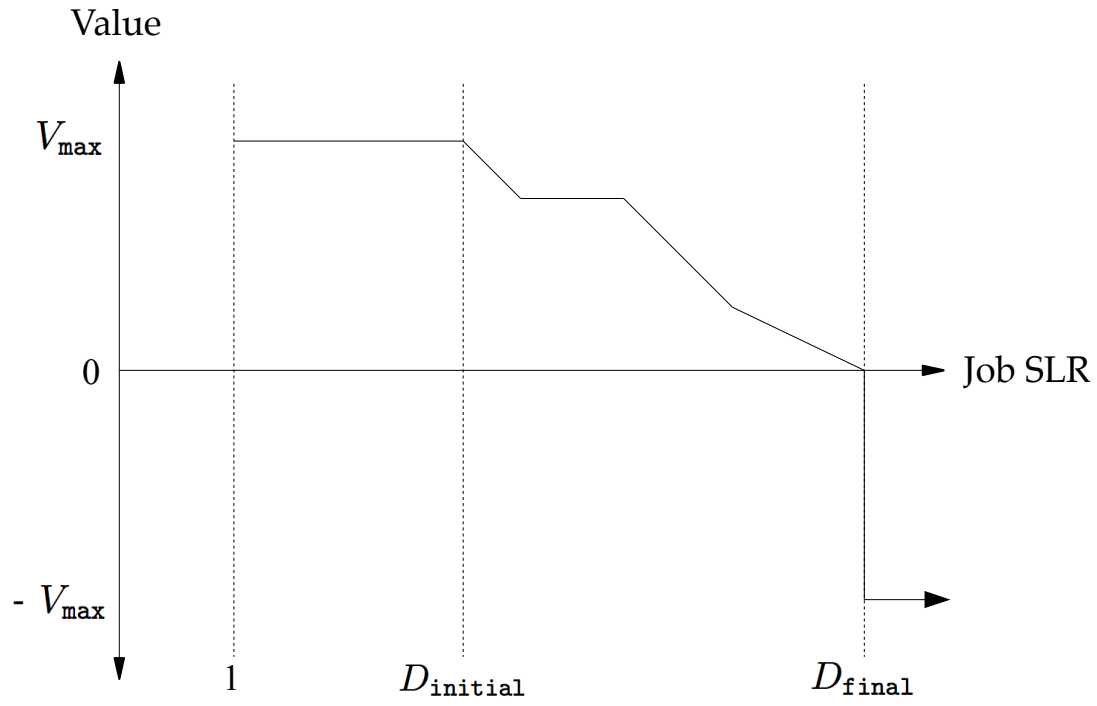


Figure 7.1: Value Curve Template

the value curve between $D_{\texttt{initial}}$ and $D_{\texttt{final}}$ are given values between 1 and 0 so that they can be scaled by $V_{\texttt{max}}$ when applied to a job. Where the SLR of the job falls between $D_{\texttt{initial}}$ and $D_{\texttt{final}}$, a fractional value dependent on the curve must be calculated. This is performed using linear interpolation between the defined points of the curve. The algorithm for calculating this factor is given in Algorithm 7.2.

The time-axis of the value curve is defined using the SLR of the job. This means that the initial and final deadlines along with the time coordinates of points on the curve are defined in terms of SLR. This is so that the value curve can easily be scaled to jobs of different sizes or lengths of critical path. The value curve is undefined before an SLR of 1, as no job can finish before the length of its critical path.

Algorithm 7.2 Value Factor Calculation

$\texttt{factor}\left(C^i, \texttt{SLR}\right)$:
  $\texttt{t\_sort} = \left[\texttt{p}\left[0\right] \texttt{ for p in } C^i\right]$
  $\texttt{v\_sort} = \left[\texttt{p}\left[1\right] \texttt{ for p in } C^i\right]$
  $\texttt{low\_index} = \left|\left[\texttt{t for t in t\_sort if t} \leq \texttt{SLR}\right]\right|$
  $\texttt{t\_low} = \texttt{t\_sort}\left[\texttt{low\_index}\right]$
  $\texttt{t\_high} = \texttt{t\_sort}\left[\texttt{low\_index} + 1\right]$
  $\texttt{v\_low} = \texttt{v\_sort}\left[\texttt{low\_index}\right]$
  $\texttt{v\_high} = \texttt{v\_sort}\left[\texttt{low\_index} + 1\right]$
  $F = \texttt{v\_low} + \left(\frac{\texttt{SLR} - \texttt{t\_low}}{\texttt{t\_high} - \texttt{t\_low}} \times \left(\texttt{v\_high} - \texttt{v\_low}\right)\right)$
  return F

Under significant overload, some tasks must starve. Using the model of value in this section, jobs can be considered to have starved if they have not completed by their final deadline. Two approaches are considered for calculating value for starved jobs. In one approach, the job's value is simply reduced to zero. The loss of value by not running a job is the penalty. Yet not finishing work by its final deadline is likely to be highly inconvenient to users and this can be reflected by applying a negative penalty of $V_{\texttt{max}}$. The advantage of the second approach is that jobs of high value give a greater penalty than those of low value. This reflects the logical idea that not completing jobs of high value causes greater inconvenience to users. Including this as part of the value calculation means that greater insight can be had into which schedulers minimise the number of jobs that do not complete as well as maximising returning the highest value overall.

As a list scheduling architecture is considered, it is important to remove starved tasks from the queue so that the queue does not fill up with work that can never return any more value. Therefore, a slight extension to the list scheduling model is proposed, where jobs that have passed their final deadline $D_{\texttt{final}}$ (or "timed out") will be removed from the queue. As a non-pre-emptive system model is considered, any tasks of the job that are running at the moment the job times out will be left to run to completion. If these tasks are the only ones that the job needs to complete, then the job will still complete. However, all tasks of the starved job that have not yet started executing will be removed from the queue.

7.2.2 Value Curve Generation

Value curves are not currently used by the industrial partner. Therefore, a range of synthetic value curves must be generated in order to apply them to the synthetic workloads from Section 5.4 used for evaluation.

The first step in generating a value curve is to supply outer bounds on the values that $D_{\texttt{initial}}$ and $D_{\texttt{final}}$ can take. These outer bounds are termed $C_{\texttt{lower}}$ and $C_{\texttt{upper}}$. To generate the two deadlines, two samples are randomly drawn from a uniform distribution in the range $[C_{\texttt{lower}}, C_{\texttt{upper}}]$. The smaller value becomes $D_{\texttt{initial}}$ and the larger value becomes $D_{\texttt{final}}$.

The next step is to decide on the number of points that will make up the intermediate curve. An upper bound on the number of intermediate points is termed $C_{\texttt{points}}$. The number of points used to generate a particular curve N is selected from a uniform distribution between one and $C_{\texttt{points}}$.

With the deadlines and the number of intermediate points fixed, the coordinates of the points on the intermediate curve can be generated. The set of x-values (SLR) are drawn from a uniform distribution in the range $(D_{\texttt{initial}}, D_{\texttt{final}})$ The set of y-values (value) are drawn from a uniform distribution in the range $(0.0, 1.0)$. In order

to ensure a non-increasing sequence of values with SLR, the smallest item in the x-value set is paired with the largest item in the y-value set to create a coordinate until no values remain.

Pseudocode to describe this process of generating value curves is given in Algorithm 7.3.

7.2.3 Synthetic Curve Parameters

In order to use the value curves for simulation, synthetic ones must be applied to a workload. In this evaluation, one thousand value curves were generated using the algorithm given in Section 7.3.

The values that bound the range of the initial and final deadlines $C_{\texttt{lower}}$ and $C_{\texttt{upper}}$ are given the values of one and ten, respectively. The lower bound of one is chosen to reflect the fact that some tasks may genuinely be so urgent that the value they deliver starts decreasing as soon as they could finish. The upper bound of ten is chosen bearing in mind the classes of jobs discussed in Chapter 5, where estimates of execution times could be reliably made within an order of magnitude. If a job had the responsiveness characteristics of a higher order of magnitude (SLR > 10), then this would be detrimental to the principle of proportional fairness. Furthermore, users suggested that once a job was taking more than an order of magnitude longer than expected, then its results would no longer be of value.

A value of twenty is used to define the upper bound of the number of points on the curve, $C_{\texttt{points\_max}}$. This value is chosen to provide an appropriate balance between the variety of curves possible and the time taken to perform calculations on those curves.

In a production implementation of a value system, the users would need to specify appropriate values for $J^k_{\texttt{Vmax}}$. As user generated values are unavailable in simulation,

Algorithm 7.3 Value Curve Generation

$\texttt{generate\_curve}\,(C_{\texttt{lower}}, C_{\texttt{upper}}, C_{\texttt{points}})$:
  $\texttt{deadlines} = \texttt{random.uniform}\,(\texttt{min} = D_{\texttt{lower}}, \texttt{max} = D_{\texttt{upper}}, \texttt{samples} = 2)$
  $D_{\texttt{initial}} = \texttt{min}(\texttt{deadlines})$
  $D_{\texttt{final}} = \texttt{max}(\texttt{deadlines})$
  $N = \texttt{int}\,(\texttt{random.uniform}\,(\texttt{min} = 1, \texttt{max} = C_{\texttt{points\_max}}))$
  $\texttt{time\_points} = \texttt{random.uniform}\,(\texttt{min} = D_{\texttt{initial}}, \texttt{max} = D_{\texttt{final}}, \texttt{samples} = N)$
  $\texttt{value\_points} = \texttt{random.uniform}\,(\texttt{min} = 0, \texttt{max} = 1, \texttt{samples} = N)$
  $\texttt{t\_sort} = \texttt{sort}\,(\texttt{time\_points}, \texttt{increasing})$
  $\texttt{v\_sort} = \texttt{sort}\,(\texttt{value\_points}, \texttt{decreasing})$
  $\texttt{inter\_points} = [(\texttt{t\_sort}[\texttt{i}], \texttt{v\_sort}[\texttt{i}])\ \texttt{for i in}\ 1..N]$
  $C^i = (D_{\texttt{initial}}, 1) + \texttt{inter\_points} + (D_{\texttt{final}}, 0)$
  $C^i_{D_{\texttt{initial}}} = D_{\texttt{initial}}$
  $C^i_{D_{\texttt{final}}} = D_{\texttt{final}}$
  $\texttt{return}\ C^i$

for the purposes of this evaluation, $J^k_{\texttt{Vmax}}$ is set to $J^k_{\texttt{exec}}$. This makes the assumption that jobs that take more compute time are also those that are more valuable. It is worth remembering the the shape of the curve is what defines the urgency of jobs, though.

All the jobs in a given workload are assigned a value curve randomly from one of these thousand synthetic curves. This assignment of value curves to jobs is fixed for all runs of a workload, so an appropriate comparison could be made between runs.

7.3 Value Metrics

To compare the value achieved by a scheduler for a given workload, it is simply the case of adding up the value of all jobs in a workload when the workload has completed. Not all workloads are the same, however, which means that the maximum value achievable will change on each run. The maximum achievable value would occur if every job in the workload were run on time. Under overload conditions, however, this may be impossible. The most effective metric, therefore, is to measure the proportion of the maximum value achievable that is actually achieved by a given scheduler in a given context. This will allow comparison between schedulers as to how well they manage overload.

$$\forall J^k \in W : \; W_{\texttt{Value\_Proportion}} = \frac{\sum \texttt{Value}\left(J^k,\, J^k_{\texttt{SLR}}\right)}{\sum J^k_{\texttt{VMax}}} \tag{7.1}$$

Jobs that pass their final deadline can be considered to have starved or timed out. As starvation will usually cause inconvenience to users, schedulers can be evaluated by how well they minimise the number of starved jobs.

$$W_{\texttt{incomplete\_by\_}D_{\texttt{final}}\texttt{\_proportion}} = \frac{\left|J^k \in W \wedge J^k_{\texttt{PSLR}} \geq J^k_{D_{\texttt{final}}}\right|}{|J^k \in W|} \tag{7.2}$$

7.4 Scheduling Policies for Value

The policies used to schedule for value are all designed to schedule workloads under periods of transient overload. During such periods, the schedulers try to prioritise those tasks/jobs that must run immediately over those that can wait for longer. This is with the aim of postponing those that will lose the least value in the process. In this section, four scheduling policies will be considered.

The time base for the value curves used by these schedulers is defined relative to the SLR of a job. The advantage of calculating value in this way is that SLR can be projected in advance using the upward ranks of tasks, as described in Chapter 6. The schedulers in this chapter use the projected SLR to create a projection of value as part of their scheduling decisions.

In the algorithm for Projected-SLR given in Section 6.1, one time unit is added to the finish time of the projected SLR in order to distinguish between large and small jobs that arrived at the same instant in time. However, this is not required when calculating for value, because the value curves themselves will give the relative importance between jobs. If the projected value of two tasks happens to be equal, then the selection of which one to run will happen in the order that the tasks' parent jobs arrived in.

7.4.1 Projected Value

The Projected Value (*PV*) scheduling policy is designed to be a baseline policy for comparison when scheduling for value. The Projected Value algorithm, as applied to each task in a workload at a given scheduling instant is given in Algorithm 7.4. While it might be natural to assume that PV should be run greedily, aiming for the tasks with the highest value first, this would actually be counter-productive. This is because the few largest tasks would be most heavily prioritised, which would cause responsiveness for the bulk of the workload to suffer similar to that of LRTF (Longest Remaining Time First, defined in Section 6.2.5). For this reason, tasks with the lowest PV are actually run first. This follows the spirit of the P-SLR policy, where tasks that are the most late are run first. It is also intended to run tasks that are about to lose all value, and hence incur a penalty.

Algorithm 7.4 Projected Value Algorithm

$\texttt{PV}\left(T^i, \texttt{curr\_time}\right)$:

$$
\begin{aligned}
J^k &= T^i_{\texttt{parent\_job}} \\
\texttt{P\_SLR} &= \frac{\left(T^i_{\texttt{R}} + \texttt{curr\_time}\right) - J^k_{\texttt{arrive}}}{J^k_{\texttt{CP}}} \\
\texttt{PV} &= \texttt{Value}\left(J^k, \texttt{P\_SLR}\right) \\
\texttt{return}\ \ &\texttt{PV}
\end{aligned}
$$

7.4.2 Projected Value Density

The trouble with PV is that although a task may promise a large value if run first, it may also require a large amount of computational resource to accomplish this. It may be possible for the scheduler to achieve greater value by running several smaller tasks that take less execution resources to achieve greater value.

The Value Density scheduling policy was originally proposed by Locke [115] in order to deal with precisely this problem. Variants of this policy are presented by Li

and Ravindran [110]. Locke [115] shows that running tasks in the order of Value Density is optimal, in the sense that it will always achieve the same or greater value as any other processor. The area in which the proof of optimality holds is limited relative to the industrial scenario considered in this thesis. Tasks must have exactly known execution times, a fixed value for completion, and all tasks must execute on a uniprocessor. Deadlines of all the tasks must also be such that they can all be satisfied by using the Earliest Deadline First (EDF) policy. However, just because the algorithm is not provably optimal outside such scenarios does not mean that it would not perform well in other conditions.

The model of task execution in this work is richer than Locke's model [115], including dependencies and inaccurate estimates of execution times. The platform model is also richer, running on a multi-core, distributed platform. The extended periods of operation under overload means that these workloads are unlikely to be schedulable using EDF [31]. However, a policy that is optimal in another context may still perform well in different contexts. This work uses the inspiration of the value density policy in Locke [115] by using the upward rank to enable value density to be calculated for workloads with dependencies.

The projected value density (*PVD*) policy considered here divides the projected value of a job by the computational requirements it would need to complete its execution. These requirements are the sum of the execution volume of all a task's successor tasks. The equation for PVD is shown in Algorithm 7.5. Tasks with the highest PVD are run first.

Algorithm 7.5 Projected Value Density Algorithm

$$\texttt{PVD}\left(T^i, \texttt{curr\_time}\right) = \frac{\texttt{PV}\left(T^i, \texttt{curr\_time}\right)}{\forall T^j \in T^i_{\texttt{succ}} \cup \{T^i\} : \sum T^j_{\texttt{exec}} \times T^j_{\texttt{cores}}}$$

7.4.3 Projected Value Critical Path Density

A slight alternative to PVD is to divide the PV by the task's upward rank, rather than by the sum of execution required. This gives a better approximation of the time taken to finish a job, rather than the effort required. In large clusters and where responsiveness is important, this may be a more useful measure. The definition of Projected Value Critical Path Density (*PVCPD*) is shown in Algorithm 7.6. Tasks with the highest PVCPD are run first.

Algorithm 7.6 Projected Value Critical Path Density Algorithm

$$\texttt{PVCD}\left(T^i, \texttt{curr\_time}\right) = \frac{\texttt{PV}\left(T^i, \texttt{curr\_time}\right)}{T^i_{\texttt{R}}}$$

7.4.4 Projected Value Density Squared

With all value scheduling, under periods of sustained overload, some jobs must starve and be timed out. If a job is likely to time out, then it is less desirable to even start it. In the same way, it is very desirable that high-value jobs get the highest priority and so continue executing quickly. Scheduling tasks by squaring the value density metric is an approach proposed in Aldarmi and Burns [4]. This gives a more extreme separation between valuable and less-valuable tasks. This is intended to make it more likely that jobs that are never going to finish will ever start. This reduces their execution penalty and the time taken by any of their tasks, which in turn makes it more likely that the most valuable jobs will be able to finish. The model proposed in Aldarmi and Burns [4] considers pre-emptive tasks, although in this work the algorithm is applied to a non-pre-emptive workload. The Projected Value Density SQuared (*PVDSQ*) algorithm is defined in Algorithm 7.7. Tasks with the highest PVDSQ are run first.

Algorithm 7.7 Projected Value Density Squared Algorithm

$$\texttt{PVDSQ}\left(T^i, \texttt{curr\_time}\right) = \left(\frac{\texttt{PV}\left(T^i, \texttt{curr\_time}\right)}{\forall T^j \in T^i_{\texttt{succ}} \cup \{T^i\} : \sum T^j_{\texttt{exec}} \times T^j_{\texttt{cores}}}\right)^2$$

7.4.5 Projected Value Remaining

The previously defined four schedulers have all been extensions of previously published policies to take into account the projection of finish time at a fixed point in time. What all these policies fail to quite capture adequately is the notion of the urgency of a task at a given point in time. While a task may be projected to be valuable or not, there is no indication of whether that value is likely to decrease with waiting much longer.

Therefore, a novel ordering policy for value is proposed, termed Projected Value Remaining (PVR). PVR uses the P-SLR to determine the earliest possible time a task could finish if it were run immediately. The metric is then the area under the value

curve remaining between the P-SLR at the current time and the final deadline $D_{\texttt{final}}$ of the job. This is illustrated graphically in Figure 7.2.

The tasks with the smallest value remaining are run first. Urgent tasks would have a steeply sloping value curve, which would give only a small area under the curve. Tasks about to time out would also have only a small area remaining. Prioritising these tasks would avoid either kind getting to their final deadline. This is designed to reduce starvation and avoid the associated penalties.

The value curves were designed using linear interpolation between the points so that the definite integral of this curve would be quickly and exactly calculable using the trapezoidal method [5]. However, the policy method generalises to any value curves where value can only decrease over time, as long as a final deadline is present. The algorithm to calculate PVR is given in Algorithm 7.8. The integration adds an extra step to the calculation of value compared to P-SLR, but this should take place in constant or $O(1)$ time. Therefore, the worst-case complexity of PVR is $O(t^2 \log t)$, the same as that of P-SLR as discussed in Section 6.1.1.

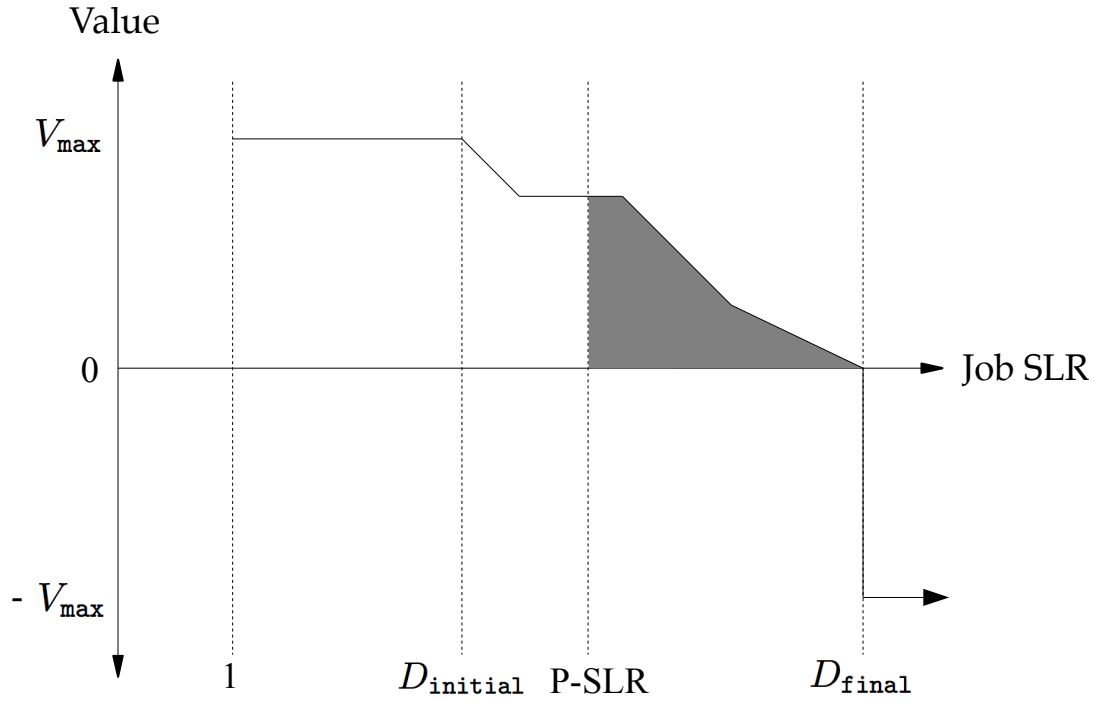


Figure 7.2: Projected Value Remaining Diagram. PVR is the shaded area.

Algorithm 7.8 Projected Value Remaining Algorithm

$\texttt{PVR}\left(T^i, \texttt{curr\_time}\right)$:

$$
\begin{aligned}
J^k &= T^i_{\texttt{parent\_job}} \\
D_{\texttt{final}} &= J^k_{C^i_{D_{\texttt{final}}}} \\
\texttt{P\_SLR} &= \frac{\left(T^i_{\texttt{R}} + \texttt{curr\_time}\right) - J^k_{\texttt{arrive}}}{J^k_{\texttt{CP}}} \\
PVR &= \int_{\texttt{P\_SLR}}^{D_{\texttt{final}}} \texttt{Value}\left(J^k, s\right)\, ds \\
\texttt{return}\ & PVR
\end{aligned}
$$

7.5 Experimental Method

The main experimental method used for the work in this thesis has already been described in Section 5.7. The evaluations in this chapter were performed in simulation using experimental Profile 4, as summarised in Table 5.4.

The performance of the schedulers will be evaluated using the proportion of maximum value metric. This will be investigated across the spectra of load, network delays and inaccurate execution times. The proportion of value metric will also be examined across the space of execution times by the decile of job execution time, for better insight into which classes of tasks schedulers are prioritising over others. This will be done by calculating the metrics of interest for jobs grouped by decile of execution time. The proportion of the workload incomplete by its final deadline will also be considered by decile of job execution time, for further insight into the kinds of prioritisation used by the schedulers and which sizes of jobs suffer most from starvation.

7.6 Value Scheduling Results

7.6.1 Load

Figures 7.3 and 7.4 show a plot of the proportion of the maximum value achievable attained by the different ordering policies compared. The clearest trend is that at low loads, all tasks can run immediately. This means that the maximum value is achieved for every job because there is never contention between them. As the load rises, especially once the arrival rate of work exceeds saturation, then the proportion of the maximum value attainable begins to decrease. This is because some jobs and tasks must necessarily suffer in relation to others. The important aspect to consider is

how well the different scheduling policies manage to balance this contention in order to achieve the highest proportion of value.

An assumption inherent in the work considered is that longer-running jobs will on average deliver greater value than shorter ones. This means that the value of the workload overall will be more heavily influenced by the proportion of value achieved for the largest tasks. While prioritising only large tasks might give a large proportion of the possible value, it may mean that the smallest tasks starve. For the results in Figure 7.3, starved tasks incur a penalty, meaning that policies that starve small tasks will never achieve the highest value. Furthermore, policies that starve small tasks for the benefit of the larger ones would undoubtedly be unpopular with users.

Despite its strength as a bin-packing heuristic for static scheduling to give low workload makespan, the LRTF policy delivers the lowest value once translated into a dynamic scheduling context. The reason for this is apparent in Figure 7.5 which shows the proportion of value achieved by decile of execution time when penalties are considered. LRTF gives the lowest proportion of value across the range of execution times and at every point other than the largest tasks, the value achieved is negative. Figure 7.6 shows the results without penalties, and LRTF also gives the lowest value for all but the largest tasks. By running the largest tasks first, all the small ones will starve. Furthermore, because the largest tasks are allowed to monopolise the grid resources, the use of LRTF may mean that other large tasks will suffer too. Confirming the findings of Chapter 6, LRTF is unsuitable for online scheduling systems, as it delivers lower value even than the Random or FIFO schedulers used as baselines for comparison.

The policies that do not consider execution time estimates also fare poorly on delivering value. These are the Random, FIFO Job/Task and FairShare policies. The effects of these is seen in Figures 7.5 and 7.6, where these policies starve small tasks by making them wait for (on average) the same length of time as larger ones. As discussed in earlier chapters, this is severely detrimental to responsiveness, and hence to value, when the range of execution times is large. By not prioritising the largest tasks either, the value of the largest decile of jobs fails to come close to the maximum possible. Nevertheless, FairShare gives the highest value (Figures 7.5 and 7.6) and lowest proportion of starved tasks (Figure 7.7) for the schedulers that do not consider execution time estimates.

Both the PVD and PVCPD policies give a higher proportion of value than the policies that do not consider execution time. This is because they succeed in prioritising the largest jobs and hence deliver a large amount of value from these, as can be seen in Figure 7.3. However, around the saturation point of the grid, they deliver significantly lower value than several other scheduling policies. Figures 7.5, 7.6 and 7.7 show the reason for this, which is that they also severely starve the shorter-running tasks in the system. Without being able to balance the priority of

work across the range of execution times, it will never be possible to achieve high levels of value.

Below saturation, PVD gives higher value, whereas PVCPD gives higher value at and above the point of saturation. This is because above saturation, PVCPD is better able to identify the tasks that will run for longest and prioritise those, hence giving the greatest value, rather than just those which may consume a large number of cores but not take as long to complete. However, they both starve smaller tasks severely (Figure 7.7), which loses the value of all the smaller tasks, meaning that they fail to achieve the highest amount of value.

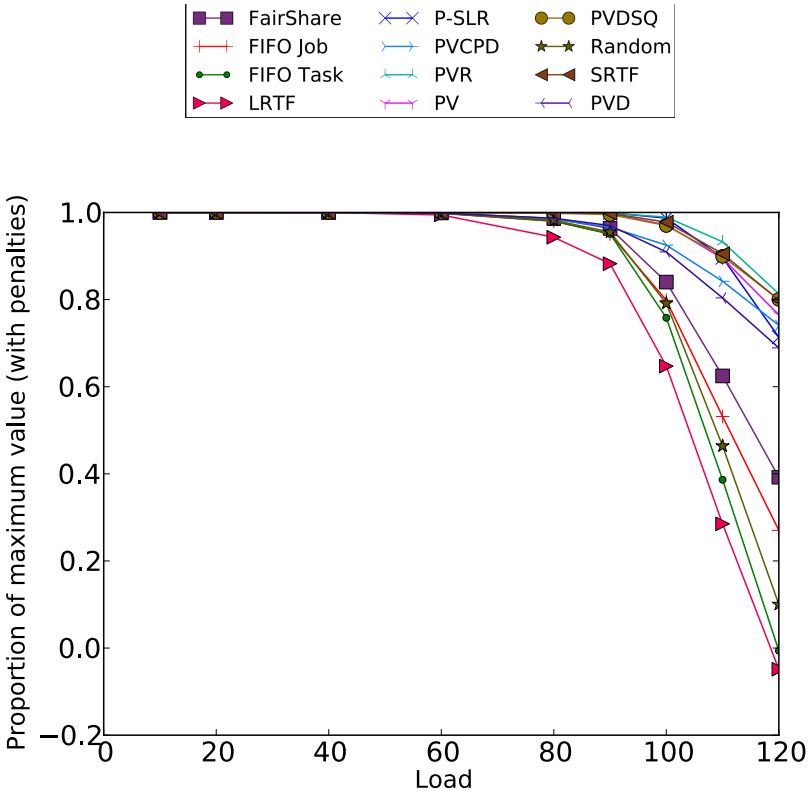
The PVDSQ policy gives higher values than most of the policies that starve a large proportion of the smaller tasks where penalties are applied to starved tasks (Figure 7.3 and 7.7). This is because it is much better at prioritising tasks around the middle of the execution time range, gaining a much higher proportion of the maximum value there (Figure 7.5). However, it still severely starves the smallest tasks, meaning that its total value still suffers compared to policies that treat the smallest tasks fairly.

Where penalties are not applied, PVDSQ gives the highest value of all the schedulers evaluated when load rises above saturation (Figure 7.4), although the difference is only statistically significant at 120% load. This is because PVDSQ has the highest proportion of value for the largest tasks except for PVD and PVCPD (Figure 7.6) whilst starving many fewer of the mid-range tasks (Figure 7.7).

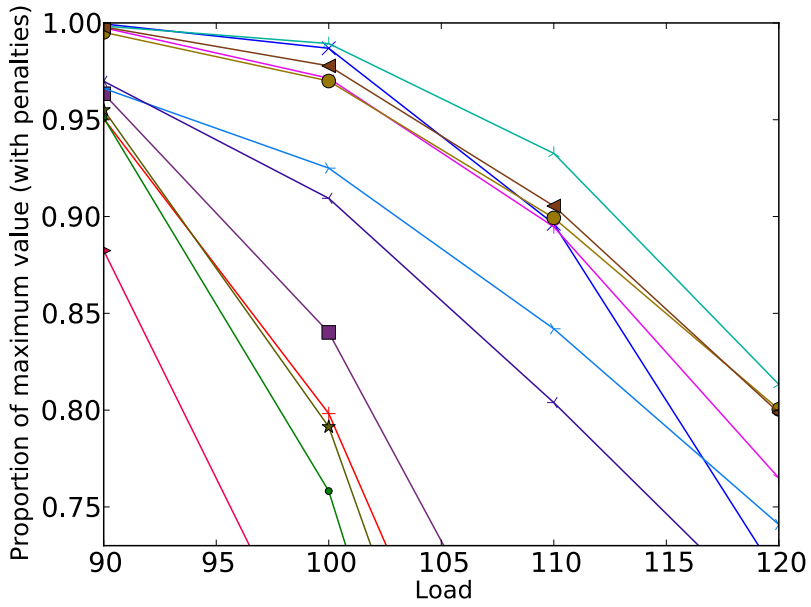
There is a high-performing group of policies that deliver particularly high value across the spectra of load and execution time (Figures 7.3, 7.4, 7.5 and 7.6). These are SRTF, P-SLR, PV and PVR. Below the saturation point of the load spectrum, P-SLR gives the highest proportion of value achievable. However, in this range all four of these schedulers attain greater than 99% of the maximum possible value attainable, so their differences are not statistically significant. The reason that these policies give such high values is that they attain high proportions of the maximum value across the space of execution times.

SRTF and PV prioritise small tasks directly. This means that they give the highest values across the execution time range except for the largest tasks, which suffer. As the largest tasks are also some of the most valuable, the total value achieved is brought down. These policies are able to have such high values for the smaller tasks because the resources freed by postponing a single large task are able to run many hundreds of smaller tasks instead. This can be seen from their relatively low proportion of starved large tasks in Figure 7.7.

The importance of preventing the large tasks from starving in order to achieve high value is shown by comparing PVR and PV in Figures 7.6 and 7.7. Although the proportion of starved tasks is only slightly higher for PV (Figure 7.7), this causes the proportion of maximum value it delivers to be reduced significantly compared to PVR (Figures 7.5 and 7.6).

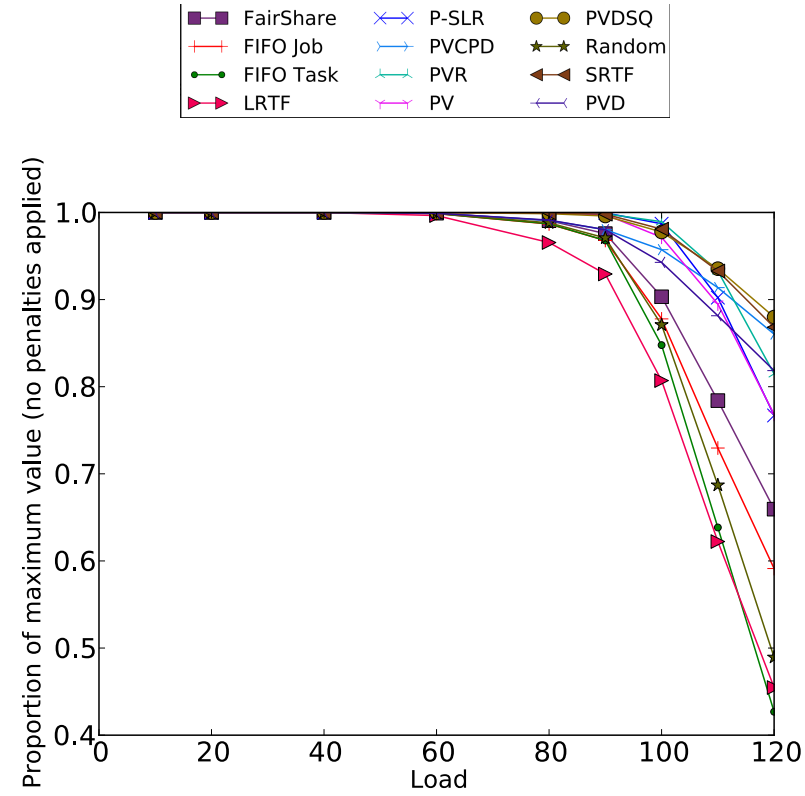


(a) Value against load (full-scale)

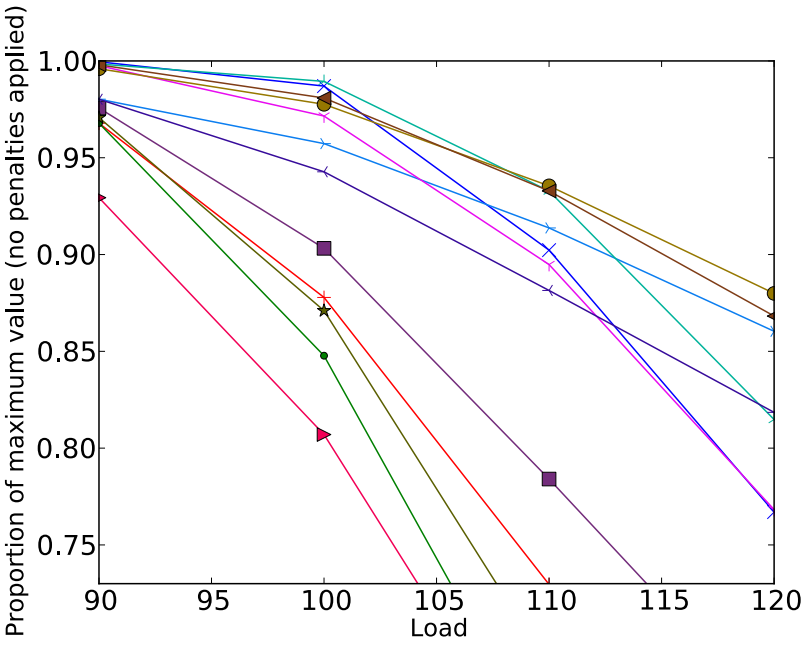


(b) Value against load (zoomed)

Figure 7.3: Value across the load spectrum with penalties

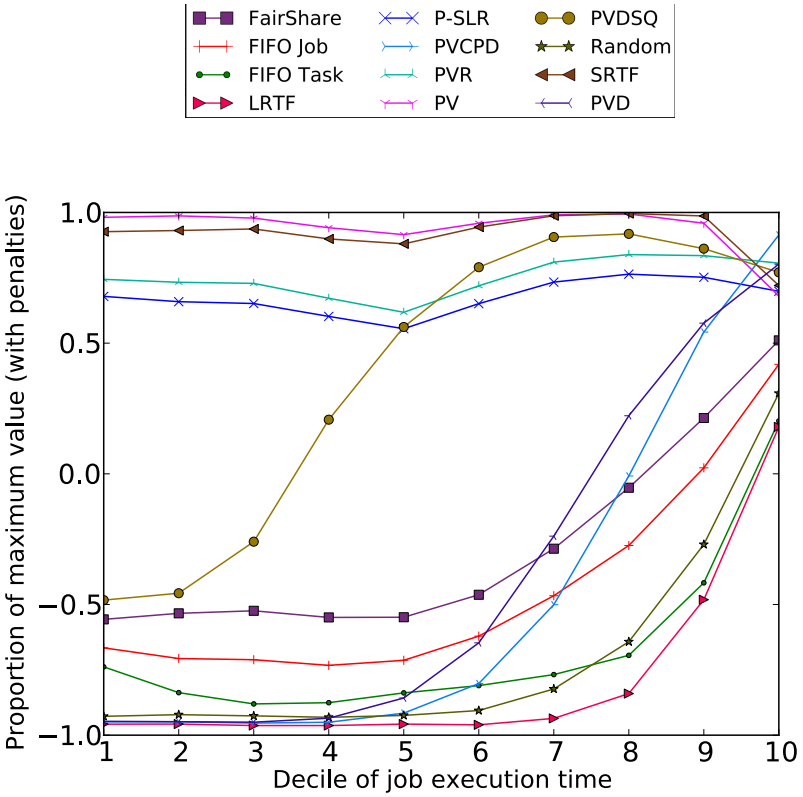


(a) Value against load (full-scale)

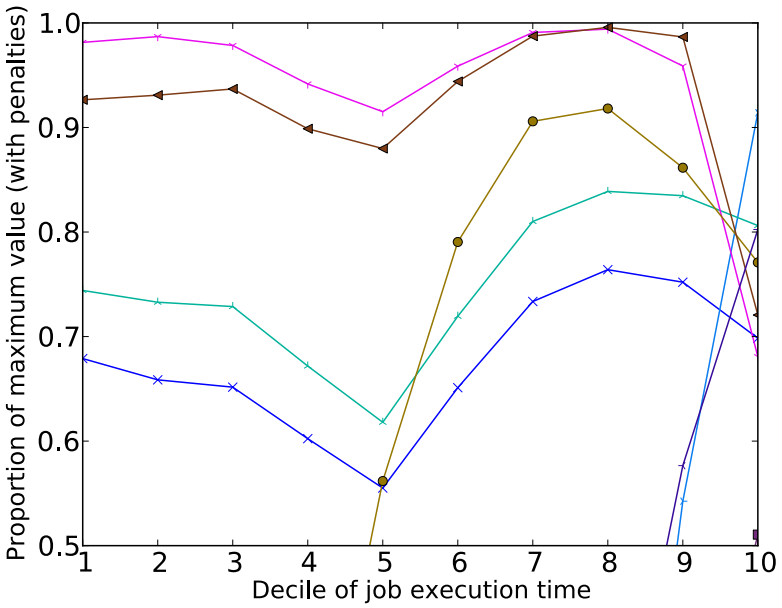


(b) Value against load (zoomed)

Figure 7.4: Value across the load spectrum without penalties

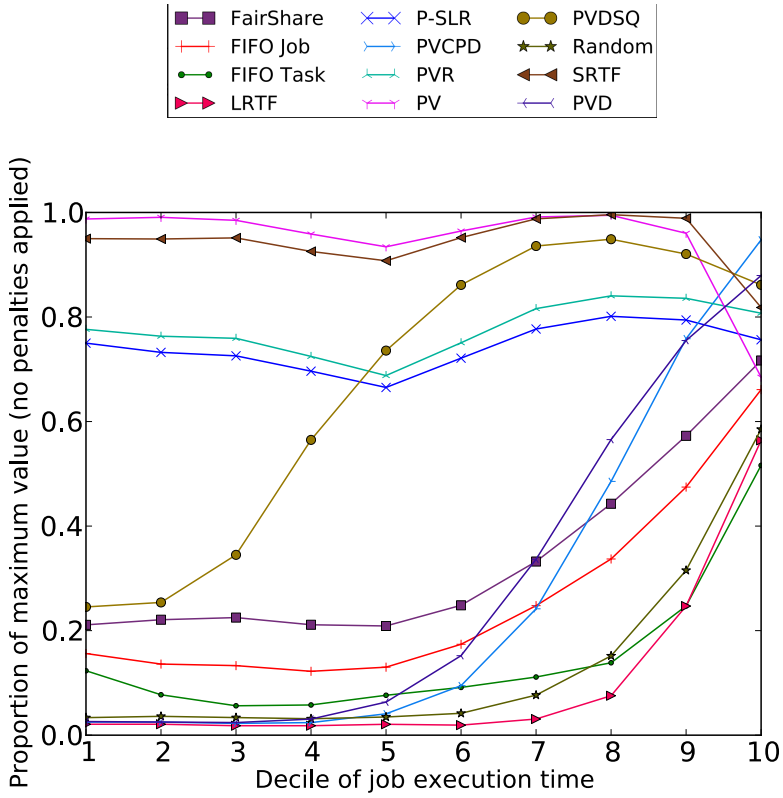


(a) Value by decile of execution time (full-scale)

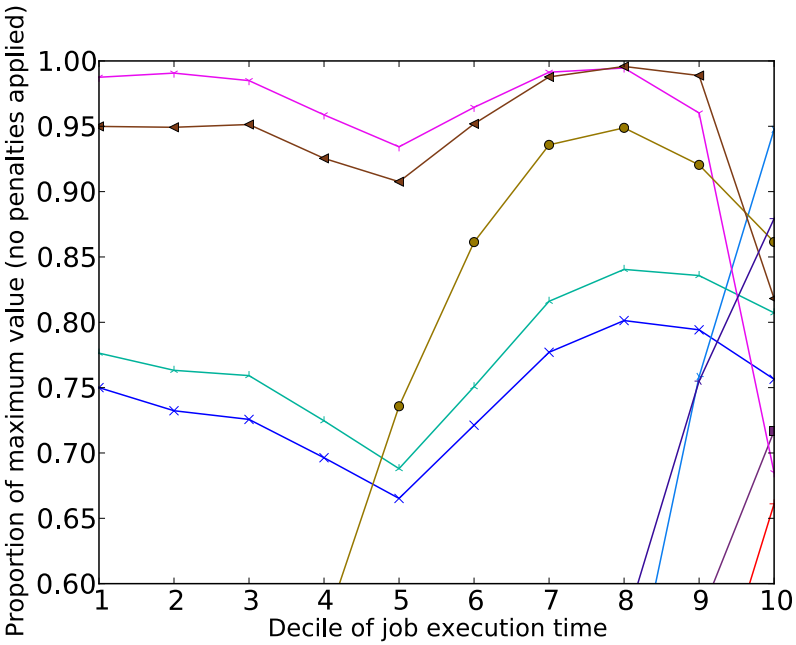


(b) Value by decile of execution time (zoomed)

Figure 7.5: Value achieved by decile of job execution time (with penalties)

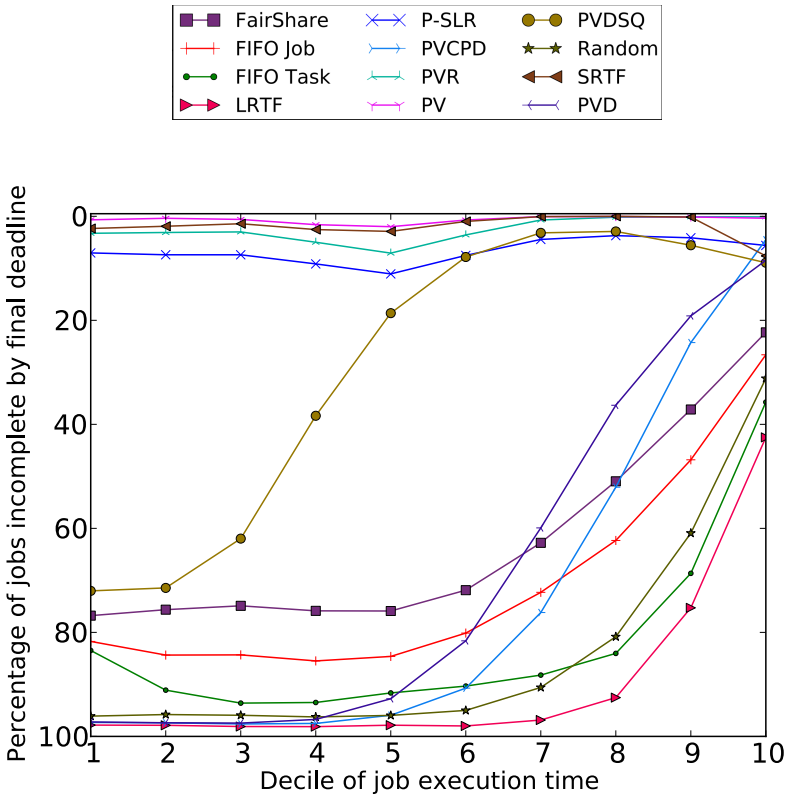


(a) Value by decile of execution time (full-scale)

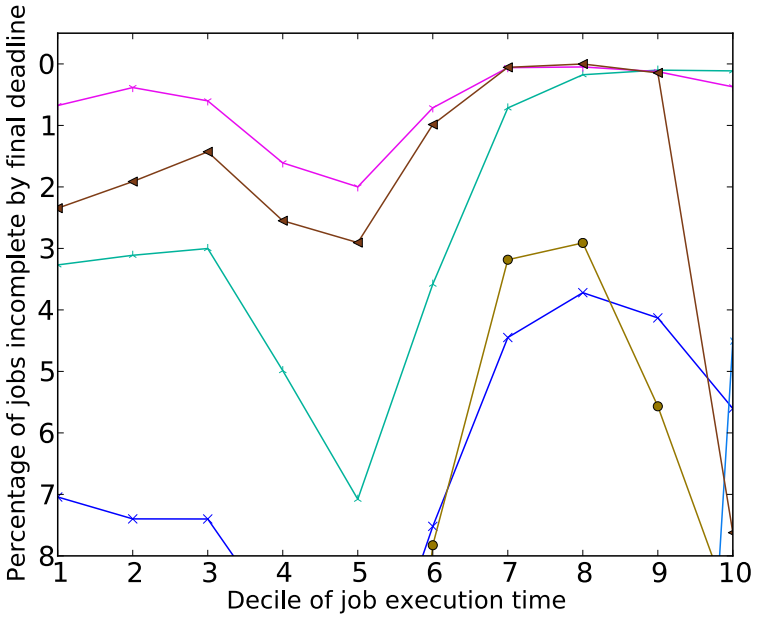


(b) Value by decile of execution time (zoomed)

Figure 7.6: Value achieved by decile of job execution time (without penalties)



(a) Proportion of starved jobs by decile of execution time (full-scale)



(b) Proportion of starved jobs by decile of execution time (zoomed)
Note: Y-scale is inverted to aid in visual comparison with the previous graphs of value against decile of execution time.

Figure 7.7: Proportion of jobs starved by decile of execution time

PVR achieves the highest levels of value under overload when penalties are considered (Figure 7.3) because it starves the lowest proportion of the largest jobs of all the schedulers evaluated (Figure 7.7). For the largest tasks, a crossover point is noticeable as SRTF, PVDSQ and PV fall behind PVR in the proportion of maximum value achieved (Figure 7.5). This crossover is how PVR is able to deliver the highest value overall, because PVR delivers high value to the large tasks while simultaneously not letting the small tasks suffer too much.

The results are slightly different where penalties are not considered (Figure 7.4). With load at the point of saturation, the PVR policy achieves the highest value, although its margin above P-SLR is small (though statistically significant). At 110% load, P-SLR falls behind in the proportion of maximum value. Instead, PVR, PVDSQ and SRTF lead jointly with the differences between them being statistically indistinguishable. At 120% load, a sizeable overload, then the PVDSQ, SRTF, PVCPD and PVD policies give a higher proportion of maximum value than PVR. Without penalties for starvation being applied, these other policies appear to do well. However, PVD, PVDSQ and PVCPD achieve these high proportionate values by starving large numbers of smaller jobs in order to run the few jobs that are the most valuable (Figure 7.7). While this may be desirable for maximising the value metric that does not include penalties, it is likely to also cause significant user dissatisfaction. For example, the PVCPD policy starves virtually all of the jobs with execution times at or below the median. SRTF and PV, on the other hand, run the largest number of jobs in the workload but starve those that are most valuable. This is also likely to cause user dissatisfaction.

This highlights three approaches used by schedulers for achieving high value under overload. Either the smallest or largest tasks are starved for the others' benefit, or a balance is necessary across the workload. In order to gain value from starving the smallest tasks, a high proportion of the workload must be starved in order to run the few largest jobs. On the other hand, only a few of the largest jobs need to be starved in order to run all the smaller ones. Yet these larger jobs are also the most valuable and therefore users are most likely to be inconvenienced due to their starvation.

When using penalties to represent the inconvenience to users due to such starvation, it is clear that the most appropriate strategy to maximise value is to aim for an even distribution of starved tasks across the range of execution times. The PVR scheduling policy is able to do this most effectively at or above saturation, and hence attains the highest value. This is also likely to be perceived by users as the most fair distribution and hence be the policy of choice.

P-SLR achieves good a value across the range of execution times (Figures 7.5 and 7.6) because it also achieves good responsiveness across this range. However, due to the model considered, responsiveness is not perfectly correlated with urgency and

hence value. PVR is able to combine the factors of responsiveness and value to measure urgency. This is possible because the X-axis of the value curve represents responsiveness, and the Y-axis represents value. PVR can combine both by using the area under the curve.

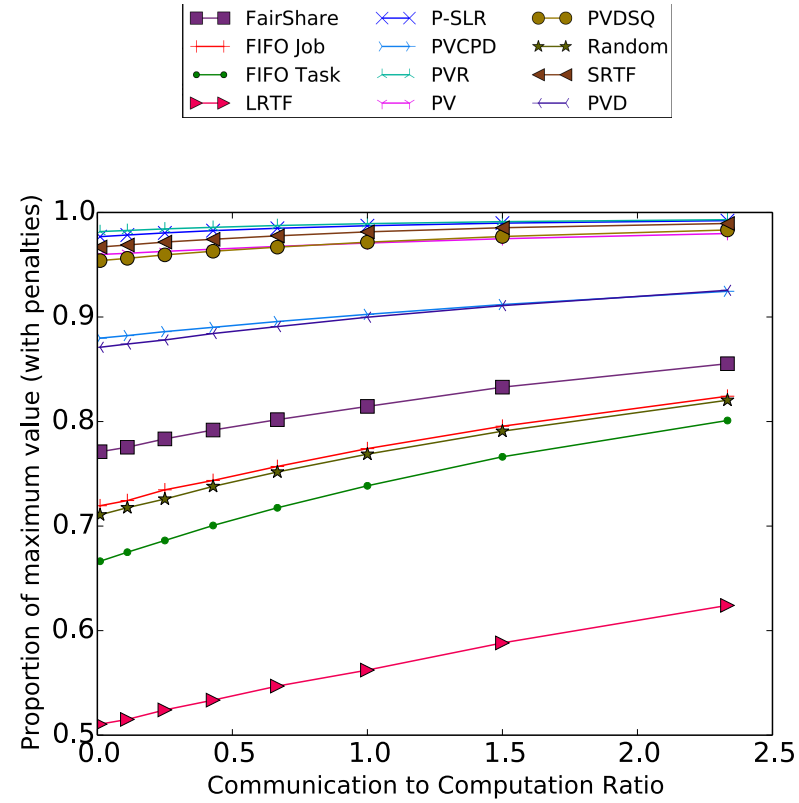
Using this technique, PVR is able to achieve better value than P-SLR above saturation (Figures 7.3 and 7.4). Where P-SLR makes all tasks and jobs suffer as evenly as possible, PVR is able to use value to discriminate between those tasks that give higher value and those that do not. It can then more intelligently starve the tasks that would never have delivered much value to begin with. This means that PVR is most effectively delivering the benefits of using a value-based policy in the zone of load where this is most needed: slight, yet continual overload.

P-SLR is designed to be starvation-free in the general case where jobs can wait as long as needed before being executed. The starvation-free guarantee means that all jobs will complete eventually when scheduling using P-SLR. However, where a value curve with a final deadline is applied, the final deadline may well be before the point at which P-SLR would be able to run the job in an overloaded system. This is why some jobs with value curves will still starve in the sense of not being completed by their final deadline, even when using the starvation-free P-SLR scheduler.

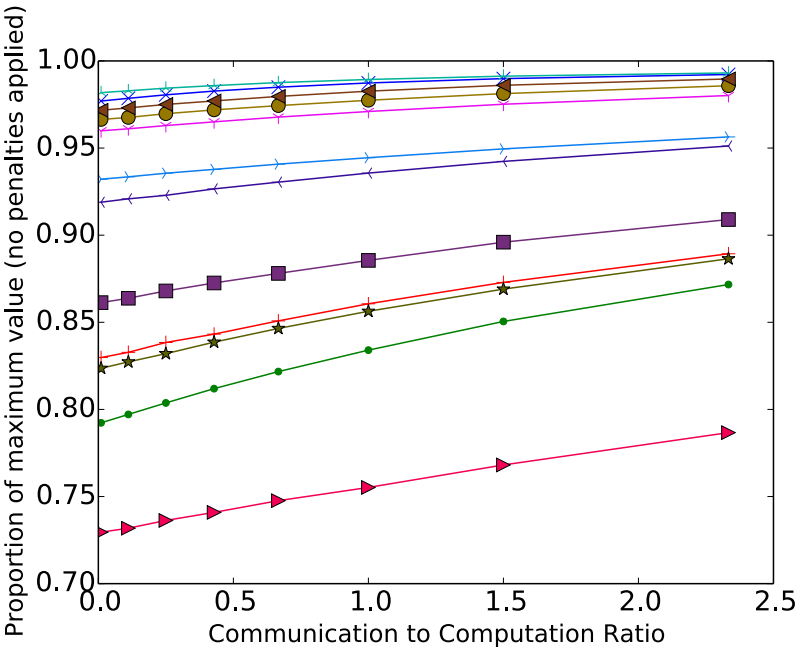
A noticeable feature of Figures 7.3 to 7.6 is that there is a dip in the value achieved in the middle of the execution time range, and this dip is exhibited by all the schedulers that achieve high levels of value. This is because these are the pieces of work that are most likely to be starved (Figure 7.7). The smallest jobs have short execution times so they will be prioritised due to their urgency. The largest jobs are prioritised to some degree because their values are so large but also because they may not suffer much by having to wait until a lull in arrival rates (such as those over the weekend) when nothing else is in the queue and they can start. In the middle lie the jobs that are neither so urgent that they must be run immediately, nor so large that not running them would cause a significant reduction in value obtained. Naturally, a scheduler seeking to maximise value should seek both the quick wins and the highest value jobs, and it is those in the middle that will get starved. The dip for SRTF is likely to be present as well because mid-range jobs will likely consume more resources and so face more contention on the cluster, whereas small tasks may be able to fit in and run with fewer resources.

7.6.2 Network Delays

Figure 7.8 compares the different schedulers' ability to achieve the proportion of maximum value across the space of networking delays for simulations with and without penalties. This figure is drawn using with results from load factors of 90, 100 and 110%, which explains why the baseline proportions with almost no network

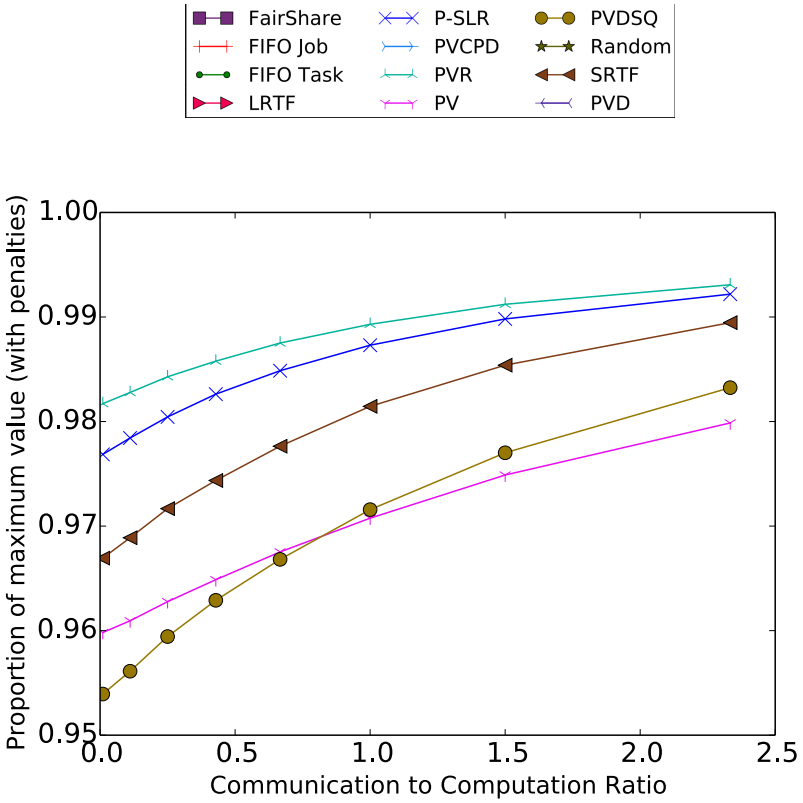


(a) Value with networking (with penalties)

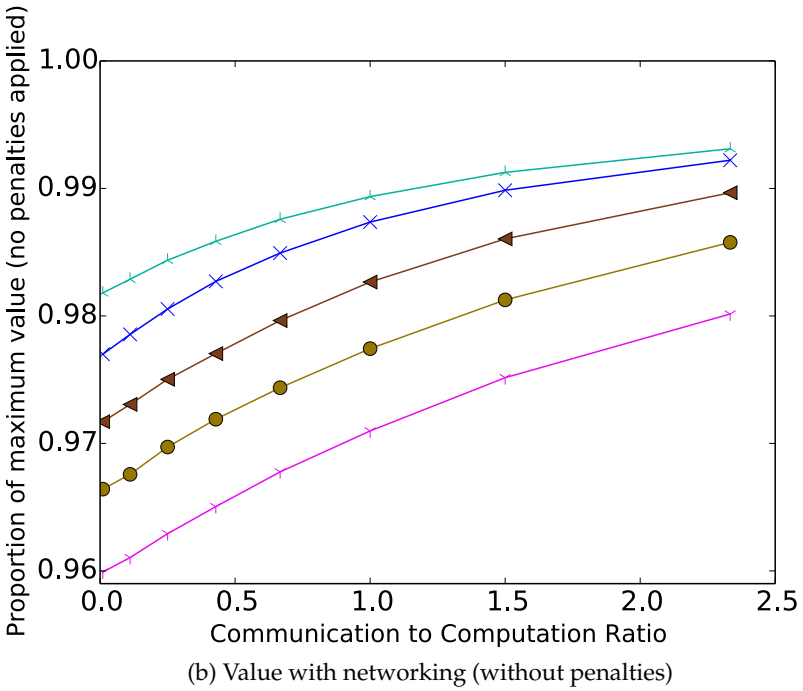


(b) Value with networking (without penalties)

Figure 7.8: Value with networking delays (full scale)



(a) Value with networking (with penalties)



(b) Value with networking (without penalties)

Figure 7.9: Value with networking delays (zoomed)

delays are high. There is a clear trend between the schedulers that as networking delays increase, the proportion of the maximum value achieved increases gradually. As network delays increase, so will the length of the critical paths of the jobs. A longer critical path reduces the SLR for the same turnaround time, improving the value.

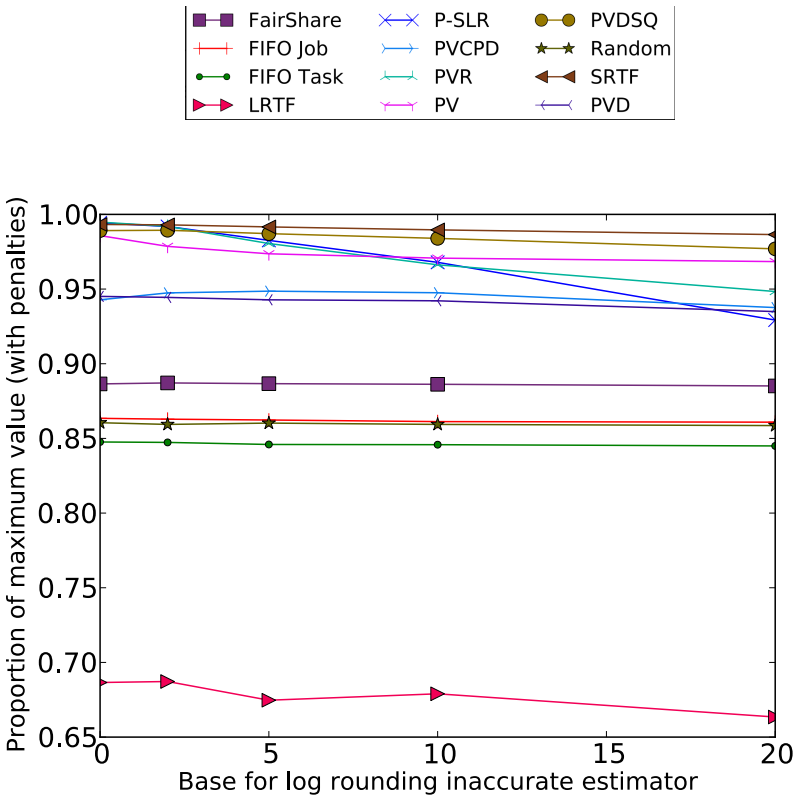
The ordering of policies is the same as examined previously when load is around the point of saturation. LRTF continues to give the lowest proportion of value across the range of networking delays. The policies that do not consider execution times (Random, FIFO Task, FIFO Job and FairShare) form a group above LRTF in value achieved, but this value is still relatively low. PVD and PVCPD are close throughout the range, becoming statistically indistinguishable (repeated measures t-test, $p = 0.05$) once communication is more time-consuming than computation.

To help distinguish between them the highest-performing policies (P-SLR, PVR, PV, PVDSQ and SRTF) are shown with a zoomed scale in Figure 7.9. These policies also see an increase in value achieved across the network spectrum, although this increase is less dramatic than the other policies. All the policies are dominated (statistically significant using a repeated measures t-test, $p = 0.05$) across the range by PVR, because of its ability to balance responsiveness and value. P-SLR also performs well, and would likely be an appropriate choice of scheduler if value curves were not available. SRTF also performs well, although not as well as P-SLR and PVR because of its tendency to starve large jobs. PV continues to suffer from its de-prioritisation of the largest tasks, meaning that it is dominated by PVR, P-SLR and SRTF. PV gives a higher proportion of value at lower networking delays than PVDSQ, although PVDSQ outperforms PV where network delays are high and penalties are considered.

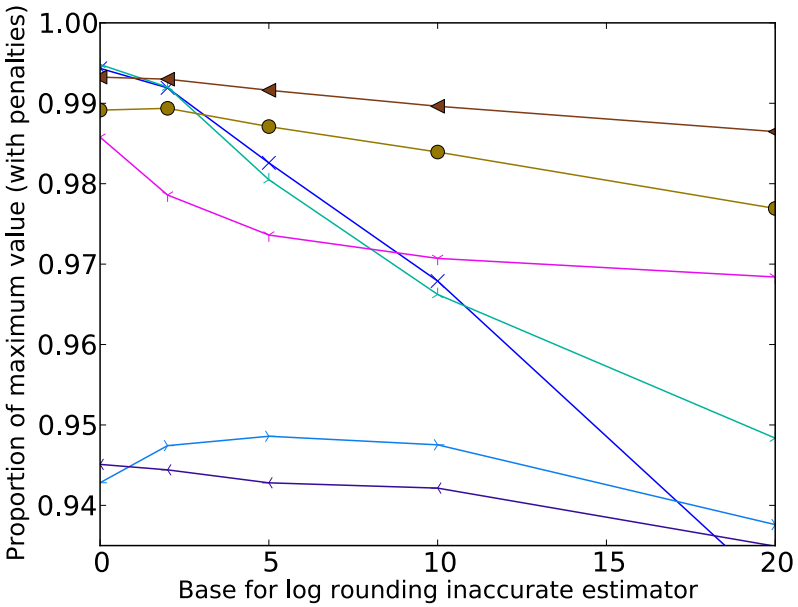
7.6.3 Inaccurate Estimates of Execution Times

Figures 7.10, 7.11, 7.12 and 7.13 show the changes in the proportion of maximum value achieved as inaccuracies in the estimates of execution times change. Figures 7.10 and 7.11 consider logarithmic rounding errors, where execution times of tasks are grouped by rounding them up to the nearest power of N, as described in Section 5.7.1.8, Equation 5.28. Figures 7.12 and 7.13 show how the proportion of maximum value achieved is impacted by introducing errors drawn from a normal distribution around the true value of execution time, using the method described in Section 5.7.1.8, Equation 5.27.

Similar to the results in Chapter 6, LRTF achieves the lowest proportion of value across the spectrum of execution time estimate inaccuracies, whether penalties are applied or not. Where estimates are normally distributed, the value achieved by LRTF increases as inaccuracies rise. This is because inaccuracies reduce its ability to achieve

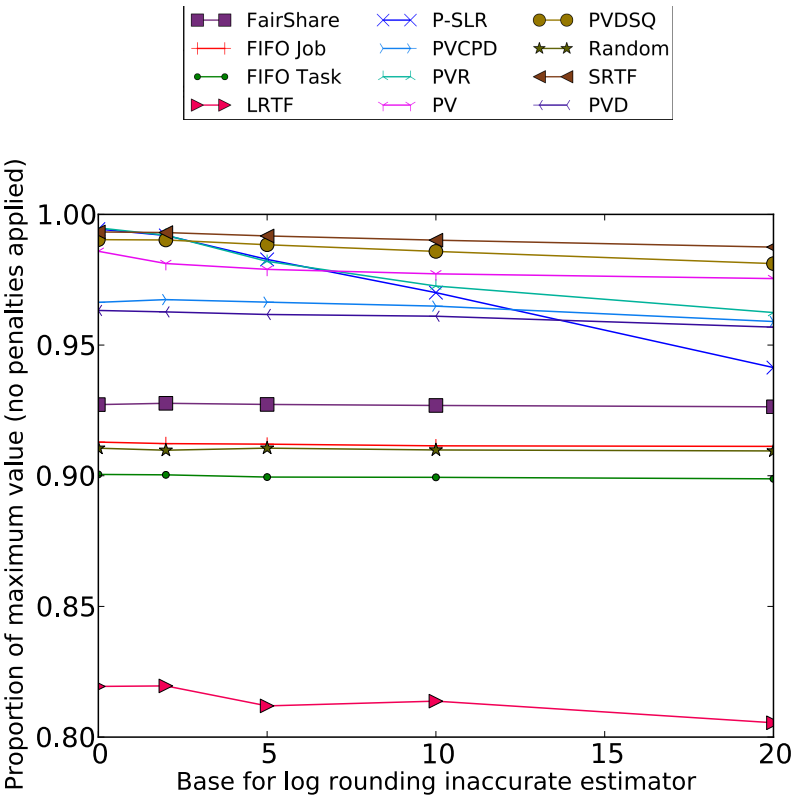


(a) Value with log-rounded inaccurate estimates (full scale with penalties)

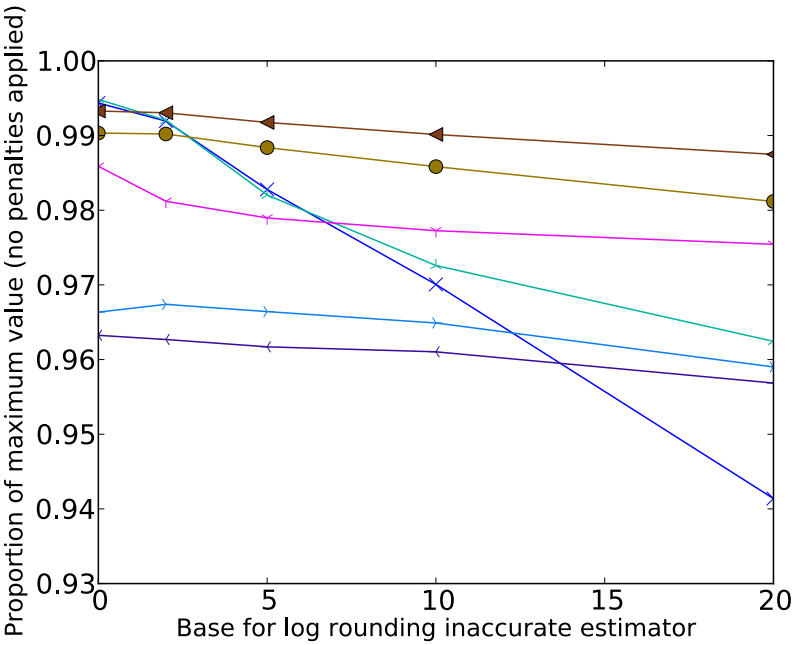


(b) Value with log-rounded inaccurate estimates (zoomed with penalties)

Figure 7.10: Value with logarithmically-rounded inaccurate estimates of execution times with penalties

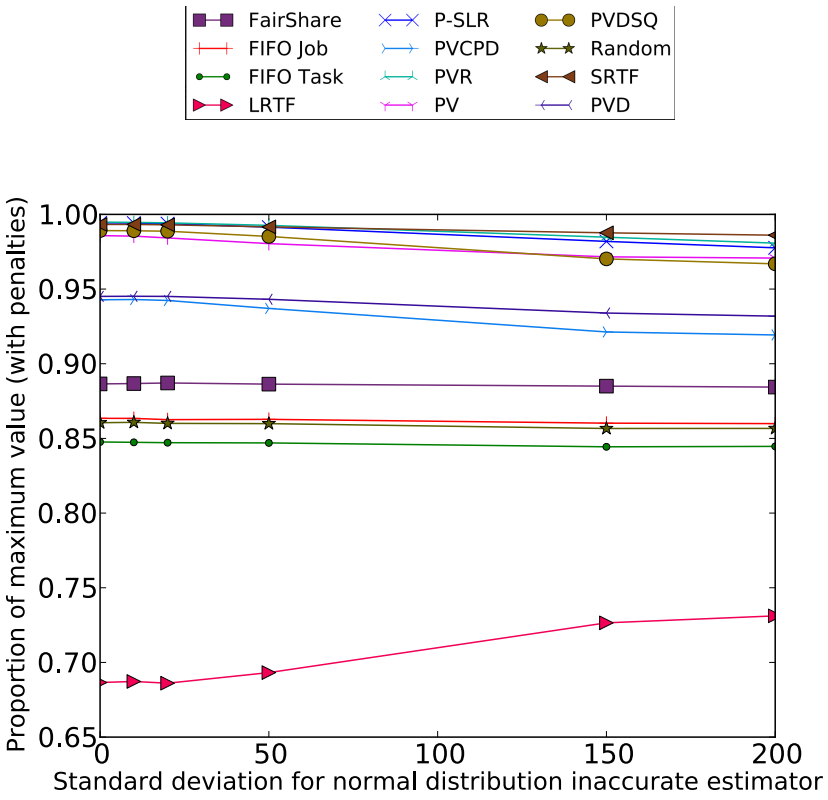


(a) Value with log-rounded inaccurate estimates (full scale without penalties)

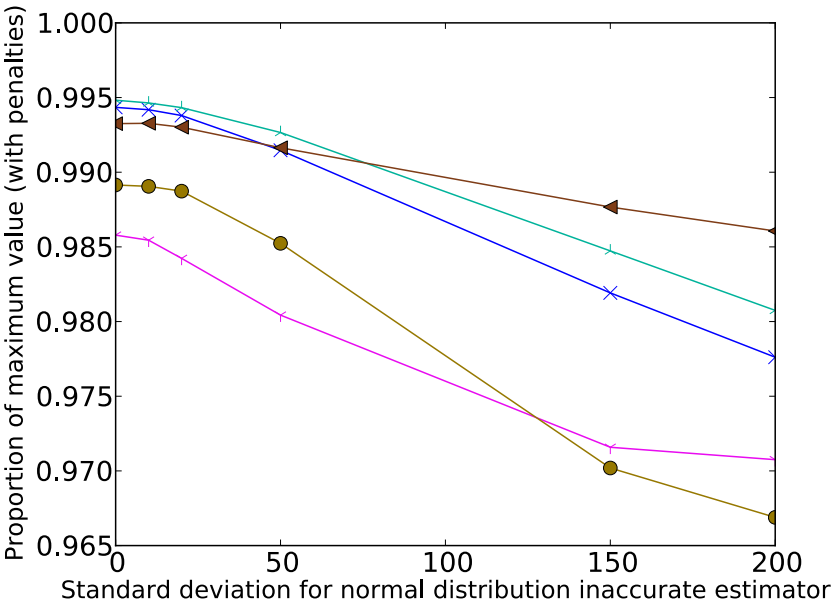


(b) Value with log-rounded inaccurate estimates (zoomed without penalties)

Figure 7.11: Value with logarithmically-rounded inaccurate estimates of execution times without penalties

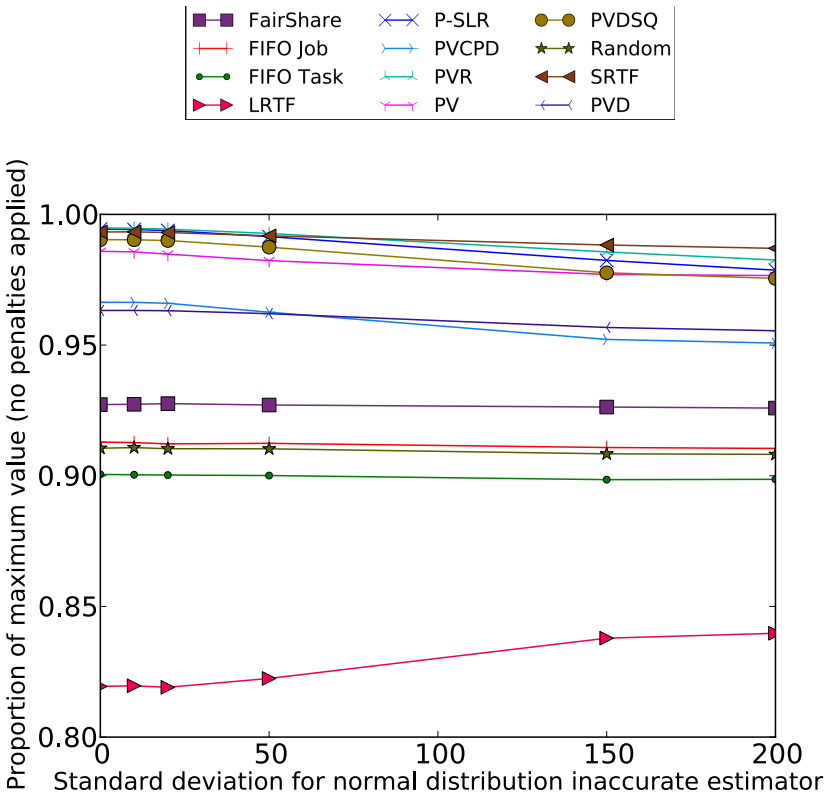


(a) Value with normal inaccurate estimates (full scale with penalties)

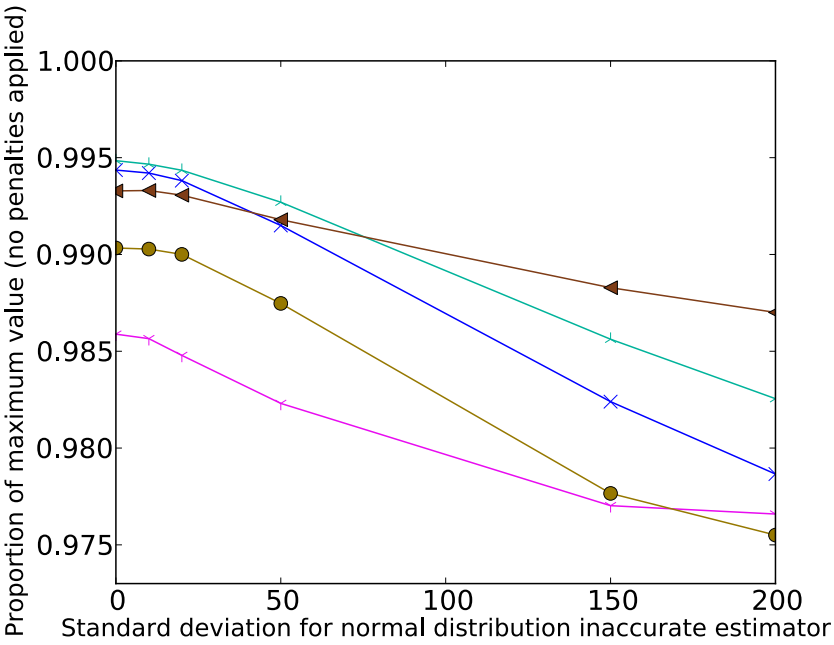


(b) Value with normal inaccurate estimates (zoomed with penalties)

Figure 7.12: Value with normally-distributed inaccurate estimates of execution times with penalties



(a) Value with normal inaccurate estimates (full scale without penalties)



(b) Value with normal inaccurate estimates (zoomed without penalties)

Figure 7.13: Value with normally-distributed inaccurate estimates of execution times without penalties

the worst case. On the other hand, the log-rounding inaccuracies decrease the value achieved by LRTF by hurting its ability to prioritise the largest tasks and achieve any value from them, as many more tasks are grouped together.

As expected, the schedulers that do not take execution times into account (FIFO Task, Random, FIFO Job and FairShare) all achieve the same proportion of value whatever the inaccuracies in execution time. It is clear, however, that even with large inaccuracies, there is significant benefit to be gained by using estimates of execution time, as all but one of the scheduling policies that do use execution times perform better than the ones that do not.

All of the scheduling policies other than LRTF that consider execution time estimates experience a fall in the proportion of the maximum value achieved when estimate inaccuracies increase. This is because as inaccuracies increase, some tasks may be treated as urgent when they are not, delaying tasks that are genuinely urgent and impacting the value achieved. Alternatively, some urgent tasks may be treated as if they had plenty of slack time, leading to a loss of value because they do not deliver results in a timely way.

When normally-distributed inaccuracies are present, PVCPD and PVD perform similarly, although PVCPD loses more value with increased inaccuracies than PVD (Figures 7.12 and 7.13). The critical path of a job with many dependencies will be usually be shorter than the total execution time of the same job. If the inaccuracy in estimation is the same, then it will have a proportionately greater impact on the value of the estimated critical path than on the total execution time. This explains the differing trend between PVD and PVCPD. When the log-rounding inaccuracies are present, however, PVCPD delivers slightly higher value than PVD (Figures 7.10 and 7.11). This is because instead of just introducing error, log-rounding groups jobs into bins of execution time. The case of log-rounding will push tasks into higher groups, which are those that PVCPD prioritises over PVD. PVDSQ experiences the same phenomenon under log-rounding, although SRTF gains higher value than PVDSQ throughout the range of inaccuracies.

The PV policy returns value higher than PVD or PVCPD throughout the range of execution time estimates. Its performance falls off the most gradually as estimate inaccuracies increase. This is because it achieved good value for all but the very largest tasks. In addition, if some of the largest tasks were inaccurately estimated to be smaller than they really are, more value will be delivered for them as well. It does not achieve the same value as PVR or P-SLR where normally distributed errors are present or where log-rounding errors are low.

At low levels of normally-distributed inaccuracy ($\mu \leq 50\%$), PVR gives the statistically significantly highest value results, followed closely by P-SLR and SRTF (Figures 7.12 and 7.13). This is the case whether penalties are applied or not. As inaccuracies get larger, however, SRTF comes to dominate. This is because of its

tendency, as described in Section 6.4.2, to become more fair with respect to responsiveness with increased inaccuracy of estimates. Although the highest values achieved overall are done by SRTF, it should be remembered that these are achieved by starving the largest tasks. This may not be desirable where a more fair treatment of work is needed. As inaccuracies grow, the likelihood of treating a small task as if it were larger grows, which causes P-SLR and PVR to have lower value as inaccuracies mount. PVR remains second best across the space analysed with normally-distributed inaccuracies (Figures 7.12 and 7.13).

Log-rounding introduces a much larger inaccuracy than the normal distribution approach. Wherever log-rounding is applied, the performance of PVR drops behind SRTF. Where log-rounding values are high, PVR also attains a lower proportion of the maximum value than PV and PVDSQ, regardless of whether penalties are applied or not (Figures 7.10 and 7.11). PVR's success in achieving high value when execution time estimate are accurate is based on it being able to perform a fine balancing act to decide which tasks are most urgent when it is impossible to run all tasks immediately. Introducing inaccurate execution times in the model used affects PVR more than other policies because it affects both axes on the value curve: the projected-SLR as well as the estimated maximum value. As the PVR is based on the area under this curve, estimate errors cause this area to change with the square of the error. Log-rounding errors tend not to affect SRTF so much because the vast majority of small jobs are all still given high priority and so rounding hardly affects them. The largest jobs will remain large even when significant rounding is applied, and these will remain penalised by SRTF.

The P-SLR policy gives results that are statistically indistinguishable to PVR where inaccuracy using logarithmically-rounded estimates is low (Figures 7.10 and 7.11). These results differ from the earlier results concerning load showing PVR giving higher value (Figure 7.5) because these are taken from a range of loads, rather than simply under an overload situation. As estimates become poorer, PVR is able to retain more value than P-SLR because it keeps more value from the largest jobs. Where normally-distributed inaccuracies are present, PVR consistently returns a higher proportion of value than P-SLR.

SRTF, PVDSQ and PV are able to attain a higher proportion of value than PVR under highly inaccurate log-rounding estimates. SRTF achieves a higher proportion of maximum value when normally-distributed errors are large. This is because these policies each starve just one extreme of the value curve (large tasks for SRTF and PV, small ones for PVDSQ). This frees up resources then used to gain good levels of value for work across the rest of the execution time spectrum. Unfortunately, starving one extreme means a class of tasks are always penalised. This is likely to lead to dissatisfaction for users with many jobs in these classes. PVR is able to give a fairer balance across the execution time spectrum by only starving work that is less

valuable. Although inaccuracies reduce its ability to achieve the maximum possible value relative to other policies, it does this without penalising any single class of jobs. Therefore, long-term user satisfaction is likely to be higher with PVR.

7.7 Summary of scheduling for Value

7.7.1 Summary of Results

This chapter considers the application of value-based policies implemented in the context of list scheduling. A model of value is described that defines value curves between an initial and a final deadline. The value curves are applied to specific jobs by scaling these curves by a value factor and the critical path length of the job defined through the SLR metric. Using SLR, a measure of responsiveness, is motivated by the observation that the value of jobs to users should be related to responsiveness. However, the responsiveness requirements of different jobs, even those of the same size, should be tuneable using value curves.

Several scheduling policies relating to value are described and formally defined. A novel policy termed Projected Value Remaining or PVR is described, which uses the integration of the value curve remaining for a job in order to prioritise tasks in a queue of work. The evaluation shows that PVR is equal or dominant in its ability to deliver a high proportion of value across the spectra of load and network delays where penalties were applied to starved tasks. When no penalties were applied, SRTF and PVDSQ gave higher value at the most extreme point of overload sampled, although these policies starve the largest and smallest tasks in the workload, respectively (see Figure 7.7), which may be an issue for production systems.

PVR is sensitive to execution time inaccuracies, no longer returning the highest value of the schedulers evaluated once inaccuracies are significant. Nevertheless, PVR returns the highest value of the policies evaluated when normally-distributed inaccuracies have a standard deviation below 50% of the original value. This kind of estimate is possible for the best user estimates to achieve [107].

Achieving the highest level of value is also dependent on treating the largest jobs appropriately. Because these jobs are very large, they are naturally closely monitored and approved to run. They are also key to the successful completion of projects. Because the largest jobs also take a long time on a human scale, their execution time estimates are likely to be of higher quality than those of the smaller tasks. This may to some degree mitigate the impact of inaccurate estimates on PVR. It is also important to note that the loss of value due to inaccurate estimates is relatively small (5% of the maximum value), even for high inaccuracies.

In the regions where PVR is not able to give the best value, SRTF gave the highest value instead. While the users who require responsiveness for small tasks would

support the selection of SRTF in an industrial context, the organisational consequences could be severe. This is because SRTF achieves high value under overload by starving the largest jobs. The largest jobs may also be some of the most critical on the development path of a project, because there are fewer of them. Therefore, delays or starvation for these tasks beyond what is proportional to everything else on the cluster may be detrimental to project deadlines.

On the other hand, due to the peaks and troughs in the arrival of work, it may be the case that SRTF is still appropriate because it is never too long to wait (relative to the execution times of the largest jobs) for a trough in arrival rates to occur so the largest jobs can reach the front of the queue. Careful monitoring of average load rates would then be necessary, however, as once load has passed saturation for an extended period, the largest tasks would starve.

Whatever scheduling policy is used, it should be a cause for concern to system administrators where load consistently exceeds the point of saturation by a wide margin. Even if high value is achieved, users who often submit jobs whose relative value is low may be dissatisfied if it is always their jobs that are starved.

Saying that, use of the PVR scheduling policy would allow effective management of a system that spends most of its time near saturation point. A particularly useful feature of using value curves in combination with the PVR policy is that responding to the daily and weekly cycles of work is not hard-coded into the scheduling system. Instead, the scheduler itself creates these desirable conditions by responding dynamically to the value curves delivered. If the mix of workloads, their patterns, or their value curves change significantly, there would be no need to change the scheduler or its configuration. This is a significant improvement over the current industrial FairShare policy which requires frequent manual adjustment of the share tree.

7.7.2 Extensions and Application of PVR

The definition of value curves will always be an activity with consequences for stakeholders, as every system which requires arbitration between workloads competing for resources will exist in the context of a socio-technical system. As long as the value curves reliably represent the needs and desires of users, the PVR policy will deliver the best value possible as long as execution time estimates are within a reasonable accuracy, given the scenarios investigated. Naturally, controls would need to be put in place so that users cannot 'game' the system by giving misleading or inflated value or execution time requirements to the system. This is no different to the FairShare system currently used, where the share allocations between groups and between users are subject to the same political tensions.

Further sociological research could be worthwhile to gain a greater understanding of how resources are allocated in organisational contexts. This could then give greater insight into how to assign value within the context of computing resources to ensure this supports rather than undermines the organisational approach.

The evaluations of the PVR policy in this chapter were conducted in a strictly dynamic scheduling scenario. Further research could well provide insight into how PVR performs in a static context. Studies into which classes of tasks are prioritised by high-performing search- and market-based policies may provide insight into possible improvements to PVR.

Rather than letting value go to zero or applying a fixed penalty when tasks reach their final deadline, others have considered value curves that extend below zero [86]. A natural extension of this work would be to evaluate the value schedulers considered here with these extended value curves.

Further work could also consider hybrid workloads where value curves have only been applied to some jobs in the workload. In this instance, a default value curve could be applied to jobs without one. The function $1/\text{P-SLR}$ could be used as a value curve, although to be suitable for the model used in this work a final deadline would need to be specified at some point. Enabling the a hybrid workload would ease the transition period for an organisation from purely responsiveness-based system to one based on value. This is likely to be of interest to organisations considering migrating workloads or grid platforms to the cloud.

The utilisation of a value-based scheduling policy is likely to be of particular interest to cloud computing providers, where their users explicitly pay for the capacity they use. Cloud computing providers could achieve the highest value possible even under overload situations.

Value is likely to be useful in the industrial scenario considered, because even if a grid is run by a single party, there are still real costs associated with its running. By using value, these costs can be charged to the projects that actually use the grid. Deploying PVR through the existing list scheduling architecture would allow the consideration of value without requiring wholesale changes to the grid infrastructure, such as would be required by a move to a market-based scheduling architecture.

If the use of value were tied to the consideration of costs on projects, this would incentivise users and managers to only submit jobs that are really required, helping to reduce load on the grid. The cost metrics also help to spread the load of the jobs over time, meaning that low-value jobs will wait until the grid is quieter early in the morning or at weekends. PVR also reduces the complexity of the system by reducing the need for an admission controller, as it will starve tasks directly.

Chapter 8

Conclusion

As this thesis is written as part of the work of an Engineering Doctorate (*EngD*), engagement with and relevance to an industrial partner organisation is an important feature. The work of this thesis is based on a detailed case study of the grid and workload of the partner. This partner is a commercial aircraft manufacturer that is increasingly using Computational Fluid Dynamics (*CFD*) software instead of physical wind tunnels to aid in the process of aircraft design.

This thesis investigates two hypotheses regarding whether it is possible to improve the responsiveness, fairness and value of work for users relative to the industrial partner's existing grid management system. The approach to achieving these aims is to change the prioritisation in the organisation's list scheduling policy to something other than the currently used FairShare policy. This chapter will describe the contributions made by this thesis in the process of investigating these hypotheses. In addition, several avenues of future work that could extend or build on the work of this thesis are outlined.

8.1 Industrial Case Study

In order to satisfy the aims of the EngD project, a close relationship with the industrial partner was important. Discussions with the users of the industrial partner's grid system as described in Chapter 2 revealed that the performance of the currently-implemented scheduling policy known as FairShare is not fully satisfactory to users. The case study describes how although FairShare achieved short-term fairness in grid utilisation, the users were far more concerned about fairness with respect to the turnaround times of their jobs, or responsiveness.

A particular contribution of this work is the access to and characterisation of the workload run on an industrial grid used for engineering design. While many previous studies have characterised workloads executing on grid infrastructures, few have been able to access industrial as opposed to academically-oriented grids.

Furthermore, having a deep understanding of the context and motivations of the industrial partner is important in an EngD to be able to suggest scheduling improvements that are practical to implement as well as theoretically sound.

A key feature of the industrial workload surveyed is the particularly large range in execution times observed that spanned seven orders of magnitude (Chapter 4). This is much larger than the range of four orders of magnitude previously noted by Chiang and Vernon [40] and Feitelson and Nitzberg [57]. The distribution of execution times is also unusual, in that it closely followed a log-uniform distribution over six orders of magnitude. This means that there are a large number of small tasks that do not contribute much of the load, and a small number of large tasks that contribute a large fraction of the load. Previous work had found grid workloads following other distributions, such as the Weibull [40] or log-normal [92]. As internet bandwidth continues to increase it is likely that high performance computing (HPC) workloads will start to be run in the cloud. In some specialised cases, this is already happening [157]. Cloud providers will then have to deal with similar execution patterns and distributions as have been observed here.

A further distinctive feature of the grid is its cycles of overload during the working day which is only caught up on overnight and at the weekend. While these daily and weekly submission patterns have been observed before [40, 172], it is unusual for a grid to operate at or very close to saturation for such sustained periods.

Few analyses of dependency patterns from a graph-theoretic perspective have been done before, and all those found [32, 73, 99, 131, 140, 160] have considered dependencies within structured algorithms, rather than between independent pieces of software composed to form a workflow. This work found dependency graphs with a wide range of degrees, with many nodes having low degree and a few very highly-connected nodes.

Algorithms are given to generate synthetic workloads that reflect the characteristics of that observed in industry. These include dependency graphs (Algorithm 4.8), execution time distributions (Algorithm 4.3) and load levels that reflect the cycles of a production environment (Algorithm 4.2). These should be relevant to future researchers wishing to replicate the observed workloads and evaluate policies regarding various aspects of resource management within grids.

The visualisation and log analysis tools developed to enable the workload characterisation were also used for contributions relevant to the industrial partner. Firstly, they were used to automatically generate a visual 'dashboard' of some of the metrics currently used by the industrial partner. This helped replace a long manual process for extracting and plotting these indicators. Secondly, they were used to improve load balancing between the industrial clusters by estimating the priorities of tasks using the knowledge of the current state of the FairShare allocations on each cluster. This enables a better spread of users' work around the grid and hence

improves responsiveness for users' tasks as they reduce the likelihood that they will contend with each other for the same share on a cluster.

8.2 Evaluation Process

A distinct contribution of this thesis is to develop abstract models representing the industrial scenario that are also suitable for use in simulation. Chapter 5 composes a number of existing application, platform and scheduling models in order to develop a framework that is suitable for simulation of a grid. The application model uses multicore tasks connected with dependencies forming a Directed Acyclic Graph. The platform model consists of clusters connected with a tree-structured network. The scheduling model follows the list scheduling architecture for a dynamic workload that does not support pre-emption. The models were chosen to be rich enough to fairly represent the industrial scheduling problem, but of sufficiently low complexity so that the industrial grid and workload could be simulated at full scale.

These models were implemented programmatically so that the scheduling of workloads over a platform can be simulated. The simulator is able to use industrial workloads derived from logs as well as synthetic ones developed from the generation algorithms in Chapter 5. This simulator is able to produce a large number of metrics pertinent to the evaluation of scheduling policies. The simulation architecture is modular so that many scheduling policies can be evaluated without changing any other parameters of the simulation.

A survey is performed of the metrics with which scheduling policies can be evaluated in Chapter 5. A wide variety of metrics have been used to evaluate schedulers but few papers discuss why they select particular metrics. Furthermore, there have been few surveys of these metrics. A contribution of this work is to perform a survey of scheduling metrics in the literature and analyse their ability to give insight into the schedules they are applied to. The Schedule Length Ratio (*SLR*) metric from [160] is shown to give the most insight into responsiveness for online workloads with dependencies. This is because unlike other metrics, SLR takes into account the structure of the dependency graph and its critical path. A useful feature of SLR is that it is applied to each job in a workload. Therefore, fairness metrics can be applied to compare how SLR is distributed within a workload.

This conclusion of the metric survey informs the metrics that are used as part of the evaluation method. In addition, these metrics have been used to help the industrial partner understand and monitor their grid system better.

8.3 Scheduling for Responsiveness and Fairness

Scheduling policies are normally evaluated using a single metric for an entire workload. Considering the wide range of execution times in the workload, a novel approach in this work has been to consider how different scheduling policies affect the metrics of jobs across this range. It can then be seen which classes of jobs are prioritised or penalised by different policies.

Where such a large range of execution times is present, it is important that the whole range is treated fairly by a scheduler, according to an appropriate definition of fairness. Otherwise, particular classes of jobs may suffer starvation which can lead to user dissatisfaction. The currently implemented FairShare scheduler does not consider execution times, which means jobs will tend to wait for the same length of time to execute. This effectively prioritises the longer-running jobs over those that have very short running times, as the waiting times of longer jobs will be proportionately much lower.

SLR is shown to be a good metric to measure responsiveness, and the distribution of SLR to be good to measure fairness. Therefore, a novel scheduling policy called Projected-Schedule Length Ratio or *P-SLR* is proposed that attempts to optimise for these metrics and also be starvation-free. P-SLR works by using the upward ranks [160] of tasks to predict a finish time, and hence a projection of SLR. The tasks that are the most 'late' with respect to P-SLR are run first.

As explained in Chapter 6, the P-SLR scheduling policy is an important contribution of this thesis, because it demonstrates that it is possible to have a list scheduler that delivers responsiveness and fairness among jobs with a wide range of execution times while remaining starvation-free. This confirms the first hypothesis of this thesis.

The P-SLR scheduler's key advantage is that P-SLR is adaptive under overload, so that responsiveness suffers for all jobs equally. P-SLR is able to do this while having equal or better responsiveness and fairness metrics when compared to the best alternative policy evaluated, Shortest Remaining Time First (*SRTF*). This is the case throughout the range of network delays. P-SLR requires estimates of execution times to be given, although it is known that obtaining accurate estimates is still difficult and is a subject of active research [105, 107, 156]. The evaluation in this thesis showed that P-SLR is still competitive at responsiveness and fairness with the best alternative scheduler (SRTF) even when execution times were within the reasonable bounds of an order of magnitude of the actual execution time value.

The evaluation also demonstrates the strengths of the SRTF scheduling policy. In particular, where execution time estimate inaccuracies are large, it is able to provide higher responsiveness and fairness than P-SLR. It is able to do this because it only starves the few largest jobs under overload, leaving the vast majority of tasks in the

workload with high responsiveness. Nevertheless, the largest jobs are also often the most critical for overall project completion in the industrial context.

A key requirement expressed by the users in Chapter 2 was for a fair distribution of responsiveness across the workload. While SRTF achieves good responsiveness for the majority, it is less fair than P-SLR because of its tendency to starve a single extreme of the jobs in the workload. In that respect, P-SLR is more likely to be favoured by users for its balanced approach to starving work across the spectrum of execution times.

8.4 Scheduling for Value

Scheduling using P-SLR is suitable where jobs with similar critical path lengths have similar urgency. However, this may not always be the case. Instead, a model of encoding urgency relative to a job's response time using a non-increasing value curve is described in Chapter 7. A further significant contribution of this thesis is the novel list scheduling policy termed Projected Value Remaining or *PVR*. PVR is designed to achieve high workload value while respecting the urgency of tasks. It works in a similar way to P-SLR, by running tasks that are considered the 'most urgent' first. PVR prioritises tasks that have the least area remaining under their value curve. PVR is not starvation-free, but is instead designed to intentionally starve the least valuable tasks under overload.

PVR is shown to dominate the FairShare policy with respect to the value obtained from a workload across the spectra of load, networking delays and inaccurate estimates of execution times. This confirms the second hypothesis of this thesis.

In addition, PVR is shown to equal or dominate all other scheduling policies evaluated with respect to the proportion of maximum value achieved across the spectrum of load when penalties were applied on job starvation. PVR achieves the highest proportion of maximum value of all the evaluated policies when the grid is overloaded. It does this by ensuring that the responsiveness, and hence value, falls fairly across the range of job execution times. This enables it to have the most graceful degradation above saturation of any of the policies evaluated, fulfilling key user requirements from Chapter 2.

The application of value penalties when jobs starve is likely to be most similar to the industrial scenario, because of the inconvenience users will experience if their submitted jobs do not complete. An alternative model is to consider the loss of value incurred by not running a job to be a sufficient loss. Where penalties are not applied to starved jobs, PVR is equal to SRTF below saturation. At the point of saturation, PVR dominates, although above this, SRTF comes to dominate. For an industrial grid

that operates very close to saturation and where penalties are not applied for jobs that do not complete, PVR is likely to be the most applicable.

The highest value under significant overload is achieved by SRTF, which starves more of the largest tasks. Although the value is higher overall, the loss of fairness in responsiveness is likely to be in opposition to the users' desire for fair treatment of work. Nevertheless, an important result of this work is to show that SRTF performs well for value under overload and does so without requiring the specification of value curves or the necessary calculations of projected value at runtime.

PVR is dominant compared to the other alternative schedulers across the spectrum of networking delays, suggesting that it is suitable for the production environment where network delays can be large. PVR also performed well where errors in execution time estimates were small and normally distributed. However, PVR fell behind SRTF, PV and PVDSQ when errors were large or due to logarithmic rounding. As found in the evaluation of P-SLR, a further contribution of this work was to show that SRTF achieved the highest proportion of maximum value when errors in execution time estimation were large. All these policies outperformed FairShare in the value achieved across the scales of networking and inaccurate execution times, however. This adds further weight in confirming the second hypothesis of this thesis.

8.5 Future Work

The findings of this thesis were produced in simulation, as the evaluation of many scheduling policies including ones known to be suboptimal on a production grid was infeasible. Having concluded that the P-SLR and PVR policies show promise in simulation, an important area of future work would be to implement and evaluate them within production scheduling systems.

Evaluating the policies in production could address several limitations of the current approach. In a real system, the amount and kind of work users submit can depend on the responsiveness they receive from the cluster. As discussed in Chapter 2, if responsiveness of the grid workloads is improved, then users will submit more work because they can do more design cycles. This may mean that changing the scheduling policy may also have an impact on the workload. Still, although the quantity of work may rise in this situation, the distributions observed Chapter 4 are less likely to change. These distributions were developed from two and half years of logs and have held reasonably constant during that time.

A further limitation of the current approach is that the model of network delays is particularly designed for low-complexity execution in simulation, and may be overly simplistic in representing the real network delays experienced in a grid. Future work

could include extending the network model to consider queueing and contention on the network links, along with network topologies that do not form a tree.

Other work has considered using value curves that extend into negative values. Extending the model of value used in this thesis to consider these negative points would be a natural extension. This may provide a more nuanced evaluation than the approaches considered here where either a fixed or no penalty is applied to jobs that have passed their final deadline.

Where value curves are not available, future work could consider extending the model of P-SLR in a different way to introduce weightings to particular classes of jobs. This could be used instead of value to handle situations where, for example, small jobs need even higher responsiveness than large ones relative to their execution time. P-SLR values could also be weighted using user or group information, to intentionally prioritise the work of some users over others.

Balancing supply and demand of resources and work is a continual issue in many HPC contexts. It is especially important where responsiveness is important to users. A dynamic measure of cluster responsiveness would be to monitor the worst-case P-SLR of the work in a cluster's queue. Future work using this dynamic value could be applied in several areas of grid management that control supply and demand. Supply of resources could be managed by the scaling up or down of cloud resources in response to changes of the queue worst-case P-SLR. In an underloaded cluster, idle machines may be powered down to save energy as long as the worst-case P-SLR is maintained below a certain threshold. Alternatively, demand could be managed using admission control. If the worst-case P-SLR in the queue passed a certain value, an admission controller could limit new admissions to the grid until the peak in load had passed.

Availability of Source Code

The Python source code of the workload generator, the scheduling simulator and the result analysis software is available under the GNU General Public License version 3 and can be downloaded from https://github.com/andieburk/fastgridsim.

Definitions

This section contains definitions of any terms specific to the thesis, including abbreviations and codes used in illustrations.

Definitions

- **Allocation**: The stage of scheduling where work is assigned to resources.
- **Application Model**: The abstract model of the workload.
- **Batch Scheduler**: A static scheduler run repeatedly.
- **Clairvoyant Scheduling**: Scheduling where the entire workload is known in advance, usually assumed in static scheduling and impossible in dynamic scheduling.
- **Cloud Computing**: The model of purchasing computing power on-demand online
- **Critical Path**: The longest path through a job's dependencies, defines the shortest time the job could run in on an unloaded cluster of unbounded capacity.
- **Dependencies**: A model of where data is required to be passed from a completed task to another task that can only start once the data is received.
- **Dynamic Scheduling**: Scheduling where work arrives and must be processed continuously.
- **Grid Computing**: Distributed computing made up of cluster resources connected by a WAN.
- **High Performance Computing**: Computing platforms designed for scale and high performance.
- **Job**: A set of tasks and the dependencies between them. Is independent (has no dependencies) on any tasks external to the job.

- **List Scheduling**: A scheduling architecture that performs ordering and allocation separately.
- **Load**: The rate at which work arrives relative to the rate at which it can be processed.
- **Workload Makespan**: The time taken to execute all the workload on a given platform.
- **Multiple Waits Problem**: An issue where jobs have low responsiveness because a job ends up waiting the length of the queue multiple times because dependent tasks are only added to the back of a FIFO queue once their predecessors have completed.
- **Ordering**: The stage of scheduling involving prioritising and sorting a queue of tasks.
- **Platform Model**: An abstract model representing the hardware platform of a grid.
- **Router**: An abstract node in a tree-structured hierarchical network that performs load balancing.
- **Saturation**: Where load on the cluster reaches 100%.
- **Scheduling Model**: The abstract structure to represent the industrial scheduling process.
- **Starvation**: When jobs never complete, or fail to complete by a final deadline.
- **Static Scheduling**: Scheduling a batch of tasks that are all known all together at the same time.
- **Task**: An indivisible unit of work that takes a certain execution time, a number of cores and requires a given architecture on which to run.
- **Thin/Fat Tree**: Models of network bandwidth in a tree where leaves or roots have the greatest bandwidth available, respectively.
- **Upward Rank**: The sum of a task's execution time with the largest critical path of any of its successors.
- **Utilisation**: The fraction of available CPU time in a grid that is being used at a given moment.
- **Workflow**: A set of tasks with dependencies. Equivalent in meaning to a job.
- **Workload**: A set of jobs.

Abbreviations

- **CCR**: Communication to Computation Ratio
- **CFD**: Computational Fluid Dynamics
- **CP**: Critical Path
- **CPM**: Critical Path Method
- **CPU**: Central Processing Unit
- **DAG**: Directed Acyclic Graph
- **EngD**: Engineering Doctorate
- **FIFO**: First In First Out, Equivalent to FCFS: First Come First Served
- **FPGA**: Field Programmable Gate Array
- **GA**: Genetic Algorithm
- **GPU**: Graphics Processing Unit
- **GS**: Generational Scheduling
- **HPC**: High Performance Computing
- **LS**: List Scheduling
- **MPI**: Message Passing Interface
- **PC**: Personal Computer
- **pmf**: Probability Mass Function
- **QoS**: Quality of Service
- **RAM**: Random Access Memory
- **SA**: Simulated Annealing
- **SLA**: Service Level Agreement
- **SLR**: Schedule Length Ratio
- **WAN**: Wide Area Network

Scheduling Policies

For Ordering

- **FairShare**: The industrial FairShare policy (see Section 2.2)
- **FIFO Job**: First In First Out by Job (see Section 6.2.3)
- **FIFO Task**: First In First Out by Task (see Section 6.2.2)
- **LRTF**: Longest Remaining Time First (see Section 6.2.5)
- **P-SLR**: Projected Schedule Length Ratio (see Section 6.1)
- **PV**: Projected Value (see Section 7.4.1)
- **PVCPD**: Projected Value Critical Path Density (see Section 7.4.3)
- **PVD**: Projected Value Density (see Section 7.4.2)
- **PVDSQ**: Projected Value Density Squared (see Section 7.4.4)
- **PVR**: Projected Value Remaining (see Section 7.4.5)
- **Random**: Random Ordering (see Section 6.2.1)
- **SRTF**: Shortest Remaining Time First (see Section 6.2.5)

For Allocation

- **EFT**: Earliest Finish Time (see Section 3.2.1)
- **EST**: Earliest Start Time (see Section 3.2.1)
- **HEFT**: Heterogeneous Earliest Finish Time (see Section 3.3.4)

List of References

[1] Advanced Micro Devices, Inc. AMD Radeon R9 series graphics specifications, November 2013. URL `http://www.amd.com/uk/products/desktop/graphics/r9/Pages/amd-radeon-hd-r9-series.aspx#5`.

[2] Ishfaq Ahmad and Yu-Kwong Kwok. A comparison of task-duplication-based algorithms for scheduling parallel programs to message-passing systems. In *Proceedings of the 11th International Symposium on High-Performance Computing Systems (HPCS'97)*, pages 39–50, 1997.

[3] Susanne Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, October 1999. ISSN 0097-5397. doi: 10.1137/S0097539797324874. URL `http://dx.doi.org/10.1137/S0097539797324874`.

[4] Saud A. Aldarmi and Alan Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of The 11th Euromicro Conference on Real-Time Systems*, pages 270–277, June 1999. doi: 10.1109/EMRTS.1999.777474.

[5] Fernando L. Alvarado. Parallel solution of transient problems by trapezoidal integration. *IEEE Transactions on Power Apparatus and Systems*, PAS-98(3):1080–1090, 1979. ISSN 0018-9510. doi: 10.1109/TPAS.1979.319271.

[6] Amazon.com. Amazon elastic compute cloud (EC2) pricing, 2010. URL `http://aws.amazon.com/ec2/`.

[7] Esther Andrés, Carlos Carreras, Gabriel Caffarena, Maria del Carmen Molina, Octavio Nieto-Taladriz, and Francisco Palacios. A methodology for CFD acceleration through reconfigurable hardware. In *Proceedings of the 46th AIAA Aerospace Sciences Meeting and Exhibit, ASME'08*. American Institute of Aeronautics and Astronautics, 2008. URL `http://oa.upm.es/4313/1/INVE_MEM_2008_59731.pdf`.

[8] Christian Anhalt, Hans Peter Monner, and Elmar Breitbach. Interdisciplinary Wing Design - Structural Aspects. SAE International, German Aerospace Center (DLR), Institute of Structural Mechanics, Lilienthalplatz 7, 38108

Braunschweig, Germany, 2003. URL http://www.dlr.de/fa/en/portaldata/17/resources/dokumente/publikationen/2003/01_anhalt.pdf.

[9] Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. University of Wisconsin-Madison Department of Computer Sciences, v0.6 edition, August 2013.

[10] Nikhil Bansal and Kirk R. Pruhs. Server scheduling to balance priorities, fairness, and average quality of service. *SIAM Journal on Computing*, 39(7):3311–3335, 2010.

[11] Albert-Laszlo Barabasi and Eric Bonabeau. Scale-free networks. *Scientific American*, 288:60–69, May 2003.

[12] Colin Barker. Data centre energy crisis looms, October 2006. URL http://www.zdnet.com/data-centre-energy-crisis-looms-3039284324/.

[13] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 35–46, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-914-5. doi: 10.1145/1730836.1730842. URL http://doi.acm.org/10.1145/1730836.1730842.

[14] Sanjoy K. Baruah. The multiprocessor scheduling of precedence-constrained task systems in the presence of interprocessor communication delays. *Operations Research*, 46(1):pp. 65–72, 1998. ISSN 0030364X. URL http://www.jstor.org/stable/223063.

[15] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 270–279, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9. URL http://dl.acm.org/citation.cfm?id=314613.314715.

[16] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Improved algorithms for stretch scheduling. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 762–771, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X. URL http://dl.acm.org/citation.cfm?id=545381.545482.

[17] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. ISSN 0922-6443. doi: 10.1007/s11241-005-0507-9.

[18] Luiz F. Bittencourt and Edmundo R. M. Madeira. Towards the scheduling of multiple workflows on computational grids. *Journal of Grid Computing*, 8(3): 419–441, 2010. ISSN 1570-7873. doi: 10.1007/s10723-009-9144-1. URL `http://dx.doi.org/10.1007/s10723-009-9144-1`.

[19] Luiz F. Bittencourt, Edmundo R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. A path clustering heuristic for scheduling taks graphs onto a grid. In *Proceedings of the 3rd ACM International Workshop on Middleware for Grid Computing, Grenoble, France*, Nov 2005.

[20] Wayne F. Boyer and Gurdeep S. Hura. Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments. *Journal of Parallel and Distributed Computing*, 65(9):1035–1046, September 2005. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2005.04.017.

[21] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölóni, Albert I. Reuther, Mitchell D. Theys, Bin Yao, Richard F. Freund, Muthucumaru Maheswaran, James P. Robertson, and Debra Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, page 15, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0107-9.

[22] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61:810–837, June 2001. ISSN 0743-7315. doi: 10.1006/jpdc.2000.1714. URL `http://dl.acm.org/citation.cfm?id=511973.511979`.

[23] Tracy D. Braun, Howard Jay Siegel, Anthony A. Maciejewski, and Ye Hong. Static resource allocation for heterogeneous computing environments with tasks having dependencies, priorities, deadlines, and multiple versions. *Journal of Parallel and Distributed Computing*, 68(11):1504–1516, 2008. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2008.06.006.

[24] James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6(3):255–276, 2008. ISSN 1570-7873. doi: 10.1007/s10723-007-9095-3. URL `http://dx.doi.org/10.1007/s10723-007-9095-3`.

[25] Edmund K. Burke, Moshe Dror, and James B. Orlin. Scheduling malleable tasks with interdependent processing rates: Comments and observations. *Discrete Applied Mathematics*, 156(5):620 – 626, 2008. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/j.dam.2007.08.008. URL http://www.sciencedirect.com/science/article/pii/S0166218X07003526.

[26] Andrew Burkimsher. Dependency patterns and timing for grid workloads. In *Proceedings of the 4th York Doctoral Symposium on Computer Science*, pages 25–33, October 2011. URL http://www.cs.york.ac.uk/ftpdir/reports/2011/YCS/468/YCS-2011-468.pdf.

[27] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. *Future Generation Computer Systems*, 29(8):2009 – 2025, October 2013. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.12.005. URL http://www.sciencedirect.com/science/article/pii/S0167739X12002257.

[28] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. Scheduling HPC workflows for responsiveness and fairness with networking delays and inaccurate estimates of execution times. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par 2013)*, volume 8097 of *Lecture Notes in Computer Science*, pages 126–137. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_15. URL http://dx.doi.org/10.1007/978-3-642-40047-6_15.

[29] Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak. A characterisation of the workload on an engineering design grid. In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 8:1–8:8, San Diego, CA, USA, 2014. Society for Computer Simulation International. URL http://dl.acm.org/citation.cfm?id=2663510.2663518.

[30] Alan Burns, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John A. Stankovic, and Lorenzo Strigini. The meaning and role of value in scheduling flexible real-time systems. *Journal of systems architecture*, 46(4):305–325, 2000. ISSN 1383-7621.

[31] Giorgio C. Buttazzo and John A. Stankovic. RED: Robust earliest deadline scheduling. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, Austin, 1993.

[32] Haijun Cao, Hai Jin, Xiaoxin Wu, Song Wu, and Xuanhua Shi. DAGMap: efficient and dependable scheduling of DAG workflow job in grid. *The*

Journal of Supercomputing, 51(2):201–223, 2010. ISSN 0920-8542. doi: 10.1007/s11227-009-0284-7.

[33] Junwei Cao, S.A. Jarvis, S. Saini, and Graham R. Nudd. Gridflow: workflow management for grid computing. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 198–205, 2003. doi: 10.1109/CCGRID.2003.1199369.

[34] Brent R. Carter, Daniel W. Watson, Richard F. Freund, Elaine Keith, Francesca Mirabile, and Howard Jay Siegel. Generational scheduling for dynamic task management in heterogeneous computing systems. *Information Sciences*, 106 (3-4):219–236, 1998.

[35] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.4634.

[36] Steve J. Chapin. Distributed and multiprocessor scheduling. *ACM Computing Surveys*, 28(1):233–235, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/234313.234410.

[37] Ken Chen and Paul Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems*, 10(3):293–312, 1996. ISSN 0922-6443. doi: 10.1007/BF00383389. URL `http://dx.doi.org/10.1007/BF00383389`.

[38] Tingwei Chen, Bin Zhang, and Xianwen Hao. A dependent tasks scheduling model in grid. In *Proceedings of the 10th Asia-Pacific web conference on Progress in WWW research and development*, APWeb '08, pages 136–147, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78848-4, 978-3-540-78848-5.

[39] T.C. Edwin Cheng and Qing Ding. Scheduling start time dependent tasks with deadlines and identical initial processing times on a single machine. *Computers and Operations Research*, 30(1):51 – 62, 2003. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/S0305-0548(01)00077-6. URL `http://www.sciencedirect.com/science/article/pii/S0305054801000776`.

[40] Su-Hui Chiang and Mary K. Vernon. Characteristics of a large shared memory production workload. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 159–187. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42817-6. URL `http://dx.doi.org/10.1007/3-540-45540-X_10`.

[41] Edgar Frank Codd. Multiprogram scheduling: parts 1 and 2. introduction and theory. *Communications of the ACM*, 3(6):347–350, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367297.367317.

[42] Edgar Frank Codd. Multiprogram scheduling: parts 3 and 4. scheduling algorithm and external constraints. *Communications of the ACM*, 3(7):413–418, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367349.367356.

[43] Gary N. Coleman and Richard D. Sandberg. A primer on direct numerical simulation of turbulence - methods, procedures and guidelines. Aerodynamics & Flight Mechanics Research Group, School of Engineering Sciences, University of Southampton, March 2010. URL `http://eprints.soton.ac.uk/66182/1/A_primer_on_DNS.pdf`.

[44] Jean-Yves Colin and Philippe Chrétienne. C.P.M. scheduling with small communication delays and task duplication. *Operations Research*, 39(4):680–684, July-August 1991.

[45] David E. Collins and Alan D. George. Parallel and sequential job scheduling in heterogeneous clusters: A simulation study using software in the loop. *Simulation*, 77(5-6):169–184, November 2001.

[46] Concentration, Heat and Momentum (CHAM) Limited. Phoenics encyclopaedia: What CFD can and cannot do. URL `http://www.cham.co.uk/phoenics/d_polis/d_info/cfdcan.htm`.

[47] Malcolm J. Cook. *An evaluation of computational fluid dynamics for modelling buoyancy-driven displacement ventilation*. PhD thesis, De Montfort University, 1998.

[48] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 2009. ISBN 9780262033848.

[49] Mohammad I. Daoud and Nawwaf Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68(4):399 – 409, April 2008. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2007.05.015. URL `http://www.sciencedirect.com/science/article/pii/S0743731507000834`.

[50] Robert I. Davis and Alan Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 0:398–409, 2009. ISSN 1052-8725. doi: 10.1109/RTSS.2009.31.

[51] Muhammad K. Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 62(9):1338 – 1361, 2002. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.2002.1850. URL `http://www.sciencedirect.com/science/article/pii/S0743731502918502`.

[52] Sofia K. Dimitriadou and Helen D. Karatza. Job scheduling in a distributed system using backfilling with inaccurate runtime computations. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 329–336, February 2010. doi: 10.1109/CISIS.2010.65.

[53] Nicolas Dube and Marc Parizeau. Utility computing and market-based scheduling: Shortcomings for grid resources sharing and the next steps. In *Proceedings of the 22nd International Symposium on High Performance Computing Systems and Applications, 2008*, HPCS 2008, pages 59–68, jun. 2008. doi: 10.1109/HPCS.2008.29.

[54] Paul Emberson. *Searching For Flexible Solutions To Task Allocation Problems*. PhD thesis, University of York, UK, 2009. URL `http://www.cs.york.ac.uk/rts/documents/thesis/emberson09.pdf`.

[55] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. In *Publication Of The Mathematical Institute Of The Hungarian Academy Of Sciences*, volume 5, pages 17–61, 1960.

[56] Liya Fan, Fa Zhang, Gongming Wang, and Zhiyong Liu. An effective approximation algorithm for the malleable parallel task scheduling problem. *Journal of Parallel and Distributed Computing*, 72(5):693 – 704, 2012. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2012.01.011. URL `http://www.sciencedirect.com/science/article/pii/S0743731512000238`.

[57] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60153-1. doi: 10.1007/3-540-60153-8_38. URL `http://dx.doi.org/10.1007/3-540-60153-8_38`.

[58] Dror G. Feitelson and Edi Shmueli. A case for conservative workload modeling: Parallel job scheduling with daily cycles of activity. In *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer*

and Telecommunication Systems, 2009., MASCOTS '09, pages 1–8, 2009. doi: 10.1109/MASCOT.2009.5366139.

[59] David Fernández-Baca. Allocating modules to processors in a distributed system. *Software Engineering, IEEE Transactions on*, 15(11):1427 –1436, November 1989. ISSN 0098-5589. doi: 10.1109/32.41334.

[60] John P. Fielding. *Introduction to Aircraft Design*. Cambridge Aerospace Series. Cambridge University Press, October 1999. ISBN 9780521657228.

[61] Kozo Fujii. Progress and future prospects of CFD in aerospace - wind tunnel and beyond. *Progress in Aerospace Sciences*, 41(6):455 – 470, 2005. ISSN 0376-0421. doi: http://dx.doi.org/10.1016/j.paerosci.2005.09.001. URL `http://www.sciencedirect.com/science/article/pii/S0376042105001016`.

[62] Fujitsu Systems (Europe) Limited. Synfiniway technical overview, January 2005. URL `http://www.fujitsu.com/downloads/EU/uk/whitepapers/synfiniwaytechnical.pdf`.

[63] Sean Gallagher. General motors is literally tearing its competition to bits ... so its 3D scanning can reverse-engineer others' vehicles, increasing speed to market., September 2013. URL `http://arstechnica.com/information-technology/2013/09/general-motors-is-literally-tearing-its-competition-to-bits/`.

[64] Michael Randolph Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975. doi: 10.1137/0204035.

[65] Michael Randolph Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY, USA, 1990.

[66] Michael Randolph Garey, Ronald L. Graham, and David S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26 (1):3–21, January/February 1978. doi: 10.1287/opre.26.1.3. URL `http://or.journal.informs.org/cgi/content/abstract/26/1/3`.

[67] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276 – 291, 1992. ISSN 0743-7315. doi: DOI:10.1016/0743-7315(92)90012-C. URL `http://www.sciencedirect.com/science/\article/B6WKJ-4BRJJ23-2S/2/fc1925064e66c33d6dd85b414435a3af`.

[68] Hashem Ali Ghazzawi, Iain Bate, and Leandro Soares Indrusiak. MPC vs. PID controllers in Multi-CPU multi-objective real-time scheduling systems. In *Proceedings of The 2012 UK Electronics Forum*, pages 77–83, August 2012.

[69] Ian Godfrey. Airbus selects SynfiniWay from Fujitsu to provide grid computing environment for aerodynamics analysis, July 2006. URL `http://www.fujitsu.com/uk/news/pr/2006/20060711.html`.

[70] Google Incorporated. Purchasing clean energy, October 2012. URL `http://www.google.com/green/energy/use/#purchasing`.

[71] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[72] Tarek Hagras and Jan Janeček. Static vs. dynamic list-scheduling performance comparison. *Acta Polytechnica*, 43(6):16–21, 2003.

[73] Robert Hall, Arnold L. Rosenberg, and Arun Venkataramani. A comparison of DAG-scheduling strategies for internet-based computing. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–9, 2007. doi: 10.1109/IPDPS.2007.370245.

[74] Claire Hanen and Alix Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. *Discrete Applied Mathematics*, 108(3):239 – 257, 2001. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/S0166-218X(00)00179-7. URL `http://www.sciencedirect.com/science/article/pii/S0166218X00001797`.

[75] Amir Hassine and Erich Barke. On modeling and simulating chip design processes: The RS model. In *IEEE International Engineering Management Conference, IEMC Europe*, pages 1–5, 2008. doi: 10.1109/IEMCE.2008.4617958.

[76] Jeffrey W. Herrmann. A history of production scheduling. In Jeffrey W. Herrmann, editor, *Handbook of Production Scheduling*, volume 89 of *International Series in Operations Research & Management Science*, pages 1–22. Springer US, 2006. ISBN 978-0-387-33115-7. doi: 10.1007/0-387-33117-4_1. URL `http://dx.doi.org/10.1007/0-387-33117-4_1`.

[77] Adán Hirales-Carbajal, Andrei Tchernykh, Ramin Yahyapour, José Luis González-García, Thomas Röblitz, and Juan Manuel Ramírez-Alcaraz. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10(2):325–346, 2012. ISSN 1570-7873. doi: 10.1007/s10723-012-9215-6. URL `http://dx.doi.org/10.1007/s10723-012-9215-6`.

[78] Arie Hordijk and Flos Spieksma. Constrained admission control to a queueing system. *Advances in Applied Probability*, 21(2):409 – 431, June 1989. URL http://www.jstor.org/stable/1427167.

[79] Naim Hossain. Modeling thermal turbulence using implicit large eddy simulation. Master's thesis, Universitat Politecnica de Catalunya, June 2012. URL http://upcommons.upc.edu/pfc/bitstream/2099.1/15782/1/MSc_thesis_MD_Naim_Hossain.pdf.

[80] Miaoqing Huang, Harald Simmler, Olivier Serres, and Tarek El-Ghazawi. RDMS: A hardware task scheduling algorithm for reconfigurable computing. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, 2009*, IPDPS 2009, pages 1–8, may. 2009. doi: 10.1109/IPDPS.2009.5161223.

[81] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, and the matplotlib development team. The matplotlib api » pyplot, Oct 2014. URL http://matplotlib.org/api/pyplot_api.html.

[82] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/0218016.

[83] Intel Corporation. Enhanced Intel SpeedStep technology for the Intel Pentium M processor - white paper, March 2004. URL ftp://download.intel.com/design/network/papers/30117401.pdf.

[84] Intel Corporation. Intel processor comparison, November 2013. URL http://www.intel.com/content/www/us/en/processor-comparison/compare-intel-processors.html.

[85] International Business Machines, Inc. IBM Platform LSF, 2013. URL http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/.

[86] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04:, pages 160–169, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-2175-4. doi: http://dx.doi.org/10.1109/HPDC.2004.5.

[87] Michael A. Iverson and Füsun Özgüner. Hierarchical, competitive scheduling of multiple DAGs in a dynamic heterogeneous environment. *Distributed Systems Engineering*, 6(3):112, 1999. URL http://stacks.iop.org/0967-1846/6/i=3/a=303.

[88] Jens Jägersküpper and Christian Simmendinger. A novel shared-memory thread-pool implementation for hybrid parallel CFD solvers. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23396-8. doi: 10.1007/978-3-642-23397-5_18. URL `http://dx.doi.org/10.1007/978-3-642-23397-5_18`.

[89] Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. *Journal of Computer and System Sciences*, 78(1):245 – 259, 2012. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/j.jcss.2011.04.003. URL `http://www.sciencedirect.com/science/article/pii/S0022000011000481`.

[90] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA (RTSS)*, pages 112–122. IEEE Computer Society, 1985.

[91] David Karger, Cliff Stein, and Joel Wein. Algorithms and theory of computation handbook: Special topics and techniques. chapter 20. Scheduling Algorithms, pages 1–34. Chapman and Hall, 2010. ISBN 978-1-58488-820-8.

[92] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production MapReduce cluster. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–103, 2010. doi: 10.1109/CCGRID.2010.112.

[93] Judy Kay and Piers Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988. URL `http://www.cs.cornell.edu/courses/cs614/2003sp/papers/KL89.pdf`.

[94] James E. Kelley, Jr. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):pp. 296–320, 1961. ISSN 0030364X. URL `http://www.jstor.org/stable/167563`.

[95] Omar Kermia and Yves Sorel. A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In *Proceedings of the ISCA 20th International Conference on Parallel and Distributed Computing Systems*, PDCS '07, Las Vegas, Nevada, USA, 2007. URL `http://hal.inria.fr/inria-00413486`.

[96] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, USA, November 1999. ISBN 1558604758.

[97] A.A. Khan, Carolyn L. McCreary, and Mary S. Jones. A comparison of multiprocessor scheduling heuristics. In *International Conference on Parallel Processing, 1994*, volume 2, pages 243–250, August 1994. doi: 10.1109/ICPP.1994.19.

[98] Dalibor Klusáček and Hana Rudová. Performance and fairness for users in parallel job scheduling. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 235–252. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35866-1. doi: 10.1007/978-3-642-35867-8_13. URL http://dx.doi.org/10.1007/978-3-642-35867-8_13.

[99] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996. ISSN 1045-9219. doi: 10.1109/71.503776.

[100] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381 – 422, 1999. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1999.1578. URL http://www.sciencedirect.com/science/article/pii/S0743731599915782.

[101] Kevin Lai. Markets are dead, long live markets. *ACM SIGecom Exchanges*, 5(4):1–10, July 2005. ISSN 1551-9031. doi: 10.1145/1120717.1120719. URL http://doi.acm.org/10.1145/1120717.1120719.

[102] Tian Lan, D. Kao, Mung Chiang, and A. Sabharwal. An axiomatic theory of fairness in network resource allocation. In *Proceedings of the IEEE INFOCOM*, pages 1–9, 2010. doi: 10.1109/INFCOM.2010.5461911.

[103] Sven Lanzan. Personal Communication, November 2011.

[104] Barry G. Lawson, Evgenia Smirni, and Daniela Puiu. Self-adapting backfilling scheduling for parallel systems. In *Proceedings of the International Conference on Parallel Processing*, pages 583–592, 2002. doi: 10.1109/ICPP.2002.1040916.

[105] Aleksandar Lazarević. *Autonomous grid scheduling using probabilistic job runtime scheduling*. PhD thesis, Department of Electronic and Electrical Engineering, University College London, University of London, 2008.

[106] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jeff Hansen. A scalable solution to the multi-resource QoS problem. In *Proceedings*

of the 20th IEEE Real-Time Systems Symposium (RTSS '99), pages 315–326, Washington, DC, USA, 1999. IEEE Computer Society. doi: 10.1109/REAL.1999.818859.

[107] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25330-3. doi: 10.1007/11407522_14. URL `http://dx.doi.org/10.1007/11407522_14`.

[108] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985. ISSN 0018-9340.

[109] Hui Li, David Groep, and Lex Wolters. Workload characteristics of a multi-cluster supercomputer. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 176–193. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25330-3. doi: 10.1007/11407522_10. URL `http://dx.doi.org/10.1007/11407522_10`.

[110] Peng Li and Binoy Ravindran. Fast, best-effort real-time scheduling algorithms. *IEEE Transactions on Computers*, 53(9):1159–1175, 2004. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2004.61.

[111] Yu Liang and Zhou Jiliu. The improvement of a task scheduling algorithm in grid computing. In *Proceedings of the First International Symposium on Data, Privacy, and E-Commerce*, volume 0 of *ISDPE 2007*, pages 292 –297, Los Alamitos, CA, USA, nov. 2007. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ISDPE.2007.17.

[112] Julie A. Litchfield. Inequality methods and tools, March 1999. URL `http://siteresources.worldbank.org/INTPGI/Resources/Inequality/litchfie.pdf`. Text for World Bank's Web Site on Inequality, Poverty, and Socio-economic Performance: http://www.worldbank.org/poverty/inequal/index.htm.

[113] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, USA, April 2000. ISBN 0130996513.

[114] Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384 –1397, November 1988. ISSN 0018-9340. doi: 10.1109/12.8704.

[115] Carey Douglass Locke. *Best-effort decision-making for real-time scheduling*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, May 1986.

[116] Muthucumaru Maheswaran and Howard Jay Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, HCW '98, page 57, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8365-1.

[117] Muthucumaru Maheswaran, Tracy D. Braun, and Howard Jay Siegel. Heterogeneous distributed computing. In *In Encyclopedia of Electrical and Electronics Engineering*, pages 679–690. John Wiley, 1999.

[118] Niladri Mandal, Manishl Malpani, and K. Ramesh Kumar. Wind tunnel model - fabrication challenges. *International Journal of Applied Research In Mechanical Engineering (IJARME)*, 1(2):22–25, 2011. URL `http://www.idc-online.com/technical_references/pdfs/mechanical_engineering/Wind%20Tunnel%20Model.pdf`.

[119] Graham Markall. Accelerating unstructured mesh computational fluid dynamics on the NVidia Tesla GPU architecture. Master's thesis, Imperial College London, 2009.

[120] Carolyn L. McCreary, A. A. Khan, J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 446–451, April 1994. doi: 10.1109/IPPS.1994.288264.

[121] Lois C. McInnes, Boyana Norris, and Ivana Veljkovic. Computational quality of service in parallel CFD. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 60439-4844, Argonne, IL; Department of Computer Science and Engineering, The Pennsylvania State University; IST Building, 16802-6106, PA, 2012.

[122] Rich Miller. Special report: The world's largest data centers. April 2010. URL `http://www.datacenterknowledge.com/special-report-the-worlds-largest-data-centers/`.

[123] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965. doi: 10.1109/JPROC.1998.658762. URL `http://dx.doi.org/10.1109/JPROC.1998.658762`.

[124] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. ISSN 1045-9219. doi: 10.1109/71.932708.

[125] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes E. Gehrk. Online scheduling to minimize average stretch. *SIAM Journal on Computing*, 34(2):433–452, 2005.

[126] Javier Navaridas, Jose Miguel-Alonso, Francisco Javier Ridruejo, and Wolfgang Denzel. Reducing complexity in tree-like computer interconnection networks. *Parallel Computing*, 36(2-3):71–85, 2010. ISSN 0167-8191. doi: DOI:10.1016/j.parco.2009.12.004. URL http://www.sciencedirect.com/science/article/\B6V12-4Y1MRPG-2/2/27aa5554ff69bfd5adb984e77d6b2283.

[127] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, January 1993. ISBN 0125184050.

[128] Jakob Nielsen. Nielsen's law of internet bandwidth, April 1998. URL http://www.nngroup.com/articles/law-of-bandwidth/.

[129] Roman Nossal. An evolutionary approach to multiprocessor scheduling of dependent tasks. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 279–287. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64359-3. doi: 10.1007/3-540-64359-1_698. URL http://dx.doi.org/10.1007/3-540-64359-1_698.

[130] NVIDIA Corporation. GeForce GTX TITAN specifications, November 2013. URL http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-titan/specifications.

[131] Alexandra Olteanu and Andreea Marin. Generation and evaluation of scheduling DAGs: How to provide similar evaluation conditions. *Computer Science Master Research*, 1(1), 2011. ISSN 2247-5575. URL http://csmr.cs.pub.ro/index.php/csmr/article/view/27.

[132] Fatma A. Omara and Mona M. Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13 – 22, January 2010. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2009.09.009. URL http://www.sciencedirect.com/science/article/pii/S0743731509001804.

[133] Oracle Corporation. N1 Grid Engine 6 administration guide - configuring the share-based policy, 2010. URL http://docs.oracle.com/cd/E19080-01/n1.grid.eng6/817-5677/i999588/index.html.

[134] Oracle Corporation and Sun Microsystems. Oracle Grid Engine, 2011. URL http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

[135] Phoenix Integration, Inc. PHX ModelCenter, 2013. URL http://www.phoenix-int.com/software/phx-modelcenter.php.

[136] Platform Computing Corporation. Platform LSF version 6.1 - administering Platform LSF - FairShare scheduling, June 2006. URL http://www-cecpv.u-strasbg.fr/Documentations/lsf/html/lsf6.1_admin/E_fairshare.html.

[137] Platform Computing Corporation. FairShare scheduling, 2008. URL http://www.cisl.ucar.edu/docs/LSF/7.0.3/admin/fairshare.html#wp215541.

[138] Platform Computing Corporation. Platform LSF: The HPC workload management standard. Online, 2011. URL http://www.platform.com/workload-management/high-performance-computing/lp.

[139] Nicolas Poggi, David Carrera, Ricard Gavalda, Jordi Torres, and Eduard Ayguade. Characterization of workload and resource consumption for an online travel and booking site. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-9297-8. doi: 10.1109/IISWC.2010.5649408.

[140] Samantha Ranaweera and Dharma P. Agrawal. A scalable task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 383–390, 2000. doi: 10.1109/ICPP.2000.876154.

[141] Zujie Ren, Jian Wan, Weisong Shi, Xianghua Xu, and Min Zhou. Workload analysis, implications and optimization on a production hadoop cluster: A case study on taobao. *IEEE Transactions on Services Computing*, 99:1, 2013. ISSN 1939-1374. doi: http://doi.ieeecomputersociety.org/10.1109/TSC.2013.40.

[142] Owen Rogers and Dave Cliff. Forecasting demand for cloud computing resources - an agent-based simulation of a two tiered approach. In Joaquim Filipe and Ana L. N. Fred, editors, *ICAART (2)*, pages 106–112. SciTePress, 2012. ISBN 978-989-8425-96-6. URL http://http://lscits.cs.bris.ac.uk/docs/ICAART%20FINAL.pdf.

[143] Gerald Sabin, Matthew Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 4376 of *Lecture Notes in Computer Science*, pages 94–114. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-71034-9. doi: 10.1007/978-3-540-71035-6_5. URL http://dx.doi.org/10.1007/978-3-540-71035-6_5.

[144] Mohammad H. Sadraey. *Wing Design*, chapter 5, pages 161–264. John Wiley & Sons, Ltd, 2012. ISBN 9781118352700. doi: 10.1002/9781118352700.ch5. URL http://dx.doi.org/10.1002/9781118352700.ch5.

[145] Ronaldo M. Salles and Javier A. Barria. Utility-based scheduling disciplines for adaptive applications over the internet. *IEEE Communications Letters*, 6(5): 217–219, 2002. ISSN 1089-7798. doi: 10.1109/4234.1001669.

[146] Peter Sanders and Jochen Speck. Efficient parallel scheduling of malleable tasks. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 1156–1166, 2011. doi: 10.1109/IPDPS.2011.110.

[147] Erik Saule, Doruk Bozdağ, 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. ISBN 978-3-642-16504-7. doi: 10.1007/978-3-642-16505-4_6. URL http://dx.doi.org/10.1007/978-3-642-16505-4_6.

[148] Arjen Schoneveld, Jan F. de Ronde, and Peter M. A. Sloot. On the complexity of task allocation. *Complexity*, 3(2):52–60, 1997. ISSN 1099-0526. URL http://dx.doi.org/10.1002/(SICI)1099-0526(199711/12)3:2<52::AID-CPLX12>3.0.CO;2-R.

[149] Daniel Schulze, Urs Baumgartl, and Tim Onnenberg (Voith Engineering Services). CFD TAU applications within the Airbus aerodynamic design process. In *Proceedings of the TAU User Meeting, October 18 + 19, 2011, DLR Braunschweig (http://tau.dlr.de/Usermeeting/)*, October 2011. URL http://tau.dlr.de/fileadmin/Talks-website/02_Schulze/TauUserMeeting_Oct2011_Voith.pdf.

[150] Dieter Schwamborn, Thomas Gerhold, and Ralf Heinrich. The DLR TAU-Code: Recent applications in research and industry. In P. Wesseling, E. O nate, and J. Périaux, editors, *Proceedings of the European Conference on Computational Fluid Dynamics (ECCOMAS CFD)*, 2006.

[151] Behrooz Shirazi, Mingfang Wang, and Girish Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10(3):222 – 232, 1990. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/0743-7315(90)90014-G. URL http://www.sciencedirect.com/science/article/pii/074373159090014G.

[152] Pankaj Shroff, Daniel W. Watson, Nicholas S. Flann, and Richard F. Freund. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *Proceedings of the Heterogeneous Computing Workshop*, pages 98–104, April 1996.

[153] M. G. Siegler. Apple's billion dollar data center will be done this year. iTunes in the cloud, anyone?, July 2010. URL `http://techcrunch.com/2010/07/20/apple-data-center/`.

[154] Ravi S. Singh, Anil K. Tripathi, Saket Saurabh, and V. Singh. Duplication based list scheduling in heterogeneous distributed computing. *IJCA Proceedings on National Conference on Advancement of Technologies - Information Systems & Computer Networks (ISCON - 2012)*, ISCON(1):24–28, May 2012. Published by Foundation of Computer Science, New York, USA.

[155] Omer Ozan Sonmez and Attila Gursoy. A novel economic-based scheduling heuristic for computational grids. *International Journal of High Performance Computing Applications*, 21(1):21–29, 2007. ISSN 1094-3420. doi: http://dx.doi.org/10.1177/1094342006074849.

[156] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 111–120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi: 10.1145/1551609.1551632. URL `http://doi.acm.org/10.1145/1551609.1551632`.

[157] Jason Stowe. Back to the future: 1.21 petaFLOPS (RPeak), 156,000-core CycleCloud HPC runs 264 years of materials science, November 2013. URL `http://blog.cyclecomputing.com/2013/11/back-to-the-future-121-petaflopsrpeak-156000-core-cyclecloud/hpc-runs-264-years-of-materials-science.html`.

[158] Prasanna V. Sugavanam, Howard J. Siegel, Anthony A. Maciejewski, Syed Amjad Ali, Mohammad Al-Otaibi, Mahir Aydin, Kumara Guru, Aaron Horiuchi, Yogish G. Krishnamurthy, Panho Lee, Ashish Mehta, Mohana Oltikar, Ron Pichel, Alan J. Pippin, Michael Raskey, Vladimir Shestak, and Junxing Zhang. Processor allocation for tasks that is robust against errors in computation time estimates. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2:122a, 2005. ISSN 1530-2075. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2005.362.

[159] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):

379–394, 2002. ISSN 1099-1425. doi: 10.1002/jos.116. URL http://dx.doi.org/10.1002/jos.116.

[160] Haluk Topcuouglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.993206.

[161] Damien Tromeur-Dervout, Gunther Brenner, David R. Emerson, and Jocelyne Erhel, editors. *Parallel Computational Fluid Dynamics 2008: Parallel Numerical Methods, Software Development and Applications*, volume 74 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2010.

[162] Spyros Tzafestas, Alekos Triantafyllakis, and George Rizos. Scheduling dependent tasks on identical machines using a novel heuristic criterion: A robotic computation example. *Journal of Intelligent and Robotic Systems*, 12 (3):229–237, 1995. ISSN 0921-0296. doi: 10.1007/BF01262962. URL http://dx.doi.org/10.1007/BF01262962.

[163] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/S0022-0000(75)80008-0. URL http://www.sciencedirect.com/science/article/pii/S0022000075800080.

[164] William Voorsluys, James Broberg, and Rajkumar Buyya. *Cloud Computing: Principles and Paradigms*, chapter Introduction to Cloud Computing, pages 1–44. Wiley Press, New York, USA, February 2011.

[165] William E. Walsh, Michael P. Wellman, Peter R. Wurman, and Jeffrey K. MacKie-Mason. Some economics of market-based distributed scheduling. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 612 –621, may. 1998. doi: 10.1109/ICDCS.1998.679848.

[166] Lee Wang, Howard Jay Siegel, Vwani P. Roychowdhury, and Anthony A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8 – 22, 1997. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1997.1392. URL http://www.sciencedirect.com/science/article/pii/S0743731597913927.

[167] Darrell Whitley. A genetic algorithm tutorial. URL http://www.cs.colostate.edu/~genitor/MiscPubs/tutorial.pdf.

[168] Adam Wierman. Fairness and scheduling in single server queues. *Surveys in Operations Research and Management Science*, 16(1):39 – 48, 2011. ISSN 1876-7354. doi: http://dx.doi.org/10.1016/j.sorms.2010.07.002. URL http://www.sciencedirect.com/science/article/pii/S1876735410000048.

[169] Min-You Wu, Wei Shu, and Jun Gu. Efficient local search for DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):617–627, June 2001. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.932715.

[170] Jianbing Xing, Chanle Wu, Muliu Tao, Libing Wu, and Huyin Zhang. Flexible advance reservation for grid computing. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *Grid and Cooperative Computing - GCC 2004*, volume 3251 of *Lecture Notes in Computer Science*, pages 241–248. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23564-4. doi: 10.1007/978-3-540-30208-7_37. URL http://dx.doi.org/10.1007/978-3-540-30208-7_37.

[171] Ming Q. Xu. Effective metacomputing using LSF MultiCluster. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 100–105, 2001. doi: 10.1109/CCGRID.2001.923181.

[172] Haihang You and Hao Zhang. Comprehensive workload analysis and modeling of a petascale supercomputer. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 253–271. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35866-1. doi: 10.1007/978-3-642-35867-8_14. URL http://dx.doi.org/10.1007/978-3-642-35867-8_14.

[173] Han Yu. A hybrid GA-based scheduling algorithm for heterogeneous computing environments. In *Proceedings of the IEEE Symposium on Computational Intelligence in Scheduling*, SCIS '07, pages 87 –92, April 2007. doi: 10.1109/SCIS.2007.367674.

[174] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. ISSN 1867-4828. doi: 10.1007/s13174-010-0007-6. URL http://dx.doi.org/10.1007/s13174-010-0007-6.

[175] Henan Zhao and Rizos Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. doi: 10.1109/IPDPS.2006.1639387.