Flexible Coscheduling

Eitan Frachtenberg
School of Computer Science and Engineering
Hebrew University of Jerusalem

December, 2001

Submitted in partial fulfillment of the requirements for the degree of
Master of Science
Supervised by Dr. Dror Feitelson

Abstract

In this thesis a novel technique is introduced for job scheduling in clusters and supercomputers with the goal of increasing the efficiency and utilization of these machines. In particular, the problems arising from heterogeneous architecture clusters and software load imbalances are addressed. The suggested technique is a variation on gang scheduling and other coscheduling methods, where several parallel jobs time-share and space-share the same machine, using varying degrees of coordination among processes.

The main idea behind this thesis is that a distributed/parallel scheduling system can gather dynamic information on the synchronization behavior of processes, and use this information to identify their different coscheduling needs. Using this information, a scheduler can make better scheduling decisions, to increase the overall system utilization and decrease the runtime of applications in a multiprogramming environment.

The contribution of this thesis is threefold: (1) addressing the problems that heterogeneous architectures and load imbalances pose to coscheduling systems; (2) a methodological system of gathering job communication information and subsequent process classification for the making of better scheduling choices; and (3) experimental results that verify the usefulness of applying dynamic communication statistics to scheduling decisions. In addition, this work includes the implementation of an efficient and flexible scheduler, with the ability to use many of the scheduling algorithms found in the literature.

The main result of this thesis is the design and development of a new approach to the identification of different process scheduling requirements and their scheduling according to these requirements. This approach is shown to be both feasible and performance-wise promising, and may also prove to be useful when integrated with other approaches. Another accomplishment of this work is the development of an extensive scheduler system that is both very efficient and flexible, and allows for testing real application behavior on real clusters, measuring real scheduling issues.

This work was done partly at the parallel systems laboratory of the Hebrew university in Jerusalem partly at the Modeling, Algorithms and Informatics group of the Computer and Computational Sciences division (CCS-3) of the Los Alamos national laboratory.

Contents

1 Introduction **7**
- 1.1 Problem Description 7
- 1.2 Outline of Proposed Approach 8
- 1.3 Main Results 9

2 Background **10**
- 2.1 Overview 10
- 2.2 Gang Scheduling 11
- 2.3 Implicit Coscheduling Schemes 15
  - 2.3.1 Dynamic coscheduling 15
  - 2.3.2 Implicit coscheduling 15
  - 2.3.3 Buffered coscheduling 16
- 2.4 Communication Patterns Characterization 17

3 Flexible Coscheduling **19**
- 3.1 Overview 19
- 3.2 Process classification 19
- 3.3 Scheduling 20
- 3.4 Characterization Heuristics 21
- 3.5 Implementation Issues 23

4 Experimental Framework **26**
- 4.1 Overview 26
- 4.2 Hardware Environment 27
- 4.3 Software Environment 27
  - 4.3.1 Design Principles 28
  - 4.3.2 Architectural Overview 29
  - 4.3.3 The Machine Manager 30

4.3.4 The Node Manager 30
4.3.5 The Program Launcher 31
4.3.6 Communication Mechanism 32
4.3.7 System Parameters 35
4.3.8 Development and Evaluation Environment 36
4.4 Scheduling Algorithms Implementation 37
4.4.1 Gang Scheduling 37
4.4.2 Flexible Coscheduling 38
4.4.3 Local Scheduling 39
4.4.4 FCFS Scheduling 41
4.5 Performance Assessment 41
4.6 Workload 43
4.7 FCS and GS Parameters 44

5 Experimental Results 45
5.1 Overview 45
5.2 Basic Tests 45
5.2.1 Response Time 45
5.2.2 Effect of Work Amount 47
5.2.3 Effect of Granularity and Variance 47
5.2.4 Effect of Multiprogramming 48
5.3 FCS Classification 50
5.4 Scheduler Comparison 51
5.4.1 Overview and Metrics 51
5.4.2 Batch Scheduling 53
5.4.3 Local Scheduling 56
5.4.4 Gang Scheduling 56
5.4.5 Implicit Coscheduling 60
5.4.6 Flexible Coscheduling 60

6 Concluding Remarks 67

List of Figures

2.1 Repeated Ousterhout matrix example 12
2.2 Repeated Ousterhout matrix with alternate scheduling 13
2.3 Coscheduling with relaxed alternate scheduling 14
2.4 Buffered coscheduling example. 17

3.1 Decision tree for FCS classification 20
3.2 Context switch algorithm for FCS 22

4.1 Elan library Hierarchy 34
4.2 Example screen shot from scheduler visualization tool 36
4.3 FCS classification heuristic 40
4.4 Main loop of synthetic benchmark application 42

5.1 Effect of multiprogramming level 49
5.2 FCS classification workload - Ousterhout matrix 50
5.3 Arrival times for mixed workload jobs 52
5.4 Work amount for mixed workload jobs 54
5.5 Work amount and runtimes with FCFS 55
5.6 Job wait times with FCFS 57
5.7 Work amount and runtimes with local scheduling 58
5.8 Job wait times with local scheduling 59
5.9 Work amount and runtimes with gang scheduling 61
5.10 Job wait times with gang scheduling 62
5.11 Work amount and runtimes with ICS scheduling 63
5.12 Job wait times with ICS scheduling 64
5.13 Work amount and runtimes with FCS scheduling 65
5.14 Job wait times with FCS scheduling 66

List of Tables

4.1 FCS and GS parameters 44

5.1 Empty job runtimes 46
5.2 Effect of work amount 47
5.3 Effect of granularity and variance 48

1 Introduction

1.1 Problem Description

Supercomputers have been used successfully during the past four decades, and supercomputing hardware m undergone immense progress during this time [70]. It is therefore somewhat surprising that production supercomputer operating systems have advanced relatively little during this period, especially when it comes to aspects of job scheduling. Some of the most widely-used job schedulers are based on batch scheduling methods, which date back to the early supercomputers [24]. While these resource management systems and others incorporate some modern features such as load-balancing, process migration and checkpointing [6], little attention has been given so far to advances in job scheduling techniques outside academia. In particular, it has been shown that various methods of coscheduling (running several globally-coordinated jobs, time-sharing the same computation resources) can be used to increase overall system performance and utilization [17, 55]. Further variations on the coscheduling technique address some of the drawbacks of these techniques, like the non-scalable requirement of global process coordination. They will be discussed in the next chapter. Yet, many super-computing centers still do not to use these techniques despite their potential benefits, due to various reasons, including implementation difficulties and the problems these schedulers pose to accounting.

An emerging trend in high performance computing is the use of computing clusters using commercial-of-the-shelf components (COTS). In the last decade, clusters and constellations (clusters of large SMPs) have increasingly replaced massively parallel processing (MPP) machines, and now have a marked presence in the top 500 supercomputer list [78]. This trend emphasizes these problems, since clusters and constellations are particularly susceptible to problems of high communication latencies and load imbalances (resulting from any of the reasons covered above). While some studies in the past tried to tackle the problem of load imbalances and scheduling in clusters ([7, 15, 54, 74]), to the best of our knowledge, this is the first study that tries to apply the advantages of

coscheduling to heterogeneous environments. Furthermore, in this study the benefits of applying dynamic communication measurements to coscheduling are tested for the first time in a real experimental setup.

This work is an effort to tackle the inefficiencies of large-scale HPC systems that arises from load imbalances. These imbalances can stem from two main sources: heterogeneous architectures and application imbalances.

Heterogeneous architectures can be found mostly in computing clusters and especially in networks of workstations (NOWs), where different nodes can have different computation capabilities, different memory hierarchy properties or even a different number of PEs per node.

Application load imbalances occur when different parallel computation threads require different computation resources, and take varying times to complete. These can occur either as a result of poor programming, or more typically, by a data set that creates uneven loads on the different computation threads.

Even when using homogeneous architectures and well-balanced software, load imbalances can occur. This can happen for instance in NOWs, when the compute nodes are not dedicated entirely to the parallel computation, and may be used for local users' programs. This uneven taxing of resources again creates a situation where some of the parallel program processes run slower than others, and a load imbalance occurs.

Load imbalances have a marked detrimental effect on many parallel programs. A large part of the HPC software can be modeled using the bulk-synchronous parallel model (BSP), in which a computation involves a number of *supersteps*, each having several parallel computational threads that synchronize at the end of the superstep [29, 44, 73]. A load imbalance can harm the performance of the parallel application because each computation thread requires a different amount of time to complete, but the entire program must wait for the slowest thread before it can synchronize. Since these computation/synchronization cycles are potentially executed many times throughout the lifetime of the program, the cumulative effect on the application run time and the system resource utilization can be quite high.

1.2 Outline of Proposed Approach

This thesis proposes to alleviate the inefficiencies caused by these factors by dynamic detection of load imbalances and flexible coscheduling that compensates for these imbalances. Dynamic detection of load imbalances is performed by (1) monitoring the communication behavior of applications, (2) defining metrics for their communication

performance that try to detect possible load imbalances, and (3) classification of the applications according to these metrics. On top of this, a flexible coscheduling mechanism is implemented that uses this application classification to make scheduling decisions. The scheduler attempts to coschedule processes that would most benefit from it, while scheduling other processes to increase overall system utilization and throughput. An application that suffers from load imbalances will not complete faster with this scheduler, compared to other schedulers; in fact, it may even take longer to complete than, for example when running in batch mode. But the proposed scheduler will not allow it to waste too many system resources, and the overall system efficiency will be increased.

1.3 Main Results

The proposed monitoring mechanism and scheduling system, were implemented and run on a real cluster. Versions of batch scheduling, implicit coscheduling and gang scheduling were also implemented and compared to the proposed scheduler, both in aspects of job runtime and wait time. In the experimental part, it is shown that the new scheduling algorithm performs at least as well as most of the other scheduling algorithms used in the comparison, and in some cases significantly better. Furthermore, it is demonstrated that the scheduler testbed that was developed for the purpose of this testing is both very efficient and flexible, and in itself consists of a possible contribution to future research into job schedulers' properties.

The rest of this thesis is organized as follows: Chapter 2 describes in detail the problem we set forth to solve, as well as previous work in this field. In Chapter 3, the proposed solution and scheduling technique is detailed. Chapter 4 describes how we tested and compared the proposed solution, while Chapter 5 presents comparative experimental results. Finally, we conclude and describe future research directions in Chapter 6.

2 Background

2.1 Overview

In this thesis, the problems arising from heterogeneity are tackled by using techniques of dynamic job scheduling and communication patterns characterization. It is therefore worthwhile to review some of the previous work in these two fields.

Scheduling in the context of a multiprogrammed parallel system refers to the execution of threads from competing programs. Generally speaking, the PEs of a parallel system can be shared in two basic, orthogonal ways, and combinations thereof: time slicing and space slicing. In time slicing, all the PEs in the system service a global queue of ready threads, whereas in space slicing, the PEs are statically or dynamically partitioned to different jobs.

Scheduling using space slicing only is typical of batch scheduling systems, and can be found in several widely-used schedulers, such as PBS [38], NQS [45] and LSF [77]. Such systems can have one or more job queues which are assigned to partitions of the machine. These queues can be prioritized or employ first-come-first-serve (FCFS) scheduling. One problem with such systems is that they cause fragmentation in resource allocation, thus reducing overall throughput and utilization. In the case that the run time of the application (or the time allotted for its running) is known in advance, a mechanism called *backfilling* can be used to improve the system's throughput. With backfilling scheduling, short jobs are allowed to be moved forward in the queue if they do not delay the first (or any) other job [2, 42, 53]. Another problem with batch scheduling is that sometimes short jobs must first wait for long jobs to finish before they are run, and in general the response time of the system does not allow interactive work. This is a serious impediment in the software development process, which typically requires frequent short runs and debug sessions.

In [28], the authors argue that the usage of parallel supercomputers for large problems in batch mode only is wrong. Time slicing can solve some of the problems with batch systems and offer similar advantages in multiprocessors to those in uniprocessor systems:

Increased utilization of resources when jobs with complementary requirements keep all parts of the system busy; and more important to most users, time slicing creates the opportunity for interactive response time. Likewise, time slicing brings the same disadvantages from uniprocessor systems to parallel systems. Of these, the most notable are increased system overhead caused by the context switching between different processes, and application performance degradation due to reduced memory cache efficiency. Like in the uniprocessor case, multiprogramming is constrained by the amount of programs that fit into main memory: if adding another job to a system causes the occurrence of swapping, the resulting performance penalties dwarf any benefit from multiprogramming. In addition, time slicing can potentially reduce dramatically the throughput of parallel jobs with fine-grain synchronization. These jobs synchronize frequently, and thus it is extremely important from the performance point of view that all the synchronizing peer processes are scheduled in a coordinated manner (either globally or locally), to reach the synchronization points together [34, 39, 40, 71]. This idea was first introduced by Ousterhout in [55], and is referred to in the literature as *coscheduling*.

Various techniques were suggested to implement coscheduling, of which the most widely used is *gang scheduling* (GS). The rest of this chapter offers an introduction to gang scheduling, as well as other relevant work on coscheduling techniques and communication pattern characterization. For a complete survey of job scheduling strategies and terminology, see [10, 24, 52].

2.2 Gang Scheduling

Gang Scheduling (GS) can be defined to be a scheduling scheme that combines these three features: [24]

1. Threads are grouped into gangs.
2. The threads in each gang execute simultaneously on distinct PEs, using a one-to-one mapping.
3. Time slicing is used, with all the threads in a gang being preempted and rescheduled at the same time.

Time slicing is obtained using a coordinated *multi-context-switch*, which occurs at regular intervals of time, called the *timeslot quantum*. The synchronization of the multi-context-switch is typically done by a central controller, but can also be done by a distributed clock synchronization algorithm. Typically, a GS system consists of a master daemon

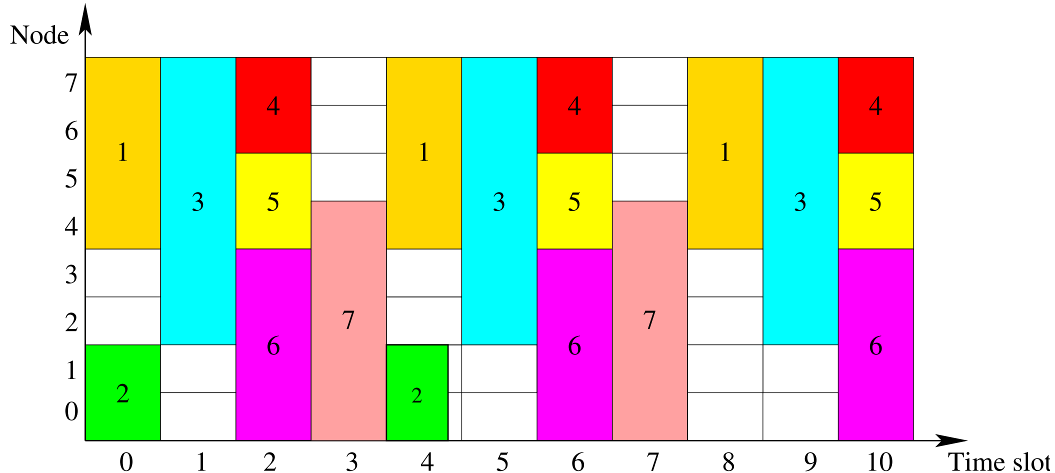

Figure 2.1: Repeated Ousterhout matrix example

(which can be distributed, or in a separate node) and node daemons that run on worker nodes. The master daemon allocates space resources for arriving jobs, and issues multi-context-switches.

The multi-context-switch requires that the receiving PEs save the state of the currently-running distributed application, and context-switch to another application. This affects the inter-processor network as well, since messages that are in transit when the multi-context-switch occurs must be dealt with correctly. For example, in Myrinet, this may require context-switching the NICs as well [18], while in Quadrics the NIC thread processor can handle multiple communicating processes, and overall system throughput can actually be increased by overlapping computation and communication of different processes [30, 32].

Another key concept of GS is the allocation matrix or Ousterhout matrix. An Ousterhout matrix is a representation of resource allocation in space and time. Figure 2.1 shows an example of a gang scheduling with a repeated Ousterhout matrix for an eight-node uniprocessor cluster. The diagram shows the resource allocation for seven jobs during eleven time slots, with a multiprogramming level (MPL) of four (so the four time-slots are being repeated approximately three times). In slot 0 we see two jobs executing, using four and two nodes respectively, while nodes 2 and 3 remain idle. In slot 2, the resource space is fully shared between three jobs, and no nodes remain idle. From slot 4 and up we see a repetition of the allocation pattern, except that job 2 terminates early in slot 4, before the context-switch, and does not execute any more (see emphasized box in Figure 2.1).

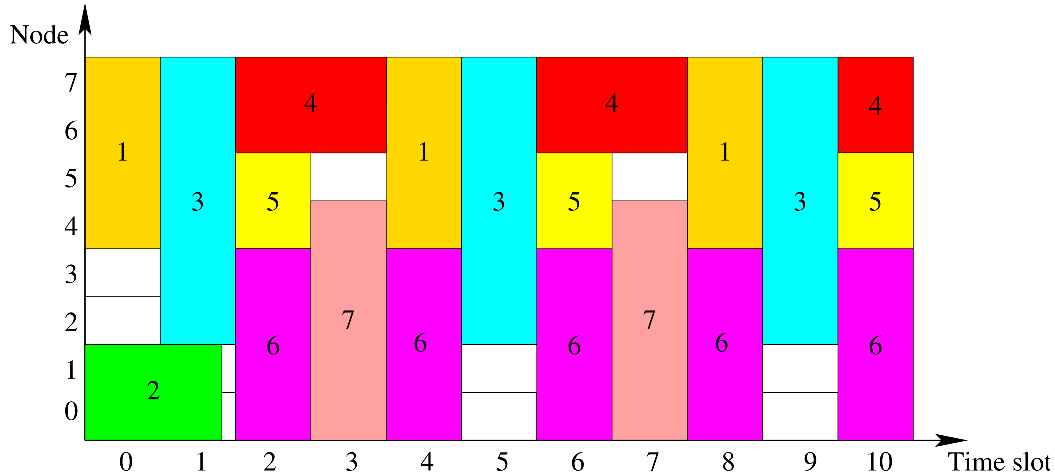

Figure 2.2: Repeated Ousterhout matrix with alternate scheduling

To reduce some of the fragmentation created with this allocation, a simple modification can be made to gang scheduling, called *alternate scheduling*. With this technique, jobs can continue running in other time slots as well, if all the required PEs for the job are idle. Figure 2.2 shows the previous allocation matrix example with alternate scheduling. The only jobs that are affected in this example are jobs 2, 4 and 6. Alternate scheduling serves to reduce the fragmentation, but does not eliminate it altogether. Ousterhout suggested to increase further the utilization of the system using a relaxed form of alternate scheduling, where processes use resources that are available in time slots other than their own, regardless of what other processes in their gang are doing. Strictly speaking, this is no longer gang scheduling, since not all the processes in the gang necessarily run together (this is called coscheduling). Figure 2.3 shows the same allocation matrix using coscheduling with relaxed alternate scheduling.

If the jobs have fine-grain communication granularity, alternate scheduling is most effective if all the processes of a job continue running, as in Figure 2.2. When coscheduling with relaxed alternate scheduling, jobs that are not coscheduled as a gang could have synchronization problems due to the load-imbalance (e.g. jobs 5 and 7). One of the important notions of this thesis is that to be more effective, coscheduling should not be done blindly, since some jobs do not benefit from it. Rather, processes should be classified according to how well they can use fragmented resources, and allocated accordingly. This notion is covered in depth in Chapter 3.

Gang scheduling is sometimes referred to as *Explicit coscheduling*, since the processes that are required to be coscheduled (the *gang*) are explicitly known to the system from

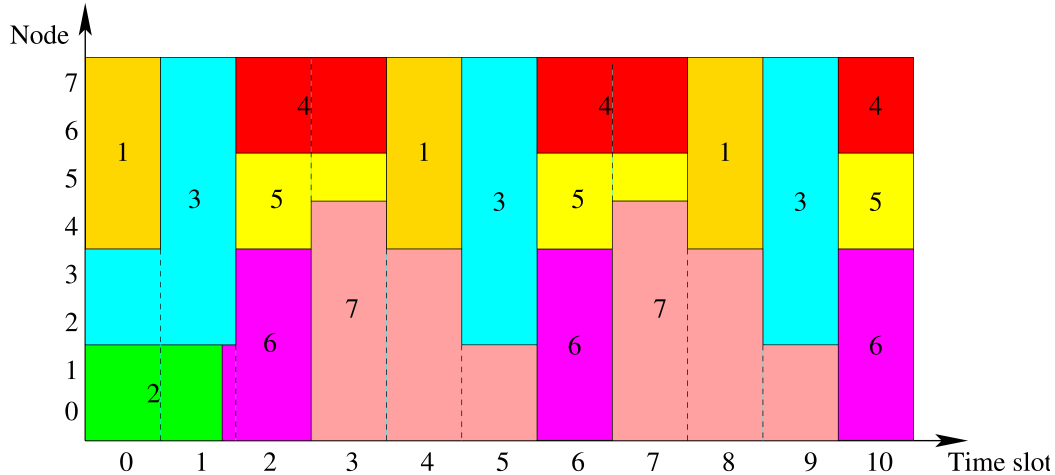

Figure 2.3: Coscheduling with relaxed alternate scheduling

the moment they are launched as a single job. In contrast, the schedulers we describe in the next section use *Implicit coscheduling*, where the communication pattern of processes determine which processes are coscheduled. Since GS requires that all threads in a gang be scheduled together (typically, a gang consists of all the processes of a job), an upper limit on the runtime of an application can be given.While coscheduling sometimes performs better than gang scheduling, it has performance implications on different applications and workloads that cannot be known in advance.

Gang scheduling supports the abstraction of a dedicated machine for each job, which is an underlying assumption for most parallel algorithms and theoretical results. This in turn means that no restrictions on the programming model are imposed. For example, busy waiting can be used for synchronization, knowing that all the processes in the job are executing simultaneously. In addition, asynchronous messages can be sent without the risk of buffer overflow, and user-level communication libraries can access the hardware directly without worrying about protection mechanisms. While these advantages are also given by simply space-sharing jobs (without time sharing), GS offers the added advantage of interactive response times. There are several studies that compare various scheduling policies and conclude that GS results in relatively high performance [11, 13, 17, 26, 33, 37, 49, 50, 65]. Gang scheduling has been implemented on several commercial platforms, such as Meiko CS-2, the Compaq SC family of Alpha-based clusters, Silicon Graphics SMPs [8], Intel Paragon [41], Cray T3E [46], and the Connection Machine CM-5 [48].

As discussed above, GS does not work well with load imbalance, since jobs can no longer gain from fast synchronization. In fact, system utilization can actually decrease

if processes spend a lot of CPU time busy-waiting for their slower peers. This and other problems are addressed by other coscheduling techniques with weaker coordination, that are described in the next section.

2.3 Implicit Coscheduling Schemes

If all the time-sharing schedulers were to be put on a scale according to their degree of coordination, we would have gang scheduling on one end as the strictest-coordinated scheduling method, and local, uncoordinated scheduling on the other end of the spectrum. Since strong coordination has both advantages and disadvantages, several techniques were devised with varying degrees of coordination. This section covers some of the better-known techniques.

2.3.1 Dynamic coscheduling

Dynamic, or demand-based coscheduling (DCS) was first proposed as a way to reduce the amount of global synchronization of coscheduling, and thus decrease the system's overhead and improve its scalability [68, 69]. The philosophy behind DCS is demand-based coscheduling:

- coordination is achieved by observing the communication between processes, and not by a master daemon,
- Communication between processes is used to deduce which processes should be coscheduled and to effect coscheduling. Thus, when a process receives an incoming message it immediately receives a priority boost. This can effectively cause an immediate local context-switch to this process, depending on fairness policy and system load.
- Processes are otherwise scheduled normally by each node's local operating system.

The reason this method results in robust coscheduling is its underlying assumption: if a process receives an incoming message, its peer(s) must be currently running on other nodes, so the process should be prioritized for immediate execution.

2.3.2 Implicit coscheduling

Like DCS, Implicit coscheduling (ICS) makes local scheduling decisions based on monitoring communication activity [3, 4]. Processes waiting for a communication action to

complete use a spin-block (SB) mechanism to relinquish control of the CPU, where first the process spins (actively waits) some time for the communication to complete, and then blocks, which consequently causes a context switch if another process is ready to run. When a communication activity of a blocked message completes, it receives a priority boost, much in the same way it would in DCS. The main difference is that DCS explicitly treats every incoming message (not just those for blocked processes) as a demand for coscheduling, causing an immediate scheduling of the receiving process as soon as it would be fair to do so. Also, in DCS a waiting process always spins and does not block (it can be preempted immediately though, when another process receives an incoming message). In [68] it is claimed that while ICS is well-suited for "bulk-synchronous" applications (those that alternate regular periods of computation and communication), DCS is more suited for less-regular applications.

A comparative study of DCS, ICS, and other variations on implicit coscheduling techniques is given in [1]. These variations include different actions for waiting processes, as well as for those receiving messages. In some cases, the author finds that simpler coscheduling mechanism can outperform ICS and DCS, depending on the OS abilities. Another comparative study presents an elaborate taxonomy and comparison of implicit coschedulers, and also suggests some new schemes based on periodic rescheduling [54]. These methods, called *Periodic Boost* (PB), are based on boosting the priority of communicating processes on a periodic basis, thus eliminating the need for an interrupt on communication events. Another extensive simulation-based study that compares different variations of PB to the other methods presented above can be found in [76]. In this study, it was found that both PB and SB to result in higher machine machine utilization than GS for most test scenarios.

2.3.3 Buffered coscheduling

Buffered coscheduling (BCS), represents a different approach to coscheduling [19, 20, 21, 22, 56]. In BCS , local scheduling decisions are based on global information of the system's status, essentially converting an on-line problem of coordinating jobs to an offline problem.

In the BCS model, each timeslot has two phases: computation and communication. During the computation phase the current job runs normally, except that all the outgoing communication is buffered for later execution. If the communication is of a blocking type, the process is preempted and another process is chosen and scheduled. The communication phase is divided into three parts: first, an exchange of information occurs between the nodes, after which every node has global knowledge of the information that

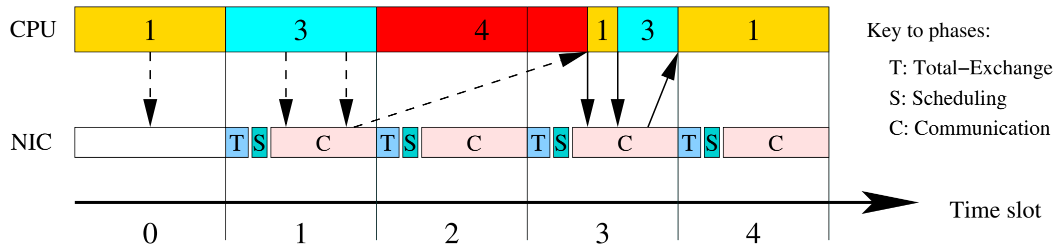

Figure 2.4: Buffered coscheduling example.

pertains to its pending incoming and outgoing communication, and possibly other information such as load, availability and status of other nodes. During the second part of the phase, the communication between the nodes is scheduled for optimal use of the network without exceeding the phase's time limit. Lastly, the messages are transmitted according to the schedule. It should be noted that the availability of advanced network hardware can enable a communication phase that is so fast, that several phases fit in a timeslot. In that case, the model can change to accommodate communication phases that service the currently running computation phase, thus reducing the need for a premature context switch.

To use the machine effectively, the computation and communication phases are overlapped, so that the communication phase for computation phase N runs concurrently with computation phase $N{+}1$. This requires that network interface card (NIC) have its own processing capabilities, as is the case for example with the Myrinet[9] and Quadrics [59, 58] NICs. Obviously, this only makes sense when there is demand for running more than one job, which is the typical case for large supercomputer centers.

Figure 2.4 shows an example of buffered coscheduling on the first five timeslots of node 7 in the Ousterhout-matrix shown in Figure 2.1. Dashed arrows represent non-blocking communication messages; regular arrows represent blocking messages. The communication phase in the NIC is divided in three parts: (T)otal exchange, (S)cheduling and (C)ommunication. Note that outgoing blocking messages cause preemption of processes, while incoming messages can enable the continuation of a process.

2.4 Communication Patterns Characterization

All the coscheduling techniques discussed above use some kind of information gathering on the communication behavior of the application and use that information for making local scheduling decisions. However, the information they gather is rather rudimentary

(mostly, they check for incoming messages), and does not serve to characterize applications into distinct classes of communication behavior. In [47], it is shown that gang scheduling does not always perform optimally for all types of applications, especially those that are I/O-bound. Furthermore, the authors conclude that messaging statistics can provide important clues to whether applications require gang scheduling, and that a scheduler needs to base its scheduling decisions on attributes of jobs in its workload.

The coscheduling schemes discussed here try to improve the definition of which processes should be coscheduled. However, they do not take into account how much applications are really affected by gang scheduling, as is suggested in this study. Other studies analyzed in detail the communication patterns of various types of scientific applications, but in an offline manner, where logs of application communications are analyzed after the execution is over [12, 14, 43, 44].

In [27], the authors make a first attempt to utilize dynamic information acquired from the network to identify "activity working sets" of processes for the purpose of coscheduling them. This study focuses mainly on identifying the sets of processes to be coscheduled, but does not suggest a classification of processes or a refined method of scheduling processes according to the information acquired.

The observations in [47] lead the authors to believe that runtime measurements of communication behavior can and should affect scheduling decisions. The main parameter they suggest to measure is message count, and correlate it to the periods when processes are coshceduled to test how much the process requires to be coscheduled with its peers. The authors also use this parameter to suggest a rudimentary classification of processes into three types: embarrassingly-parallel, workpile and synchronization intensive. The first two classes describe processes that have little synchronization needs, and may or may not, require just some load-balancing, respectively. The third type is for processes that synchronize frequently, and these processes explicitly require coscheduling. The authors indicate that a coscheduler should use this information to make better scheduling decisions, and suggested the name "Flexible Coscheduling" for it. Unfortunately, this study does not implement or test the effects of such a scheduler. Furthermore, the paper shows that in some cases, message count alone is not enough to offer a reliable indication of progress.

In the next chapter, a coscheduling scheme is suggested that offers a methodological way of gathering meaningful statistical information from the communication layer and using this information to schedule processes according to their specific coscheduling needs.

3 Flexible Coscheduling

3.1 Overview

To address the problem described in Chapter 1.1 we propose a novel scheduling mechanism called Flexible Coscheduling (FCS). The main motivation behind FCS is the improvement of overall system performance in the presence of heterogeneous hardware or software using dynamic measurement of applications' communication patterns and classification of application to distinct types. The scheduler can thus make better local scheduling decisions based on the class information of different processes and applications.

3.2 Process classification

FCS employs dynamic process classification and schedules processes using this class information. Processes are categorized into one of three classes:

1. CS (coscheduling): These processes require coscheduling with their peers, and are currently successfully coscheduled. The way to measure this success is explained in Section 3.4.
2. F (frustrated): These processes require the synchronization gains obtained with coscheduling, but coscheduling them is unsuccessful. These are typically those processes that suffer from system heterogeneity or load imbalance, originating for example from uneven decomposition of the data set.
3. DC (don't-care): These processes rarely synchronize, and can be scheduled in almost any possible way without penalizing the system's utilization. For example, a job using a coarse-grained workpile model would be categorized as DC.

Note that some processes, called RE (rate-equivalent), are included in the DC category. These processes have low synchronization requirements, but need coarse-grain load bal-

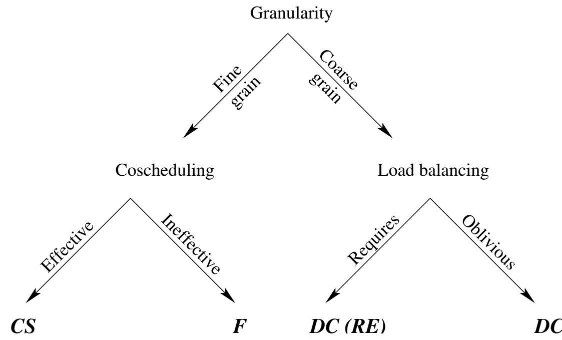

Figure 3.1: Decision tree for FCS classification

ancing, so they require the same amount of normalized CPU time, even if this does not necessarily occur in the same time slots[1]. Figure 3.1 summarizes this classification.

Processes of the same job will typically, but not always, be of the same class. Some local traffic patterns can create subgroups of processes with their own synchronization patters. To allow for these cases, and to avoid global exchange of information, processes are categorized on a per-process basis, instead of a per-job process.

This classification differs in two important ways from the one suggested in [47]. First, we differentiate between the CS and F classes, so that even processes that require gang scheduling would not tax the system too much if heterogeneity prevents them from having proper GS. Second, there is no separate class for RE, or embarrassingly parallel applications. These are indistinguishable (from the scheduler's point of view) from DC processes, and are scheduled in the same manner. Embarrassingly parallel applications are hard to detect, since it is hard to predict their need for load balancing. Thus, they are designated as F processes so that they may be given approximately the same CPU time on all nodes, but not if they are predicated by heterogeneity (which would cause a detrimental load imbalance for them anyway).

3.3 Scheduling

The principle behind scheduling in FCS is as follows: CS processes should be coscheduled and should not be preempted; F processes should be coscheduled but can be preempted

[1] The detection of RE processes would require the exchange of information about processes across nodes. This may pose some scalability and performance problems. This is outside the scope of this work which attempts to schedule processes in a more distributed manner. Buffered coscheduling, described in the previous section, attempts to tackle the these issues, and uses hardware features to implement efficient global exchange of information.

when synchronization is not achieved; and lastly, DC processes impose no restrictions on scheduling. Like with BCS, the node daemon in FCS receives multi-context-switch messages from the master daemon, but also has autonomy to make local scheduling decisions.

Figure 3.2 shows the basic algorithm of the node daemon upon receipt of a multi-context-switch message. The basic idea is to allow the local operating system the freedom to schedule DC processes according to its usual criteria (fairness, I/O considerations, etc.), and also to use DC processes as "Lego blocks" to fill in the gaps that F processes have because of their synchronization idiosyncrasies to gain better machine utilization. An F process that waits for pending communication should not block immediately, but rather spin for some time to avoid unnecessary context switch penalties.

The fact that FCS collects communication statistics for each process allows the scheduler to determine a competitive spinning time on a per-process basis. Two more principles differentiate this scheduling method from DCS and ICS

1. A CS process in FCS cannot be preempted before the time slot expires even if an incoming message arrives for another process (processes classified as CS have "proven" that it is not worthwhile to deschedule them in their time slot).

2. The local scheduler's freedom to choose among processes in the DC time slots and F gaps is affected by the communication characterization of processes, which could lead to less-blocking processes and higher utilization of resources.

3.4 Characterization Heuristics

FCS uses dynamic characterization of processes based on their communication behavior, to continually adapt processes' classification and associated parameters over time. The local scheduler monitors six variables for every process:

1. C_{CS} - The message count (in and out) for the process while it is being coscheduled (i.e. scheduled as either CS or F).

2. C_{DC} - The message count (in and out) for the process while it is scheduled as DC.

3. T_{CS} - The total amount of time spent waiting for communication attempts to complete while being coscheduled.

4. T_{DC} - The total amount of time spent waiting for communication attempts to complete while scheduled as DC.

```
procedure context_switch (current_process, next_process)
begin
    if current_process == next_process then return
    switch (on type of next_process)
        if CS then
            run next_process for its entire time slot
        if DC then
            let local OS scheduler schedule among all DC processes
            for entire time slot, and if there are no DC processes
            waiting to run, allow local scheduler to choose from F
            processes first and then from CS processes
        if F then
            loop for entire time slot
                1) run next_process until it blocks for communication
                2) Do until communication unblocks for next_process:
                       let local OS scheduler schedule among all
                       DC processes for entire time slot, and if there
                       are no DC processes waiting to run, allow local
                       scheduler to choose from F and then CS processes
        else if next slot is empty then
            if current process is DC or F, continue process
            else suspend current CS process and run DC in next slot
end
```

Figure 3.2: Context switch algorithm for FCS

5. $W_{CS} = \frac{T_{CS}}{C_{CS}}$ - The average wait time for a message while being coscheduled.

6. $W_{DC} = \frac{T_{DC}}{C_{DC}}$ - The average wait time for a message while scheduled as DC

These variables refer only to synchronous communication operations, and only while the process is scheduled (time spent suspended is not counted). Asynchronous operations are not counted or monitored (waiting for an asynchronous operation to terminate, using for example MPI_Wait(), counts as a synchronous operation). Applications that have little synchronization needs do not require to be coscheduled. Therefore, processes that rarely communicate synchronously are automatically classified as DC.

Initially, when a process is launched, it is tagged as CS, and is scheduled as such for a fixed number of time slots, while updating the C_{CS} and T_{CS} variables. This allows for short jobs, which account for a significant amount of many production workloads [16, 51] to enjoy the benefits of gang scheduling and thus terminate early. Then, the process is tagged as DC for another fixed number of time slots while updating the C_{DC} and T_{DC} variables. After this initial period, the variables are updated during the entire lifetime of the process, and the process scheduling class is determined using these principles:

- If the process rarely communicates (both C_{CS} and C_{DC} are lower than a given threshold DC_{thresh}, compared to the total accumulated run time of the job), the process is either DC or RE and is classified as DC;

- Otherwise, If the process always has relatively long periods of waiting for communication to complete (compared to a threshold F_{thresh}), it is of type F[2];

- Otherwise, If the process communicates significantly better while being coscheduled than not (as determined by the ratio $\frac{W_{DC}}{W_{CS}}$ and a threshold $MAX_{\frac{W_{DC}}{W_{CS}}}$), the process is of type CS;

- Otherwise, the process is of type DC.

The thresholds to determine what "low" and "high" ratio and count values are tun-able parameters.

3.5 Implementation Issues

It is worth noting the following implementation issues:

[2] These are F processes rather than DC, since they communicate relatively frequently, albeit not very efficiently. If they were designated as DC and not coscheduled, they would probably suffer even larger delays in communication.

1. DC and F processes are allowed to use alternate scheduling, and continue into the next time slot if it is not allocated to any other process. The reasons for this is that DC processes do not need coscheduling and F processes can sometimes benefit from the extra CPU time, and if not, they will not hurt the system much since DC processes can fill the gaps they create.

2. One possible optimization would allow for processes with effective, fine-grain communication (that is, their average count of synchronous communication operations in a timeslice exceeds a threshold CS_{thresh}), to never switch from the initial CS state to the initial DC state. This makes sense since such processes do not really belong to the F or DC classes.

3. For implementation purposes, it is safe to assume that the average wait time while being coscheduled does not exceed the the average wait time while not coscheduled (i.e., $W_{DC} \geq W_{CS}$), so that it is enough to check W_{CS} for determining whether the average wait is longer than a threshold (and thus classify the process as F).

4. As described in Section 3.3, F processes spin ("busy-wait") for some time before blocking and possibly yielding the processor. The amount of spin time should be large enough to allow "normal" communications to complete with little overhead, and small enough to detect stalled communication adequately (stalled communication is typically caused by a load imbalance in an F process). One advantage of collecting communication statistics about process is that the spin time can be both adaptive and individual per process. In the current implementation, the spin time was chosen to be $spin = min(2 * MAX_{non-block}, MAX_{spin})$, where $MAX_{non-block}$ is the maximum time that a non-blocking communication took in the lifetime of the process, and MAX_{spin} is an absolute upper bound on the spin time.

5. Frequent re-classification of processes should be avoided, since it incurs extra overhead and reduces the reliability of the process statistics. Therefore any process should spend some minimum amount of time MIN_{ctime} in any given class before being evaluated for re-classification. Furthermore, a higher amount of time $MIN_{init-ctime}$ should be spent on the two initial CS and DC periods, to allow the process some time for initialization. Note that in both cases, CPU time in class is measured, as opposed to wall-clock time.

6. While running in DC slots, or when using DC processes to fill idle fragments, we retain the option to run F and CS processes when all DC processes are blocked for communication or I/O. One way of implementing such a mechanism is detecting

that all DC processes are blocked and launching F or CS processes. This detection may be potentially difficult to implement, since a watchdog process must monitor the status of the DC processes. Another implementation option is to allow all processes to run, in three descending priority classes for DC, F, and CS processes. If the priorities are set properly, F processes will almost never run in these slots, unless all DC processes are blocked, and CS processes will almost never, unless all DC and F processes are blocked. Either one of these mechanism can also be used to "fill in" idle periods on CS and F timeslots, if the coscheduled process is blocked for a long I/O operation.

7. Another subtle issue regarding DC processes occurs when running on SMP nodes. It is possible that some PEs in the node will be running a coscheduled process (F or CS) while one or more PEs are running in DC mode. Since more than one DC process may be running, it is possible that the local UNIX scheduler will assign some of these DC processes to PEs that are running a coscheduled process, for load balancing. This predicament has an adverse effect on the coscheduled processes and should be avoided if possible. One way to avoid it is using *processor affinity*, where processes are "locked" to certain PEs, so that DC processes only run in PEs that are are not running coscheduled processes (but can be load balanced within their assigned PEs). This feature is operating-system dependent, and is supported for example in Linux 2.4 and Tru64. Processor affinity also hold sometimes the added advantage of a process maintaining its cache state for a longer period.

8. To determine process granularity, which is required to distinguish the DC class, the actual running time of a process must be measured, so that periods when the process is descheduled or blocked are not counted. Linux and other operating systems offer system calls that facilitate this measurement[3].

9. It may be beneficial to let the scheduler to reset process classes from time to time, and allow them to go again through the initial classification process. The purpose of this resetting is to allow long-running processes to start gathering fresh statistics and possibly acquire a different classification, thus adapting to changing process behavior of correcting a previous, incorrect classification. A process is thus set back to the INIT_CS class if it has spent more than MAX_{ctime} CPU time in one class.

[3]For more information, see Linux man pages on times(), rusage() and clock().

4 Experimental Framework

4.1 Overview

This chapter describes in detail the hardware, software and run time environments that were used to test FCS and compare it to other scheduling algorithms. Both the hardware and software environments are among the fastest currently available in the market, and offer performance that compares favorably with modern systems.

On the hardware side, a 64-PEs cluster was using, with a Quadrics interconnection network. This network offers high-bandwidth, low latency point-to-point communication, along with extremely fast collective operations. Section 4.2 describes in detail the hardware environment.

The scheduler system that was developed for these tests is currently the only system that can run MPI programs over the Quadrics network, aside from Quadrics' own RMS, but offers better response time and scalability than RMS [32]. Furthermore, its flexible design allows for a simple implementation of most of the scheduling algorithms in the literature, and several of those were implemented to enable comparison with FCS. The scheduling system is described in Section 4.3 while Section 4.4 covers implementation details of the scheduling algorithms that were used in the comparison. A more detailed description of the scheduler software can be found in [31].

A special benchmark was developed for this work, that allows extensive comparison of scheduler behavior under different types of applications and workloads. This benchmark is described in Section 4.5. A realistic model of a production-environment workload was used, and is described in Section 4.6.

It is important to note that this simulation contains real life measurements and not a simulated hardware environment. The scheduler's efficiency is measured on a relatively fast cluster using MPI applications. Thus, all the overheads and effects associated with scheduling, process management, network contention, etc., are measured and accounted for.

4.2 Hardware Environment

The hardware used for experimentation was the 'crescendo' cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550) and Quadrics inter-processor network switch [63, 79] (using 32 of the 128 ports). Each compute node is composed of the following:

- Two 1.13 GHz Pentium-III
- 1 GB ECC RAM
- 18 GB hard-disk
- Two independent 66MHz / 64-bit PCI buses
- A Quadrics QM-400 Elan3 NIC [61, 62, 79] for data network
- An Ethernet-100 network adapter for management network
- Red Hat Linux 7.1 with Quadrics kernel modifications and user-level libraries [60] and the processor affinity patch [80].

The Quadrics interconnection-network (ICN) is one of highest-performing interconnects currently available, and can offer a sustained MPI data bandwidth of more than 300 MB/s, and synchronization latencies lower than $7.5\mu sec$, even for 1024 nodes [59, 57]. It is currently being used in Pittsburgh Supercomputing Center at the machine ranking number 2 in the top 500 list, and will be used in the ASCI 30-teraops machine, to be deployed at LANL. It was used in the experiments both as the underlying device layer of the MPI library linked to the test applications, and as the internal communication layer of the scheduler for its own information exchange. More details on the advantages of using the Quadrics ICN are outlines in Subsection 4.3.6 below.

4.3 Software Environment

To verify the FCS scheduler and test how it performs in comparison to other schedulers, a complete scheduling testbed system was designed and implemented. This section describes the main points of the design and implementation of this testbed. Additional low level implementation details and a detailed background of the Quadrics software environment can be found in [31].

4.3.1 Design Principles

1. **Flexibility:** An important feature for the testbed is to support as many modern and future scheduling algorithms as possible, so that it can be used as a valuable research tool. With this in mind, the testbed currently supports GS, SB (ICS), FCS, FCFS and local scheduling, but is also designed so that other scheduling methods, including those described in Section 1, can be readily added to the system. In fact, one of the imminent future research directions at LANL includes the implementation of BCS and other schedulers, for in-depth study of their properties.

2. **Performance:** To make the experimental results both valid and comparable to current state-of-tart systems, the design should strive for best scheduler and application performance. From the software point of view, this requirement translates to a lightweight, efficient scheduler, with fast user-level internal communication and a relatively low-overhead implementation. From the hardware point of view, an environment where test applications can perform well compared to other systems is required.

3. **Scalability:** The scheduler testbed should be as scalable with the machine size as possible, both to allow testing on larger machines, and to obtain good scheduling algorithms. This implies some design constraints such as having the testbed control daemons work as asynchronously as possible, and using broadcast/multicast mechanisms for any global coordination messages

4. **Simplicity:** The testbed should not be over-complicated, so that maintenance and augmentation of new scheduling algorithms will incur little overhead. This in turn means that some nice-to-have features such as those described below are not included in the design goals. This also means that parallel applications should not be changed to accommodate the system, or at most be linked again with an instrumenting version of MPI.

5. **Portability:** The testbed should be designed so that porting it to other hardware platforms, ICNs or even operating systems will be relatively simple, to allow for extended testing and enhancing. To this end, the system should intervene as little as possible with the operating system. Furthermore, the most hardware-dependent module of the testbed, the underlying communication layer, should be encapsulated in a small, isolated module (the final version actually included two implementations of this layer, one that works over the Quadrics ICN, and another that works over any implementation of MPI, and is used mainly for debugging purposes).

Since the testbed's main purpose is for scheduling research and not a production system, the following issues were not included in the design goals:

1. **security:** The system takes no special precautions to avoid rogue requests and does not check for access control rights. However, since it is run by the user in user-mode, using her own user-id and group-id, the scope of potential security violations is limited to that of any application the user might run.

2. **reliability:** No special attempt was made to make the system fault tolerant or have graceful degradation in case of software/hardware errors. However, the communication layer is designed to be reliable, and the scheduler includes various sanity checks to allow many kinds of errors to be trapped.

3. **manageability:** The testbed system has no cluster or job management tools except for a basic set of scripts, and does not provide most of the features in modern cluster management systems (CMS) [6].

4. **Ease-of-Use:** The testbed has a relatively simple command-line, scripts, and files interfaces, and offers no GUI or other high-level tools.

4.3.2 Architectural Overview

The testbed system consists of a single wrapper application running in many copies, where each copy of the program assumes one of three possible roles according to its node location and sequence number. These three types of programs consist of a machine management (MM) process, node management processes (NM) and program-launcher processes (PL), and they communicate among themselves with a well-defined protocol (See 4.3.6). In general, the MM is in charge of the initial allocation of resources to programs and the coordination of the NMs; the NMs are responsible for local scheduling on each node and communicate with the MM and the PLs; the PL's only function is to fork() and execute new applications, and report back when these terminate. The functions of each process are detailed in Subsections 4.3.3-4.3.5. Subsection 4.3.6 describes both the details of the internal communication mechanism between the scheduler modules, and the augmentations to the user-level communication library that the test application use. Finally, subsection 4.3.7 Covers the system parameters that control the behavior of the testbed system.

4.3.3 The Machine Manager

The machine manager has two roles: the dispatching of new jobs, initial allocation of resources to them, and the coordination of node managers through heartbeat messages. The workload of jobs is fed into the MM using a workload file, with a simple format that describes, in each line, the following parameters:

1. Time to run the application (in some predefined units).

2. Number of PEs the application requires.

3. Command-line of application to run, including parameters for the application

The MM reads this file, which is sorted by application start time (Item 1), one line at a time, and sleep()s until the time the application should be launched or a heartbeat message should be sent. It would then wake up to perform the communication, and possibly to read the next line of the workload file, before going back to sleep. It can also be awaken by a message from one of the NMs, e.g. announcing the termination of a process.

For initial allocation of resources, we use a library called rm_dynbt (Resource Management using DYNamic Buddy Trees), developed by Uri Lublin and Dan Tsafrir for the ParPar cluster scheduling system [18, 25, 81]. This resource management library employs the buddy-tree allocation algorithm [23]. Its modular design allows it to be easily plugged into different schedulers and use various allocation schemes. A wrapper layer was written to access its functions, and also includes the functions to allow the library the updating of its internal data structures due to status change of nodes or jobs. This is done by propagating the appropriate messages (e.g. process termination) from the NM to the MM, which then calls the wrapper module's appropriate callback function.

The MM occasionally exchanges messages with the node managers in a well-defined asynchronous communication protocol. These messages include broadcast information from the MM to the NMs (e.g. new jobs or heartbeat message), and incoming messages from the NMs (e.g. process termination notification). See Section 4.3.6 for a complete description of the protocol.

4.3.4 The Node Manager

Node managers are responsible for the initial launching and scheduling of processes on each node. For simple gang-scheduling, the role of the NM is limited to executing commands issued by the MM: launching and preempting jobs. With some of the the other

scheduling schemes, where local scheduling decisions are made based on locally-collected information, the NMs are in charge of the information collection and the local scheduling decisions.

The NMs hold a copy, through messages from the MM, of the Ousterhout-matrix information pertaining to their node. This allows them to know which processes to run on a context-switch, as well as to make better-informed local-scheduling decisions. For some of the scheduling schemes, the NM also requires information about communication among the processes. Information regarding in-node processes are gathered by augmenting one of the layers of the MPI library (the ADI layer) so that it informs the NM of relevant communication events. A small library using BSD message queues was implemented for this purpose.

Each node in the cluster runs exactly one copy of the NM, regardless of the number of PE's the node has. (In debug mode, several PEs or even several nodes can be simulated on any smaller number of nodes, including one). This NM assigns and preempts processes to all the PEs in this node. This policy can offer a performance benefit when making local scheduling decisions in SMPs, since the NM has more degrees of freedom for making such decisions.

4.3.5 The Program Launcher

The program launcher has a very simple role: execute a command from the NM to launch a program. Upon receiving a new job to run, the PL fork()s a new process, sets up pipes so that the standard input, output and error streams of the new process are redirected to Quadrics' RMS (and in turn, to the management node), and exec()s the new job. Each PL is associated with the node NM with which it communicates using a well-defined communication protocol. The number of PLs in each node is determined by the number of PEs and the MPL we choose when initializing the system. One copy of the PL is required for each PE and for each timeslot in the system. A new program can be run only if the PL assigned to its PE and timeslot (or any PL in local scheduling) is currently available.

An available PL simply waits for an event from the NM, giving it the details of a program to run (command-line, etc.), and then fork()s, and exec()s the program. The PL itself blocks with a waitpid() call. When the user process terminates, the PL wakes up and notifies the NM of the process termination and becomes available for a new program again.

It should be noted that in an ideal implementation, the PL is actually redundant with the NM and not really required as an independent process. However, the system was

designed this way since it allows running application processes with Quadrics capabilities without modifying the Quadrics Resource Management System (RMS). This limitation arises from the fact that currently, only RMS can assign a Quadrics capability (access to the Elan NIC), to processes, so an implicit role of the PLs is actually to reserve this capability for the application process.

4.3.6 Communication Mechanism

For the correct operation of the scheduler, an efficient communication mechanism is required to connect the various modules. This section describes the communication protocol that is used between the modules, and some of the implementation details of the communication layer. In particular, the implementation over the Quadrics network and libraries is described.

There are two important issues when considering the communication aspects of the testbed. The first is the internal system communication layer, that connects the MM, NMs, and PLs. The second is the set of adjustments that is required to make the application communication layer cooperate with the scheduler subsystem for some of the scheduling algorithms (For example, BCS, FCS, DCS and a few other require information about synchronous communication operations of the applications).

Internal protocol

The first issue is resolved by defining a lean communication layer API and a communication protocol to connect the scheduler modules. This layer can later be implemented over any underlying hardware, and has currently two implementations in our system: one over Quadrics for maximum performance and one over MPI for flexibility and ease of debug. The API defines primitives for sending and receiving messages from one process to another, and broadcasting messages from the MM to the NMs. All messages are assumed to be sent asynchronously, and the layer supports polling of the communication channel for incoming messages. The communication protocol allows for the following types of messages (with an appropriate, context-sensitive payload):

1. Heartbeat (multi-context-switch) with the new timeslot number. This is broadcast from the MM to NMs.

2. Process-ID information, from NM to PLs at initialization.

3. New job-launch, with timeslot and allocated PEs information, broadcast from MM to NMs and from NM to PLs.

4. Process-ID of a newly-launched process, from PL to NM.
5. Failure notification if launching of process failed, from PL to NM.
6. Process termination message, from PL to NM.
7. Process termination message, from NM to MM (contains more information than previous message)
8. Shutdown message when all work is completed, broadcast from MM to NMs and from NMs to PLs.

The processes are designed to be as state-less as possible, and receive the incoming message at asynchronous fashion and in almost any order. However, they do perform sanity checks on the source and content of every message, to trap for erroneous message (for example, message from NM to NM are invalid, or a message to launch a new job on a previously allocated PE/timeslot). Furthermore, the protocol is designed so that the MM only talks to the NMs, and the PLs only talks to the NMs. This allows for having two different implementations of the communication layer, depending if it is with the MM or the PLs. In fact, one version was implemented where all the communication between the NM and the PLs on its node is done via shared memory instead of Quadrics. This allows enhances the performance for NM $\leftrightarrow$PL communication and simplifies the handing down of Quadrics capabilities to the real executing applications (since they are not being used by the PLs).

Implementation details for the Quadrics internal communication layer

The network interface of the Quadrics network (Elan) can be programmed using several programming libraries [61], as outlined in Figure 4.1. These libraries trade off speed with machine independence and programmability. Starting from the bottom, Elan3lib provides the lowest-level, user-space programming interface to the Elan3. At this level, processes in a parallel job can communicate with each other through an abstraction of distributed virtual shared memory. Each process in a parallel job is allocated a virtual process id (VPID) and can map a portion of its address space into the Elan. These address spaces, taken in combination, constitute a distributed virtual shared memory. Remote memory (i.e., memory on another processing node) can be addressed by a combination of a VPID and a virtual address. Since the Elan has its own MMU, a process can select which part of its address space should be visible across the network, determine specific access rights (e.g., write- or read-only) and select the set of potential communication

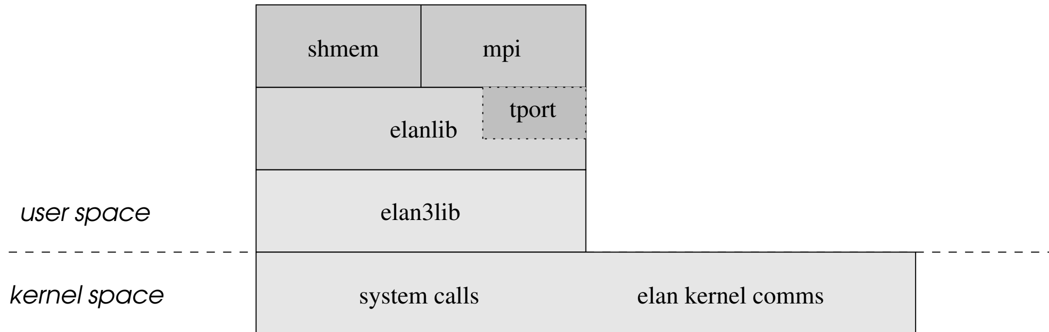

Figure 4.1: Elan library Hierarchy

partners. Communication at Elanlib level is vary fast, with a base latency of only $2 \mu sec$ and asymptotic bandwidth in excess of 320 MB/sec.

Another interesting feature of the Quadrics network is the native support for collective communication. The switches in the network can support a tree-based multicast that is combined with the execution of multiple active messages on the destinations network interfaces. For example, a source node can inject a multicast packet into the network, and this packet is forwarded in a tree-like manner to multiple destinations. The packet open a tree of circuits and on these circuits multiple transactions can be executed synchronously. A typical transaction is to check the status of variable on all destinations, merge the results of all these nodes using a combining tree and perform other transactions, based on the results of the previous transaction. These mechanisms allow the fast implementation of collective communication patterns.

The communication library implemented for this scheduler leverages both mechanisms, the remote DMA and the multicast, to provide fast communication and synchronization between all the processes. More specifically, the heartbeat (multi-context-switch) is implemented with a hardware multicast, which can be delivered in few microseconds, with good scalability, even in the presence of background traffic. The communication between the NMs and the MM uses remote DMAs

Application communication layer

These processes communicate with each other through helper threads that run on the Quadrics NIC. This way, the communication incurs almost no penalty to the compute

processes, and can benefit from Quadrics' network advantages, like fast collectives and communication that requires no intervention from the main CPUs.

For the communication of the test applications, we use MPI [35, 36, 67] as the underlying communication layer. MPI was chosen due to its widespread use in parallel applications and the availability of a high-performance implementation of MPI over the Quadrics ICN. It is important that the PL close all the open Quadrics handlers it has before executing the user program, so that they become available to the application process. Further, Some changes are required to the underlying Elan3[1] level, to trick the application into seeing a world of MPI processes that consists only of its peer processes, and not all the NMs, PLs, and other running applications. This in turn requires that information about the program[2] be available to the Elan3 layer. This is done by the NM posting this information (as a mapping from job virtual IDs to Elan virtual IDs) in a shared-memory region in main memory. MPI_Init() was enhanced so that whenever a new program is launched and calls it, it will read that mapping from the shared-memory area, and set it up as an Elan group. From that point forward, all other processes are transparent to the application, and all MPI calls work normally.

4.3.7 System Parameters

A set of environment variables determines the operational parameters and behavior of the testbed system. These variables control the following:

- The machine configuration, whether real or simulated (number of nodes, number of PEs per node).
- Scheduling algorithm to use.
- The multiprogramming level (MPL) - the number of timeslots in the Ousterhout matrix, for the algorithms that use it. No more than MPL processes can be assigned to a given PE at any time.
- Timeslice for each timeslot, in microsecond resolution.
- Debug level - a set of binary flags controlling which aspects of the system should be instrumented, debugged, or verified with further sanity checks.

[1] The Elan3lib is the lowest-level library that allows access to the Elan NIC thread processor.
[2] Specifically, information on the program's timeslot, or row, in the Ousterhout-matrix

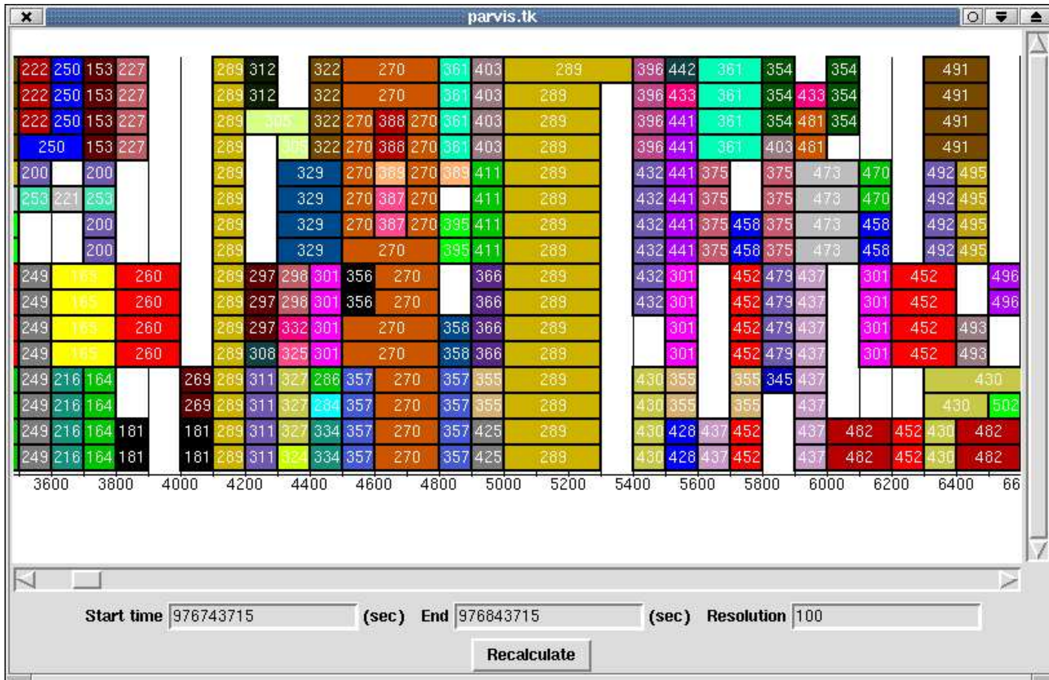

Figure 4.2: Example screen shot from scheduler visualization tool

4.3.8 Development and Evaluation Environment

The development and testing environment includes a set of preporcessing and postprocessing scripts that handle the following tasks:

- Format the output of the workload model to a workload file adapted for the synthetic benchmark
- Verify consistency among the program parameters and environment variables (for example, check that the MPL is set to 1 before running an FCFS batch).
- Launch the MMs, NMs and PLs at their corresponding nodes via RMS [64]or mpirun [67].
- Process the output log file of the scheduler and calculate the performance metrics.

Another set of Perl and TCL/TK scripts was used to analyze scheduler log files and display the scheduling decisions over time in a visual manner, similarly to Figures 2.1-2.3. Figure shows an example screen shot from these tools.

4.4 Scheduling Algorithms Implementation

This section describes the implementation details of each of the scheduling algorithms that were implemented in the current version of the testbed.

4.4.1 Gang Scheduling

The description of the implementation of gang scheduling in the testbed system can serve as a basis to describe other scheduling algorithms as well. In fact, the system is designed so that there are very little structural differences for the various implementations. Most notably, these include a different implementation of the context switch and are mostly confined to the NMs.

In the gang scheduling implementation, the MM has the responsibility over scheduling decisions (total global coordination), and the NMs have therefore very little autonomy. Their role is limited to that of carrying out communication between the MM and the PLs. The following describes the order of events when the MM decides to launch a new job (after its start time is has arrived and resources were successfully allocated to it):

1. The MM broadcasts a job-launch message to all the NMs with the details of the new job and its allocated resources.

2. Each NM verifies the request, and checks if the new job runs on some or all of its node PEs.

3. If it does, the NM forwards the request to the appropriate PLs (each PL is responsible for one PE and one timeslot, so the job's PLs are well-defined).

4. The PLs receive the request and try to fork() and launch the program. Note that for performance reasons, it is best that a copy of the program and its data files be kept in a local hard-disk, rather than accessing it through NFS. The same applies for the output files of the program.

5. If successful, the PL returns the PID of the new process to the NM, and waits for it to terminate using the waitpid() system call. If the program could not be run, the PL returns a failure message to the NM.

6. The NM stores the PID for future reference. It also suspends the newly-run process if its allocated timeslot is not the currently active timeslot (so that it does not interfere with the currently-running job).

Later on, when the user process terminates, the following sequence of events occurs:

1. The PL awakes and sends a termination message to the NM with the process ID.
2. The NM matches the PID with its tables, and sends the MM information about the process.
3. The MM deallocates the process resources in rm_dynbt, and checks to see if it can use them for a new job.

On every timeslice expiration, the MM broadcasts a heartbeat message to the NMs, with the new timeslot number. Each NM checks to see if is required to switch to another process in the next timeslot (for each PE), and if so, suspends the currently running process and resumes the next process (using SIGSTOP and SIGCONT signals). Note that processes will continue running beyond their timeslot if no other process is supposed to run on the next timeslot (In other words, they will employ a relaxed kind *alternate scheduling*, see 2.2). This may not be overly useful for fine-grain, synchronization-bound jobs, but could be beneficial to compute-bound processes.

4.4.2 Flexible Coscheduling

The FCS algorithm can be divided into three logical parts: communication monitoring (instrumentation), process classification and scheduling. We can therefore focus on these parts independently.

Communication monitoring

Instrumentation of the application communication is achieved through a monitoring library (libmonitor.a), that is used both to record communication events and to retrieve the information. The first part is achieved by simple modifications to the MPI library that take effect when running under FCS, so that whenever an application calls a synchronous MPI function, the modified MPI library function registers the event. Furthermore, these calls are also modified so that they perform a spin-block mechanism, instead of the default blocking. The amount of time to spin is read from a global variable from libmonitor.a, and can be adaptive, as described in Section 3.5. The spin-block mechanism is also used in the implementation of ICS.

The monitoring information is stored in an event queue, at a well- known shared memory region, so that the queue can be emptied by the NM at fixed intervals or when requested by the monitoring library. This request will occur when the queue is nearly

full, and whenever the spinning time of the communication call is expired, and it starts blocking. This is done by the means of a UNIX signal, and enables the NM to react immediately to a blocking communication call and reschedule processes as needed.

Process classification

One of the key issues in FCS is the classification heuristics, that tries to determine which scheduling requirements are best for each process, based on its communication behavior. Thus, on periodic intervals and whenever a process performs a blocking MPI call (and a signal from the monitoring library is received), the process class is reevaluated. The classification heuristics uses most of the optimizations described in 3.5, and contains several parameters that were tuned by trial-and-error to find relatively good characterization precision. Figure 4.3shows the re-classification heuristics. Note that if the process actually changes class, some immediate scheduling decisions may have to be taken (for example, if a process changes from DC to CS or F, the rest of the DC processes (if any) must be suspended for the rest of the timeslot.

Process scheduling

Like in GS, process scheduling is the responsibility of both the MM and the NM, and executed mainly by the NM's context-switch function. However, in contrast to GS, the NM has a relatively large autonomy for making scheduling decisions, and may deviate from the Ousterhout matrix allocation when it deems it beneficial to do so. The implementation of the context switch function follows the algorithm described in Section 3.3 and Figure 3.2. Two implementation details are worth noting.

1. When performing a context-switch for a reason other than a regular heartbeat (i.e. because a process terminated), DC processes are scheduled till the end of the timeslot.

2. When switching from an F process to a different process, care must be taken that the process may have been blocked for communication, and therefore DC processes might be running and should possibly be suspended.

4.4.3 Local Scheduling

Local scheduling is the least-coordinated and easiest to implement scheduling mechanism for an NOW or a cluster. In local scheduling, parallel jobs are launched from one management node, but all scheduling decisions are made locally by every compute

```
procedure reclassify_process (proc)
begin

    ctime = total CPU time of proc in current class
    events = total no. of blocking comm. events in current class

        // Don't reclassify too early:
    if ctime < MIN_{init-ctime} then return  // For initial CS or DC
    if ctime < MIN_{ctime} then return   // For other classes

        // Reset process class if too old:
    if ctime > MAX_{ctime}  then reset class and return

        // If at initial CS, switch to initial DC:
    if proc.class == initial CS then
        proc.class = initial DC and return

        // Find class according to process statistics:
    if ctime/events > DC_{thresh} then proc.class = DC

    else if W_{CS} > F_{thresh} then proc.class = F

    else if W_{DC}/W_{CS} > MAX_{W_{DC}/W_{CS}} then proc.class = CS

    else proc.class = DC

end
```

Figure 4.3: FCS classification heuristic

node, typically using their default OS scheduler. Since jobs are not coordinated across machines, this scheduling mechanism is expected to provide the worst performance for parallel jobs, especially if they synchronize frequently. Of all the algorithms described above, local scheduling is the easiest to implement, since it requires no coordination at all. It is provided in the testbed as a baseline for comparison with other schedulers.

In the testbed implementation, job launching is done in the same manner as it is done for GS and all other algorithms: When the job's arrival time is reached, the MM sends the job to the NMs, which in turn send the job to the PLs for actual launching. The same goes for the job termination process: the PL reports the termination of a process to the NM, which forwards it to the MM for resource management accounting. The MM takes care of not allocating more than MPL processes per PE at a time, just as it does for every other scheduling algorithm.

There are two important differences between local scheduling and gang scheduling. In local scheduling, heartbeat messages are not sent by the MM, and the NM considers such a message an erroneous condition. Furthermore, the NM never suspends a process, so in theory, every PE could have up to MPL processes waiting for it to run them, and it is the responsibility of the local UNIX scheduler to assign processes to processors using the usual UNIX schedule() kernel function [72]. In Linux, there is only one run-queue for all the processes, and while the scheduler tries to schedule processes to processors in a consistent manner, it is quite possible in practice that processes would have different assignments of CPUs and CPU time on every run cycle.

4.4.4 FCFS Scheduling

The implementation of FCFS (batch) scheduling is easily obtained by limiting the MPL of the system to 1. Thus, no more than one process is allocated to a PE by the rm_dynbt module, and jobs wait in the MM queue, sorted by their arrival time. Whenever a resource becomes available (e.g. by termination of a process), the MM retries to allocated the first jobs in the queue.

4.5 Performance Assessment

The benchmarking of the different scheduling algorithm is based on a versatile synthetic application that allows the exposing of various scheduler aspects [29, 47, 54, 76]. The synthetic application is essentially a job comprising of processes that loop that performs some computation, some I/O, and then some exchanges synchronous message with peer processes, as illustrated in Figure 4.4 . A constant buffer size of 4KB was used for the

```
for i := 1 to iterations_num do
    compute (basic_work + variance (basic_work, variance factor))
    do_IO (block_amount)
    exchange_communication (pattern)
end
```

Figure 4.4: Main loop of synthetic benchmark application

communication exchanges and a buffer 1KB for disk I/O [76]. Using these constants, the average times for exchanging one message (T_{msg}) and writing one disk I/O synchronously to the disk (T_{IO}) were measured, and found to be approximately $10\,\mu sec$ and $17.5\,msec$ respectively.

The following non-constant parameters also control the behavior of the application:

1. N, or the number of processes in the job. This is determined in the workload file in a per-job basis.

2. W, or total amount of time to be run, in seconds. This is given in the application command line, and is determined by the workload.

3. G, or granularity, expressed as a ratio of the timeslice. For example, a ratio of 0.1 represents an application that exchanges communication 10 times in one timeslice.

4. The communication pattern, (*pattern* in Figure 4.4), which can be either nearest-neighbor (*NN*) or all-to-all (*AA*). These two cases represent extreme cases of communication. Another "communication pattern" is provided for testing, where no messages are actually sent. The communication pattern and the number of processes in the job N determine together the amount of message exchanges per iteration, C.

5. B, or the amount of 1KB blocks to write to the disk in each iteration (*block_ amount* in Figure 4.4). This parameter controls how dominant the I/O factor is in the application (and can also be set to zero, to disable I/O).

6. w or the basic amount of work per iteration (*basic_ work* in , in seconds of computation. This is determined by estimating the amount of time the communication and computation will take per iteration (by evaluating the expression $C \cdot T_{msg} + B \cdot T_{IO}$), and subtracting it from the granularity in seconds (expressed as $G \cdot timeslice$). Di-

viding W by w, we can derive the number of iterations, (*iterations_num* in Figure 4.4).

7. V, or the variance factor, expressed as a ratio of B, or percentage. Thus, a basic work unit of $w = 1$ second and a variance factor of $V = 20\%$ would mean that in each loop, the application would do something between 0.8 and 1.2 seconds of computation. The actual value is determined in a uniform-random manner every iteration, where each process has its own separate pseudo-random number generator (PRNG) seed. This is in fact one of the most important parameters of the experiments, since it controls the amount of of load imbalance the application has. The higher the variance, the higher the probability that when a process exchanges a message, a matching send or receive was not posted yet.

4.6 Workload

For the purpose of creating a workload that would be realistic and representative of real production workloads, a model introduced in [51] was used, that in turn is based on run logs from several supercomputing sites. Using this model, a workload file defining arrival- and run-times of jobs can be created for any number of jobs and nodes, which has similar mathematical and statistical properties the workloads extracted from these log files. For these experiments a workload file for 32 PEs was created (so that two experiments can be run in parallel in two separate partitions of the cluster), with a 512 jobs, and an average load of 0.75[3], representing approximately one week of real jobs. All the run-times and arrival times in the workload were "compressed" by a constant factor (100), that is large enough so that experiments run fast enough to make an extensive comparison practical, and short enough so that run-times are still several orders of magnitude above the system latencies. Thus, running the entire 512-job workload typically takes approximately $1.5-3$ hours, depending on the algorithm and parameters being used. Every job chooses its granularity in a uniform random manner from one of three values: fine (10% of the timeslice), medium (50% of the timeslice) and coarse (250% of the timeslice). Likewise, each jobs selects in a uniform random manner one of three values for variance (taken from [76]): no variance (0%), low variance (10%) and high variance (75%). The communication pattern is also chosen randomly, with the following probabilities: No communication: 10%; Nearest-Neighbor : 45%; and All-to-All: 45%. A low probability is assigned to

[3]This is not the load measured from the experiments. Rather, this is a pre-run estimation of the ratio of CPU-time request divided by total available CPU-time. Actual load depends naturally on the efficiency of the scheduling mechanism and the behavior of the applications.

Parameter	Value	Description
T	$200\mu sec$	Timeslice: Interval between heartbeats, length of timeslot
MPL	4	Multiprogramming level
$Nodes$	16	Number of SMP nodes
$MAX_{\frac{W_{DC}}{W_{CS}}}$	2.0	Maximum value of $\frac{W_{DC}}{W_{CS}}$ for a process to be considered DC
F_{thresh}	$\frac{T}{100}$	Threshold for average W_{CS}, above which a job is considered F
DC_{thresh}	$1 * T$	Threshold for coarse granularity, above which process remains DC
CS_{thresh}	$\frac{T}{9}$	Threshold for fine granularity, below which a process remains CS
MAX_{ctime}	$500 * T$	Maximum CPU time for process in same class before resetting
MIN_{ctime}	$2 * T$	Minimum CPU time for process in a class before reclassification
$MIN_{init-ctime}$	$5 * T$	Minimum CPU time for process in the initial CS and DC classes

Table 4.1: FCS and GS parameters

the no-communication case, since it is assumed that most parallel applications require synchronous communication, hence the need for all the coscheduling techniques that were developed to tackle this need.

4.7 FCS and GS Parameters

Table 4.1 shows the values for all the various GS and FCS parameters used in the experiments. All the tunable FCS parameters were selected using extensive sets of tests to determine which values might produce reasonable and stable results for the given hardware and software platform.

5 Experimental Results

5.1 Overview

This chapter describes the experiments the were performed with the scheduler system, and the results that were obtained. In Section 5.2, several simple tests are used to demonstrate the basic abilities and performance properties of the scheduler system. Section 5.3 contains a special test scenario to examine the workings of the FCS classification heuristics, while in Section 5.4 the performance of FCS is compared to several other scheduling algorithms: ICS, GS, FCFS and local scheduling.

5.2 Basic Tests

The purpose of this section is to offer some insight into the basic performance aspects of the scheduler system, before divulging into the benchmark results. We explore some of the overheads associated with launching simple jobs of different types in the system. Note that some of this these tests involve small quantities in their input parameters or output measurements. Since the experiments were run on a real machine, with UNIX daemons in the background occasionally interrupting the computation, and minute differences in hardware, the results occasionally fluctuate. These fluctuations are in orders of magnitude of fractions of a percent, and do not affect much the longer experiments, but may affect some of the shorter ones. Unless otherwise mentioned, all these tests were run with batch (FCFS) scheduling.

5.2.1 Response Time

Perhaps the simplest of cases is running a single job that does not compute, communicate or perform any I/O, to measure the incurred overhead of launching a job. This overhead can be regarded as the basic response time of the system, since it would be added to the runtime of any job, regardless of its size. The overhead stems from the cost to process a

#	G	B	N	W	$T_{arrival}$	T_{start}	T_{end}	Runtime
1	-	0	1	0.0001	2	2.19	3.61	1.42
2	-	0	32	0.0001	10	10.17	11.59	1.42
3	-	0	64	0.0001	20	20.13	21.55	1.42

Table 5.1: Empty job runtimes

job, send its information to the NMs and from there to the PLs, execute the job, wait for the job to terminate on all nodes, and pass the information to the MM. Furthermore, we may expect a delay of up-to a full timeslice after the job's arrival time before starting it, and up-to a full timeslice after its termination before the MM receives the notification. The reason for this is that the MM only checks for jobs' arrival times and messages from the NM at every timeslice interval. Table 5.1 shows the runtimes when running a set of single-job , zero-work tests. All jobs are batch scheduled (FCFS), do not perform any communication or I/O, with a negligible amount of work (parameters G, B and W, respectively) and vary only in their amount of processes (N). The times $T_{arrival}$, T_{start}, and T_{end} denote the designated arrival time, actual launching time and actual time of removal from the system in seconds respectively, measured by the MM. Runtime is simply $T_{end} - T_{start}$. All times are given in seconds. The rest of the run-time parameters are used with their default values, detailed in Sections 4.5-4.7. From this table we can see that the average overhead associated with launching a job is approximately $1.4\,sec$ or less. This overhead can broken down to the following elements:

- Time gap from the arrival time until the MM actually allocates resources to it: up to one timeslice (the MM sleeps between timeslices).
- For the same reason, up-to one more timeslice accounts for the time between the NM notification until the MM noticing it.
- The time it takes the NM to notify the PLs, the PLs to execute the empty program, and report back to the NM: approximately one timeslice.
- The other 0.8 seconds are nearly constant, and are divided between the processing of the job in the MM, local context-switches in a node (all nodes run one PL per PE, in addition to an NM, and an MM in node 0). Communication latencies between the MM and the NMs account for a few hundreds of microseconds and can be considered negligible.

#	G	B	N	W	$T_{arrival}$	T_{start}	T_{end}	Runtime
1	-	0	32	1	1	1.20	3.67	2.464
2	-	0	32	10	3	3.09	14.59	11.498
3	-	0	32	100	15	15.05	116.51	101.456
4	-	0	32	1000	120	120.13	1121.59	1001.458

Table 5.2: Effect of work amount

Another interesting feature arising from Table 5.1 is that there is no noticeable difference in overhead between running a single-process job, a 32-process job or a 64-process job. This implies good scalability of the scheduler and demonstrates the importance of efficient communication collectives (in this case, from the MM to the NMs) for the implementation of a scalable scheduler. For example, Quadrics' gang-scheduler (RMS) uses TCP/IP over ethernet to communicate between its versions of the machine-management daemon and the node-management daemons, and employs a somewhat inefficient resource allocation mechanism [32]. The result is that launching a job of 32 processes can several seconds (typically 5–15).

5.2.2 Effect of Work Amount

In this test, the effect of various amounts of work on the runtime is analyzed. Table 5.2 shows the runtimes for jobs of work amounts from 1 to 1000 seconds. We observe no significant increase in the amount of overhead associated with running longer jobs. The deviations in the overhead from the average 1.42 seconds can probably be attributed to a minute increase of overhead or small fluctuations in the experiment. This implies that the scheduler system does not have adverse effect on longer jobs that run alone.

5.2.3 Effect of Granularity and Variance

To test the effect of communication granularity and variance on the runtime of single jobs, five tests were run with different granularity and variance values, corresponding to the values of 'coarse', 'medium' and 'fine' granularity and 'low' and 'high' variance, respectively. All the tests were run separately, so all have an arrival time of 1. The communication pattern that was used in all the experiments is all-to-all, which represents a worst-case scenario from the communication load point of view.

Table 5.3 Shows the results of these tests. Tests 1-3 show the effect of refining the granularity. We can notice a small increase in the runtime of jobs as the granularity grows finer, which is associated with the increased amount of communication: Partly

#	G	V	B	N	W	$T_{arrival}$	T_{start}	T_{end}	Runtime
1	2.5	0	0	32	10	1	1.14	12.64	11.499
2	0.5	0	0	32	10	1	1.16	12.67	11.503
3	0.1	0	0	32	10	1	1.14	12.76	11.623
4	0.1	0.1	0	32	10	1	1.15	15.79	14.644
5	0.1	0.75	0	32	10	1	1.18	17.10	15.923

Table 5.3: Effect of granularity and variance

because of the small overhead that is associated with each communication operation, and partly because the higher number of communication operations increases the chance of delayed communications, which can occur for example when one process is descheduled by the local UNIX scheduler, in favor of one of its daemons. However, the increase is not very large, representing approximately 1% of added overhead when increasing the communication amount by a factor of 25.

The next two tests use the fine granularity and test the effect of introducing variance to the communication (fine granularity was used, again to represent a worst-case scenario). The increase in runtime is quite noticeable, demonstrating the detrimental effect of variance and heterogeneity on applications. Since the amount of computation per iteration as well as the number if iterations is fixed in tests 3-5, we can see that even the introduction of a relatively small variance (10%, test 4) increases the amount of time spent in communication by about 34%, and the number grows to about 50% when using a variance of 75%.

5.2.4 Effect of Multiprogramming

One of the basic properties of a multiprogramming scheduler is the cost of context-switching between jobs/processes. The higher the cost, the more time is wasted on overhead, especially with higher multiprogramming levels. In this test, several copies of the same program were run together (each with a work amount of 100, with All-to-All communication and fine granularity), and the cumulative effect on their runtime is observed. Figure 5.1 shows the average runtime of jobs running together with MPL values ranging from 1 to 8. This job does not use a lot of memory, and therefore does not expose problems arising from processor cache flushing, but these should not be noticeable with this timeslice (the single-processor, normal UNIX timeslice is much shorter). It can be seen that the curve is completely linear, meaning that no significant overhead is caused by the additional context-switches, at least for these values of timeslice. In contrast,

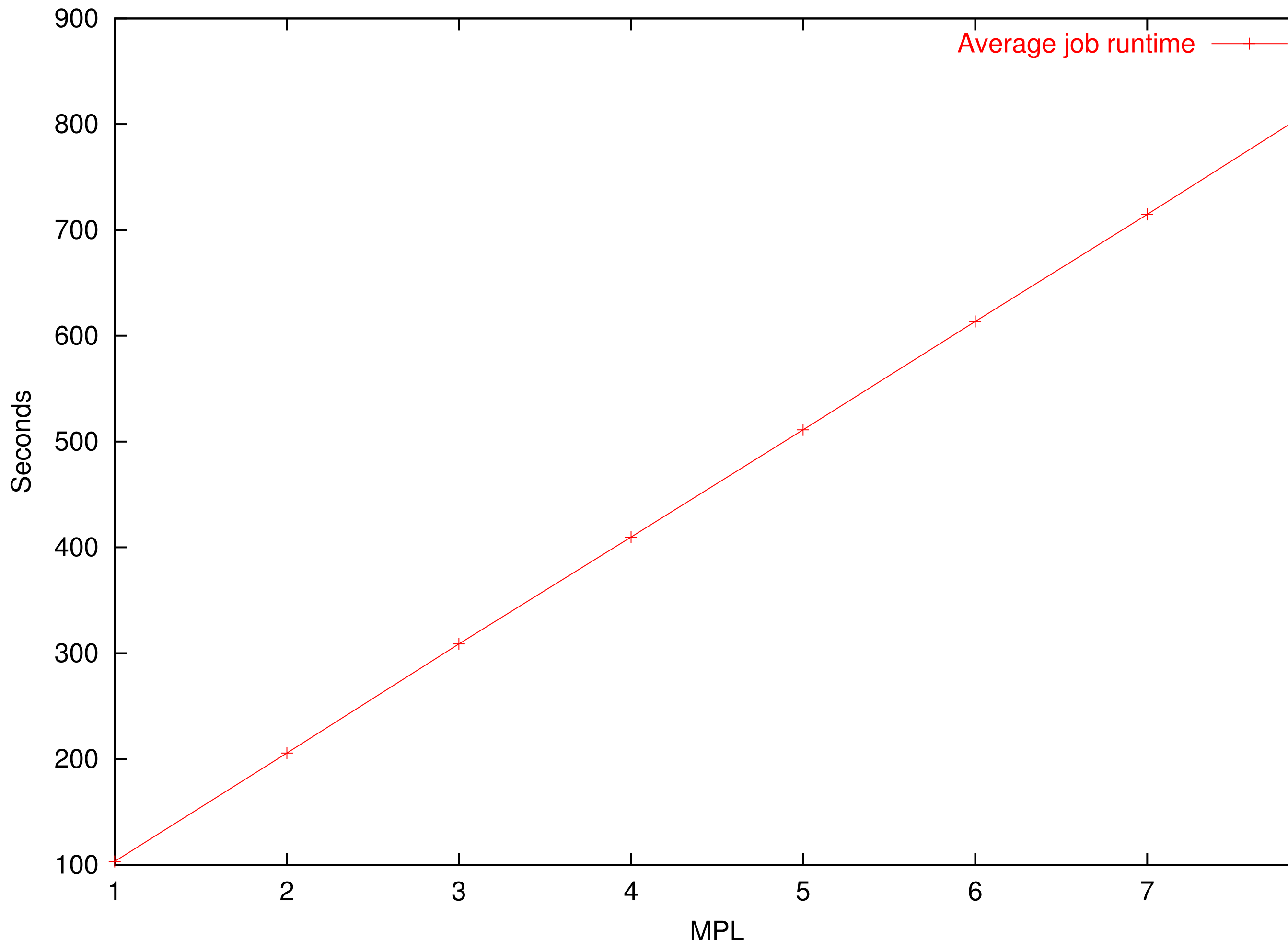

Figure 5.1: Effect of multiprogramming level

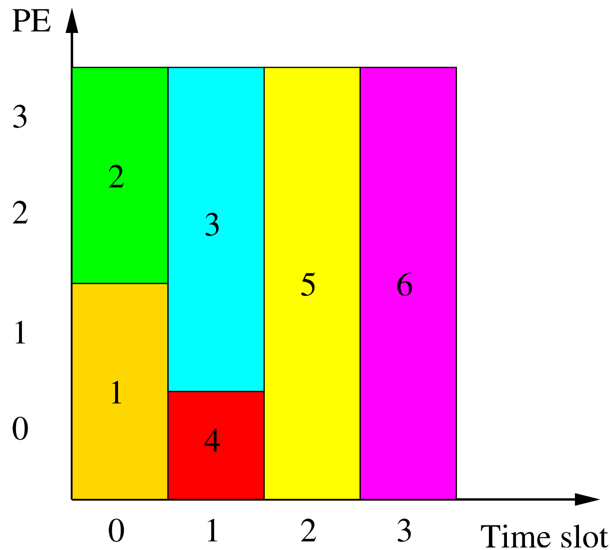

Figure 5.2: FCS classification workload - Ousterhout matrix

RMS exhibits noticeable performance problems for timeslice values even as large as 10 seconds [32]. Normalizing the average runtimes shown in the graph by the MPL yields negligible differences between the different run times.

5.3 FCS Classification

The purpose of this experiment is to understand and verify the classification heuristics of FCS for different types of jobs and processes. To this end, a special workload of six jobs was constructed for 4 PEs, and was allocated to a 2-node partition using the Ousterhout matrix shown in Figure 5.2. All jobs have a work amount of 100, and a communication pattern of all-to-all (if any). The following is the description of the different properties of each job:

1. A 2-PE job with medium granularity and no variance.
2. A 2-PE job with fine granularity and no variance.
3. A 3-PE job with fine granularity, and one process significantly slower than the other two (performs 50% more computation work per iteration, simulating a slower or loaded machine).
4. A single PE job (with no communication).

5. A 4-PE job with medium grain communication and 75% variance for the first two processes.

6. A 4-PE job with low grain communication and 10% variance for all processes.

All the jobs had the same start time and were launched together, and detailed log files were collected and analysed. The following is a description of the resulting classification:

1. This job went through the initial CS and DC periods, before the scheduler decided to switch both processes to CS, never changing their class again.

2. This job communicates relatively very well, so it was never changed from the initial CS class.

3. With this job, one process (the slower one) was eventually classified as CS, while the other two processes were classified as F. This confirms expectations, since the slower process benefits from coscheduling (the other two are always waiting for it), while the faster processes cannot communicate effectively with it (long average blocking wait time).

4. This process started with the initial CS class, and since it does not communicate, it was not re-evaluated until the time came for a periodic re-evaluation of all jobs. It was then classified as DC since it had a zero synchronous communication count.

5. All this job's processes were classified as F, due to their long wait time for blocking calls.

6. The processes of this job were eventually all tagged as DC: the low granularity and low variance created a situation were there was no big difference between the time it takes to complete communications in DC and CS modes.

This classifications seem reasonable, and suggest that the scheduler might be able to use this information to make better scheduling decisions.

5.4 Scheduler Comparison

5.4.1 Overview and Metrics

In this section, we compare the performance of FCS and four other scheduling algorithms: ICS, GS, FCFS and local scheduling. The workload that was used is described in Section 4.6. Figure 5.3 shows the arrival time per job in seconds. Note that this workload is

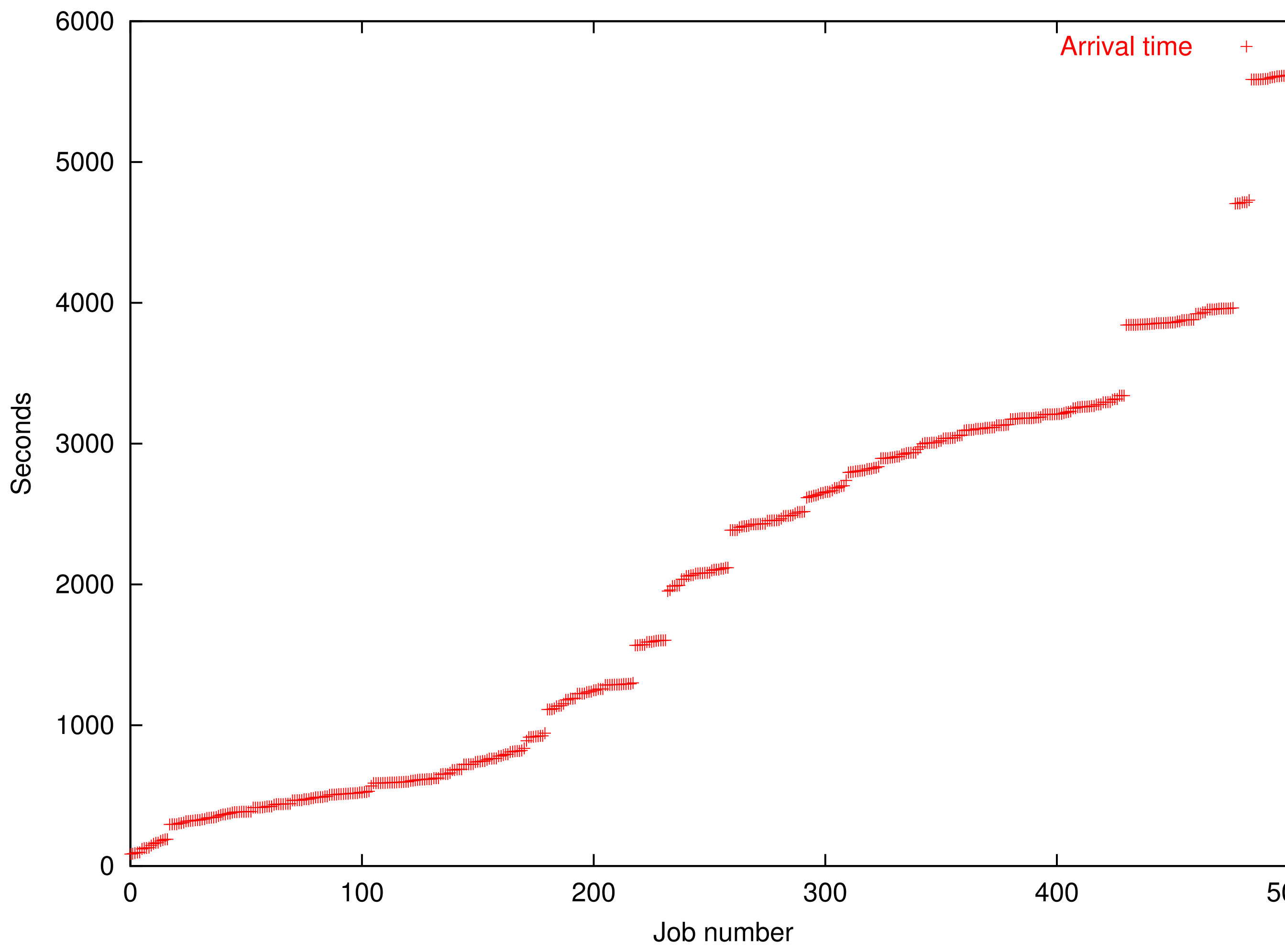

Figure 5.3: Arrival times for mixed workload jobs

light enough so that none of the scheduling algorithms reaches saturation. An interesting research venue in the future might compare the saturation point of different scheduling algorithms. For each run, an elaborate log file was created, and then analysed using Perl scripts to produce the following metrics:

1. The arrival, start, and end time of each job, as well as the amount of work it should perform (in seconds).

2. The average wait time of each job (i.e. the difference between its start time and arrival time).

3. The average response time, i.e. the difference between the actual end time of each job, and its arrival time.

4. The average slowdown, which is defined as the ratio between the actual runtime of the job and its given amount of work.

These metrics are measured separately for two groups of jobs: 'short' jobs, and 'long' jobs, defined arbitrarily as those with less than one minute of work (short jobs) and the other jobs. (using one real-world minute, divided by the 'compression ratio' of 100). This divides the entire workload into two roughly equal-size groups, one representing more-interactive jobs, where wait time is the important metric, and the other representing longer computation jobs, where runtime is the more important metric. Figure 5.4 shows the amount of given work per job in the workload, and how it relates to the short-job threshold. Overall, this workload contains 236 short jobs with an average work amount of 0.203 seconds(or real 20.3 seconds), and 276 jobs with an average work amount of 68.737 seconds (or ≈ 114 minutes of real time).

5.4.2 Batch Scheduling

FCFS scheduling is a good test case as a baseline for other multiprogramming schedulers, since jobs receive a dedicated partition with it, and thus we can expect no better runtimes (per job) than with batch scheduling. Indeed, if we look at the average runtimes we see an optimistic picture: the average runtime of short jobs is 1.57 seconds, while that of long jobs is 70.536 seconds. The small difference from the work amount (1.367 seconds for short jobs and 1.8 seconds for large jobs) is explained by the system overheads described in Section 5.2. Figure 5.5 shows the runtimes of the jobs in FCFS, superimposed on the work amount. For the largest jobs, the difference can be significant if the communication

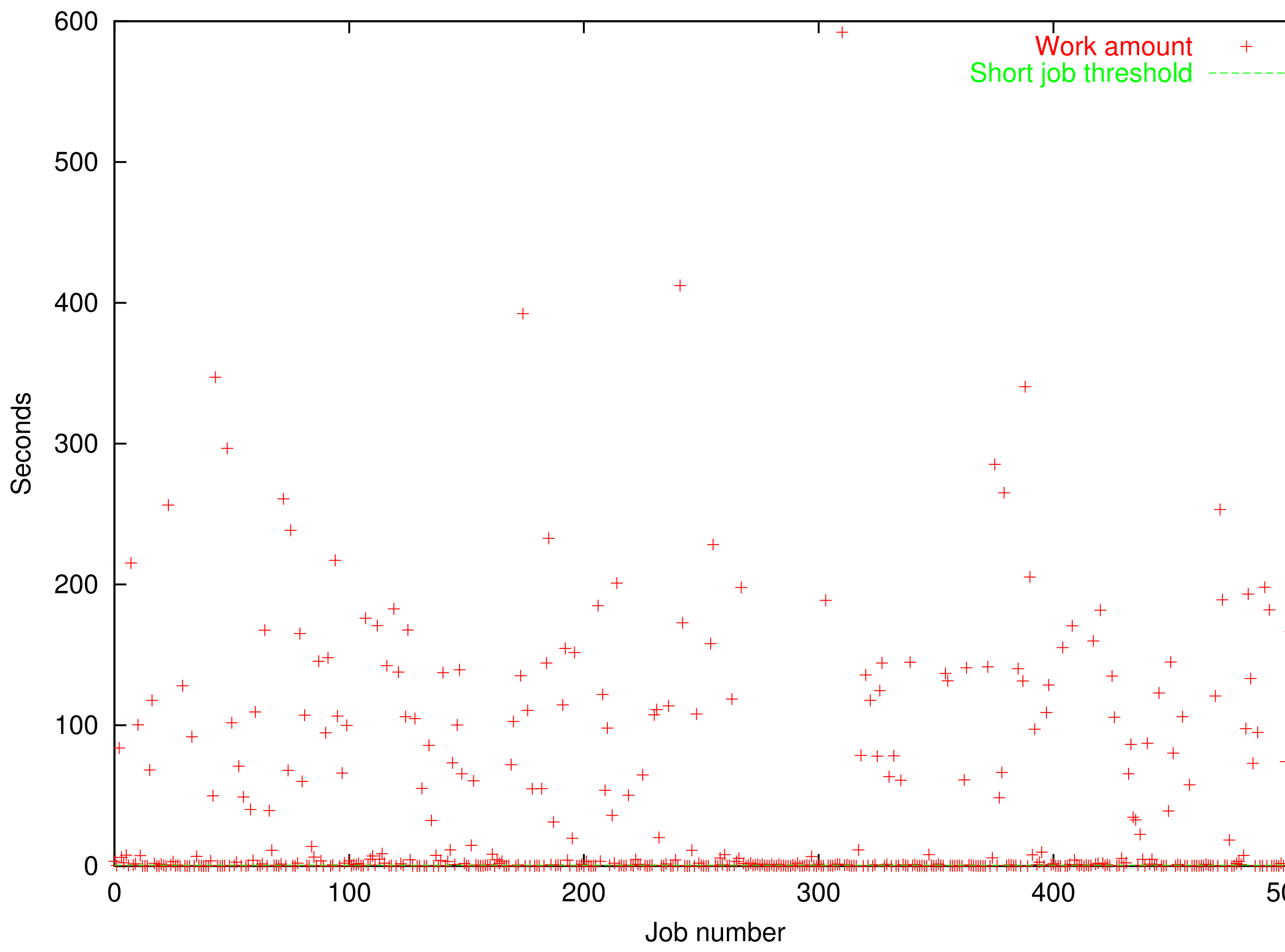

Figure 5.4: Work amount for mixed workload jobs

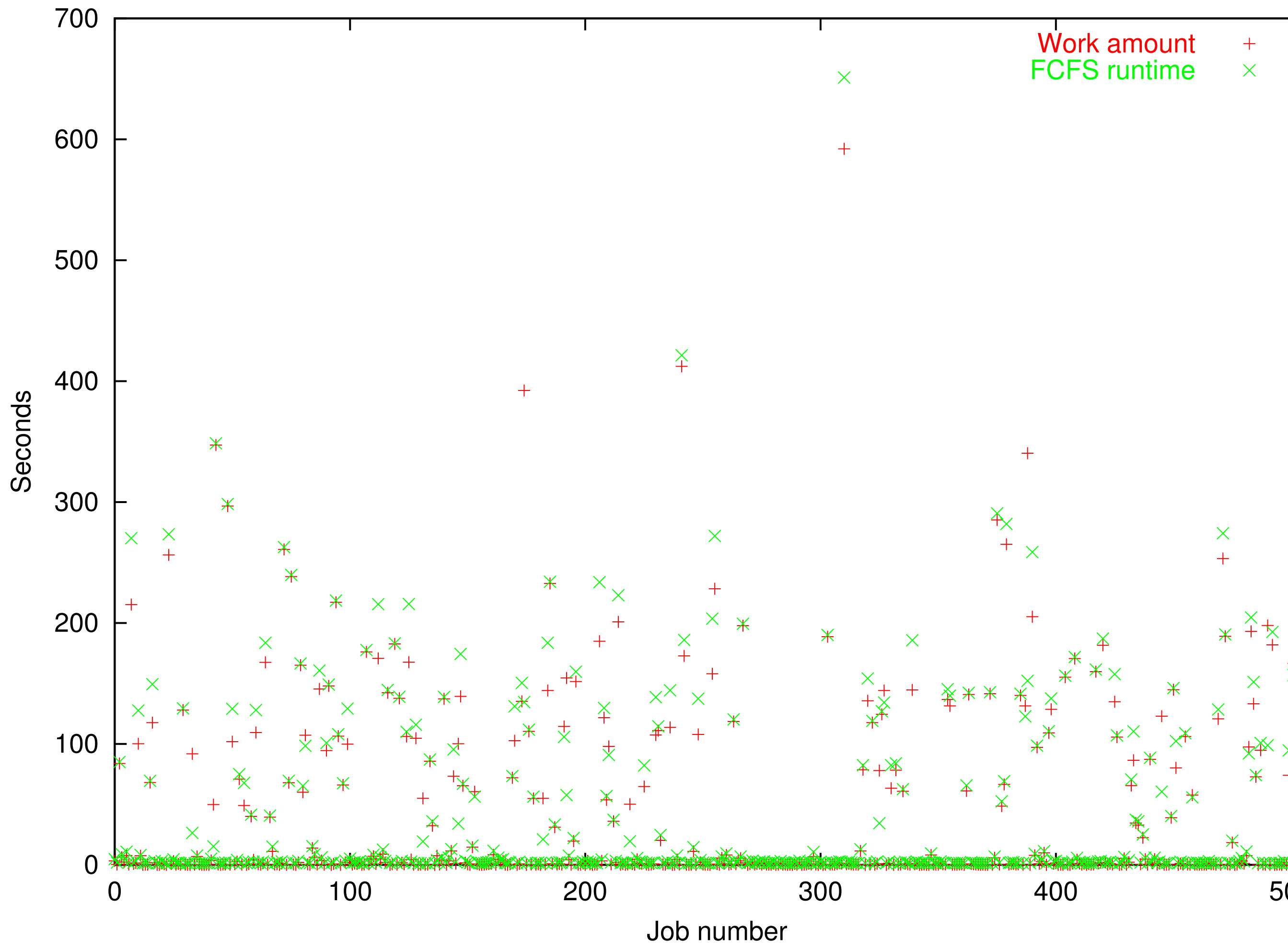

Figure 5.5: Work amount and runtimes with FCFS

granularity is fine and with high variance, as explained in Section 5.2.3. However, the overall effect of these cases is relatively small.

The picture is entirely different when we look at average wait times (and response times). Since no two jobs can occupy the same PE, most jobs have to wait until all previous jobs have completed, and for moderate and heavily loaded workloads, these times can easily accumulate so that late-arriving jobs might wait for very long periods before being allocated to a partition, especially if their PE requirement is high (a process with a low requirement for PEs needs to wait less time on average until that number of PEs becomes available). This can be readily seen in Figure 5.6, which shows how the wait time increases over time and even passes 3500 seconds (97 hours of real time). The average wait time for short jobs is 2233 seconds - far more than is acceptable for interactive jobs.

5.4.3 Local Scheduling

Local scheduling represents the other extreme in scheduling coordination, with no coordinated scheduling at all. We might therefore expect the runtime performance of jobs, especially communicating jobs, to be rather poor. On the other hand, the average wait time for jobs should be much lower than that of FCFS, because multiprogramming allows the allocation of jobs to non-dedicated processes. Still, since an MPL of 4 is enforced, and since jobs take relatively long to terminate, in some events it can take a significant amount of time until resources become available for allocation. These expectations are confirmed in Figures 5.7 and 5.8. It can be readily seen that runtimes are much longer in local scheduling, especially for long jobs (averaging 215 seconds, more than three times the amount of work). Wait times are relatively low for most cases, but on periods when the system is particularly loaded it can rise dramatically, and increase the average wait times (84.1 and 156.9 seconds for short and long jobs, respectively).

5.4.4 Gang Scheduling

Gang scheduling was shown in previous studies to perform better than FCFS in terms of wait time, and better than local scheduling in terms application runtime performance. Indeed, the average runtime of short and long jobs for gang scheduling are 1.782 and 191.972 seconds respectively. Average wait times for short and long jobs are 52.40 and 99.26 seconds respectively, much better than FCFS scheduling, but also significantly better than local scheduling, probably due to the fact that jobs terminate relatively quickly (since they have no synchronization difficulties from mis-scheduling), and free

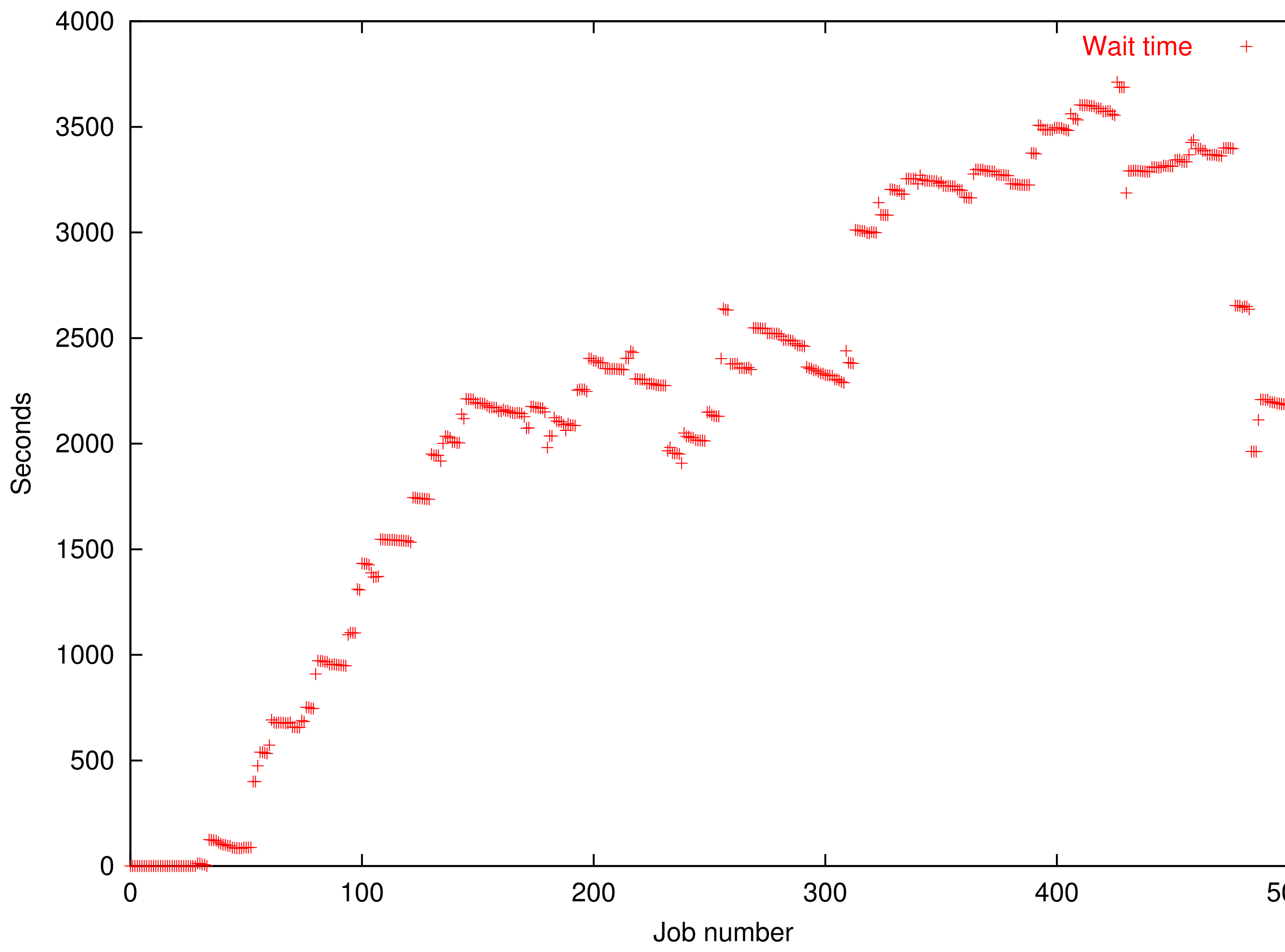

Figure 5.6: Job wait times with FCFS

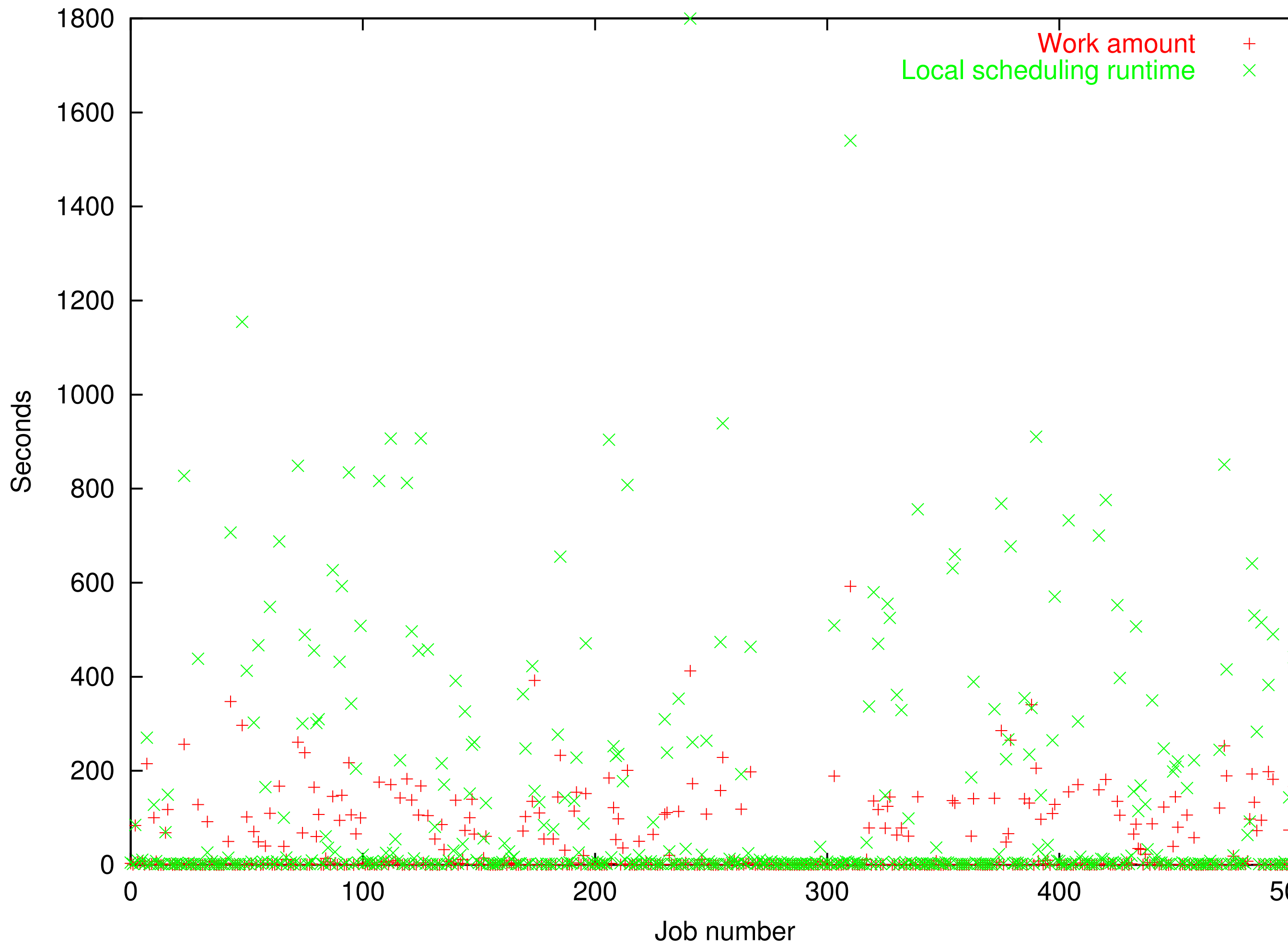

Figure 5.7: Work amount and runtimes with local scheduling

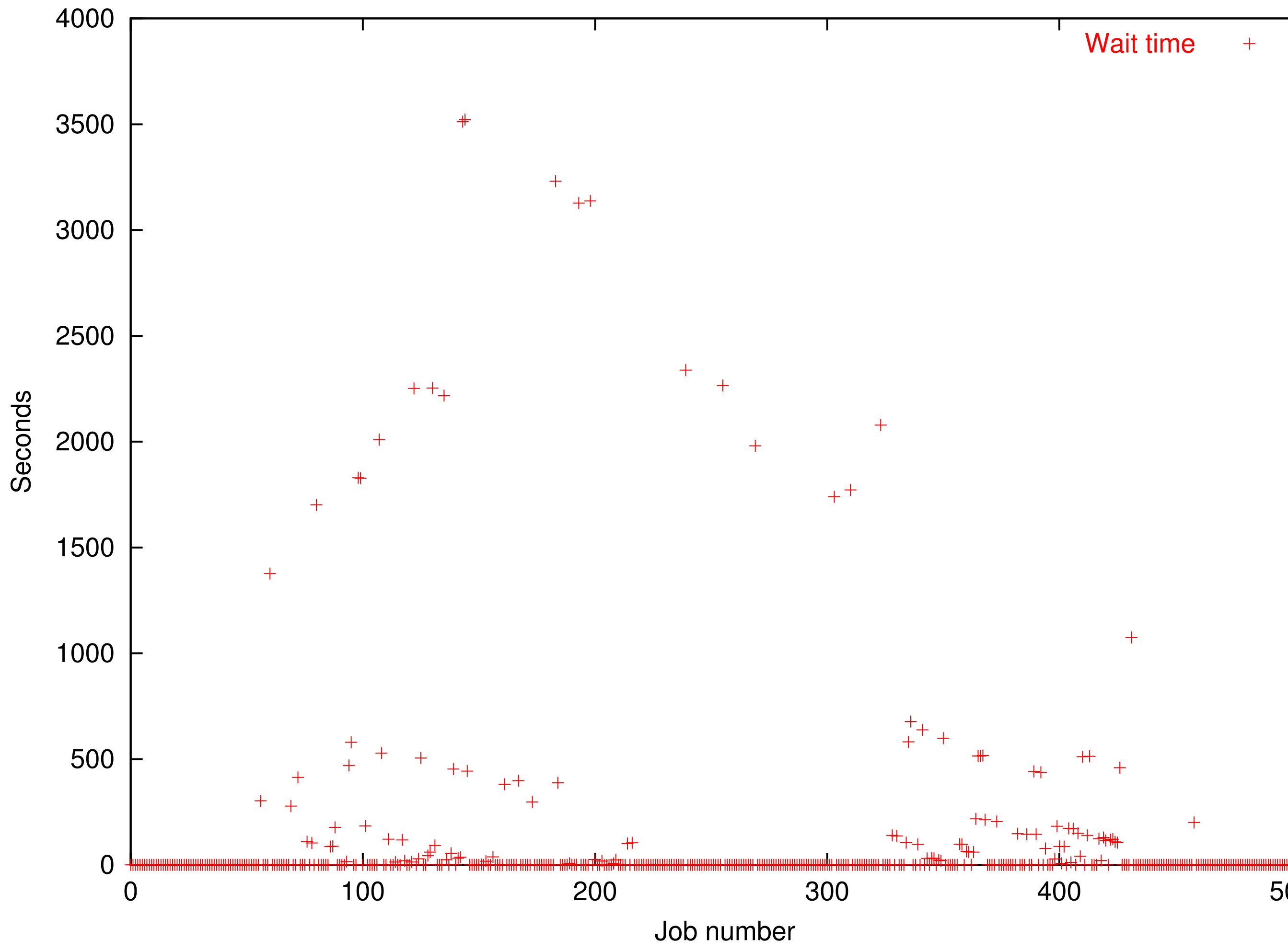

Figure 5.8: Job wait times with local scheduling

resources promptly. Figures 5.9 and 5.10 show the runtimes and wait times of jobs with gang scheduling, respectively.

5.4.5 Implicit Coscheduling

In several studies ICS was shown to be one of the best coschedulers for application performance (see Chapter 2). Since ICS deschedules processes that block on synchronous communication (after an initial spin time), it also offers a basic method to tackle load imbalances, assuming it has enough processes to fill in the gaps. Figures 5.11 and 5.12 show once more the runtimes and wait times of the workload jobs while running under ICS. It can be seen that a significant amount of the jobs has a near-zero wait time, although surprisingly, some jobs have relatively high wait times even when compared to GS. On the average, the performance is relatively good, with average runtimes of 1.776 seconds and 182.59 seconds for short and long jobs respectively, and average wait times of 67.07 and 115.91 seconds.

5.4.6 Flexible Coscheduling

It should be interesting to see whether FCS can cope with this mixed workload better than ICS and GS. Figure 5.13 shows the runtimes for jobs under FCS, and seems somewhat similar to that of ICS, with some of the slower jobs running better in FCS than GS or ICS (there should be little difference for short jobs, since they are initially coscheduled anyway). Indeed, the average runtime for long jobs under FCS is 181.03 seconds, not much lower than ICS, but still somewhat better due to the reduction of runtime in for long jobs. For short jobs the average runtime is actually slightly higher than ICS, (1.778 seconds), which could be either attributed to the switch some of these jobs do to the initial DC state (and thus possibly hamper the job's need for synchronization), or to fluctuations in the experiments. The situation with the average wait time is also positive, as is shown in Figure 5.14. The average wait times for short and long jobs are 65.21 and 112.26 seconds respectively. Overall, FCS performs better than most of the tested algorithms in terms of both job runtime and wait time, although the differences with ICS are not large. This could be either because ICS is as well adapted to this kind of workload as FCS is, and may change under heavier workloads.

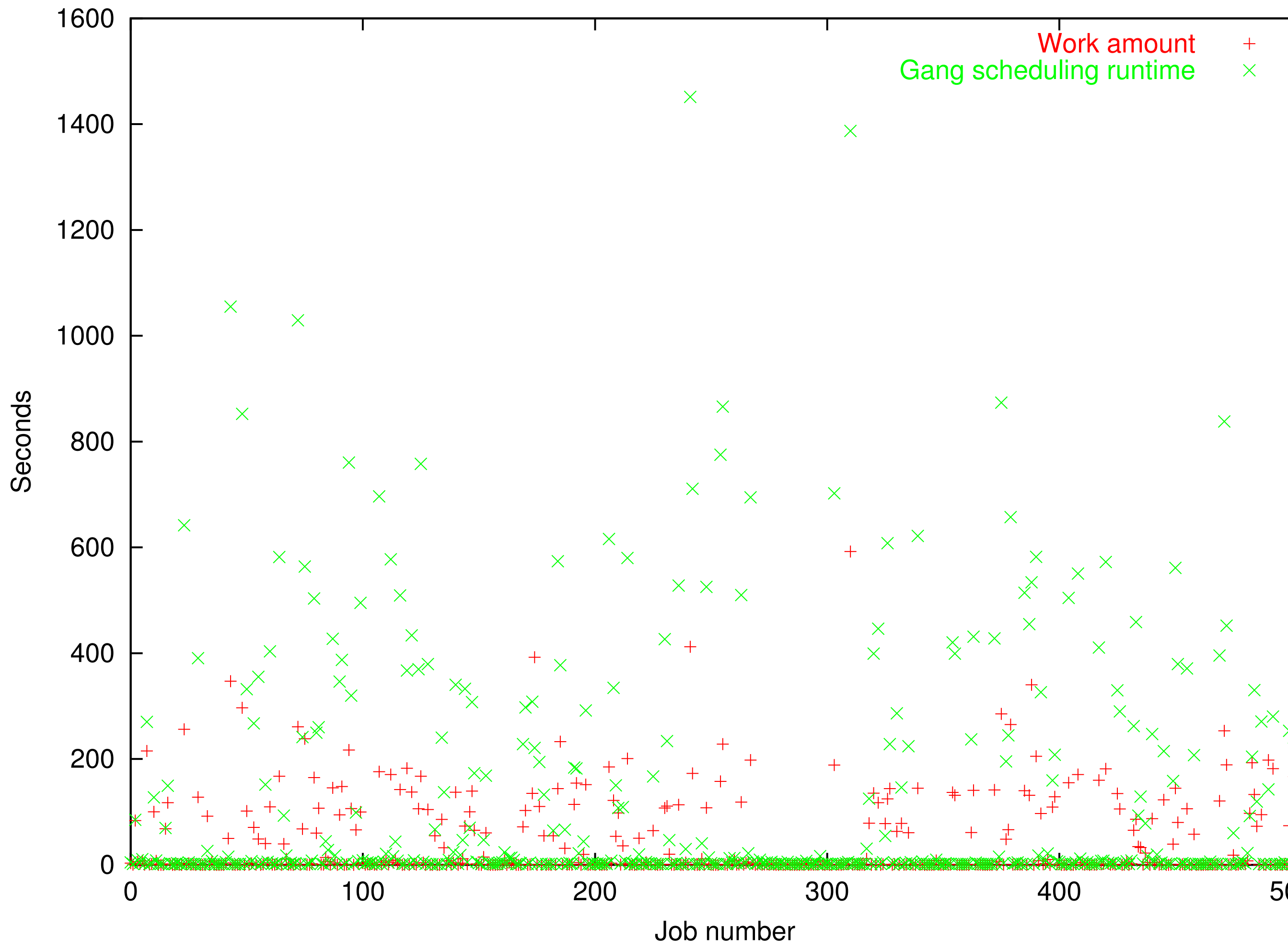

Figure 5.9: Work amount and runtimes with gang scheduling

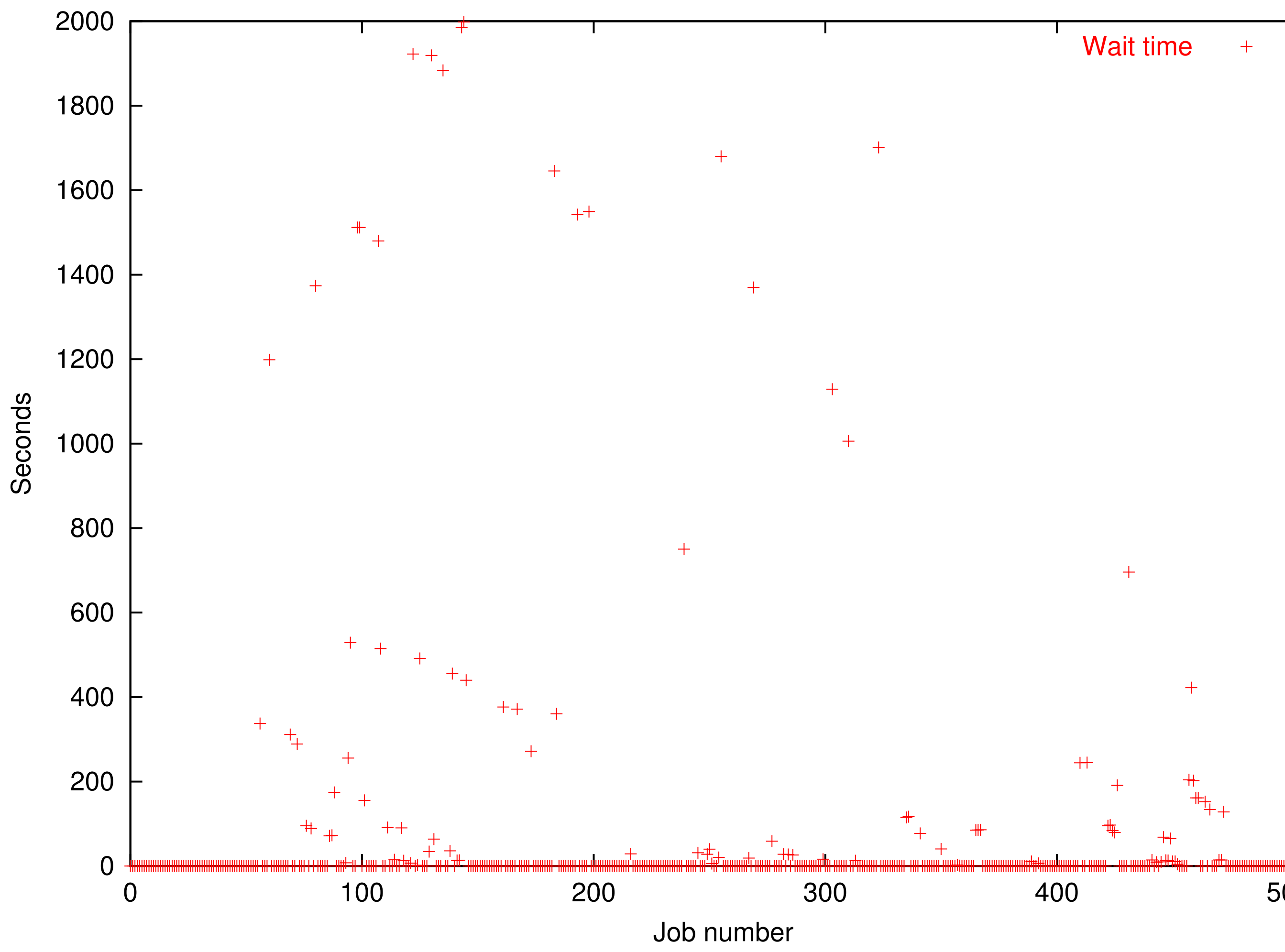

Figure 5.10: Job wait times with gang scheduling

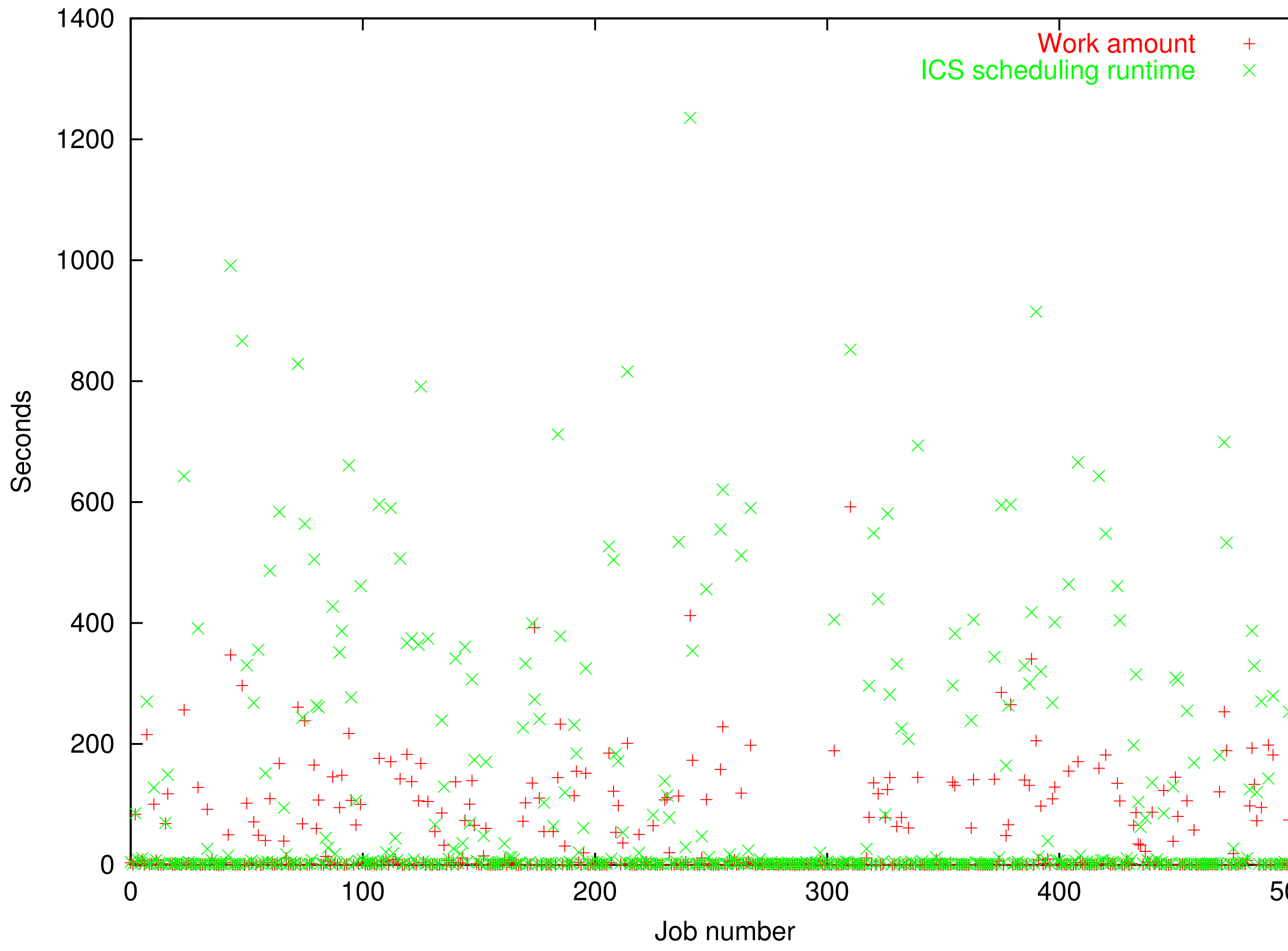

Figure 5.11: Work amount and runtimes with ICS scheduling

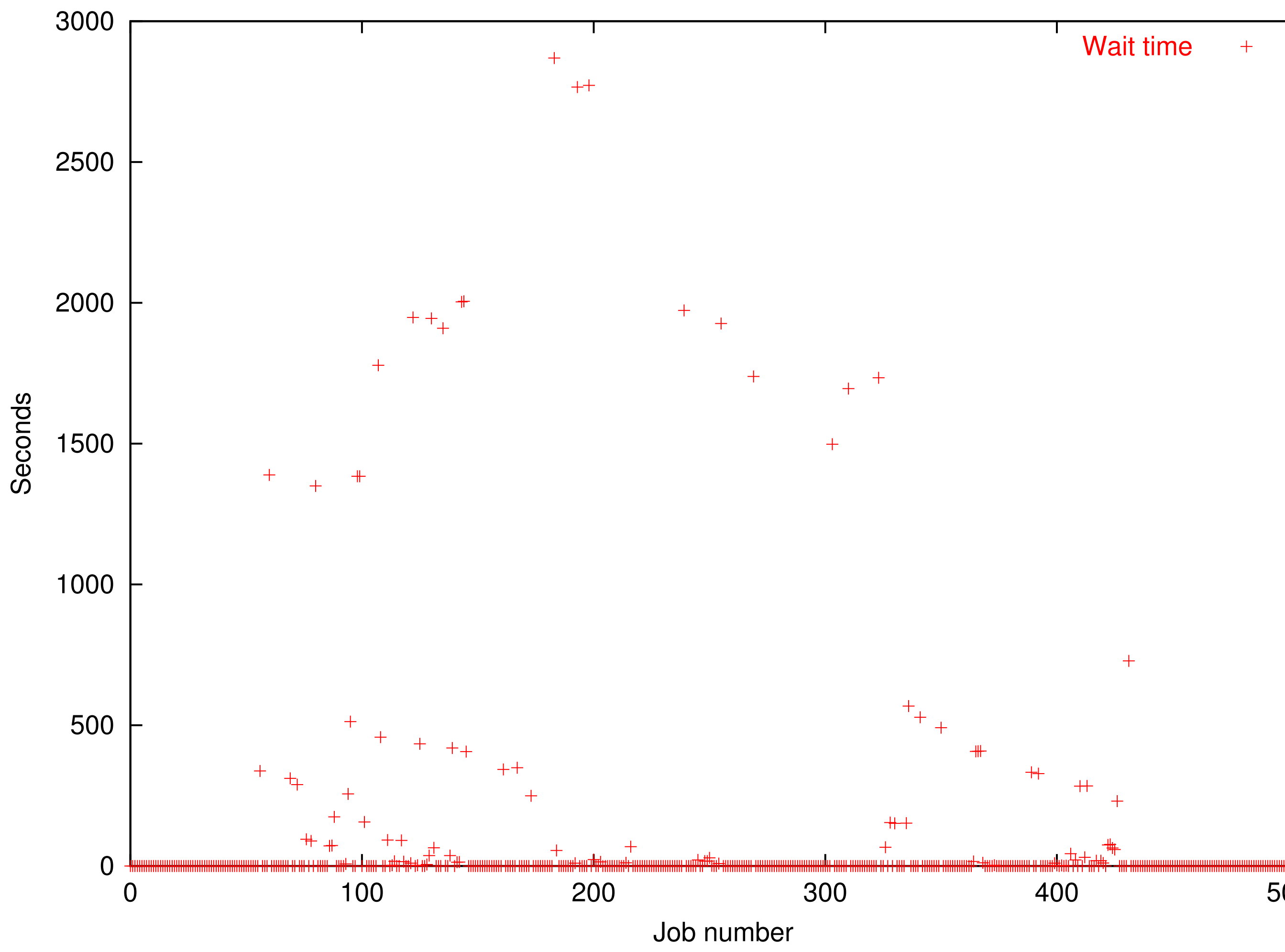

Figure 5.12: Job wait times with ICS scheduling

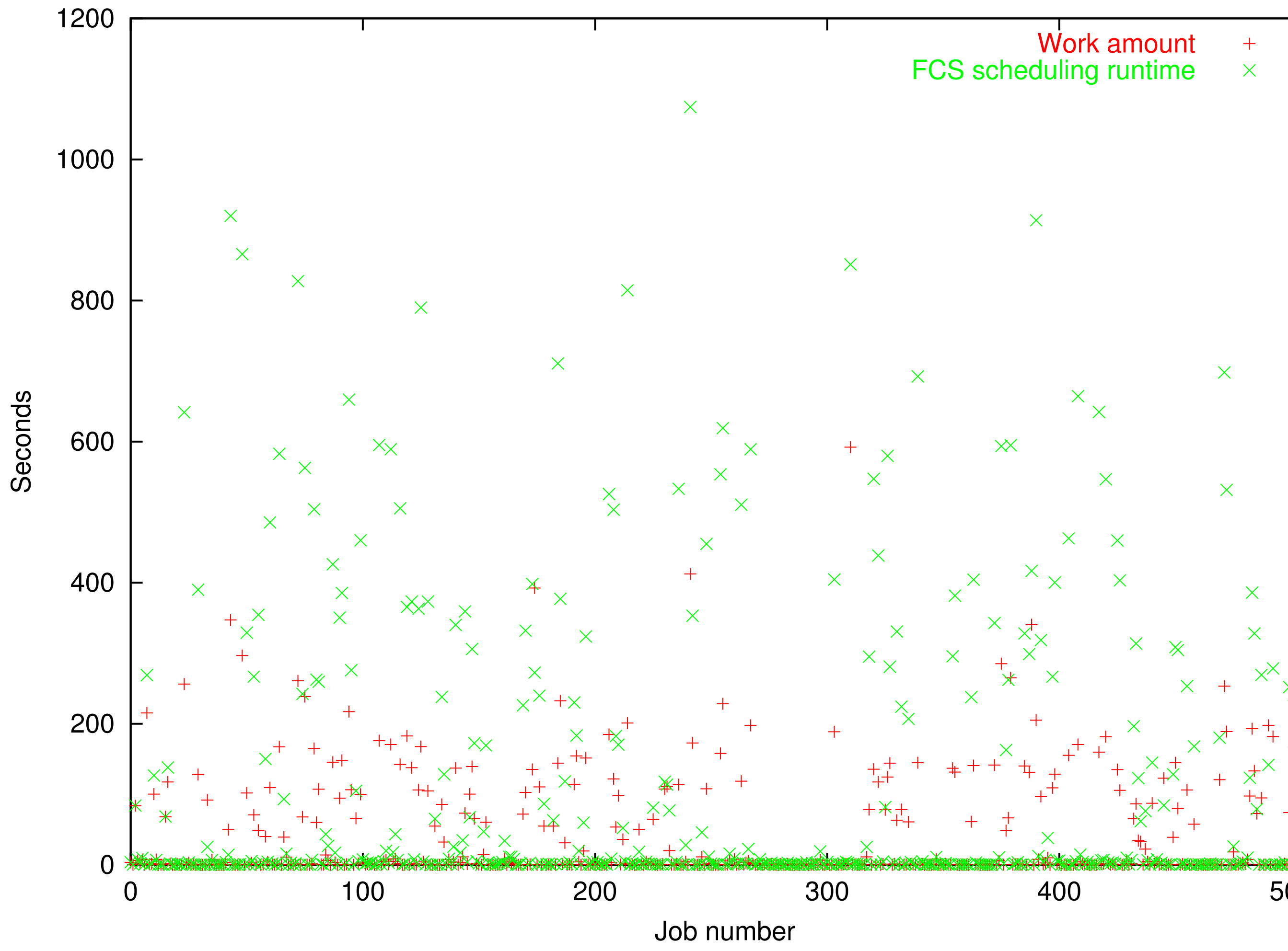

Figure 5.13: Work amount and runtimes with FCS scheduling

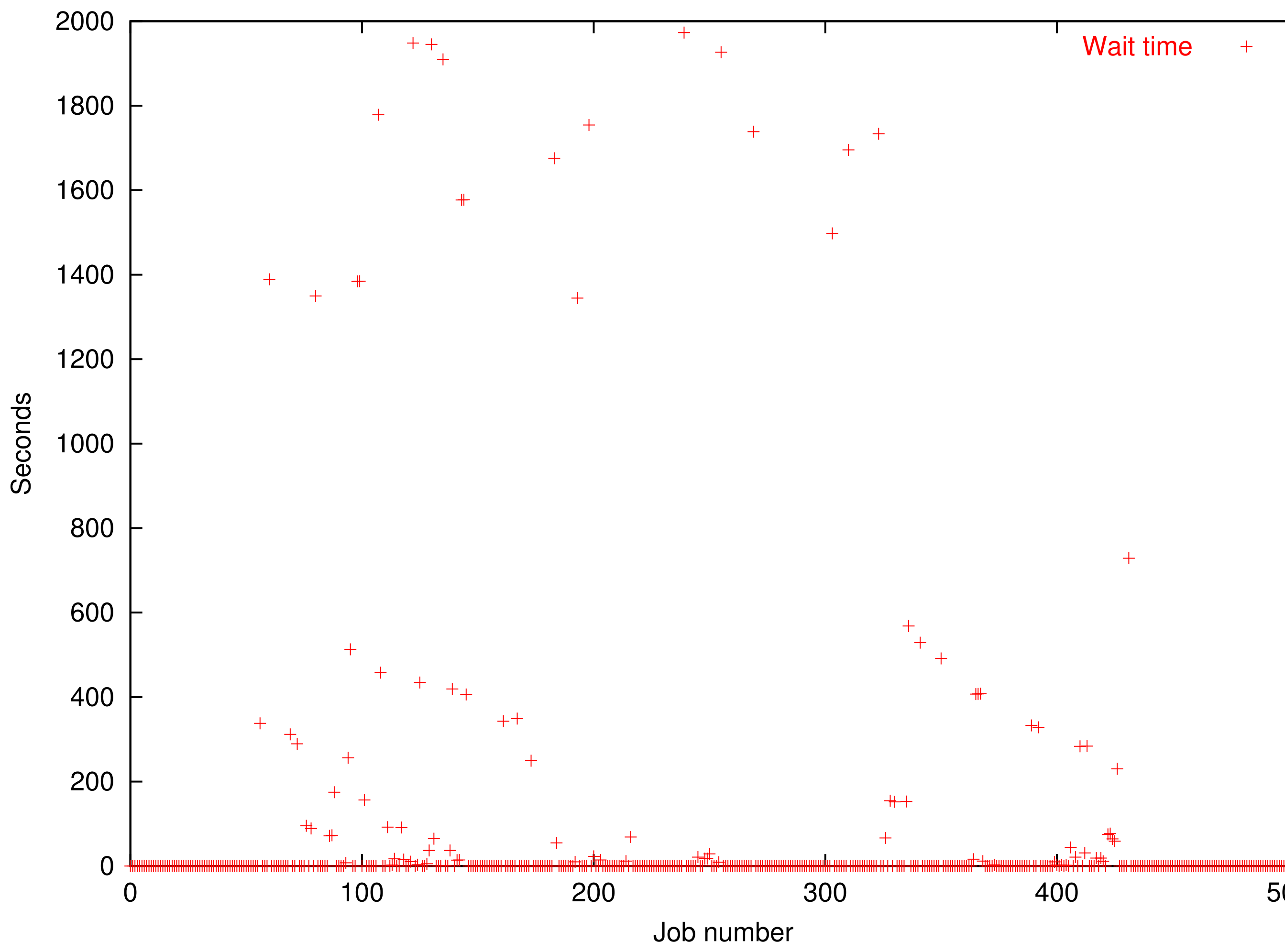

Figure 5.14: Job wait times with FCS scheduling

6 Concluding Remarks

This thesis addressed the issue of load imbalances in parallel programming, and ways to tackle them. A new method was suggested to collect dynamic information on the communication behavior of the applications, and use this information to classify processes according to their coscheduling needs. A novel scheduling algorithm then uses this information to make intelligent scheduling decisions that increase the utilization of the system, while not harming tightly-coupled jobs. We have seen that this new classification method can classify correctly various types of processes. Furthermore, the scheduler compares favorably with all four other scheduling algorithms, when running a workload with a mix of jobs of various jobs.

Possibly as important, this thesis contributes an implementation of an advanced platform for the evaluation of scheduling algorithms, that runs real MPI applications on real, high-performance clusters. The scheduler testbed was shown to be very efficient in terms of overhead and scalability (both in number of nodes and multiprogramming level), especially when comparing it to the commercial RMS scheduler. This testbed should be the basis for many fruitful future studies into various aspects of jobs scheduling.

Future Work

While this work is a complete study of a versatile scheduling system and a new scheduling algorithm, there are still many interesting venues for research that can be based on it, and will be covered in future studies. Of those, the most important is probably a continuing study of the properties of FCS in particular and how dynamic communication monitoring can assist in making intelligent scheduling decisions in general. To this end, further work on FCS could compare different workloads and job mixes, to find for example the saturation point of each scheduling algorithm, and better-define the cases were FCS performs best (or worst). Another extension to the FCS algorithm can include the detection of RE processes using exchange of information between the NMs.

Another interesting venue would be to add more scheduling algorithms to the system,

and compare them against those already implemented. In particular, BCS can a very interesting candidate, since it offers a different approach for tackling synchronization problems.

Another important issue that is outside the scope of this work but can easily enhance it is the effect of I/O jobs on each scheduler. This is a somewhat complex subject, due to the many aspects of I/O that must be identified and modeled.

Once a good understanding of the basic properties of FCS and other scheduling algorithms under various conditions is obtained using synthetic benchmarks, real applications can be tested. This can include application from production sites, such as LANL, and benchmark application suites such as SPLASH [66], SPLASH-2 [75, 82] and NAS NPB-2 [5, 83]. Also, an heterogeneous cluster can be assembled to test whether hardware heterogeneity is different from software heterogeneity.

Acknowledgments

First and foremost, I am indebted to my adviser Dr. Dror Feitelson and supervisor Dr. Fabrizio Petrini. Without their guidance, critique and inspiration, this work could never have succeeded.

Several people have assisted in the many complex technical aspects of this work: Salvador Coll, Juan Fernandez Peinador and the Quadrics team made the implementation of the Quadrics communication layer possible. I owe many thanks to my colleagues from the parallel systems laboratory at Hebrew university for their help with this work. Among them, Yoav Etsion and Dan Tsafrir guided me through the mazes of ParPar; Uri Lublin assisted with his workload model; and especially Tomer Klainer, who helped enormously in numerous aspects, ranging from programming issues to system administration.

To Prof. Darren Kerbyson I owe many thanks for reviewing this work and helping to bring to work to light. In addition, Duncan Roweth of Quadrics helped in gaining understanding of the RMS scheduler and properties. Andrea Arcangeli of Suse assisted with the the processor affinity patch, and provided an excellent tutorial on the Linux kernel.

Last but not least, my family deserve the warmest thanks for their infinite patience, support, and faith. Without it, this work may never have been completed.

Glossary

AA	All-to-All (communication pattern)
API	Application Programmer's Interface
BCS	Buffered Coscheduling
BSP	Bulk-Synchronous Parallel
CMS	Cluster Management System
COTS	Commercial-of-the-shelf
CPU	Central Processing Unit
DCS	Dynamic Coscheduling
ECC	Error Correcting Code
FCS	Flexible CoScheduling
FCFS	First-Come-First-Serve
FCFS-BF	First-Come-First-Serve with backfilling
GS	Gang Scheduling
GUI	Graphical User Interface
HPC	High-Performance Computing
ICN	Interconnection Network
ICS	Implicit Coscheduling
I/O	Input/Output
LANL	Los Alamos National Laboratory
LSF	Load Sharing Facility
MM	Machine Manager (daemon)
MPL	Multi-Programming Level
MPP	Massively Parallel Processing
NIC	Network Interface Card
NN	Nearest-Neighbour (communication pattern)
NM	Node Manager (daemon)

NOW	Network of Workstations
NQS	Network Queueing System
OS	Operating System
PB	Periodic Boost
PBS	Portable Batch System
PE	Processing Element
PID	Process Identifier
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memorys
RMS	Resource Management System
SB	Spin-block
SMP	Symmetrical Multi Processing
PL	Program Launcher (daemon)
VPID	Virtual Process ID

Bibliography

[1] Cosimo Anglano. A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations. In *Proceedings of the Ninth International Symposium on High Performance Distributed Computing (HPDC 2000)*, Pittsburgh, PA, August 2000.

[2] Olaf Arndt, Bernd Freisleben, Thilo Kielmann, and Frank Thilo. A comparative study of online scheduling algorithms for networks of workstations. *Cluster Computing*, 3(2):95–112, Sep 2000.

[3] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[4] Remzi Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Amin Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[6] Mark A. Baker, Geoffrey C. Fox, and Hon W. Yau. Cluster computing review, npac technical report, sccs-748, syracuse university, november 1995.

[7] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[8] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 45–69. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[9] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[10] 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.

[11] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, number 6, pages 12–24, Nov 1994.

[12] Sucheta Chodnekar, Viji Srinivasan, Aniruddha S. Vaidya, Anand Sivasubramaniam, and Chita R. Das. Towards a communication characterization methodology for parallel applications. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 310–319, San Antonio, TX, February 1997.

[13] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, number 3, pages 590–597, December 1991.

[14] Robert Cypher, Alex Ho, Smaragda Konstantinidou, and Paul Messina. A quantitative study of parallel scientific applications with explicit communication. *The Journal of Supercomputing*, 10(1):5–24, 1996.

[15] Fred Douglis and John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[16] Allen B. Downey. A Parallel Workload Model and its Implications for Processor Allocation. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC 97)*, Portland, OR, August 1997.

[17] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[18] Yoav Etsion and Dror G. Feitelson. User-Level Communication in a System with Gang Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, IPDPS2001*, San Francisco, CA, April 2001.

[19] Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, volume 16, Cancun, MX, May 2000.

[20] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 2000.

[21] Fabrizio Petrini and Wu-chun Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, April 2000.

[22] Fabrizio Petrini and Wu-chun Feng. Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements. In *6th Workshop on Job Scheduling Strategies for Parallel Processing*, Cancun, MX, May 2000.

[23] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – Proceedings of the IPPS'96 Workshop*, volume 1162, pages 89–110. Springer, 1996.

[24] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. technical report, ibm t. j. watson research center, 1997. rc 19970 - second revision. Technical report, Yorktown Heights, NY, 1997.

[25] Dror G. Feitelson, Anat Batat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc Volovic. The ParPar System: a Software MPP. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and systems, pages 754–770. Prentice-Hall, 1999.

[26] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, 1997.

[27] Dror G. Feitelson and Larry Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[28] Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[29] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 1998.

[30] Eitan Frachtenberg and Fabrizio Petrini. Overlapping of Computation and Communication in the Quadrics Network. Technical Report LAUR 01-4695, Los Alamos National Laboratory, August 2001.

[31] Eitan Frachtenberg and Fabrizio Petrini. Scheduler Testbed System Design. Technical Report LAUR 01-4694, Los Alamos National Laboratory, August 2001.

[32] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang Scheduling with Lightweight User-Level Communication. In *2001 International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[33] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette. An evaluation of parallel job scheduling for ASCI Blue-Pacific. In *Proceedings of Supercomputing (SC99)*, Nov 1999.

[34] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Syetems. In *6th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '96)*, pages 1–9, Annapolis, MD, October 1996.

[35] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message Passing Interface. In *Second International Euro-Par Conference, Volume I*, number 1123 in LNCS, pages 128–135, Lyon, France, August 1996.

[36] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference*, volume 2, The MPI Extensions. The MIT Press, 1998.

[37] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–132, May 1991.

[38] R. L. Henderson. Job scheduling under the portable batch system. *Lecture Notes in Computer Science*, 949:279–294, 1995.

[39] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo. A scalable time-sharing scheduling for partionable distributed memory parallel machines, January 1995.

[40] A. Hori, H. Tezuka, Y. Ishikawa, and N. Soda. Implementation of gang-scheduling on workstation cluster. *Lecture Notes in Computer Science*, 1162:126–139, 1996.

[41] Intel Supercomputer Systems Division. *Paragon User's Guide*, Jun 1994.

[42] James Patton Jones and Bill Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[43] S. Karlson and M. Brorsson. A comparative characterization of communication patterns in applications using mpi and shared memory on an ibm sp2. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC)*, Las Vegas, NV, February 1998.

[44] JunSeong Kim and David J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC)*, pages 202–216, Las Vegas, NV, February 1998.

[45] William T. C. Kramer and James M. Craw. Effective use of cray supercomputers. In *Proceedings of the Supercomputing 89*, pages 721–731, New York, NY, 1989. ACM Press.

[46] Richard N. Lagerstrom and Stephen K. Gipp. PScheD: Political scheduling on the CRAY T3E. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 117–138. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[47] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[48] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. Number 4, pages 272–285, Jun 1992.

[49] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. pages 226–236, May 1990.

[50] Shau-Ping Lo and Virgil D. Gligor. A comparative analysis of multiprocessor scheduling algorithms. Number 7, pages 356–363, Sep 1987.

[51] Uri Lublin. A workload model for parallel computer systems, 1999. Master's thesis, Hebrew University, 1999. (In Hebrew).

[52] Jon Mauney, Dharma P. Agrawal, Y. Choe, Edwin A. Harcourt, S. Kim, and W. J. Staats. Computational models and resource allocation for supercomputers. In *Proceedings of the IEEE*, volume 77, pages 1859–1874, December 1989.

[53] 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, June 2001.

[54] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[55] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[56] Fabrizio Petrini, Federico Bassetti, and Alex Gerbessiotis. A New Approach to Parallel Program Development and Scheduling of Parallel Jobs on Distributed Systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume I, pages 546–552, Las Vegas, NV, July 1999.

[57] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects 9*, Stanford University, Palo Alto, CA, August 2001.

[58] Fabrizio petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware-Based and Software-Based Collective Communication on the Quadrics Network. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, October 2001.

[59] Fabrizio Petrini, Adolfy Hoisie, Wu chun Feng, and Richard Graham. Performance Evaluation of the Quadrics Interconnection Network. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.

[60] Quadrics Supercomputers World Ltd. *Elan Kernel Communication Manual*, December 1999.

[61] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

[62] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[63] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.

[64] Quadrics Supercomputers World Ltd. *RMS User Manual*, April 2000.

[65] Mark K. Seager and James M. Stichnoth. Simulating the scheduling of parallel supercomputer applications. Technical Report UCRL-102059, Lawrence Livermore National Laboratory, Sep 1989.

[66] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[67] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. The MIT Press, 1998.

[68] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[69] Patrick Sobalvarro and William E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[70] L. Kent Steiner. Evolution of supercomputers. In *ACM annual conference on The range of computing : mid-80's perspective*, pages 112–116, Denver, CO, October 1985. Assoc. for Computing Machinery, ACM PressNew York, NY, USA.

[71] P. Steiner. Extending multiprogramming to a dmpp. *Future Generation Comput. Syst.*, 8(1-3):93–109, July 1992.

[72] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX Scheduling for Large Systems. In *Proceedings of the USENIX Winter Conference*, pages 111–139, Denver, CO, 1986.

[73] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[74] Jon B. Weissman and Xin Zhao. Scheduling Parallel Applications in Distributed Networks. *Cluster Computing*, 1(1):109–118, 1998.

[75] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, , and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[76] Yanyong Zhang, Anand Sivasubramanian, José Moreira, and Hubertus Franke. Impact of workload and system parameters on next generation cluster scheduling mechanisms. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):967–985, September 2001.

[77] Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, 1992.

[78] http://www.top500.org.

[79] http://www.quadrics.com.

[80] ftp://ftp.us.kernel.org/pub/linux/kernel/people/rml/cpu-affinity/v2.4/.

[81] http://www.cs.huji.ac.il/labs/parallel/parpar.shtml.

[82] http://www-flash.stanford.edu/apps/SPLASH/.

[83] http://www.nas.nasa.gov/Software/NPB/.