# The Artaras Job Scheduler
# for Multicomputers

Dion A.C. Wooning

March 2005

**TU**Delft

**Delft University of Technology**

# The Artaras Job Scheduler
# for Multicomputers

THESIS

to obtain the title of
Master of Science in Computer Science
at Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Parallel and Distributed Systems Group

by

Dion A.C. Wooning

March 2005

# Graduation Data

Author     : Dion A.C. Wooning
Title        : The Artaras Job Scheduler for Multicomputers
Date       : March 21, 2005
Location  : Faculty EEMCS, Mekelweg 4, Delft

**Graduation committee**

| | |
|---|---|
| Prof. dr. ir. H.J. Sips (chair) | Delft University of Technology |
| Ir. dr. D.H.J. Epema | Delft University of Technology |
| Ir. F. Ververs | Delft University of Technology |

## Abstract

In order to efficiently share a multicomputer among multiple users, a job scheduler is used to allow control and monitoring of the programs executed on the machine. In this work, a job scheduler is designed and implemented for scheduling parallel jobs on a multicomputer. The scheduler can be adapted to support different multicomputers and scheduling algorithms. An implementation is provided for a virtual multicomputer on which parallel programs can be run, allowing experimentation with scheduling algorithms and aiding in providing simulation of a multicomputer environment for debugging parallel applications prior to running them on a production machine.

# Preface

This report is my Master's thesis at the Parallel and Distributed Systems Group of the Department of Software Technology at Delft University of Technology. It is the result of my efforts to design and implement a job scheduler for running and controlling parallel jobs on multicomputers. This thesis was preceded by a literature study on the scheduling of parallel jobs and the provision of fault tolerance through the use of checkpointing mechanisms.

    I would like to thank friends, family and last but not least my supervisor D.H.J. Epema for their patience and support during this project.

<div align="right">

Dion Wooning
Delft, March 2005

</div>

iv

# Contents

# Chapter 1

# Introduction

Multicomputers that are shared by a number of users without some mechanism to aid this sharing are often used in a straightforward but primitive way. Each user who wishes to run a program and finds the machine in use will have to check regularly whether the last running program has finished and the machine has become available. If such is the case, he then has to start the new program manually.

Clearly, this pattern of access to the multicomputer is not very efficient. The machine is left idle for extended periods between running programs, especially during nights and weekends. In addition, users need to negotiate among themselves for turns to use the machine, which becomes difficult to manage if there are many users sharing the multicomputer.

The situation can be improved by the use of a facility called a *job scheduler*. The next section provides a short introduction to this subject. It is followed by an overview of the scheduler designed and implemented for this thesis. Finally, the organisation of the remainder of this thesis is given.

## 1.1 Job Scheduling and Checkpointing

A job scheduler improves sharing a machine among a number of different users or applications by satisfying following primary goals:

- Provide users with easier and more comfortable access to the machine.

- Support sharing of the multicomputer by a larger user base.

- Allow the system to be used more efficiently.

- Give administrators the ability to regulate the machine's usage.

Instead of starting programs directly on a multicomputer in a situation without job scheduler, when a scheduler is used users will *submit* their programs as *jobs* to be placed in a *job queue*, where these jobs remain waiting until they are selected to run. The actual execution of jobs is controlled by the job scheduler: it decides at what time each job will be taken from the queue to be run and on which processors it will be executed. Such scheduling of parallel jobs can be done in several ways, which are discussed more elaborately in [3, 4, 8].

Additionally, it is often desirable to provide multicomputers with a certain degree of fault tolerance in order to reduce the risk of loosing computation time due to system failures. The probability of a failure occurring in multicomputers is much higher than in single-processor systems because the former contain many more components than the latter [7]. In environments where the run-times of the programs range from several hours to several days, the amount of potentially lost computation time is significant.

*Checkpointing* is a technique in which a job makes regular safety stops called *checkpoints* or *recovery points* that will allow it to restart from the last stop made in case of failures, thus minimising the amount of computation time lost. A discussion and overview of various checkpointing techniques can be found in [2, 9, 10].

## 1.2 The Artaras Scheduler

In the following chapters, the design and implementation are discussed of a parallel job scheduler called *"Artaras"* that supports the use of checkpointing. Prior to this project a literature study was performed on the subjects of parallel job scheduling and the provision of fault tolerance through the use of checkpointing techniques [12]. In this literature study, it was also examined what functionality users and administrators would ask of a job scheduler for the DEMOS multicomputer [6, 14]. This has been taken as the basis for the functionality to be provided by Artaras.

The scheduler has been designed to be extensible so that it can be adapted to schedule jobs on multicomputers of different architectures using various scheduling algorithms. At this time, an implementation is provided for a simple virtual multicomputer called *"Virtua"* that can be run on single-processor systems, simulating a multicomputer while running actual parallel programs. Besides aiding development of the scheduler itself, this allows two other interesting possibilities:

- Experimentation with various scheduling algorithms.

- Help creating an environment similar to a real multicomputer to aid testing and debugging parallel programs prior to running them on production machines.

One possible target for further development of the scheduler would be the **D**istributed **A**SCI **S**upercomputer (DAS) [13].

The scheduler has been implemented in C++ for the Linux operating system using the GNU Compiler Collection (GCC) [15] and the GPS developer environment [16]. During development two books on C++ have been found to be of invaluable use and are highly recommended [1, 11].

Artaras has been tested by running test jobs on the Virtua machine. These jobs were computationally trivial, as the computation performed by them was not relevant to the correctness of the scheduler. No extensive analysis has been made of the scheduler's performance at this time.

**Artaras.** This name is not an acronym but in a dramatic break with tradition derived from the combination of *âr, aran*, meaning "king or lord of a region", and *tass, táras*, meaning "labour or task" in the ancient Sindarin language[1].

## 1.3   Organisation of this Thesis

The remainder of this report is organised in the following way:

- Chapter 2 discusses the functionality to be provided by Artaras. It provides an overview of the scheduler's main features, followed by a more detailed look at several aspects.

- In Chapter 3, the scheduler user interface and the ways in which the administrator can control the multicomputer through the scheduler are considered.

- Chapter 4 deals with the technical details of the scheduler. It gives an overview of the internal structure of the scheduler, and takes a closer look at the workings of the various components such as the scheduler daemon and shell commands.

- Chapter 5 discusses the machine and scheduling interfaces that are used to separate machine-specific details and scheduling algorithms from the rest of the scheduler. The implementation of the machine interface for the Virtua and the scheduling algorithm used to schedule on the Virtua are also described.

- In Chapter 6 conclusions are presented about the usability of the job scheduler and recommendations for future work are made.

- Appendix A contains a guide to installing Artaras for further development.

- Appendix B gives an overview of the modules that make up the project.

- Appendix C discusses the formats of the files used by Artaras.

---

[1]As invented by the author and linguist J.R.R. Tolkien, prof. of Anglo-Saxon at the University of Oxford (1892-1973).

# Chapter 2

# Functional Design

In this chapter the functionality offered by the Artaras scheduler will be discussed. First, an overview is given of the scheduler's most important aspects. Then we discuss in more detail the scheduling policy, the handling of computation-time quota, and finally the checkpointing of jobs by the scheduler.

## 2.1 Scheduler Overview

In this section, an overview is presented of the functionality offered by Artaras.

### 2.1.1 Using the Scheduler

Users that wish to access the multicomputer must do so through the scheduler. It allows them to run programs and control and monitor the execution of these programs. The scheduler can be used directly from any workstation that has network access to the multicomputer (see Figure 2.1).

The following actions can be performed by users of the machine:

- Submission of jobs.

- Getting an overview of the job queue and multicomputer/scheduler status, showing the predicted order of execution of jobs.

- Removal of jobs, either running or waiting.

- Moving jobs to a different position in the job queue.

- Getting an overview of current computation-time quota.
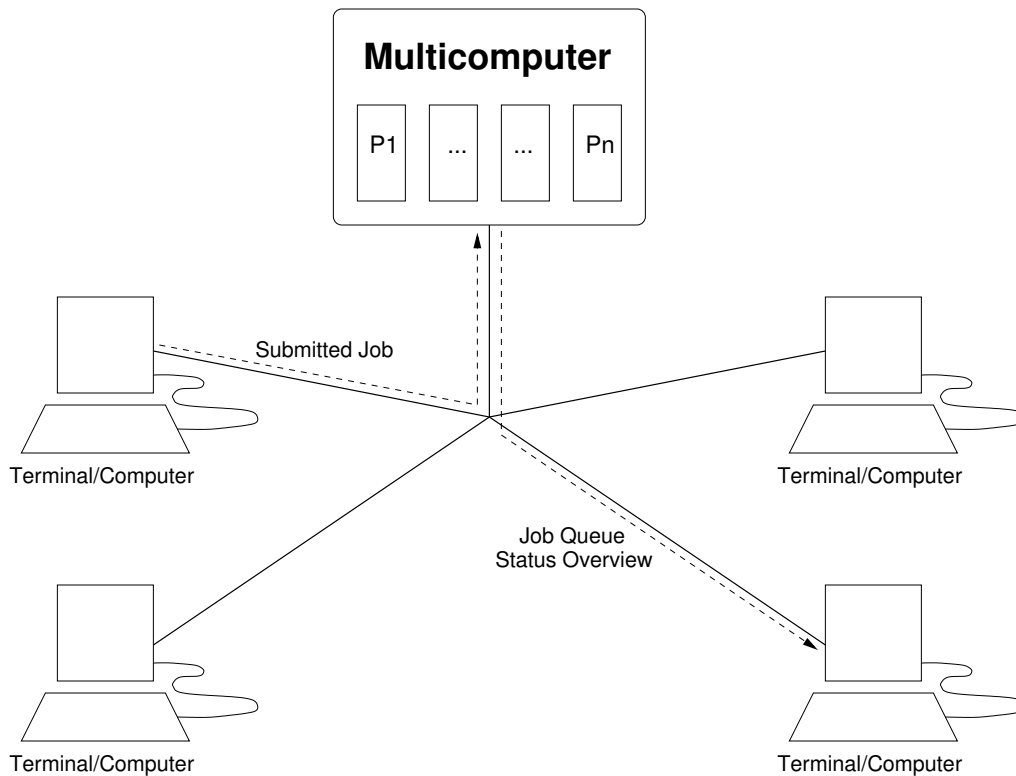
Figure 2.1: Using the scheduler.

The user interface through which users can perform these actions consists of a set of shell commands. Section 3.1 of the next chapter provides a detailed description of each of the available commands.

The administrators of the multicomputer can perform each of the above actions on any user's jobs to control the way in which the machine is used[1]. In addition administrators can:

- Perform scheduler startup and shutdown.

- Configure the scheduler.

- Set user computation-time quota.

- Enable logging of various scheduling events.

The extra administrator shell commands and configuration files through which these actions can be performed are discussed more elaborately in Section 3.2.

---

[1]The exception being that the administrator cannot submit other users' jobs, which is an operation that is generally not needed.

### 2.1.2 Scheduling Strategy

The *scheduling strategy* used by a scheduler describes in what manner jobs are executed. The chosen strategy is *gang scheduling* [12]. This is a method of time slicing under which all processes of a parallel job are scheduled on separate nodes at the same time. This keeps the blocking of processes while communicating to a minimum, and allows a job to make full use of the multicomputer's communication capabilities.

To ensure that applications fit in physical memory, each processing node is used by only one job at any given time. If a job reaches the end of its time slice, its state is stored on disk (*checkpointed*) so that it may be continued from that point during a next time slice, and the job is removed from all nodes it has in use. Since this transfer to disk often requires a fair amount of time during which no useful computation is performed, time slices will be large, and in most cases jobs will be executed from start to finish in the same time slice. Checkpointing is discussed in more detail in Section 2.4.

If the machine is configured as a single partition, only one job will be run on the multicomputer at a given time. If multiple partitions are available separate jobs can be executed simultaneously, one on each of these partitions. In both cases, one scheduler is used to handle all jobs.

### 2.1.3 Job Types

The scheduler supports the execution of jobs of three different types: *normal*, *priority* and *interactive* jobs.

Normal jobs and priority jobs are jobs that can be entirely executed without any user interaction while running. For this reason, they are often called *non-interactive* or *batch* jobs. Non-interactive jobs are executed detached from the shell from which they were submitted, and do not allow input or output to occur with the shell used for submission. The difference between normal and priority jobs lies in the order in which they are run: while normal jobs are executed in the order in which they are submitted to the scheduler, priority jobs may be executed ahead of others, if their results are needed urgently (before a certain deadline).

On the other hand, interactive jobs are meant to be run interactively by the user that submits them. Common purposes for using this type of job are visualisation, demonstration, and testing and debugging. Interactive jobs remain attached to the shell from which they were submitted, so that input and output operations can be performed from and to that shell.

The exact order in which jobs of each type are selected for execution will be discussed in detail as part of the scheduling policy in Section 2.2.

### 2.1.4 Program Classes

The scheduler can handle jobs that belong to one of several program classes. Program classes are used to describe the structure of jobs to the scheduler so that they can be started, suspended, and removed in the appropriate way. Note that the program class to which a job belongs is independent of and should not be confused with the job type as discussed in the previous section.

The following classes are provided as standard:

- SPMD. SPMD stands for *Single Program, Multiple Data*. Jobs in this class consist of a single executable that must be run on each processor on which the job is scheduled for execution.

- Scripts. This class consists of (shell) scripts that launch their processes on each of the appropriate processors.

### 2.1.5 Reliability and Availability

Special care is taken on the part of reliability of the scheduler. It is guaranteed that once a user has submitted a job and the scheduler has accepted it, the job will have been stored in the job queue and will be waiting to be executed according to the scheduling policy described in Section 2.2. The contents of the job queue will be retained at all times, across orderly scheduler shutdowns and restarts but also across reboots of the multicomputer or when the scheduler is unexpectedly killed.

As the scheduler can be used from any machine in the same network as the multicomputer, most of the actions mentioned in Section 2.1.1 can still be performed even if the scheduler itself or the multicomputer are not up and running, albeit with the following limitations:

- New jobs can be submitted, except for interactive ones. Note that although they are queued, no jobs will be started during the time the scheduler is down.

- An overview of the job queue can be obtained. This overview properly reflects any jobs submitted to, moved within, or removed from the queue since the scheduler was stopped. However, the status display of jobs that were running when the scheduler was stopped will remain unchanged until the scheduler is restarted.

- Jobs waiting in the queue can be removed. Jobs that are running cannot be stopped and removed via the scheduler as long as it is down. The removal of such jobs will be postponed until the scheduler is restarted.

- Jobs may be moved to another position in the queue as usual.

- An overview of current computation-time quota can be obtained. The numbers reported take into account any changes resulting from the submission or removal of jobs.

### 2.1.6 Computation-time Quota

The amount of computation time available to each user can be controlled by the system administrator to regulate the use of the machine and to prevent users from monopolising the system. Separate quota exist for normal, priority and interactive jobs. Quota are discussed in more detail in Section 2.3, while Section 3.2.3 of the next chapter describes how the administrator can set the quota for each user.

### 2.1.7 Authorisation

The scheduler itself runs with enough privileges to allow it complete control of the multicomputer. The jobs that are started by the scheduler are executed with the privileges of the users that submitted them.

Since it is not desirable that users can interfere with other users' jobs running on the machine, the scheduler ensures that all commands that access the multicomputer or parts of it can only be successfully run by the user whose running job would be affected.

## 2.2 Scheduling Policy

The *scheduling policy* defines in what order jobs are taken from the job queue and executed on the multicomputer. In this section, the policy of the scheduler is described for each of the different job types: normal, priority and interactive.

### 2.2.1 Normal Jobs

The principal way in which jobs are scheduled is *first-come, first-served (FCFS)*[2]. Under this method, jobs are executed in the order in which they were submitted. Each time a ring of processors becomes available, the scheduler picks the next job from the queue and starts it on the ring. Once started, the job is executed from start to finish. When the job quits, the next job is selected from the queue, and the above process is repeated.

To be able to make predictions as to when each job will start and finish, users are required to provide a maximum run-time when submitting a job. If, during execution, it appears a job is going to exceed this limit, it is preempted and *checkpointed*, see Section 2.4. The scheduler starts checkpointing jobs early enough to ensure checkpointing is

---

[2]Sometimes also referred to as *first-in, first-out (FIFO)*

completed before the given maximum run-time will be reached. After the job has been checkpointed, its processes are removed from the machine. The job does not return to the job queue, but if needed, a user can restart his job at a later time by resubmitting it, as discussed in Section 2.4.

The advantage of the FCFS approach combined with the possibility of predicting worst-case job starting and finishing times, is that users get a clearer idea of when to expect their jobs to be ready. However, in the following two cases, the FCFS policy proves to be awkward or inflexible:

- Running interactive jobs. Since under FCFS, jobs are executed in the order of arrival, interactive jobs would be queued while waiting for the preceding jobs to be executed, most likely forcing users to wait for unacceptably long times.

- The execution of priority jobs: non-interactive jobs whose results are needed with some urgency. Under FCFS, such jobs would have to wait for the completion of all jobs that were submitted earlier and are therefore ahead in the queue, irrespective of the relative urgency with which the results of these jobs are actually needed.

To deal with both scenarios, the scheduler executes interactive and priority jobs in an order different from FCFS. The exact ways in which they are scheduled are discussed in Sections 2.2.2 and 2.2.3.

**Acceptance of Normal Jobs**

When a normal job is submitted to the scheduler, it is first checked whether the job will be able to complete under the normal-job quotum of the user. This is found to be the case if the job's maximum run-time that was provided by the user is less than or equal to the current quotum. If this is not the case, submission will fail, and the user will receive an error message stating this.

A user submitting a job may optionally indicate a specific partition on which the job is to be executed[3] and/or specify the number of processors the job requires. The default behaviour in case the machine consists of one single partition or multiple equal-sized partitions is for the job to be able to execute on any partition, with the number of processors set equal to the number of processors in the partition. If there are partitions of different sizes and the user does not indicate a partition to be used, the number of processors is no longer optional and must be supplied by the user. Finally, if a user specifies a partition, the default number of processors will be set to the number of processors in the indicated partition. Submission will fail and the user will be notified if it is found at submission time that the scheduler will not be able to successfully run the job:

- The indicated partition does not exist, or

---

[3]This allows the execution of jobs that use hard-coded processor IDs.

- No partition exists with at least the requested number of processors, or

- The specified partition has fewer processors than was asked for.

For interactive and priority jobs the acceptance procedure is similar, using the quota corresponding to those job types instead. For exact details see Sections 2.2.2 and 2.2.3, respectively.

**Execution of Normal Jobs**

When a partition becomes available due to the fact a job either finished or was checkpointed and removed, the scheduler will search the queue for the next job to be executed on that partition. Assuming the queue is ordered with the jobs that were submitted earliest first, the scheduler starts by examining the first job and proceeds toward the rear of the queue until a job is found that can be executed on the available partition. This depends on the following two conditions:

1. The available partition consists of exactly the number of processors that was specified during submission of the job under consideration. If not, the partition must either be the smallest in the system that does have a larger number of processors than required by the job, or one of several equal-sized smallest partitions, if more than one exist. The partition is then sized down by executing dummy processes on the extra nodes that pass on received data if necessary, effectively creating a partition of the size requested by the job.

2. Either the user has specified that the job is able to run on any partition of the correct size, or the available partition is the specific partition indicated by the user.

If no suitable job has been found after examination of the entire job queue, the partition will remain unused until the submission of new jobs will cause the scheduler to re-examine the queue and find a job that satisfies the above conditions.

Note, that according to the two conditions above it is possible for a partition to remain unused even if there is a job in the queue that could potentially be executed on it if the partition were resized, if that partition is not the smallest in the system. This is the result of the decision to make only the smallest possible partition available for downsizing, as running jobs in this fashion will slow down execution and is considered wasteful. The policy therefore is that although it should be possible to run jobs that need partitions to be downsized, they will have to wait for the smallest possible partition to become available, while the larger partitions will remain free for use by larger jobs that make more efficient use of them.

### 2.2.2 Interactive Jobs

Unlike normal jobs, for which it is acceptable that the time between submission and actual execution may be quite large depending on whether there are other jobs queued or running, it is essential for interactive jobs that they are started without too much delay once they have been submitted. Therefore, the scheduler deals with the submission and execution of interactive jobs in one go: either the submission succeeds immediately and the job starts running with a short delay, or if this is not possible, submission fails.

**Acceptance of Interactive Jobs**

The procedure for accepting interactive jobs is similar to the the one described for normal jobs in the previous section. The submission of an interactive job may fail because of the following reasons:

- The provided maximum run-time exceeds the user's quotum for interactive jobs.

- No suitable partition according to the same conditions as described for normal jobs in Section 2.2.1 exists or is available at submission time to run the job on; in addition, no such suitable partition can be made free within a short period of time. The freeing of partitions for use by interactive jobs will be described more fully below. Suffice it to say here that if freeing is not possible, this is because all suitable partitions already have an interactive job executing on them.

- The provided maximum run-time exceeds the maximum allowed duration of an interactive job as configured by the administrator.

**Execution of Interactive Jobs**

If a suitable free partition was found on which the interactive job can be executed, it is started immediately on that partition as part of the submission procedure. If one or more suitable partitions do exist but are not available at the time of the submission, the scheduler will free one as quickly as possible, by performing the following steps:

1. The scheduler chooses from the set of suitable partitions one on which no other interactive job is currently running. If several partitions fulfil this criterion, the one on which the job that has been running longest relative to its maximum run-time is selected. For jobs that have been restarted from their checkpoints, counting starts from the time at which they were restarted. This prevents that the same job will suffer every time an interactive job is submitted.

2. The job currently using the partition is checkpointed and removed. This may take a certain amount of time, depending on the job, causing the delay between acceptance and actual startup of the interactive job mentioned earlier.

The checkpoint time of jobs should be kept small enough to keep this delay reasonably small. There are several possibilities:

(a) Jobs with a checkpoint time larger than some configurable amount of time are only allowed to run at times when it is unlikely interactive jobs will be executed: at nights and weekends. Long jobs that cannot be completed within one night or weekend are checkpointed each time when daytime starts and restarted again that same evening, or

(b) Jobs with too large checkpoint times are executed mostly at nights and weekends; once started, they can only be interrupted by interactive jobs at certain times. The user submitting the interactive job is told how long it would take to wait for the checkpoint to be made, or

(c) Remove the running job without checkpointing, keeping a checkpoint made earlier. For jobs with a large checkpoint interval this method potentially causes the loss of a large amount of computation time.

Methods (a) and (b) both seem reasonable, although the first has the advantage that interactive jobs are not hindered by jobs with large checkpoint times at all during daytime. Both methods require the scheduler to be extended with a daytime/nighttime mode. The third option has the advantage of keeping the FCFS order in which normal jobs are executed intact. For now, we will assume the third method to be satisfactory, and choose one of the other alternatives if practice shows that the combination of interactive jobs and jobs with large checkpoint intervals occurs too often.

3. Finally, the interactive job is started on the partition that has now become available.

Once an interactive job is running, it is never preempted by another job, until a certain time limit is exceeded. There are two limits to the execution of interactive jobs: the maximum run-time provided by the user, and an upper limit to the duration of interactive jobs. The latter is imposed on interactive jobs to prevent users from running long, non-interactive jobs under the guise of interactive ones, and can be configured by the administrator (see Section 3.2.3). The scheduler applies the lower of the two limits. If the actual run-time approaches the maximum time to within a configurable amount of time, say 10 minutes, a warning informing the user is printed in the shell from which the user is running the job. Once the maximum time is used up, the job is removed from its partition. No checkpoints will be made during the execution of an interactive job, nor will a final checkpoint be made when the maximum time is reached.

If the execution of an interactive job caused another job to be checkpointed and removed, this job will be automatically restarted by the scheduler from its last checkpoint after the interactive job has finished.

### 2.2.3   Priority Jobs

For priority jobs, it is important that their results will be available at some given time in the future. The latest point in time at which the job must be fully completed is called its *deadline*. The job may be completed earlier than its deadline, but certainly not later. The scheduler supports this principle by allowing priority jobs to start ahead of other jobs, based on deadlines supplied by users.

**Acceptance of Priority Jobs**

At submission time, the scheduler performs the same checks that were described for the acceptance of normal jobs in Section 2.2.1. In addition the scheduler predicts whether or not it will be possible to complete the priority job before the given deadline, using the provided maximum run-time and the information it has about other (priority) jobs. If the prediction turns out to be positive, the job is accepted. By doing so, the scheduler guarantees the job will be started and completed in time, albeit with the following two restrictions:

1. The scheduler must not be suspended or quit before it has had the opportunity to start the priority job. Needless to say the multicomputer needs to remain available as well. If the scheduler is interrupted, it will try to keep as much of the deadlines as possible after it has been restarted, but there cannot be a guarantee any longer.

2. No interactive jobs must be submitted and executed in between. This is a matter of policy: the decision has been made that priority jobs may be preempted by inter-active jobs to give optimal support to interactive jobs. However, this implies that a priority job may potentially miss its deadline if one or more interactive jobs pre-empt it. Fortunately, this will normally be by a relatively short time, as interactive jobs have short run-times.

To summarise, submission of priority jobs may fail because of these reasons:

- The user's quotum for priority jobs is too low. In this case, a warning will be issued and the job will be accepted as a normal job if the user has a normal job quotum that is high enough, otherwise this will fail as well.

- No suitable partition exists according to the same conditions as were described in Section 2.2.1 for normal jobs.

- The deadline cannot be met because of existing deadlines of other jobs, or an inter-active job is running that will not finish soon enough. In this situation, the user will receive a warning that the deadline cannot be fully met and the job will be started as soon as possible, thus still receiving priority over normal jobs.

**Execution of Priority Jobs**

The scheduler tries to treat priority jobs as much as possible in the same way as normal jobs. If it appears a priority job is able to finish in time for its deadline without giving it priority over running jobs or jobs ahead in the queue, it will be queued just like any other normal job. If a priority job must be started earlier because it would otherwise miss its deadline, the scheduler will move the job to the latest position in the queue that still allows it to be started early enough. If the scheduler finds that according to its information, no suitable partition will become available in time, it will wait for as long as possible given the deadline of the priority job, and then free one of the partitions by checkpointing and removing one of the running jobs. After this has been done, the priority job is started, allowing it to finish in time.

If the execution of a priority job caused the checkpointing of another job, the latter will be restarted from its last checkpoint right after the priority job has finished.

## 2.3   Quota Computations

For each user, computation-time quota are specified for each of the different job types (normal, interactive, and priority). For each type, the quotum consists of the maximum amount of time available on a monthly basis to a given user for running jobs of the corresponding type, regardless of the number of processors each job will use. Optionally, an end-date may be set, after which no longer any time will be available to the user.

At the beginning of each month, if the end-date has not passed, the quota for each job type are raised to the monthly limits. This means that users will never have more hours available than the limits set for them, and that they cannot save up hours if they have still some time left at the beginning of the month. If the end-date has passed, all quota are set to zero, disabling the user from submitting any new jobs. Jobs that are already queued will still be executed.

User quota are reduced each time a job is successfully submitted. The amount of time subtracted equals the maximum run-time specified by the user during submission. If afterwards it turns out the job actually needed less computation time than was indicated at submission time, the excess amount of time is returned to the quota. The amount of computation time left reported to the user at any given moment is thus the amount of time available to newly submitted jobs. When returning amounts of time, care is taken that unused time from jobs that were submitted in a previous month is discarded. If the end-date has passed, the unused time will not be returned to the user since he may no longer use the machine. To stimulate users to provide reasonably accurate maximum run-times when submitting jobs and not just any value, the amount of unused time is returned only partially. A value of 75% seems reasonable for this purpose.

For each successfully submitted job, the quotum of the corresponding job type is

reduced, and afterwards the amount of unused time will be returned to this same category. For priority jobs, a slightly different method is used, so that the quotum for such jobs is only used if jobs are actually executed with priority. As usual, the maximum run-time is initially subtracted from the priority quotum. If the job is executed with priority, the unused time will will be returned to the priority quotum as expected. However, if the job has been run without having to revert to priority execution, the entire amount of time (meaning 100%) that was subtracted is returned to the priority quotum, and the quotum for normal jobs is reduced instead as it would have been if a normal job had been executed. In case the normal job quotum is not sufficiently large to do so, the quotum for priority jobs will be charged.

When the scheduler is started, it checks whether any job that was running when it went down is still executing on the machine (see Section 3.2.1). If this seems to be the case, the scheduler assumes the job has been running all the time and will count this time accordingly. However, when the job is no longer present, the scheduler will only count the time the job had been using until the scheduler went down.

The time used for the periodic creation of safety checkpoints (for checkpointing, see Section 2.4) during the execution of jobs and the time needed to make a final checkpoint when the job reaches its maximum run-time are deducted from the user's quota. In contrast, the user will not be charged for the time needed to take checkpoints due to the preemption of a job on behalf of interactive or priority jobs or as part of a system shutdown.

## 2.4 Checkpointing

*Checkpointing* is a technique under which the state of a job is written to disk, thereby creating a so-called *checkpoint*. The scheduler uses checkpointing for two purposes:

- Periodic safety backups. By creating checkpoints periodically, the amount of computation time lost when failures occur on the machine is minimised. Only the time that has passed since the creation of the last checkpoint is lost.

- The ability to preempt and, if needed, restart jobs afterwards when they must give up the machine.

In this section it is discussed how jobs are supposed to deal with being checkpointed and restarted.

### 2.4.1 Checkpointing Jobs

The method of checkpointing that has been chosen is a *semi-automatic technique* [10]. As some action is required by a job that is to be checkpointed, it is shown in this section how this should be handled.

The interface between the scheduler and user jobs uses the standard UNIX signal mechanism. When the scheduler decides a job needs to be checkpointed, it sends this job a checkpoint signal. The job must install an appropriate signal handler for the handling of these checkpoint signals. A job may receive one of two different types of signal:

- A periodic-checkpoint signal. This signal is sent periodically to jobs in order to make safety backups. The interval is specified by the user during submission. Upon reception of this signal, the job should continue execution until all threads reach the end of the current time step in the computation. When this point is reached, the job must create its own checkpoint by storing all data that needs to be preserved on disk. The advantage of creating a checkpoint at the end of a time step is that the amount of data that needs to be saved is at its lowest. After the checkpoint has been taken, the job is allowed to continue execution.

- A preemption-checkpoint signal. This signal is sent to a job when it is going to be preempted shortly. This can happen when it is going to be replaced by another job, when it reaches its maximum run-time or when the machine will be shutdown. Upon reception of this signal, the job has to act exactly like it would after receiving a periodic signal, the only difference being that the job must exit after the checkpoint has been completed. The scheduler will remove the job if it does not quit by itself.

## 2.4.2   Restarting Jobs

A job may be restarted from a checkpoint by the scheduler. Since such restarts are identical to a fresh start of the job, the job must determine whether or not it is being restarted. The job can do this in several ways, one of which may be to examine the existence of a checkpoint file. If it finds it has been restarted, it must first retrieve the data stored in its checkpoint and then continue its computation where it left off.

The scheduler may optionally assist in making it easy to recognise a restart by starting a job with the additional command-line argument -Restart. Upon startup, a job must then examine its list of arguments for the occurrence of this special argument to determine whether or not is is being restarted. By default, this method is used. This behaviour may be configured for each program class, see Section 3.2.3.

# Chapter 3

# User and Administrator Interface

In this chapter the functional aspects of the user and administrator interface to Artaras are described. In the first part, the various commands that make up the user interface are discussed. The second part describes the way in which the administrator can control and configure the scheduler.

## 3.1   User Interface

The user interface through which jobs can be handled with the scheduler consists of a number of shell commands. In this section, they are described in detail.

### 3.1.1   Job Submission

Submission of new jobs to the job queue can be done using the `submit` command. In order for Artaras to be able to execute the job properly, the user has to describe certain characteristics of the job as part of the submission. The following information has to be provided; unless told otherwise, items are required:

- Name and path of program executable or script, including arguments.

- Program class (e.g., SPMD or script).

- Job type: normal, interactive, or priority.

- Partition to use (optional). By default, the scheduler will determine the partition on which the job will be executed. If a partition is specified, the scheduler ensures the job will be eventually executed on that partition only, queueing the job if needed even though other partitions may be available.

- Number of processors needed. Usually, this number will be the size of one of the existing partitions. For jobs requiring sizes different from any of the existing partitions, the scheduler will automatically create a partition of the correct size, see Section 2.2.1. This item is optional if a partition has been specified by the user, in which case the number of processors in that partition will be the default.

- Estimated maximum run-time. This is the maximum time it will take the job to run till completion, measured as wall-clock time.

- Deadline. Only for priority jobs.

- Checkpoint interval and checkpoint duration. The former indicates how much time must pass between two checkpoints, the latter specifies how much time it will take at most to create a checkpoint.

The command supports the following three ways in which the required information can be provided:

- First of all, if the user simply starts the submit command without any arguments, he is prompted for the value of each of the items mentioned above in the shell from which the command was executed.

- The second way is for the user to put the values for certain items in a so called *job description file*, and to supply the name of this file as a command-line argument to the submit command. Such description files are handy for setting those values that do not change when a job is submitted more than once. After the job description file has been processed, items for which no value was found in the file will be prompted for interactively as described earlier.

- Finally, the value of each of the items may be supplied to the submit command as command-line arguments. If an additional job description file is provided, items not present on the command line will be taken from the file, whilst the values of those items provided via the command line override the ones that are found in the job description file. Items not specified on the command line nor present in a supplied job description file will be prompted for.

For each job successfully submitted, a Job ID is returned that can be used to allow additional job control via any of the other commands. A warning will be issued in addition when the user's quota are getting low. If submission fails, a clear description will be given of the cause, and the user will be returned to the shell. For possible causes, see Section 2.2.

When a normal or priority job is accepted, the submit command will return immediately after the submission has been completed. For interactive jobs that have been successfully submitted, the user will not be returned to the shell. Instead, the input of

the user in the shell from which the job was submitted will be passed on to the job, and the output of the job will appear in that same shell. If the scheduler cannot start the interactive job right away, but has to suspend a running job first, the user will be given an indication of how long these preparations will take. The user is then given the option to abort submission if this time is too long to his liking.

### 3.1.2   Job Queue and Multicomputer Status Overview

The `js` command can be used to obtain an overview of the job queue and the current status of the multicomputer. With regard to the status of the multicomputer, the following information is displayed:

- Whether the machine and scheduler are up and running, or currently down. If down, the estimated time when the machine will be available again is given.

- The configuration of the machine. For multicomputers with fixed partitions, a list is reported of the existing partitions and the processors in each partition.

- The current time, to make using the other displayed times more convenient.

The overview of the job queue shows the following information, for each job either running on the machine or waiting in the job queue:

- Job ID. This ID can be used to manipulate the corresponding job with one of the scheduler commands.

- User name. The name of the user that submitted the job.

- Job status (S). Indicates the current status of the job. Possible values are:

    - `Running (R)`: the job is currently executing on the multicomputer.
    - `Waiting (W)`: the job is waiting in the job queue.
    - `Quitting (Q)`: the job is checkpointing and exiting.

- Job type (T). The type to which the job belongs. Possible values are

    - `Normal (N)`.
    - `Priority (P)`.
    - `Interactive (I)`.

- Job location (L). The partition on which the job is running, indicated by the single letter that is its name. If a job is waiting for a specific partition to become available, the name of that ring is shown. Otherwise, a dash (-) is displayed.

- Starting time. For running jobs, this is the time the job was started. For waiting jobs, this is an estimation of their starting time.

- Completion time. Both for running and waiting jobs, the estimated time of completion.

- Application name and arguments. The name of the executable started by the scheduler, followed by any arguments given[1].

In Figure 3.1, an example is given of an overview generated by the `js` command.

```
Current time: Tue Jul 2 14:12.
Scheduler is up and running.
Configuration:
  Partition A (#8): 0-1-2-3-4-5-6-7
  Partition B (#8): 8-9-10-11-12-13-14-15

JID User     S T L  Starting time     Completion time    Job

101 Ape      R P A      -      09:35  Thu Jul  4 15:20   MolSim
102 Note     R N B      -      13:09  Wed Jul  3 10:09   3pMD
103 Meece    W N -  Wed Jul  3 10:09  Wed Jul  3 12:50   Dynamol
```

Figure 3.1: Example output of `js`.

Using an option, a longer overview can be obtained instead that shows for each job in addition to the above, the following information:

- Program class (e.g., SPMD or script).

- Number of processors needed.

- The time at which the job was submitted.

- The deadline, if specified.

- Checkpoint interval and duration.

- The time at which the last checkpoint was taken.

---

[1]The length of this field is adjusted to fit the space remaining on each line of the user's shell.

### 3.1.3 Removing Jobs

Using the command `kill`, jobs can be removed from the job queue, if waiting, or from the multicomputer itself, if running. Ordinary users can only remove their own jobs. The following actions may be performed:

- Removal of a job by specifying its Job ID. Multiple jobs may be removed in one go by specifying multiple Job IDs.

- Removal of all queued and running jobs belonging to the user executing the command, via an option.

The administrator is capable of removing any job of any user. When the administrator uses the command however, it behaves slightly differently for the option that removes all queued and running jobs. It is used to remove all jobs of a specified user instead. To prevent accidental removal of all jobs of all users, the user name is required for the command to work. A special option is available to enable the removal for all users.

### 3.1.4 Moving Queued Jobs

The command `move` allows users to move jobs to a different position within the job queue. This is done by specifying the ID of the job that is to be moved, followed by the intended destination, which consists of the ID of the job after which the former should be placed. An ID of `0` can be used to indicate the job should be placed at the foremost position of the queue.

Ordinary users can only move jobs that belong to themselves; in addition, they can only move jobs to a position lower in the queue, that is, a position resulting in an estimated starting time which is later than the currently predicted one. These restrictions prevent a user from shuffling jobs around in order to gain himself an advantage. The command is intended to allow users to voluntarily give up their position in the queue and have someone else's job start earlier.

The administrator is allowed to move jobs around without any restrictions. This feature is desirable to enable the administrator to have additional control of the job order in special or unforeseen situations. If the change of job order causes jobs to miss their deadlines, a warning is given.

### 3.1.5 Computation-time Quota Overview

Users can obtain an overview of the quota they have left via the `quota` command. The command shows the following information:

- The amount of computation time obtained every month for each of the job types (normal, interactive and priority), in hours and minutes (hh:mm).

- The amount of computation time left this month, for each of the job types, again in hours and minutes (hh:mm).

- The date after which the monthly quota will no longer be available, thereby disabling the user from using the machine. For example, an end-date of `Apr 2005` means that the quota will be available up to and including the whole month of April 2005.

Ordinary users can only get an overview of their own quota. The administrator may look at any user's current quota by specifying the user's name as an argument to `quota`.

Figure 3.2 is an example output of this command:

```
                Normal    Priority    Interactive   End-date
Limit/month     200:00     20:00          0:00      Apr 2005
Left             50:15      5:02          0:00
```

Figure 3.2: Example output of `quota`.

## 3.2 Administrator Scheduler Control

In this section, it is described what actions are taken by the scheduler at startup and shutdown, and what control the administrator has over these events. It is also shown how the administrator can configure certain aspects of the scheduler.

### 3.2.1 Scheduler Startup

When the scheduler is started, it first reads the existing configuration files and adjusts its behaviour accordingly. The various aspects that may be configured are discussed in Section 3.2.3.

After its initial configuration, the scheduler examines the multicomputer to see whether there are processes present that are left over from a previous session of the scheduler. If so, they will be allowed to run as if the scheduler had been present all the time. If jobs have exceeded their maximum run-time in the absence of the scheduler, they will be immediately checkpointed and removed. Any other processes found by the scheduler will be removed.

Next, if jobs are found that were stopped and checkpointed due to a scheduler shutdown, they will be restarted from their checkpoint.

Finally, the scheduler examines the job queue and starts scheduling the jobs that are present as described earlier in Section 2.2.

### 3.2.2 Scheduler Shutdown

The scheduler can be shut down in an orderly fashion via the `shutdown` command. Optionally, a begin and expected end-time of the shutdown may be specified. If a begin-time is present, the shutdown will wait until then. If omitted, the shutdown will proceed right away. The expected end-time is used to inform users: it allows the adaptation of the estimated times given in overviews of the queue via the `js` command, see Section 3.1.2.

On a normal shutdown, the scheduler takes following steps:

1. Running jobs are checkpointed by sending them a checkpoint signal (see Section 2.4).

2. All processes left on the multicomputer are removed.

3. The scheduler exits.

Using a command-line option, the scheduler can be made to simply exit and stop controlling the machine without disturbing the running jobs. In effect, only step 3 will be performed. A second option can be used to perform a more urgent shutdown in which no checkpoints are made but only steps 2 and 3 are performed.

If the scheduler is quit or killed without using `shutdown`, running jobs will be checkpointed nor killed, but simply left undisturbed.

### 3.2.3 Configuration

The administrator can configure the behaviour of the scheduler in several ways.

**Computation-time Quota**

As described in Section 2.3, for each user there is a monthly limit on the amount of computation time he can use, until a specified end-date is reached after which no time will be available any longer.

The configuration can be made by editing a file which is read by the scheduler at startup time. The file consists of the user names that are allowed to run jobs on the multicomputer, followed by the three different quota for each of the job types and the end-date after which the user will not receive any more quota. Users that have no entry in this configuration file will be treated as if their quota would all have been set to zero.

An example file is given in Figure 3.3.

**Settings and Options**

The administrator can set several options that modify the behaviour of the scheduler. These are specified in a file, one option per line. The following settings can be made:

```
# User     Quota/month,     Quota/month,     Quota/month,     End-date
#          normal           interactive      priority

Ape        200:00           20:00                 0:00        Apr 2005
Note       200:00           20:00                50:00        Jul 2005
Meece      100:00           10:00                 0:00        Jan 2006
```

Figure 3.3: Example quota configuration file.

- Interactive limit. Allows maximum duration of interactive jobs to be specified (in hh:mm format). Setting it to 0:00 hours will prevent all interactive jobs from being accepted. Similar limits may be specified for various other items as well, such as the length of checkpoint intervals and checkpoint durations.

- Interactive Warn limit. Allows the time to be specified (also in hh:mm format) that users running interactive jobs are warned in advance with when their jobs approach their maximum run-time or the interactive limit, whichever comes first.

- Log Entries. This setting can be used to specify the kind of events the scheduler will log (see Section 3.2.4. Separate keywords are available for each of the possible events. Specifying the keyword corresponding to a certain type of event will enable logging of such events.

An example settings file is shown in Figure 3.4.

```
# Scheduler settings file

Interactive limit = 0:45
Log entries       = startup, shutdown, jobsubmit, jobfinish
```

Figure 3.4: Example scheduler settings file.

**Program Classes**

Settings can be specified for program classes which modify the way the scheduler (re)starts, quits and checkpoints a job of that class. An example configuration script is shown in Figure 3.5.

Checkpoint periodic signal allows the specification of the signal number send periodically to start a safety checkpoint. Checkpoint preempt signal is the signal

```
# Class    Checkpoint     Checkpoint     Restart
#          periodic sig   preempt sig    action

SPMD            30             31          arg
script          30             31          arg
```

Figure 3.5: Example job class configuration file.

number used when a job is requested to make its final checkpoint prior to quitting. Finally, Restart action specifies whether the scheduler will supply the special argument -Restart when restarting jobs from their checkpoint.

### 3.2.4   Logging

The scheduler keeps a log file in which an entry is made for all actions taken. The following actions and events may be logged:

- Scheduler startup. The entry contains the startup time and the Job IDs of any jobs found on the machine.

- Scheduler shutdown. Two entries will be made: the first entry is written when the shutdown is started. The entry contains the shutdown start time and, if given, the estimated time of restart. The second entry is made when the shutdown has been completed. It shows the shutdown finishing time and the Job IDs of jobs that are left running on the multicomputer, if any. Between both entries, entries corresponding to the checkpointing of jobs may be found.

- Submission of a job. This entry contains the following information:

  – Job ID.
  – Time of submission.
  – User name.
  – Name and path of program executable or script, including arguments.
  – Program class (e.g. SPMD, script).
  – Whether this is a normal, interactive or priority job.
  – Partition to use, if provided
  – Number of processing nodes needed, if provided.

- – Estimated maximum run-time.
- – Deadline, for priority jobs.
- – Checkpoint interval and duration.
- – Starting time, as estimated at submission.
- – Completion time, as estimated at submission.

- Job startup and restart. The entry contains the Job ID, the starting time, and the updated estimated completion time.

- Checkpoint creation. This entry is made each time a job is checkpointed. Besides the time the checkpoint was started, it also displays whether this checkpoint is a periodic or a final checkpoint.

- Job finishing or killing. Besides the Job ID, this entry contains the actual starting time, the time when the job exited, the estimated maximum run-time and the actual run-time.

- Removal of waiting jobs from the queue. The entry shows the Job ID and the time of the removal.

- Moving of jobs in the queue. This entry shows the ID of the moved job and the ID of the job it has been moved after, and the time when this happened.

- Changes in quota. These entries are created when quota are reduced due to the submission of jobs and when quota are returned after jobs have exited.

- Errors and warnings generated by the scheduler, for instance when at startup it appears a configuration file has a bad format.

## 3.3   Manual Pages

Several manual pages are available, written in the hypertext language HTML:

- User manual pages. For each of the scheduler commands described in Section 3.1, a description of its use is given.

- Administrator manual pages. Manual pages are available about starting and stopping the scheduler, as described in Sections 3.2.1 and 3.2.2. In addition, separate manual pages describe the syntax and semantics of the various configuration files discussed in Section 3.2.3.

# Chapter 4

# Technical Design

This chapter discusses the technical design of the Artaras scheduler. First, an overview of the structure of the scheduler is given. Then, the scheduler daemon and the shell commands are each looked at in more detail. Finally, the handling of job schedules is discussed.

## 4.1 Overview of Scheduler Structure

This section provides an overview of the technical structure of the job scheduler. First we will see how a distributed approach was used in the scheduler's design. After that, it will be discussed in what way adaptability to different multicomputers guided the design.

### 4.1.1 Distributed Approach

The job scheduler has been designed as a distributed application that operates in a local network environment. As such, the scheduler application has been divided into a number of executables that each take care of a separate part of the scheduler's functionality. These executables can be run independently from each other on any machine in the network and used simultaneously by different users. The data used throughout the scheduler application is shared by its executables through a set of files.

The way in which the scheduler gets its work done is by having the executables perform their designated actions on the contents of one or more of the data files, signal the so-called *scheduler daemon* if needed, and then exit.

An example is given in Figure 4.1. The four scheduler processes in this figure are the result of running the corresponding executables on machines in the network; also shown are the network through which the processes can communicate with each other, and two data files on disk storage that can be accessed by the processes through this network. For

reasons of simplicity, the example shows only the activities of the two processes submit
and scheduler daemon.
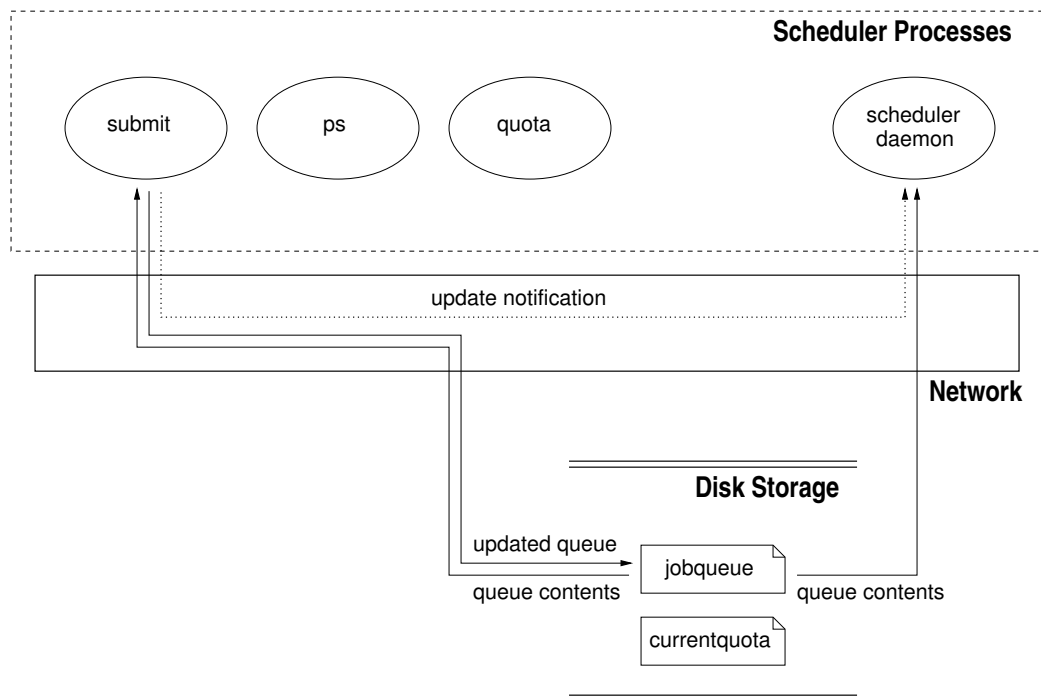


Figure 4.1: Distributed approach used in scheduler.

In the example we see how the submit command goes about adding a job to the
job queue which is stored in the file jobqueue. The command starts by reading the
current contents of the job queue into memory, inserts a new entry corresponding to the
job that is to be submitted, and then calculates a new job schedule. After that, it writes the
rescheduled job queue back into its file and notifies the scheduler daemon that changes
have been made to the job queue. After receiving the notification, the daemon first re-
examines the job queue by reading the updated schedule, then decides whether further
scheduling activities are to be taken.

The distributed approach enables users to work with the scheduler directly from any
terminal or workstation in the network without having to login to a specific machine, as
was promised in Section 2.1.1 of the functional design. More importantly, the approach
prevents from becoming dependent on a single machine to take care of all scheduling
operations, as would be the case in a more traditional centralised scheduler. Specifically,
the design maximises the ability of the scheduler to make the best of situations in which
the multicomputer and/or host running the scheduler daemon are temporarily unavailable
and allows Artaras to provide as much of its functionality as possible in such situations,

as was specified in Section 2.1.5.

The scheduler's executables and data files will be covered in more detail in Section 4.2. The communication that occurs between scheduler processes is also further discussed there.

### 4.1.2 Adaptability and Extensibility

Artaras has been designed with the possibility of running the scheduler on different multicomputers in mind. It accomplishes this by defining two interfaces through which machine-dependent operations are performed:

- Machine interface. This interface deals with managing a multicomputer, such as launching a thread on a processing node and monitoring its execution, actions needed to create partitions, etc.

- Scheduling interface. This interface is used to access the scheduling algorithm. Not all multicomputer architectures support the same scheduling strategies, especially in the area of partitioning (for a taxonomy see [3]).

Support for a particular multicomputer can be added to Artaras by providing an appropriate implementation of each interface. Of the two interfaces, the machine interface requires the more specific implementation, whereas the implementation of a given scheduling algorithm can be successfully used to schedule across a variety of multicomputers that support the same scheduling strategy.

Both interfaces will be discussed in more detail in Chapter 5. This chapter also describes the virtual multicomputer Virtua for which both the machine interface and a scheduling algorithm have been implemented.

## 4.2 Scheduler Components

In this section, we will look in more detail at the two categories of components of which the scheduler is composed: executables and data files. After that, the communication between components of the distributed application will be discussed.

### 4.2.1 Executables

As was shown in Section 4.1.1 earlier, Artaras consists of a number of separate executables that together make up the application.

The first five executables are shell commands with which users are able to perform the associated actions:

- submit. Allows new jobs to be submitted to the scheduler.

- js. Gives an overview of the current status of the multicomputer and the job queue.

- kill. Removes a running job or a job waiting in the queue.

- move. Moves a job to a different position within the queue.

- quota. Gives an overview of a user's current computation-time quota.

The last two executables that are part of Artaras are used by the administrators:

- schedd. The *scheduler daemon*: controls the execution of jobs and monitors the multicomputer. The daemon is started by launching schedd on the multicomputer and leaving it running.

- shutdown. A shell command that causes the scheduler daemon to exit gracefully.

The scheduler daemon will be discussed in detail in Section 4.3, while the shell commands (for both users and administrators) are examined more closely in Section 4.4.

### 4.2.2   Data Files

The data on which the scheduler operates, such as the job queue and computation-time quota, are stored in a number of files that are used by the different parts of the scheduler. The files are made accessible to the multicomputer and the workstations in the network via the Network File System (NFS).

The scheduler makes use of the following files:

- jobqueue. This file contains the job queue, maintained by the scheduler.

- currentquota. Contains the users' current computation time quota.

- machine.config. Contains information about the current configuration of the multicomputer. This file is created by the administrator.

- jobclasses.config. Configuration file containing settings for job classes.

- quota.config. Configuration file that contains monthly limits to users' computation times.

- settings.config. This configuration file contains various settings and options that modify the behaviour of the scheduler.

- log. The scheduler logfile, containing entries for various actions by Artaras and events that take place.

- users' job description files. These files can be used to describe the characteristics of jobs.

Since each file may be accessed by parts of the scheduler that run independently of each other, file locking must be used to prevent corruption of the file. A process that wants to modify one of the files first obtains an *exclusive lock* on it, protecting the file from accesses by other processes. Then, it reads the contents into memory, and modifies the data. After the changes have been completed, the process first writes the data into a new file, and if no errors occurred, then replaces the original file with the newly created file. Finally, it releases the lock. Processes that only need to read a file obtain a *shared lock* prior to reading. This lock does allow other processes to read the file at the same time, but does not allow processes to write to it.

Table 4.2.2 gives an overview of the accesses to each of the data files that are required by the different executables. The formats of the files will be specified in Appendix C.

| | Data Files | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Executable** | job queue | current quota | demos. config | jobclasses. config | quota. config | settings. config | log | job descr. |
| schedd | r/w | r/w | read | - | read | read | write | - |
| submit | r/w | r/w | read | read | - | read | write | read |
| js | read | - | read | - | - | - | - | - |
| kill | r/w | r/w | - | - | - | read | write | - |
| move | r/w | - | - | - | - | read | write | - |
| quota | - | read | - | - | read | - | - | - |
| shutdown | - | - | - | - | - | read | write | - |

Table 4.1: Overview of Data Files Accesses.

Locking files, if done carelessly, might cause *deadlocks* when two processes, each already having obtained a lock on a file, are waiting for the other to release its lock. The following two precautions are taken to prevent deadlocks:

- Wherever possible, a process has only one file locked at a time. This way, no deadlock can possibly occur, since it is guaranteed a lock will ultimately be released.

  This approach is used by the various parts of the scheduler when accessing the files machine.config, jobclasses.config, quota.config and settings.config for reading or the log file for writing. It is also used when access is needed to only one of the files jobqueue or currentquota.

- Multiple locks on multiple files can only be obtained in one specific order, which is the same for all parts of the scheduler. Because of this restriction, two processes can

never wait for each other's lock, thus preventing deadlocks. This method is known as *hierarchical allocation* [5].

Throughout the scheduler, this method is used to access the job queue and current quota files. Whenever both files are needed for update, the file `jobqueue` is always locked first, followed by `currentquota`.

### 4.2.3   Communication

The scheduler is designed in such a way that its different parts can operate mostly independently of each other. Nevertheless, in some cases coordination is needed between two scheduler parts. Such coordination always involves the scheduler daemon and one of the shell commands. We can discern the following two types of coordination:

1. A shared file has been updated by one of the shell commands. A typical example is that of a new job having been added to the job queue. Since the scheduler daemon keeps a copy of the queue in memory to work with, it must be made aware of the changes made to it, and re-read it from the job-queue file.

2. Synchronisation is required between the scheduler daemon and one of the shell commands. This occurs in the following situations:

   - an interactive job is submitted.
   - a job running on the multicomputer is removed.
   - the scheduler daemon is asked to perform a shutdown.

The required coordination is achieved through communication. In the first situation, it is sufficient for the shell command to send the scheduler daemon a message after is has completed its update of the shared file; the command does not need a reply and no further coordination is necessary. This kind of communication is called *asynchronous* since it does not cause a sender (in this case a shell command) to synchronise with the receiver (here: the scheduler daemon).

Asynchronous communication is performed within the scheduler by using the *udp* protocol via a BSD socket. Under this protocol, messages called *datagrams* are sent to their destination without any confirmation of their arrival. A shell command that wishes to notify the scheduler daemon of a change in one of the data files simply sends the latter a datagram. The advantage of using datagrams is, that the shell command may continue immediately after sending, without waiting for the message to be received. In case the scheduler daemon is not running, not having to wait for a time-out is a great advantage.

The second situation does require the daemon and shell command to synchronise with each other. In order to do so, *synchronous* communication is used, which causes the sender to wait for reception by the intended receiver.

Within the scheduler, the *tcp* protocol is used to obtain such behaviour. Under this protocol, sender and receiver connect to each other first, thereby synchronising. After a connection has been made, data can be transported reliably in both directions via a channel. This feature is used to allow the scheduler daemon and shell command to perform their required actions in turn. The scheduler uses a separate socket for use by the tcp protocol.

In both cases, the need to communicate is initiated by one of the shell commands, while the scheduler daemon waits passively for the former to attempt to communicate. This model of communication is usually called a *client-server* model. In our case, the scheduler daemon waits passively for either the reception of a datagram on its first socket or an attempt to connect via the tcp protocol on the other socket. In the latter case, it accepts the connection and receives the message. After the reception of a message on either socket, the scheduler daemon takes the appropriate action. This is discussed in Section 4.3.

## 4.3   Scheduler Daemon

In this section, the scheduler daemon that is the part of Artaras that controls and monitors jobs running on the multicomputer is discussed in more detail.

At startup, the scheduler daemon accesses the files `settings.config` and `quota. config`. The next step is for the daemon to determine the current configuration of the multicomputer. Configuration settings are read from the file `machine.config`, which is created by the administrator. As the final part of starting up, the daemon will read the job queue from the file `jobqueue`.

The daemon now enters its main loop. Within this loop, it performs a number of actions. First, it monitors the multicomputer. If a partition becomes available because a job exits, it takes the next suitable job from the queue and starts the job on the partition. If a job must be suspended because it exceeds its maximum run-time, it is checkpointed and removed by the daemon. Every time the job queue changes due to an event on the multicomputer or a decision by the scheduler daemon, the job queue file is updated.

Secondly, Artaras's daemon waits for changes in the job queue by one of the shell commands, for example the addition of a new job entry by `submit` or the removal of a job by `kill`. The scheduler will notice such changes in the job queue in two ways:

1. The scheduler is sent a message by a shell command after the latter has changed the job queue (see Section 4.2.3).

2. The scheduler uses polling with a configurable regular interval, e.g., 1 minute, to determine whether the queue has been changed by examining the modification date of the file `jobqueue`. This is a backup measure, because as described in Section 4.2.3,

the delivery of messages sent to the scheduler is not guaranteed under the used udp protocol.

The daemon uses the real-time timer associated with its process to receive a signal regularly, upon which the polling is initiated.

Each time the scheduler notices a change in the job queue, it will re-read the job-queue file and examine the change.

All actions performed by the daemon and other notable events happening on the multicomputer (such as the completion of a job) are logged in the scheduler log file by adding an entry for each event.

## 4.4   Shell Commands

This section describes the shell commands of the job scheduler. Each of the commands will be discussed in detail in the following sections. Finally, the last section discusses how a potential problem is circumvented involving the locking of files that contain shared data.

### 4.4.1   The Artaras `submit` Command

Upon startup, the `submit` command first reads the files `machine.config`, `settings.config` and `jobclasses.config`.

Next, the submit command will obtain information from the user about the job that is to be submitted. After it has completed this input, the job queue and quota are read, after which `submit` will check whether submission is possible. If so, it adds the new job to the queue and reduces the quota with the maximum run-time provided by the user. After that, the job queue and quota are written back to disk, and the scheduler is notified of the change using a datagram.

The submission of interactive jobs requires some coordination between the scheduler daemon and the submit process, so a two-way stream is set up between the daemon and the submit process. Using this connection, `submit` signals the scheduler to examine the job queue and to find a suitable partition, freeing one if needed. As soon as a partition becomes available, the daemon will respond to the submit process, upon which the job will be started on the multicomputer and its input/output will be handled on the workstation from which it it was submitted.

Finally, the submit command adds an entry for the submission of a new job to the scheduler log file `log`.

### 4.4.2 The Artaras `js` Command

The `js` command requires only read access to two files: `machine.config` and `jobqueue`. After these files have been read successfully, the command displays this information in a nicely readable way.

### 4.4.3 The Artaras `kill` Command

The kill command starts by reading the job queue and current quota files. If the job to be killed is queued, its entry is simply removed. If it is running, the job's status is marked for removal. After that, the job queue and current quota files are updated on disk and the scheduler daemon is signalled of the change, using a datagram. After examining the queue, the daemon will shut down any marked jobs running on the multicomputer.

### 4.4.4 The Artaras `move` Command

The `move` command reads the job queue and verifies whether or not the user may move the job to the new position. If this is allowed, the job queue is updated and written back to disk. After that, the daemon is signalled using a datagram.

### 4.4.5 The Artaras `quota` Command

This command simply reads the files `currentquota` and `quota.config` and displays the entries for the desired user in a nicely formatted way.

### 4.4.6 The Artaras `shutdown` Command

This command sends the scheduler daemon a message using a two-way stream, which informs the scheduler a shutdown is desired. Options such as whether or not to checkpoint jobs as part of closing down are sent along.

### 4.4.7 Preventing Suspension of Commands Holding Locks on Files

Since the shell commands use locking to access the data files, care must be taken that the user cannot suspend these commands while having locks on files. This would cause all other processes accessing the locked files to wait until execution of the suspended command is resumed.

A process can be suspended by sending it one of the following signals:

- `SIGTSTP`. Caused by a user pressing control-Z.

- SIGTTIN. Sent to a process when it tries to read from standard input while running in the background.

- SIGTTOU. Sent to a process when it tries to perform output to the shell while running in the background.

- SIGSTOP. The cause and purpose of this signal are not documented.

The problem can be solved by installing signal handlers for each of the above signals, so that the shell commands cannot be suspended while having locks.

Unfortunately, the last signal in the list (SIGSTOP) cannot be caught this way, as no handler can be attached to it. Although this leaves a potential way in which the scheduler can be disrupted, it is unlikely this signal will be sent to one of the scheduler processes. For now, we will consider catching the first three signals to be an adequate solution.

## 4.5   Handling Job Schedules

In this section, we discuss the job queue that is used by the scheduler to keep track of the submitted jobs. First we will look at the structure of the job queue. Then we discuss how the job queue is maintained and how the scheduler daemon selects jobs to run from the queue.

### 4.5.1   Structure of the Job Queue

Since the job queue must be accessible by commands such as used for the submission or removal of jobs even if the scheduler daemon is not running, a copy of the job queue is maintained on disk. When a command or the scheduler daemon needs to examine the queue or make changes to it, the queue is read from disk into memory, any changes needed are made, and it is written back to disk again.

In memory, the job queue is kept as a list of job entries, together with some properties describing the queue, such as the highest job id given to any job thus far. A list structure was chosen for maintaining the jobs while it makes it easy to insert and remove entries at any position, operations which will typically be used often while modifying or reordering job schedules. A second advantage of lists is that it does not place an upper bound on the amount of jobs that can be contained within the job queue[1].

Instead of using a common doubly-linked list structure (figure 4.2(a)), a slightly more complex structure that we will call a *multilist* is used (figure 4.2(b)). In a common list, a single order is imposed on the nodes, which may be traversed from head to tail or vice-versa. Figure 4.2(a) shows the nodes in the order of 1, 2, 3. The multilist allows

---

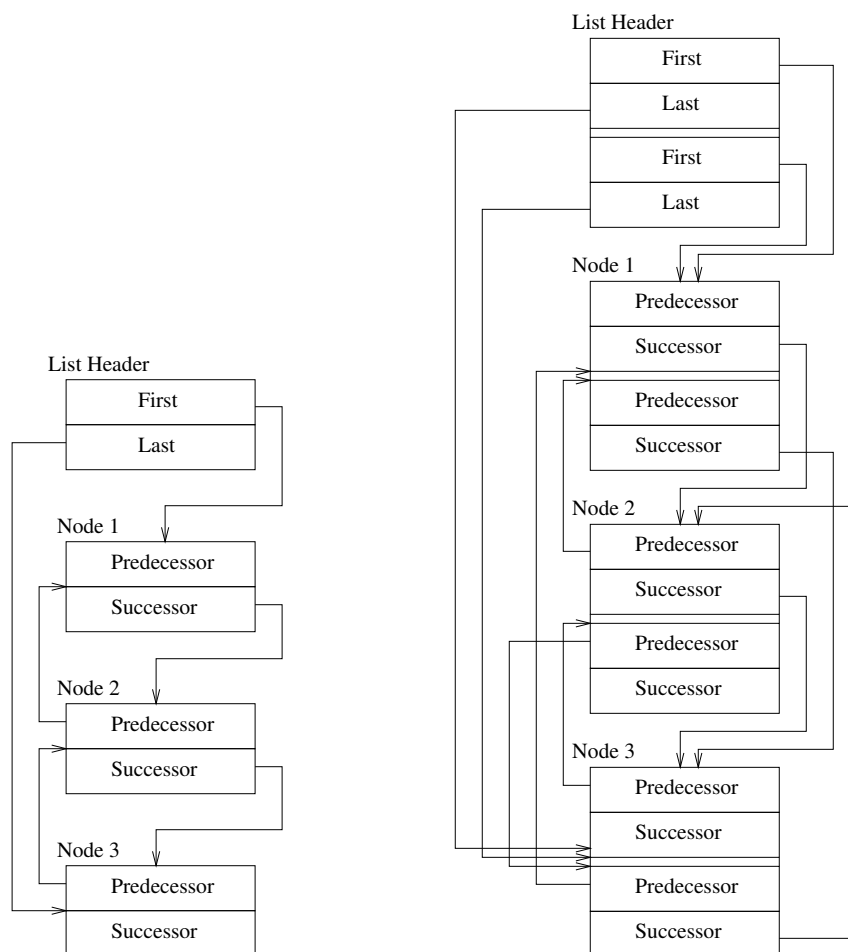[1]The amount of available memory remains of course a limitation.

Figure 4.2: List (a) and multilist (b).

multiple orderings to be kept for the nodes simultaneously by adding an extra pair of links to predecessor and successor nodes for each ordering. The multilist in figure 4.2(b) shows the nodes both in the orders 1, 2, 3 and 1, 3, 2. Since the scheduling algorithm needs to examine the jobs ordered according to several different criteria, such as first-in first-out (FIFO) and earliest deadline first, storing them in a multilist allows efficient access to the job queue. The alternative would be to repeatedly sort the list of jobs to obtain the desired ordering, or keep separate arrays for the extra orderings.

The jobs in the job queue are kept ordered according to the following three criteria:

- Scheduled order. This is the order in which the jobs are scheduled to run, in increasing (estimated) start times.

- FIFO order. The order in which the jobs would be executed were it not for dead-

lines. It corresponds roughly to the order in which jobs are submitted, except for interactive jobs and jobs whose position in the queue has been explicitly changed by a user or administrator.

- Deadline order. This order exists only for priority jobs. They are sorted on the latest possible start time that still allows them to complete before their deadline.

### 4.5.2   Maintaining The Job Queue

The scheduler keeps the three orders contained in the job queue up-to-date by reapplying the scheduling algorithm whenever a change to the job queue is needed, such as the addition of a new job or the removal of a queued job. The job entries in the queue contain the predicted start and completion times as determined by the scheduling algorithm, and the location on the multicomputer where the job will be executed (i.e. which nodes).

### 4.5.3   Executing Jobs from the Queue

The scheduler daemon runs jobs in the order indicated by the job queue's scheduled order as calculated by the scheduling algorithm. In normal circumstances where the multicomputer is available to execute jobs, the first job entries in the scheduled order correspond to jobs that are currently running.

A job is ready to be launched when the processor nodes become available for which it is first in line as determined by the scheduling algorithm. The scheduler daemon then launches each of the job's threads on the specified nodes and starts monitoring the execution of the job. When the job finishes or is removed, the corresponding job entry is removed from the queue. As after any change to the job queue, the scheduling algorithm is applied to update the order of the jobs in the queue, after which the above process is repeated.

# Chapter 5

# Machine and Scheduling Interfaces

In this chapter we will look at the two interfaces that were defined to allow Artaras to be adaptable to different multicomputers. First we look at the machine interface, followed by a discussion of how this interface was implemented for the Virtua multicomputer. After that, the scheduling interface is described, followed by the fixed partitioning scheduling algorithm that is used to calculate job schedules on the Virtua.

## 5.1  Machine Interface

In order to allow Artaras to be adaptable for use on different multicomputers, machine-specific operations and data structures have been grouped together and singled out from the other parts of the scheduler. This machine-dependent part must be implemented for each machine that is to be supported.

To prevent the need for changes to the machine-independent parts of the scheduler when adapting the scheduler for use with another multicomputer, the interface that allows the machine-specific part to be accessed by the rest of the scheduler is the same for all supported machines. The interface defines a number of operations and opaque handles to data structures, which will be discussed in more detail in the following sections.

### 5.1.1  Machine Configuration

The first and most obvious difference between machines is the way in which their behaviour may be configured. Examples include the amount of available processors and partitions, the (maximum) size of partitions, special actions to be undertaken before or after running jobs, etc.

The interface offers a function that determines the current configuration of a given multicomputer, either by reading settings from a configuration file or determining them by examining the system, or a combination of both. This configuration information can then

be used by the other machine-specific operations. In addition, some of this information is relevant to and used by the scheduling algorithm. For instance, for fixed partitioning, the amount of partitions and their sizes are required (See also Section 5.4).

## 5.1.2   Job Control and Monitoring

Another area in which machines may differ from each other, is the way in which jobs are launched, and after that, controlled and monitored.

Therefore, the interface supports the following actions on jobs:

- Start the execution of a job on the multicomputer.

- Quit a job already running on the multicomputer.

- Cause a running job to checkpoint and/or stop itself.

- Check the status of a running job: whether it has already completed or is still running.

## 5.1.3   Locations

As discussed in Section 5.4, the scheduling algorithm assigns to each job a location where it will run on the multicomputer. Such a location indicator will differ between machines. For instance, for multicomputers that use fixed partitioning, a partition number or name is a natural choice. But for machines that use more flexible partitioning methods, it might be better to store a list or range of processing-node numbers.

The machine-dependent part offers functions to read, write and display locations through opaque handles, allowing the machine-independent parts to handle such location indicators without becoming dependent upon the actual contents of the location data.

## 5.1.4   Run-time Job Information

In order to perform the job control and monitoring mentioned earlier, certain run-time information needs to be kept, such as the location of every thread of the parallel job on the multicomputer. Since this information is only needed by the machine-dependent part of the scheduler, it is retained by it internally.

As the scheduler may be stopped and restarted while jobs are still running on the multicomputer, the job execution information must survive from one session of the scheduler to the next. This is done by storing, for each running job, the run-time information in the job queue in combination with the other attributes of the job.

Much like the location information discussed in the previous section, run-time job information can be handled by the machine-independent part of the scheduler by calling upon functions for reading and writing run-time job information, provided by the machine-dependent part of the scheduler.

## 5.2 The Virtua Machine

In order to facilitate testing of the scheduler, a virtual machine has been called into existence, offering all of the functionality described above. In this section, this machine, dubbed the Virtua, will be described.

The Virtua simulates a parallel machine that supports fixed partitioning, on a single host running under the Linux operating system. The number of fixed partitions and their sizes are configurable through the configuration file `machine.config`, which is read when trying to obtain the machine configuration.

The actual architecture of the host is not relevant to the Virtua, since the simulation only deals with it on the process level. When starting a new job on a partition, as many processes are created using `vfork()` and `execl()` as there are computation nodes used by the job, and their process IDs are recorded as belonging to that job. When a job is killed or checkpointed, the appropriate signal is sent to each process whose ID has been previously recorded, using the `kill()` system call.

A job's status is monitored by using the recorded process IDs and the `waitpid()` system call. The job has not finished as long as any of the launched processes still exists and no process has exited with an error status.

A location on the Virtua consists simply of a number that indicates an entire fixed partition. Partitions are numbered from zero to the amount of partitions minus one.

Finally, the job execution information that is read from or written to the job queue consists, for any given job, of the list of the process IDs recorded when the job was launched on the Virtua.

## 5.3 Scheduling Interface

The scheduling algorithm is responsible for determining the various orders in which jobs are kept in the job queue as described in Section 4.5. Such an algorithm is dependent on the possibilities with regard to the ways in which the processors can be shared that are offered by the machine on which the scheduling is to be performed. The more flexible the multicomputer, the more decisions and trade-offs must be made by the algorithm.

In order to allow Artaras to be extended with different scheduling algorithms, the scheduling algorithm is separated from the rest of the scheduler and accessed through a

number of functions. These functions correspond each to a particular change that needs to be made to the job queue:

- Adding a new job to the queue.

- Removing a job from the queue.

- Moving a job from one position in the queue to another.

In combination with the machine interface described in Section 5.1, it is possible to use the same scheduling algorithm for an entire class of machines that support the same forms of time-slicing and/or partitioning. For instance, the fixed partitioning algorithm described in the next section is used to schedule on the virtual multicomputer Virtua, but could also be used to calculate schedules on any other fixed-partitioning machine for which the machine interface is implemented.

## 5.4   Fixed Partitioning Algorithm

The scheduling algorithm discussed here deals with machines that support only fixed partitioning, and do not use time-slicing.[1] The model that can be used to handle such machines is quite simple. It describes the machine using the following two parameters:

- The number of partitions the machine has.

- For each individual partition, the number of processing nodes within.

How the nodes within each partition are named or numbered, is not important to the scheduling algorithm. Such information is only needed when actually controlling and monitoring jobs on the machine as discussed in Section 5.1.2.

When a new job is added to the queue, or an existing one removed or moved to another position, the algorithm first examines which type of job it is: normal, priority or interactive. Then, for each type, the appropriate actions are taken, ultimately resulting in a new job schedule. This process will be discussed separately for each of the types in the following sections.

### 5.4.1   Handling Normal Jobs

In this section we will see how normal jobs are handled when they are added to, removed from or moved within the job queue.

---

[1]Or only very coarse-grained, using checkpointing.

- Addition of normal jobs. This is one of the more simple operations that may be performed on the job queue. They are added at the end of the scheduled and FIFO job lists, and the estimated start and completion times are updated. The start time equals the estimated completion time of a job's predecessor, while its completion time is equal to its start time plus its maximum run-time.

- Removal of normal jobs. When a normal job is removed from the queue, it is first removed from the FIFO list. However, it does not suffice to simply remove it from the scheduled list and leave it at that, because the earlier presence of the job may have caused other jobs with deadlines to be moved upward in the schedule in order to satisfy their deadlines. Since the former job will now be removed, these jobs must now return to their previous position in the schedule.

  The new schedule is obtained by first taking the FIFO job order as the initial order. Starting with the last job, and working upward toward the head of the list, each priority job is checked in turn against its deadline. Its deadline is satisfied only if the estimated start time plus the run time is less than the specified deadline. If the job's deadline is not satisfied in its current position, the job is moved upward until its new estimated start time allows it to finish in time to satisfy its deadline. After that, the jobs below it are checked and moved upward in the same fashion. This process is repeated until all jobs have been checked and all deadlines found satisfied.

- Moving a normal job to another position in the queue. This situation is handled similar to the removal of a job described above, the difference being that the job is not removed from the FIFO list but put in another position into it.

## 5.4.2   Handling Priority Jobs

This section describes how priority jobs are handled by the scheduler.

- Addition of priority jobs. As described in the functional design, priority jobs are scheduled in the same manner as normal jobs unless doing so would cause the priority job to miss its deadline. In order to obtain such behaviour, new priority jobs are added at the end of the scheduled and FIFO lists, and inserted into the deadline list at the appropriate point. The latter operation is a simple linear traversal of the deadline list, starting at the head, inserting the new job before the first job encountered with a latest possible start time which is higher then the one of the new job. The latest possible start time of a job is determined by subtracting its run time from the deadline.

  Before a new schedule is constructed, it is tested first whether this is at all possible. In situations in which the deadlines become too tight after adding a new priority

job, any order chosen would cause one or more jobs to violate their deadlines. This condition is tested using the ordering of jobs on deadline. It assumes all priority jobs will be executed after one another in the order given by the deadline list of jobs, and checks whether the resulting start times allow all priority jobs to finish in time of their respective deadlines. If all deadlines are met, a new schedule is possible.

The new schedule is calculated in almost the same manner as the schedule was recalculated after removing a normal job from the queue: starting with the FIFO job ordering as initial order, and moving priority jobs upward until all deadlines are met. However, since the scheduled order prior to adding the new priority job already incorporates the deadlines existing at that time, it suffices to add the new job at the end of the scheduled list, and checking and moving priority jobs upward from there until again all deadlines are satisfied. This way, the entire schedule does not need to be recalculated starting with the FIFO order, typically saving a lot of shifting jobs around.

- The removal of priority jobs and moving them to other positions in the job queue. These actions require the schedule to be recalculated starting with the FIFO order, in the same manner as was described for normal jobs.

### 5.4.3   Handling Interactive Jobs

Interactive jobs are added to the scheduled and FIFO lists right after any running job, or if none, at the head. After the addition has been done, the scheduler will start the interactive job right away. If a job was running, it is checkpointed and removed from its partition first.

# Chapter 6

# Conclusions and Future Work

In this report, the design and implementation of a job scheduler called Artaras has been discussed. It has been shown how Artaras is constructed as a distributed application, allowing increased availability of the scheduler. In addition, we have described how Artaras is designed to be extensible both in multicomputers supported and scheduling algorithms. Finally, support has been implemented for the virtual multicomputer Virtua and a scheduling algorithm supporting different job types. This has allowed testing of the scheduler, in addition to which this will be a helpful feature for further development and experimentation.

Although Artaras is in its current form a functional scheduler, work remains to be done. First of all there is the implementation of two features that were described in the design but due to time limitations could not be implemented: handling computation-time quota and support for interactive jobs.

To make Artaras of more practical use, the addition of implementations supporting one or more actual multicomputers is needed. Of both practical and experimental value would be adding scheduling algorithms, e.g., those that can handle variable or dynamic partitioning.

An interesting topic for further study could be ways of separating scheduling *strategy* and *policy*. By this we mean splitting up a scheduling algorithm into the part that knows how to use a given type of architecture (for instance one that allows variable partitioning) and the part of the algorithm that adds all kinds of organisational considerations such as priorities, night-time/day-time schedules, etc.

Finally, job schedulers are open to the addition of a virtually unlimited number of features to make the life of both users and administrators easier. Some useful enhancements to Artaras in this area include:

- A graphical or web interface, allowing comfortable display and handling of the job queue.

- The possibility to receive an e-mail or instant message upon selected scheduling

events, of which of most interest would be the completion of a job.

- The introduction of a mechanism to allow "niceness" when executing jobs, perhaps in the form of an additional job type. The idea is to enhance the spirit of cooperation between users by enabling them to specify their jobs to be "nice" by allowing other jobs to be executed first even though submitted at a later time. A deadline would be specified after which the job stops being nice.

*"Incipere multost quam impetrare facilius"*
*("It is much easier to begin a thing than to finish it")*
*– Alfred Bester*

# Bibliography

[1] Alger, Jeff, *"C++ for Real Programmers,"* 2nd ed., Academic Press Limited, London, 1998.

[2] Deconinck, Geert; Vounckx, Johan; Cuyvers, Rudi and Lauwereins, Rudy, "Survey of Checkpointing and Rollback Techniques," Technical Reports O3.1.8 and O3.1.12 of ESPRIT Project 6731 (FTMPS), ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, June 1993.

[3] Feitelson, Dror G., "A Survey of Scheduling in Multiprogrammed Parallel Systems," Research Report RC 19790 (87657), IBM T.J. Watson Research Center, October 1994.

[4] Feitelson, Dror G. and Rudolph, Larry, "Parallel Job Scheduling: Issues and Approaches," in *Job Scheduling Strategies for Parallel Processing*, D.G. Feitelson and L. Rudolph (eds.), Springer, Berlin, 1995, pp. 1-18.

[5] Finkel, Raphael A., *"An Operating Systems Vade Mecum,"* 2nd edition, pp. 140-141, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[6] Grol, H.J.M, *"Traffic Assignment Problems solved by Special Purpose Hardware with emphasis on Real Time Applications,"* PhD-thesis Technische Universiteit Delft, CIP-Data Koninklijke Bibliotheek, Den Haag, October 1992.

[7] Leon-Garcia, Alberto, *"Probability and Random Processes for Electrical Engineering,"* Section 3.10, Addison-Wesley, Reading, 1989.

[8] Ousterhout, John K., "Scheduling Techniques for Concurrent Systems," in *3rd International Conference on Distributed Computing Systems*, October 1982, pp. 22-30.

[9] Plank, James Steven, *"Efficient Checkpointing on MIMD Architectures,"* Dissertation at Princeton University, Department of Computer Science, June 1993.

[10] Plank, James S.; Beck, Micah; Kingsley, Gerry and Li, Kai, "Libckpt: Transparent Checkpointing under Unix," in *Conference Proceedings, Usenix Winter 1995 Technical Conference*, New Orleans, LA, January 1995, pp. 213-223.

[11] Stroustrup, Bjarne, *"The C++ Programming Language,"* 3rd ed., Addison-Wesley Publishing Company, Reading, 1997.

[12] Wooning, D.A.C, *"Job Scheduling and Checkpointing for the DEMOS Multi-computer,"* literature study, Delft University of Technology, Delft, May 1996.

[13] *"The Distributed ASCI Supercomputer (DAS),"* webpage available at:
http://www.cs.vu.nl/das/

[14] *"DEMOS overview,"* webpage available at:
http://www.cp.tn.tudelft.nl/research/demos/

[15] GNU Compiler Collection (GCC) webpage available at:
http://gcc.gnu.org/

[16] *"GPS: The GNAT Programming System,"* webpage available at:
http://libre.adacore.com/gps/

# Appendix A

# Installation Guide

In this appendix it is described how Artaras can be compiled and installed in three steps: unarchiving, compiling, and installing and running Artaras.

## A.1   Unarchiving the project

The Artaras project is distributed in the form of a gzipped tar (.tgz) file. The project is extracted from the archive file into a directory named `artaras` in the current directory by using the following command:

```
> tar -xzf artaras.tgz
```

or, alternatively on Unixes with a `tar` command that doesn't support gzip internally:

```
> gzip -dc artaras.tgz | tar -xf -
```

The above will have created the directory `artaras`, inside of which can be found:

- File `Readme`. This file contains the latest information and instructions about setting up the project.

- File `Makefile`. The project's main makefile.

- File `scheduler.gpr`. The GPS project file (see next section).

- Directory `src`. This contains the project's source files and accompanying Makefile.

- Directory `doc`. Contains the project's documentation.

- Directory `config`. Contains default and example configuration files.

## A.2 Compiling the project

Artaras was developed using the GPS developer environment [16], but is not dependent on it. To compile the project using this environment, either open the project file `scheduler.gpr` from within the GPS application and use the 'build' option, or at the command line use:

```
> gprmake -Pscheduler
```

If one wishes to compile the project using standard `make`, a `Makefile` is also provided for this purpose. Using a simple

```
> make
```

from within the `artaras` directory will suffice.

## A.3 Installing and Running Artaras

After obtaining the executables in the previous step, the scheduler can be installed using

```
> make install
```

This will copy the scheduler executables to a preferred location and set up an empty job file and default configuration files. If desired, the location in which Artaras is installed can be changed by altering the path defined near the beginning of the makefile.

# Appendix B

# Modules

This appendix gives a short overview of the way in which the scheduler is divided into modules, together with a short description of each module.

## B.1   Overview of Modules

The scheduler project consists of the following modules:

- `schedd.o`. Contains the main routine and help functions that make up the scheduler daemon.

- A separate module for each of the shell commands, containing the main routine and help functions. Each module is named after the corresponding shell command:

    - `submit.o`.
    - `js.o`.
    - `kill.o`.
    - `move.o`.
    - `quota.o`.
    - `shutdown.o`.

- `communication.o`. Provides routines to perform communication between the parts of the scheduler.

- A separate module for each supported multicomputer, providing functions for controlling and monitoring jobs on the multicomputer, and obtaining machine configuration information.

– `machine_virtua.o.`

- `schedule.o.` The scheduling algorithm, containing functions to calculate job schedules and adapt user quota after changes to the job queue.

- A separate module for each supported class of machines with regard to time- and/or space-slicing, providing functions to recalculate a job schedule after adding jobs to, removing jobs from, and repositioning jobs within a job queue.

  – `schedule_fixed.o`

- A separate module for accessing the information stored in each of the shared data files. Each module is named after the data file it accesses:

  – `jobqueue.o.`
  – `currentquota.o.`
  – `jobclassesconfig.o.`
  – `quotaconfig.o.`
  – `settingsconfig.o.`
  – `log.o.`
  – `jobdescription.o.`
  – `machinesconfig.o.`

- `list.o.` Provides basic multilist and list datatypes and the functions to work with such lists.

- `error.o.` Routines to handle errors and exceptions, and report them.

Most parts of the scheduler are machine-independent, and should not require any changes when moving the scheduler to another machine. Normally, when a new and different machine is to be supported by the scheduler, a `machine_(name).o` module must be written specifically for that multicomputer, implementing the required machine-specific functionality (see also Section 5.1).

In addition, if the time-slicing and partitioning possibilities offered by the multicomputer do not fit any of the scheduling algorithms already available through the different `schedule_(class).o` modules, a new module will have to be written implementing the required scheduling algorithm. Once written, such a module can be used to schedule successfully across an entire range of machines (see also Section 5.3).

# Appendix C

# File Formats

In this appendix, the format of the various shared data files used by the scheduler is discussed.

## C.1 Format of `jobqueue`

The file `jobqueue` contains the queue of jobs. The format of this file is quite simple. For each job, the file contains one entry; the entries of multiple jobs are placed one after another. Each job entry starts with a special marker, followed by a number of items that describe the job's characteristics, each item on a separate line. Optional items may be left blank.

A job entry consists of the following items:

1. Job ID.

2. User name.

3. Job status.

4. Name and path of program executable or script, including arguments.

5. Job class.

6. Job type: normal, priority or interactive.

7. Ring to use (optional), or job location if running.

8. Number of processor boards needed.

9. Estimated maximum run-time.

10. Deadline (only for priority jobs).

11. Checkpoint interval and checkpoint duration.

12. Submission time.

13. Starting time.

14. Completion time.

15. Time of last checkpoint.

16. Location on multicomputer (machine-dependent)

17. Run-time job information (machine-dependent)

## C.2   **Format of** `currentquota`

This file contains for each user on a separate line, the following information:

- User name. Name of the user whose quota are described on the line.

- Normal Quota. Amount of time left for the execution of normal jobs in the current month.

- Interactive Quota. Amount of time left this month for executing interactive jobs.

- Priority Quota. Amount of time left this month for executing priority jobs.

## C.3   **Format of** `log`

The log file is made up of the different log entries. Each log entry is contained on a separate line, preceded by the time at which it was added.

## C.4   **Format of configuration and job description files**

The formats of the configuration (`.config`) and job description files are described in the functional design (see Section 3.2.3).