Performance Analysis of Processor Co-Allocation in Multicluster Systems

Anca Bucur
Propositions

Performance Analysis of Processor Co-Allocation in Multicluster Systems

Anca Bucur
March 8, 2004

1. Co-allocation is a viable solution for job scheduling in multicluster systems. [This thesis]

2. When a scheduling policy for co-allocation provides separate queues for single- and multi-component jobs, and does not explicitly favour the latter, the performance of single-component jobs is little influenced by the presence in the system of multi-component jobs. [This thesis]

3. The performance of a scheduling policy may be improved by taking into account the users' behaviour, but certainly not by counting on their cooperation. [This thesis]

4. The effort of debugging a simulation program grows faster with the number of undocumented features in the simulation package used than with the complexity of the model simulated.

5. Badly applied statistics are very frequent in everyday life. The two main errors are ripping off the context, which turns a partial entailment into an absolute entailment, and a set of experiments that is too small.

6. In 90% of the cases, papers are finished and submitted on the night of the deadline, if possible during the hours earned due to the difference in time zone.

7. Ignoring the unfortunate accidents of some of its peers proves to be a well calculated risk for the feline if it finds its favourite food.

8. The number of misconceptions in one's mind is directly proportional to the number of hours spent in front of the TV.

9. A man never reaches the dizzy height of wisdom that he can no longer be led by the nose [Mark Twain]. This is why advertisements never cease to work.

10. A chocolate shop is not the right place to start a diet.

These propositions are considered defendable and as such have been approved by the supervisor, Prof.dr.ir. H.J. Sips.
Stellingen

Performance Analysis of Processor Co-Allocation in Multicluster Systems

Anca Bucur
8 maart 2004

1. Coallocatie is een levensvatbare oplossing voor job scheduling in multicluster-systemen. [Dit proefschrift]

2. Wanneer een scheduling policy voor coallocatie voorziet in afzonderlijke wachtrijen voor jobs met één en met meerdere componenten en de laatste niet expliciet begunstigt, ondervinden de jobs met één component weinig hinder van de aanwezigheid in het systeem van jobs met meerdere componenten. [Dit proefschrift]

3. De prestatie van een scheduling policy kan worden verbeterd door rekening te houden met gedrag van de gebruikers, maar zeker niet door op hun medewerking te rekenen. [Dit proefschrift]

4. De moeite om simulatieprogramma’s van fouten te ontdekken groeit sneller met het aantal ongedocumenteerde eigenschappen van het gebruikte simulatiepakket dan met de complexiteit van het gecompliceerde model.

5. Slecht toegepaste statistieken komen veel voor in de dagelijkse praktijk. De belangrijkste twee fouten zijn het weglaten van de context, waardoor slechts gedeeltelijk geldige conclusies verwerkt tot absolute conclusies, en een te kleine versameling experimenten.

6. In 90% van de gevallen worden papers voltooid en ingediend in de nacht van de deadline, mogelijkerwijls tijdens de uren die gewonnen worden door het verschil in tijdzone.

7. Het negeren van de onfortuinlijke ongelukken die sommige van zijn soortgenoten overkomen blijkt een goed ingebed risico te zijn voor de kat die zijn favoriete voedsel vindt.

8. Het aantal misvattingen dat bij iemand heerst is direct evenredig met het aantal uren dat hij voor de TV doorbrengt.


10. Een chocolade-winkel is niet de goede plaats om met een dieet te beginnen.

Deze stellingen worden verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor, Prof.dr.ir. H.J. Sips.
Performance Analysis of Processor Co-Allocation in Multicluster Systems

Anca Bucur
Performance Analysis of Processor Co-Allocation in Multicluster Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College van Promoties,
in het openbaar te verdedigen op maandag 8 maart 2004, om 13:00 uur

door Anca Ioana Daniela BUCUR

inginer, Universitatea Politehnica Bucureşti
geboren te Boekarest, Roemenië
Contents

1 Introduction 1
   1.1 Setting the context 1
   1.2 The problem of processor co-allocation 3
   1.3 Outline of the thesis 4

2 A model for co-allocation in multicluster systems 7
   2.1 The DAS system 7
   2.2 The structure of the system 8
   2.3 The structure of jobs 10
   2.4 The communication structure within jobs 11
       2.4.1 Explicitly modeling the communication structure 11
       2.4.2 Extending single-cluster service times 12
       2.4.3 Using total run-time measurements 12
   2.5 The workloads 13
       2.5.1 Job sizes 13
       2.5.2 Job service times 15
   2.6 The job-placement rules 16
   2.7 The queuing disciplines 18
   2.8 The scheduling policies 19
   2.9 Related work 22
       2.9.1 Co-allocation and grid computing 22
       2.9.2 Scheduling in parallel systems 24
       2.9.3 Common issues with NUMA architectures 25
       2.9.4 Non-FCFS queuing disciplines 26
       2.9.5 Maximal utilization and bin-packing-related research 28
       2.9.6 Research in the context of the DAS 29

3 The capacity loss in multicluster systems 31
   3.1 The maximal utilization and the capacity loss 32
   3.2 Reasons for capacity loss 33
   3.3 Formulas for FCFS 33
       3.3.1 Ordered requests 33
       3.3.2 Unordered requests 35
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Simulation methods</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>The accuracy of the approximation and the simulations</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>Results</td>
<td>38</td>
</tr>
<tr>
<td>3.6.1</td>
<td>The influence of the job-size and service-time distribution</td>
<td>39</td>
</tr>
<tr>
<td>3.6.2</td>
<td>The influence of the queuing discipline and of the placement policy</td>
<td>41</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Using data from the logs</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>Asymptotic behaviour</td>
<td>42</td>
</tr>
<tr>
<td>3.8</td>
<td>Conclusions</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Co-allocation with GS without communication</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>The component-size and service-time distributions</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1</td>
<td>The influence of the component-size distribution</td>
<td>46</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The influence of the service-time distribution</td>
<td>48</td>
</tr>
<tr>
<td>4.1.3</td>
<td>The influence of the request type</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>The influence of the queuing discipline</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Equal versus different cluster sizes</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparing systems with different numbers of clusters</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>The fractions of jobs with different numbers of components</td>
<td>53</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Jobs with components from the same distribution</td>
<td>54</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Jobs with the same total request size</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusions</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>Co-allocation with GS and communication</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Parameter settings</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>The influence of the communication-speed ratio</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>The influence of the placement policies for flexible requests</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Mixes of job request types</td>
<td>62</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>Scheduling policies for co-allocation</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Parameter settings</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Policies with multiple variations</td>
<td>67</td>
</tr>
<tr>
<td>6.2.1</td>
<td>The LS policy</td>
<td>67</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The LP policy</td>
<td>70</td>
</tr>
<tr>
<td>6.2.3</td>
<td>The EQ policy</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance comparison of the policies</td>
<td>73</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Dealing with both single- and multi-component jobs</td>
<td>74</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Dealing with only single- or multi-component jobs</td>
<td>78</td>
</tr>
<tr>
<td>6.3.3</td>
<td>The unbalanced case</td>
<td>78</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Local versus global queues</td>
<td>80</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Balanced versus unbalanced local queues</td>
<td>80</td>
</tr>
</tbody>
</table>
Contents

6.4 Conclusions ........................................... 81

7 Simulations based on real workloads ....................... 83
7.1 Trace-based simulations .................................. 84
7.1.1 Comparing the policies ............................ 84
7.1.2 Limiting the total job size ........................ 88
7.1.3 Setting the job-component-size limit ............... 89
7.1.4 Gross versus net utilization ....................... 91
7.1.5 Conclusions ........................................ 94
7.2 Description of the applications ......................... 95
7.2.1 The Ensflow application ............................ 95
7.2.2 The Poisson application ............................ 96
7.3 Runtime measurements on the DAS ...................... 97
7.3.1 Total runtime of the Ensflow application .......... 98
7.3.2 Total runtime of the Poisson application .......... 98
7.4 Simulations based on runtime measurements ............ 100
7.4.1 Evaluation of the scheduling policies ............. 101
7.4.2 Gross versus net utilization ....................... 107
7.4.3 Conclusions ........................................ 111
7.5 Simulations based on detailed application measurements ........................................... 114
7.5.1 Measurements ...................................... 114
7.5.2 Simulation results .................................. 114
7.5.3 Conclusions ........................................ 115

8 Co-allocation versus no co-allocation .................... 117
8.1 Unordered requests versus flexible requests ............ 117
8.2 The capacity loss ..................................... 118
8.3 Communication represented by the communication-speed ratio ......................................... 119
8.4 Communication included through runtime measurements ........................................... 121

Appendices ............................................... 122

A Communication-time measurements ....................... 123
A.1 Per-process communication time for exchanging borders ........................................... 123
A.2 Synchronized and non-synchronized operation ........ 123
A.3 Total communication time for exchanging borders ........................................... 124
A.4 Data transfer rate of exchanging borders ............. 124

B Simulation details ....................................... 129
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>130</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>135</td>
</tr>
<tr>
<td>Summary</td>
<td>137</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>139</td>
</tr>
<tr>
<td>List of publications</td>
<td>141</td>
</tr>
<tr>
<td>Curriculum vitae</td>
<td>143</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Setting the context

Over the last decade, clusters and distributed-memory multiprocessors consisting of hundreds or thousands of standard CPUs have become very popular. Compared to single-cluster systems, multic Peace: systems made up of multiple, geographically distributed clusters interconnected by high-speed wide-area networks can provide a larger computational power. Instead of smaller groups of users with exclusive access to their single clusters, larger groups of users can share the multic Peace: The potential lead to lower job turn-around times and higher system utilizations, and makes larger job sizes possible by allowing jobs to use processors in multiple clusters simultaneously. The possibility of creating multiclusters fits with the recent interest in computational and data grids [4, 28], in which it is envisioned that applications can access resources (hardware resources such as processors, memory, and special instruments, but also data resources) in many different and possibly widely dispersed locations at the same time to accomplish their goals. Applications running on multiclusters, and more generally, in grids, may then require (processor) co- allocation, i.e., the simultaneous allocation of resources (processors) in different clusters or subsystems of a grid.

There are two problems that may occur when employing processor co-allocation in multic Peace: systems. First, applications may not be very well suited for multi-cluster execution because they cannot deal well with the relatively slow wide-area links interconnecting the component clusters, which degrades their performance too much to justify such execution. Second, scheduling policies for such systems have to consider more restrictions than schedulers for single clusters in that components of single jobs may need to fit in separate clusters. Most of the research on processor scheduling in parallel computer systems has been dedicated to multiprocessors and single-cluster systems, but little attention has been devoted to multic Peace: systems. In this thesis, we study the performance of scheduling policies employing processor co-allocation in multic Peace: systems.

Many scheduling strategies have been developed for parallel systems [25] in
an attempt to improve their performance. Techniques such as gang scheduling—using time sharing across multiple processors—and different forms of backfilling—allowing certain jobs not at the head of the queue to start execution—have been devised. However, a simple and yet, due to its practicality, often used strategy is to allow only rigid jobs scheduled by pure space sharing, i.e., jobs requiring fixed numbers of processors that are executed on these processors exclusively until completion. This is also the strategy considered in this thesis.

The performance metrics employed are the average response time of jobs, which is a user-oriented metric, and the maximal utilization for which the system is still stable, which is system-oriented. They are the most frequently used metrics for assessing the performance of scheduling in parallel systems. From a practical point of view, a high utilization of real systems is often seen as a measure of successful (and profitable) operation. From a theoretical perspective, a very important problem in mathematical models of parallel job scheduling is to find the values of such parameters as the arrival rates, job sizes, and service times for which the system is stable.

An example of multicluster systems is the five-cluster 400-processor second-generation Distributed ASCI Supercomputer (DAS) [1, 11] (and its predecessor) (see Figure 1.1), which was an important motivation for our work. This system
was designed and deployed by the Dutch Advanced School for Computing and Imaging (ASCII) in the Netherlands to assess the feasibility of running parallel applications across wide-area systems [10, 36, 44], and more generally, to perform research in parallel and distributed computing. In the most general setting, grids and grid resources are very heterogeneous; in this thesis we restrict ourselves to homogeneous multicluster systems such as the DAS and to processor co-allocation. Showing the viability of processor co-allocation in such systems may be regarded as a first step in assessing the benefit of co-allocation of various types of resources in more general grid environments.

1.2 The problem of processor co-allocation

In our model for studying processor co-allocation in multiclusters, the system consists of a set of clusters of possibly different sizes with fast intra-cluster links and slow inter-cluster links. We assume jobs to be rigid and to be scheduled by policies employing space sharing. Jobs may consist of multiple components, each component being scheduled on a different cluster. We allow four request types in that jobs may specify only the total number of processors they need—flexible requests—or the numbers of processors needed in the separate clusters—unordered requests—or in each of the clusters—ordered requests—of the multicluster system. For comparison with the single-cluster case, we introduce total requests, which only specify the total number of processors needed, equal to the number required by (unordered requests, but this time in a single cluster. We design six scheduling policies for three different queuing structures: one with only a global queue, one with only local queues, and four with both, in which case single-component jobs go to the local queues and multi-component ones to the global queue. For some of the policies we define several variations.

We define the gross utilization as the utilization computed from the actual service times experienced by jobs, which for multicomponent jobs includes the time spent in the slow wide-area communication. The net utilization is defined as the utilization computed from the single-cluster service times of jobs of the same total size, which gives a measure of the throughput of the system. When there is no co-allocation, there is no wide-area communication, and the net utilization is equal to the gross utilization.

Our two performance metrics are the mean job response time as a function of the utilization, and the maximal utilization at which multicluster systems are stable. In studies of scheduling in single clusters it has been shown that the achievable (maximal) utilization may be much less than 100%, a problem that may be aggravated in multicluster systems. We assess the maximal utilization when co-allocating jobs in multiclusters for the case of a single global queue, both with analytic means (we derive exact and approximate formulas when the service-time distribution is exponential), and with simulations with synthetic workloads and with workloads derived from the logs of actual systems.

In the case of a single global queue, the performance of co-allocation in multi-
cluster systems is evaluated depending on such parameters as the structure and the sizes of job requests, the queuing discipline, the number and sizes of the clusters making up the multicluster, and the communication-speed ratio, which is the ratio between the inter-cluster and the intra-cluster communication speeds. The queuing disciplines considered are First Come First Served (FCFS) and First Fit Processors First Served (FFFS), which is a form of backfilling. While with the first discipline the scheduler always respects the queuing order, for the second discipline jobs may be allowed to leapfrog the job at the head of the queue when it does not fit.

Using co-allocation does not mean that all jobs have to be split into components and spread over the clusters, small jobs can also be submitted as single-component jobs and go to a single cluster. In general, there is in the system a mix of jobs with different numbers of components. In this context, an important decision to make is whether there will be one global scheduler with one global queue in the system, or more schedulers, and in the second case how jobs will be divided among the schedulers. We evaluate various system configurations defined by such parameters as the number of schedulers and queues in the system, the way jobs with different numbers of components are distributed among these queues, and the priorities imposed on the schedulers. We study the influence each of these choices have on performance for various fractions of jobs with different numbers of components and for synthetic workloads.

In addition to synthetic workloads, we also consider workloads derived from traces of actual systems and from measurements of two application on the DAS. We investigate and compare the total runtimes of single-cluster and multicluster execution on the DAS of these two parallel applications, which model physical phenomena. Subsequently, we assess the performance of three scheduling policies, one for each queuing structure, for co-allocation in multiclusters, with simulations using the runtime measurements.

Finally, we compare systems that use co-allocation with systems that do not, to find out whether co-allocation can be beneficial even in cases when it can be avoided, which is when all jobs fit in a single cluster. Multiclusters that do not employ co-allocation either keep jobs local to single clusters, or they share the load among the clusters, and we consider both choices.

### 1.3 Outline of the thesis

In this thesis we assess the performance of co-allocation in multicluster systems for a wide variety of parameter settings. We have designed and evaluated six scheduling policies for co-allocation, some of which with multiple variations. Our conclusions are presented in the conclusions sections of the chapters. The material in this thesis is structured as follows.

- In Chapter 2 we describe our model of multicluster systems, the workloads, the queuing disciplines, and the scheduling policies.
1.3. **Outline of the thesis**

- Chapter 3 assesses the maximal utilization when co-allocating jobs in multicloud systems with analytic means and with simulations, using both synthetic workloads and workloads obtained from the logs of actual systems.

- In Chapter 4 we simulate multicloud systems without communication, and evaluate the performance of co-allocation for different structures and sizes of job requests, different numbers and sizes of clusters, and two queuing disciplines. The scheduling policy employed in this chapter uses a single global queue.

- In Chapter 5 we maintain the scheduling policy with a single global queue and introduce communication. We simulate multicloud systems with the FCFS queuing discipline and with several policies for placing flexible requests, under workloads consisting of a single or of a mix of request types and for different communication-speed ratios.

- In Chapter 6 we assess the response times of both single- and multi-component jobs in multicloud systems for the six scheduling policies with different queuing structures defined in our model and for synthetic workloads. We vary the numbers of queues, the priority orders in which jobs are scheduled from multiple queues, and the fractions of jobs with different numbers of components.

- In Chapter 7 we restrict ourselves to three representative scheduling policies for co-allocation in multiclusters, one for each queuing structure. We use workloads derived from a DAS log and from runtime measurements of single-cluster and multicloud execution of two parallel applications on the DAS. We have also performed detailed measurements of the time spent in communication of one of the two applications. Because the results of these measurements are not used in the simulations, they are included in Appendix A.

- Chapter 8 compares systems with and without co-allocation, to derive various sets of parameters for which co-allocation is beneficial.
Chapter 2

A model for co-allocation in multicluster systems

In this chapter we present our model of multicluster systems based on the Distributed ASCI Supercomputer (DAS). We start in Section 2.1 with a description of the DAS system. In Sections 2.2–2.4 we introduce our assumptions concerning the structure of the system, the structure of the job requests, and the communication structure of the applications. Section 2.5 describes the workloads we use when simulating the model we defined, and Sections 2.6–2.8 discuss our scheduling decisions. In Section 2.9 we review related work.

2.1 The DAS system

The Distributed ASCI Supercomputer is a wide-area computer system located at a number of Dutch universities. The system is now in its second generation, was designed by the Advanced School for Computing and Imaging (ASCI, in the Netherlands) and is used for research in parallel and distributed computing.

The DAS1, the first-generation DAS system [2, 11], consisted of four clusters of identical Pentium Pro processors, one with 128, the other three with 24 processors each. The clusters were interconnected by ATM links for wide-area communications, while for local communication inside the clusters Myrinet LANs were used. The local and wide-area application-level bandwidths and roundtrip latencies were 50 MByte/s and 0.55 MByte/s, and 40 μs and 2.5 – 3.0 ms, respectively, yielding a ratio of the communication speeds of about two orders of magnitude. The operating system employed was RedHat Linux. On single DAS clusters a local scheduler (prun) was used that allowed users to request a number of processors bounded by the clusters' sizes, for a time interval during the day not exceeding 900 seconds. The Globus toolkit [5] was installed on the DAS system, and an interface was created between prun and Globus to allow the submission of jobs spanning multiple clusters. In addition, multicluster jobs could also be submitted directly and
all components simultaneously started on the clusters without Globus, by using a special option of prun. This way around Globus was used for the measurements in Section 7.5.

The DAS2, the second-generation DAS system which was installed at the end of 2001 when the first-generation DAS1 system was discontinued [1, 11], is a wide-area computer system consisting of five clusters of identical, 1 GHz Pentium III, dual-processor nodes, one with 72, the other four with 32 nodes each, for a total of 400 processors. The clusters are interconnected by the Dutch university backbone for wide-area communications (100 Mbit/s), while for local communications inside the clusters again Myrinet LANs (1,200 Mbit/s) are used. On single DAS clusters now the PBS scheduler [6] is employed, while jobs spanning multiple clusters can be submitted with Globus [5].

2.2 The structure of the system

We model a multicluster distributed system consisting of \( C \) clusters of processors, with cluster \( i \) having \( N_i \) processors, \( i = 1, \ldots, C \). We assume that all processors have the same service rate.

![Diagram](image)

Figure 2.1: A multicluster system with only a single (global) scheduler.

By job we mean a parallel application requiring some number(s) of processors, possibly in multiple clusters simultaneously (co-allocation). We assume that jobs only request processors, so we do not include in the model any other types of resources. Jobs are rigid, so the numbers of processors requested by and allocated to a job are fixed, and cannot be changed during its execution. We call a task the part of an application that runs on a single processor. Tasks can communicate by exchanging messages over the network. All tasks start and end at the same time, which implies that all the processors allocated to a job are being simultaneously
occupied and released. Preemption is not admitted, nodes being released only when the tasks running on them end.

When we explicitly model the communication in the structure of jobs (see Section 2.4), we assume that the communication network is complete—any two processors can communicate with each other—and that the communication between the tasks of a parallel application is synchronous—a task doing a send operation can only continue when it knows the receiving task has received the message. All
intra-cluster communication links are considered to have the same capacity and latency, as are all the inter-cluster links. The parameter related to communication in our model is the *communication-speed ratio*, defined as the ratio between the time needed to complete a single synchronous send operation between processors in different clusters and in the same cluster.

The system can have a single central scheduler with one global queue (see Figure 2.1), a local scheduler with its own queue for each cluster (see Figure 2.2), or both global and local schedulers (see Figure 2.3). When there are both global and local queues in the system, all the multi-component jobs are stored in the global queue and all the single-component jobs in the local queues.

When $C$ equals 1, the system is a single cluster. In many cases we compare the performance of a multiclus-ter system with that of a single cluster system with the same total number of processors. We refer to a system with $C$ clusters of identical sizes $n$ with the notation $C \times n$.

### 2.3 The structure of jobs

Jobs that require co-allocation have to specify the number and the sizes of their components, i.e., of the sets of tasks that have to be run in separate clusters. A job is described by a tuple of $C$ values (the same as the number of clusters), at least one of which is non-zero.

We will consider four cases for the structure of jobs, which are differentiated by the flexibility of their requests:

1. In an *ordered request* the positions of the request components in the tuple specify the clusters from which the processors must be allocated.

2. For an *unordered request*, by the components of the tuple the job only specifies the numbers of processors it needs in the separate clusters, allowing the scheduler to choose the clusters for the components (see Section 2.6 for the placement policies considered).

3. A *flexible request* specifies the total number of processors needed, obtained as the sum of the values in the tuple, letting the scheduler decide how to spread the tasks over the clusters (see Section 2.6 for the different ways of splitting up flexible jobs).

4. For *total requests*, there is a single cluster and a request specifies only the total number of processors needed, again obtained as the sum of the values in the tuple.

As long as we do not take into account the communication, the cases of flexible and total requests amount to the same. The way we choose the job component sizes for ordered and unordered requests and the total job sizes for flexible and total requests, makes the results for the four cases comparable.
Ordered requests are used in practice when a user has enough information about the complete system to take full advantage of the characteristics of the different clusters. For example, the data available at the different clusters may dictate a specific way of splitting up an application. Unordered requests, especially when grouping request components on the same cluster would be allowed (we do not need to explicitly consider this case for unordered requests since it is included in the general notation introduced above), are modeled by applications like FFT, where tasks in the same job component share data and need intensive communication, while tasks from different components exchange little or no information. Flexible requests are the best from the system’s point of view because of their lack of restrictions concerning the placement of their tasks gives the scheduler the possibility to improve the overall performance of the system.

2.4 The communication structure within jobs

The communication among the tasks of a job is an important part of our model; in general, co-allocation introduces communication over the slow wide-area links, which may decrease the performance of the system. The way jobs are split among clusters influences the amount of global communication impacting this way the performance. For example, a job requiring 16 nodes would send no wide-area messages during an all-to-all personalized communication step if scheduled on a single cluster. The same job would send 128 (slow) wide-area messages if it had two components of size 8, and 192 of such messages for four components of size 4.

There are three different ways in which we include communication in our model.

2.4.1 Explicitly modeling the communication structure

The most detailed way of including communication takes as a model for the structure of jobs a parallel application, each process of which performing the same algorithm and alternating between computation and communication steps; in the latter, we assume that an all-to-all personalized message exchange is performed. The communication steps achieve also the synchronization among the tasks. A task sends its messages successively and synchronously, which means that it waits until a message has been received before it sends the next, and that in the mean time, it receives all messages sent to it by the other tasks. After all these messages have been sent and received, a task starts a computation step.

Although the sizes of the messages are supposed to be equal, the time costs will not be the same, depending on whether the sender and the receiver are in the same cluster or not. The total service time of a job is the sum of the durations of the computation steps and of the communication steps.

The effect of including communication in our model is that the service times of tasks are extended by the times needed for communication. When a task of a job with service time $t$ sends $m_i$ intra-cluster (local) messages and $m_w$ inter-cluster (wide-area) messages, its total service time including communication is
$t + m_l r l$, with $l$ the time (assumed deterministic) needed for a local message, and $r$ the communication speed ratio. Because different tasks in the same job will have different values for $m_l$ and $m_w$, the total service time varies among the tasks in the same job. As jobs only release their processors when all tasks have finished, the total service time of a job is determined by the longest tasks. With the communication speed ratios we consider, these are the tasks with the largest values of $m_w$, which are part of the smallest job component (also see Section 2.6).

We use this method of including communication in Chapter 5 and Section 7.5.

2.4.2 Extending single-cluster service times

A second way to include communication in the model is to start by taking single-cluster service times, and for all multi-component jobs to extend these service times with a factor, independent of the number of components, in order to account for the wide-area communication. To choose a realistic value for this extension factor we have looked at different relevant results presented in the literature and also ran our own experiments. In [49], the performance of four parallel applications in wide-area systems is assessed by comparing the speedups of the original applications on a 64-cluster system with the speedups of versions of the applications optimized for wide-area execution on a 4x16 multicluster system. For three of the four applications, these speedups are 60.6 and 53.9, 60.9 and 61.1 (which is counter-intuitive, but the reason is explained in [49]), and 55.4 and 52.5, respectively; the fourth application (All-pair Shortest Paths) has very poor multicluster performance. So for these three applications, the extension factor for multicluster operation does not exceed 1.12. In [9], experiments are presented with distributing a large, wide-area optimized application solving Einstein's equations across four supercomputers in two locations using Globus. For a total of 1140 processors, an efficiency of 88% is cited, which amounts to an extension factor of 1.14. In addition, in [21] it is concluded that it pays off to use co-allocation when the extension factor does not exceed 1.25.

In Section 7.3 we show the extension factors obtained for our applications on the DAS and, based on the results presented above and on our experiments, we choose an extension factor which we use in the simulations in Section 7.1.

2.4.3 Using total run-time measurements

A third way to include wide-area communication in the model is to directly measure the execution times of the applications we study, in several configurations of processors and with different numbers of components and to use the results of these measurements in the simulations. We use this method in Section 7.4. In our simulations based on execution-time measurements the service times include the communication times exactly as they were measured when running the applications on the DAS.
2.5 The workloads

We define the workload by specifying the arrival process of jobs, the total job sizes or the sizes of the job components, and the service times of the jobs. The arrival process is always assumed to be Poisson. For both job sizes and service times, we use synthetic distributions, distributions derived from logs of actual systems, amongst which the DAS, or the values as measured on the DAS.

2.5.1 Job sizes

The synthetic distribution used for the non-zero sizes of the job components is either the uniform distribution $U[n_1, n_2]$ on some interval $[n_1, n_2]$ with $0 < n_1 \leq n_2$, or the distribution $D(q)$ defined as follows: $D(q)$ takes values on some interval $[n_1, n_2]$ with $0 < n_1 \leq n_2$, and the probability of having job-component size $i$ is $p_i = q^i/Q$ if $i$ is not a power of 2 and $p_i = 3q^i/Q$ if $i$ is a power of 2, with $Q$ such that the $p_i$ sum to 1. This distribution favours small sizes, and sizes that are powers of two, which has been found to be a realistic choice [15].

For total and flexible requests we use the sum of multiple copies of the distributions defined above for job-component sizes. Also when comparing co-allocation to no co-allocation we use sums of copies of these distributions (see Chapter 8).

Although co-allocation is (was) possible on the DAS2 (DAS1), so far it has not been used enough to derive statistics on the sizes of the jobs' components. Before we started our simulations we could not obtain a relevant log from the recently installed DAS2. However, from the log of the largest cluster (128 processors) of the DAS1 we found that over a period of three months, the cluster was used by 20 different users who ran 30,558 jobs. The sizes of the job requests took 58 values in the interval [1, 128], for an average of 23.34 (see Table 2.1); their density is presented in Figure 2.4. The results are in line with those presented in the literature for other systems, in that there is an obvious preference for small numbers and powers of two [15] (see also Table 2.2).

For the job sizes of actual systems, we use in our simulations the log of a three-month period of the first-generation DAS1 system discussed above, and the log of the Cornell Theory Center (CTC) of Feitelson's parallel workload archive [3]. As both logs are for single-cluster systems, they only contain the total sizes of jobs. Statistics of the two logs are presented in Table 2.1 (cv stands for coefficient of variation). When using these logs, we only consider unordered and total jobs, clusters of equal size, and generate the (dependent) job-component sizes as described above.

Since all our simulations using the logs of the DAS1 and of the CTC were performed for systems with a total of 128 processors (single clusters with 128 processors and multiclusters with 4 clusters of 32 processors each), we restricted the total job size to this value and excluded from the CTC log all the larger jobs. By CTC-128 we refer to the CTC log cut at 128.

By sampling the job-size distribution as measured on the DAS1 we derive two distributions from the DAS log, which we use in our simulations for the total job
Table 2.1: Statistics of the total job-size distributions derived from the logs.

<table>
<thead>
<tr>
<th>log</th>
<th>number of jobs</th>
<th>mean</th>
<th>cv</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTC</td>
<td>79,302</td>
<td>10.72</td>
<td>2.26</td>
</tr>
<tr>
<td>CTC-128</td>
<td>78,865</td>
<td>9.53</td>
<td>1.83</td>
</tr>
<tr>
<td>DAS-s-128</td>
<td>30,558</td>
<td>23.34</td>
<td>1.11</td>
</tr>
<tr>
<td>DAS-s-64</td>
<td>29,947</td>
<td>21.46</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Table 2.2: The fractions of jobs with sizes powers of two in the DAS1 log.

<table>
<thead>
<tr>
<th>total job size</th>
<th>fraction of the jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.091</td>
</tr>
<tr>
<td>2</td>
<td>0.130</td>
</tr>
<tr>
<td>4</td>
<td>0.087</td>
</tr>
<tr>
<td>8</td>
<td>0.066</td>
</tr>
<tr>
<td>16</td>
<td>0.090</td>
</tr>
<tr>
<td>32</td>
<td>0.039</td>
</tr>
<tr>
<td>64</td>
<td>0.190</td>
</tr>
<tr>
<td>128</td>
<td>0.012</td>
</tr>
</tbody>
</table>

sizes. DAS-s-128 is based on the entire log, while DAS-s-64 is obtained from the log cut at 64; the maximum size of a job is reduced to half excluding only 2% of the jobs from the log—the percentage of jobs that require more than 64 processors. The reason to introduce DAS-s-64 was to check whether limiting the total job size improves the performance (see Section 7.1.2).

When we use the logs for total requests in a single cluster, the total job-size distribution is directly used. In the multicluster case, a size limit is set for the job components. The number of components of a job is computed depending on the size limit, with jobs having as few components as possible. As long as the number of components is smaller than the number of clusters no component is larger than the size limit; only when the number of components is equal to the number of clusters may the size limit be exceeded. Once we decide on the number of components of a job, we split the job into components of sizes as equal as possible. We consider three size limits for the components: 16, 24, and 32; with the size limit we vary the numbers and the sizes of the job components (see Table 2.3). Besides the obvious reason of being able to fit them on clusters with 32 processors, we split the jobs and vary the maximum sizes of their components to observe the effect of limiting the component sizes on the performance (see Section 7.1.3).
2.5. The workloads

Figure 2.4: The density of the job-request sizes for the largest DAS1 cluster (128 processors).

Table 2.3: The fractions of jobs with the different numbers of components for the DAS-s-128 distribution and the three job-component-size limits.

<table>
<thead>
<tr>
<th>job-component-size limit</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0.513</td>
</tr>
<tr>
<td>24</td>
<td>0.738</td>
</tr>
<tr>
<td>32</td>
<td>0.780</td>
</tr>
</tbody>
</table>

2.5.2 Job service times

The synthetic job service-time distributions we use are the deterministic, the exponential, and the hyperexponential distributions. For data from actual systems, we again use the CTC and DAS1 logs. Table 2.4 gives statistics of the service times in the logs. In DAS-t only the jobs for which the service times could be computed are included, and DAS-t-900 only contains jobs with service times at most equal to 900 seconds.

We made the choice to use both synthetic distributions and distributions from actual systems because with the second type of distribution we may obtain a more accurate, realistic evaluation of the performance in those systems, but at the same time these distributions might be very specific and make our results hard to compare to those for other systems. On the other hand, the synthetic distributions may be less realistic but more general and more suited for analysis.

In the DAS1 log, 28,426 jobs were recorded with both their starting and ending times, and we could compute their service times. Influenced by the fact that during working hours jobs are restricted to 15 minutes of service (they are automatically killed after that period), 94.45% of the recorded jobs ran for less than
Table 2.4: Statistics of the service-time distributions derived from the logs.

<table>
<thead>
<tr>
<th>Service-time statistics</th>
<th>number of jobs</th>
<th>mean</th>
<th>cv</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTC</td>
<td>79,302</td>
<td>10,983.42</td>
<td>1.65</td>
</tr>
<tr>
<td>CTC-128</td>
<td>78,865</td>
<td>11,006.09</td>
<td>1.65</td>
</tr>
<tr>
<td>DAS-t</td>
<td>28,426</td>
<td>356.45</td>
<td>5.37</td>
</tr>
<tr>
<td>DAS-t-900</td>
<td>26,850</td>
<td>62.66</td>
<td>2.05</td>
</tr>
</tbody>
</table>

Figure 2.5: The density of the service times for the largest DAS1 cluster (128 processors).

15 minutes. Figure 2.5 presents the density of service time values on the DAS1, as it was obtained from the log. The average service time is 356.45 seconds and the coefficient of variation is 5.37. In our simulations, we use for the service-time distribution the distribution derived from the log of the DAS, cut off at 900 seconds (DAS-t-900). See Table 2.4 for the average service time of the jobs in the cut log and their coefficient of variation, and for a comparison to the CTC log.

Still, not all jobs in the DAS1 log were short: the longest one took around 15 hours to complete. Figure 2.6 divides the service times of the jobs into eight intervals: < 10s, 10–30s, 30–60s, 60–300s, 300–900s, 900–1800s, 1800–3600s, and > 3600s.

### 2.6 The job-placement rules

For ordered and total requests the way of placement is specified by the request. Based on the numbers of idle processors in the clusters and the sizes of the job components it is also clear whether a job fits on the system.
2.6. The job-placement rules

Figure 2.6: The density of the service times of the jobs in the DAS1 log divided into eight intervals.

For unordered and flexible requests the placement is less obvious; in this section we discuss the different possibilities for job placement we consider for these request types.

In order to determine whether an unordered request fits, we first order the job component sizes, and then try to schedule the components in decreasing order of their sizes. Whatever way of placement is used, if placing the job components in this order does not succeed, no other order will. Possible ways of placement include First Fit (FF; fix, once and for all, an order of the clusters and pick the first one on which a job component fits), Best Fit (BF; pick the cluster with the smallest number of idle processors on which the component fits), and Worst Fit (WF; pick the cluster with largest number of idle processors). We will only study FF and WF. If we consider the influence each placement has on the jobs following in the queue, WF can be expected to give better results than the other placement methods when combined with the FCFS policy, because it leaves in each cluster as much room as possible for subsequent jobs.

For flexible requests, we first determine whether there are enough idle processors in the whole system to serve the job. If so, the job is placed on the system according to one of the following three placement policies, Cluster Filling (CF), Load Balancing on the Smallest number of clusters (LB-S), and Load Balancing on All clusters (LB-A). For CF and LB-S, the clusters on which the job will be scheduled are chosen in a WF manner by taking the smallest set of clusters with enough idle processors. CF completely fills the clusters with the largest numbers of idle processors until all the tasks are distributed; LB-S distributes the request over these clusters in such a way as to balance the load (in the sense of leaving (almost) equal numbers of processors idle). LB-A does exactly the same, but with all clusters.

The reason for the way of selecting clusters for CF and LB-S is that because
of the higher cost of intercluster communication, it is preferable to schedule a job on as few clusters as possible. CF has the potential advantage of a smaller total number of intercluster messages among the tasks in a job, while LB-S has the advantage of maintaining comparable loads. As an example, consider a flexible request of 18 processors coming in an empty system consisting of four clusters with 8 processors each. When LB-S is used, the job is split into three components of 6 processors each and there are 216 intercluster messages in a single all-pairs message exchange among the job’s tasks. With CF the job is again spread over three clusters, but two of the components are of size 8 and the third of size 2, which yields only 192 intercluster messages for CF. When all the requests in the system are flexible, LB-S does not change the maximal utilization and CF may provide better performance, but when combined with other types of requests (ordered for example), the advantage of having comparable loads and of avoiding completely filling some of the clusters while leaving some others emptier is obvious. Using CF for flexible requests would be very obstructive towards both ordered and unordered requests, and would result in a loss of capacity. While LB-A can bring larger communication costs, it may yield better performance for mixed workloads than the other two ways of placement. We will deal with these issues in Section 5.4.

2.7 The queuing disciplines

To observe the contribution of the scheduling scheme to the system’s performance, apart from the First Come First Served (FCFS) policy, the Fit Processors First Served (FPFS) policy explained below was implemented.

FCFS is the simplest scheduling scheme, processors being allocated to the job at the head of the queue. When this job does not fit, the scheduler is not allowed to choose another job further down in the queue. This restriction has a negative influence on the maximal processor utilization, since processors may have to stay idle even when one or more jobs in the queue do fit.

With the FPFS policy, the scheduler searches the queue from head to tail for jobs that fit. It is similar to backfilling [7], except that the durations of jobs (which we assume unknown) are not taken into account, so the usual requirement of backfilling that the job at the head of the queue should not be delayed is not enforced. To avoid starvation (a job is never scheduled), we use an aging mechanism with counters: each job in the queue has a counter which is incremented each time the job is leapfrogged by another job (another job situated behind it in the queue is scheduled first); when a job’s counter reaches a chosen limit, the scheduler is not allowed to overpass that job anymore. This way, the effectiveness of scheduling is preserved. Of course, when the limit is equal to zero, FPFS becomes FCFS. FPFS has the potential advantage of an increased maximal utilization of the system compared to FCFS.
2.8 The scheduling policies

In a multicluster system where co-allocation is used, jobs can be either single-component or multi-component, and in a general case both types are simultaneously present in the system. We make this distinction since the single-component jobs do not use co-allocation while multi-component jobs do. A scheduler dealing with the first type of jobs can be local to a cluster and does not need any knowledge about the rest of the system. For multi-component jobs, a scheduler needs global information for its decisions. Treating both types of jobs equally or keeping single-component jobs local and scheduling only multi-component jobs globally over the entire multicluster system, having a single global scheduler or a scheduler local to each cluster, these are decisions that influence the performance of the system.

We define six policies, one with only a global queue, one with only local queues, and four with both. For some of the policies we define several variations, depending on the order in which jobs are scheduled from the queues. The rules of the scheduling policies are invoked either when a job departs or when a job arrives at an empty queue. We will describe below only the behaviour of the system at job departures since job arrivals at empty queues are a degenerate case of it.

In our description of the policies we say that a queue is enabled when its corresponding scheduler is allowed to start jobs from that queue. In the initial state when all queues are empty, all queues are disabled. When a queue is enabled, the job at its head is scheduled if it fits. If the job does not fit, the queue (and its corresponding scheduler) becomes disabled and may only get enabled again at a subsequent job departure. Queues that schedule their last job and become empty are also disabled.

When a job departs from the system all or only some of the non-empty queues are enabled, depending on the scheduling rules. Empty queues are not enabled at departures. The enabled queues are then repeatedly visited in some order, starting in each round at most one job from each (enabled) queue, until no job can be scheduled anymore according to the policy rules. At this point all queues, both the empty and the non-empty ones, are disabled. When a job arrives at an empty queue, the queue becomes enabled and the job can be scheduled right away if it fits, unless there is a policy rule saying that the queue should stay disabled.

When there are multiple queues in the system, the scheduling discipline in each queue is always FCFS. For the policy with a single global queue we also consider FPPS in Section 4.2. In all the other occurrences of this policy FCFS should be assumed.

1. [GS] The system has one global scheduler with a global queue, for both single- and multi-component jobs. All jobs are submitted to the global queue. The global scheduler knows at any moment the number of idle processors in each cluster and based on this information chooses the clusters for each job.

2. [LS] Each cluster has its own local scheduler with a local queue. All queues receive both single- and multi-component jobs and each local scheduler has
global knowledge about the numbers of idle processors. However, single-component jobs are scheduled only on the local cluster. The multi-component jobs are co-allocated over the entire system. At each departure, all non-empty queues are enabled; depending on the enabling order we define four variations of LS.

[LS-OR] The queues are enabled in a fixed order, starting with the same queue.

[LS-RD] The queues are enabled in a fixed order, starting with a queue randomly chosen. All queues have the same probability to be enabled first.

[LS-RO] The queues are enabled in the order in which the processors in the corresponding clusters are set to be released by the departing job, which is assumed to be in the decreasing order of the job-component sizes (this is the same as the order in which the processors are allocated in the case of unordered requests). If the departing job has fewer components than the number of clusters, the queues local to the clusters not holding any component are enabled last.

[LS-DO] The queues are enabled in the same order in which they were disabled last.

All the remaining policies define both a global queue and local queues. Multi-component jobs go to the global queue and are scheduled by the global scheduler using co-allocation over the entire system. Single-component jobs are placed in one of the local queues and are scheduled by the local scheduler only on its corresponding cluster.

For the policies below, the order in which the local queues are enabled does not matter since the jobs in them are only started on the local clusters.

3. [GP] The local queues are enabled only when the global queue is empty, so the global queue has priority.

4. [LP] With this policy the local queues have priority: the global queue is only enabled when at least one local queue is empty. If no local queue is empty only the local queues are enabled and repeatedly visited; the global queue is enabled and added to the list of queues which are visited as soon as at least one of the local queues becomes empty. If one or more of the local queues are empty both the global queue and the (non-empty) local queues are enabled. Then, depending on the order in which this happens, we differentiate the following variations.

[LP-LF] First the local queues are enabled and then the global queue.

[LP-GF] The queues are enabled starting with the global queue.

[LP-RD] Either the global queue is enabled first or the local queues, with equal probability.
5. [EQ] When a job departs all non-empty queues are enabled and repeatedly visited in a pre-defined order. With this policy local and global queues have equal priority. We consider three different ways in which queues are enabled at departures.

- [EQ-LF] First the local queues are enabled and then the global queue
- [EQ-GF] First the global queue is enabled, followed by the local queues.
- [EQ-RD] Either first the local schedulers are enabled and then the global one, or the other way around, both choices occurring with equal probability.

6. [LQ] At each moment, either the non-empty local queues or the global queue are enabled, depending on the lengths of the queues. The global queue is enabled if it is longer than all the local queues, otherwise only the local queues are enabled. With this policy the states of the queues change also when the order of their lengths changes, and not only at departures, at arrivals at empty queues, or when jobs do not fit.

For comparison, we consider the single-cluster case where there are only single-component jobs and we use FCFS as scheduling policy ([SC]). This policy is similar to GS, but for a single-cluster system.

In the extreme case, GP can indefinitely delay the single-component jobs, and LP can do the same with the multi-component jobs. In practice, an aging mechanism has to be implemented in order to prevent this behaviour.

The LQ policy might seem to favour the local schedulers (the global scheduler is only permitted to schedule jobs when its queue is longer than all the others), but our results show that this is not the case. It takes into account the fact that each of the local schedulers accesses just one cluster, so they can be simultaneously enabled. To allow the local schedulers to work only when more of their queues are longer than the global queue would be much to the disadvantage of the local schedulers, especially if their queues are unbalanced.

When the local queues receive only single-component jobs, the local schedulers manage disjoint sets of resources (a local scheduler starts jobs on a single cluster) and there is no need for coordination among them. However, for systems with both a global scheduler and local ones, or when the local schedulers also deal with the multi-component jobs and may use more clusters, the access to the data structures used in the process of scheduling (numbers of idle processors, queue lengths) has to be mutually exclusive since we make the choice to keep those data consistent at all moments. The global scheduler always uses global information since it does co-allocation over the entire system. In our simulations we do not address these issues and do not take into account the scheduling overhead that results.
2.9 Related work

Over the last decade, clusters and distributed-memory multiprocessors consisting of hundreds or thousands of standard CPUs have become very popular. In addition, recent work in computational and data grids [4, 27] enables applications to access processors (and other resources) in different and possibly widely dispersed locations simultaneously—that is, to employ processor co-allocation [19]—to accomplish their goals, effectively creating single multicenter systems. Most of the research on processor scheduling in parallel computer systems has been dedicated to multiprocessors and single-cluster systems, but hardly any attention has been devoted to multicenter systems.

In this section we first discuss the research in co-allocation and grid computing, then we summarize some relevant papers concerning scheduling in parallel systems and discuss aspects of co-allocation in multicenter systems also occurring in NUMA multiprocessors. We also include results related to non-FCFS queuing disciplines such as backfilling, and to bin-packing algorithms. Finally, we briefly present research going on in the context of the DAS.

2.9.1 Co-allocation and grid computing

Not much work has been done related to co-allocating rigid jobs with space sharing in multicenter systems. In [43], a queuing model in which jobs require simultaneous access to multiple resources is studied. The interarrival and service-time distributions are only required to be stationary. Feasible job combinations are defined as the sets of jobs that can be in service simultaneously. A linear-programming problem based on an application of Little’s formula for these feasible job combinations is formulated for finding the maximal utilization, regardless of the scheduling policy employed.

In [21], co-allocation (called multi-site computing there) is studied, with as performance metric the (average weighted) response time. Jobs only specify a total number of processors, and are split up across the clusters. The slow wide-area communication is accounted for by a factor $\alpha$ by which the total execution times are multiplied (i.e., the extension factor). Co-allocation is compared to keeping jobs local and to only sharing load among the clusters, assuming that all jobs fit in a single cluster. One of the most important findings is that for $\alpha$ not exceeding 1.25, it pays off to use co-allocation.

In [22], several ways to enhance the performance of a scheduler that uses co-allocation are presented. The performance is measured in terms of average weighted completion time, and communication and migration overheads are considered in the model by extending the service time (there called completion time). It is shown that imposing a lower bound on the job size at which the scheduler is allowed to split a job and restricting the maximum number of components improves the performance of the scheduling scheme. The paper also evaluates an adaptive scheduler that compares the completion time of the single-site execution to which the waiting time for enough idle processors is added, to the completion
time of the multi-site execution to decide whether it pays off to split the job or not. It is concluded that such an adaptive scheduler which only splits a job when it leads to a shorter response time performs better than a non-adaptive scheduler.

In [45] co-allocation in computational grids is advocated. The authors combine time-shared workstations and remote space-shared supercomputers to run a work-queue application. It is shown that a scheduler which uses dynamic information to co-allocate jobs on workstations and immediately available supercomputer nodes can perform better than several traditional non-co-allocating schedulers. In [46], a performance comparison of two meta-schedulers is presented. It is shown that dedicating parts of subsystems to jobs that need co-allocation is not a good idea.

One approach to scheduling on the grid is to provide a global load distribution system (LDS) on top of the local job management system (JMS) at each site. Users submit jobs directly to their local JMS. The LDS monitors the load at each site and if some sites become heavily loaded while others are lightly loaded, it attempts to equalize the load across the grid system by moving jobs among the sites. Classical load balancing attempts to maximize system throughput by keeping all processors busy. In [39] this goal is extended to fully utilizing all resources at each site. The resource average load index (RA) used in previous work to detect load imbalance is not very precise for multi-resource servers since it does not express the load differences for each resource, but a global average of all resources at a server. In [39] a resource balance index (RB) is introduced to measure how balanced the job mix at a server is with respect to their different resource requirements. Using these indices the LDS attempts to move the system to a state where the load is balanced across the servers and the job mix at each server matches the resource capabilities provided by that server. These extensions to load balancing provide an increase of 5 – 15% to the system throughput. There is a persistent performance gap of 15% between a central-queue approach to workload distribution and the load balancing algorithms presented in [39], which may persist even if the load is perfectly balanced—in the latter case servers may execute only jobs from their local queues and even if there is a job in the system which would fit to the unused resources, it cannot be scheduled if it is in the queue of a different server.

In [51], scheduling single applications consisting of multiple components that may be spread across multiple sites is considered, with the goal of reduced completion time. The site resources and the network capabilities are characterized solely in terms of delivered performance to applications. Meta-applications with a statically known communication pattern are considered and cost functions are constructed for component computation times, inter-component communication times, the total communication time for components, and the startup overhead. The meta-application completion time $T_{CT}$ is then defined in terms of the component cost functions. The scheduling problem is to determine an assignment of schedulable components to site resources that minimizes $T_{CT}$. It was previously shown [50] that selecting the best single site for single-component parallel applications can be done efficiently. In [51], a scalable scheduling heuristic with performance within 10 – 20% of optimal on average is presented, in a simulation study of three classes of meta-applications. It is concluded that effective scheduling of
meta-applications is possible if sufficient cost information concerning application and system resources is available.

The problem of mapping a set of applications in a heterogeneous computing system, where application tasks require concurrent access to multiple resources of different types (the resource co-allocation problem), is considered in [8]. The applications are represented using two graphs: A Directed Acyclic Graph (DAG), and a Compatibility Graph (CG). The DAG describes the precedence constraints among tasks while the CG represents the resource sharing constraints. Tasks are unable to run concurrently if they have either precedence constraints or resource sharing constraints. The objective is to determine a matching of tasks to compute resources and schedule their executions satisfying all resource requirements so as to minimize the overall schedule length (makespan) for the set of applications. It is assumed that estimates of the computation times of each task on each machine are given, that each task needs concurrent access to a set of resources, and that the entire set of applications is known. After selecting a maximal independent set of tasks, they are allocated to compute resources using several heuristics. The simulations show that the number of machines and resources did not have a significant impact on the performance of the allocation heuristics and selection strategies. All the considered heuristics seemed to perform equally well, indicating that a simple heuristic would suffice. The results indicate the importance of considering the co-allocation requirements during mapping decisions, since the performance of the heuristics increases as the total number of tasks increases.

In [29], considering as the primary goal of building a metacomputer to optimize resource usage, it was attempted to bridge the gap between the analytically studied algorithms and those used in existing metacomputing environments; they developed a generic simulation environment based on a model reflecting the real world scenario as close as possible in which they measured the quality of different scheduling algorithms. It was argued that in a real environment the metacomputer scheduling algorithm can only access the available machines indirectly, by submitting (sub-)jobs to the local queuing systems, while the local users will continue submitting jobs directly, bypassing the metacomputer scheduler.

### 2.9.2 Scheduling in parallel systems

Scheduling parallel jobs in single-cluster systems has received much attention (see, e.g., [26]). In the most simple model, rigid jobs (which have predefined, fixed sizes) are scheduled according to the FCFS policy with space sharing, in which jobs run to completion on exclusively allocated processors. In order to improve the performance, techniques such as malleable jobs (which may vary in size over their lifetimes), gang scheduling (using time sharing across multiple processors), and different forms of backfilling (allowing certain jobs not at the head of the queue to start execution) have been devised. In [52] three techniques are analyzed—backfilling, gang-scheduling and migration—that could replace space sharing in large parallel systems in order to improve the performance. Combinations of the above mentioned techniques are evaluated and it is concluded that a strategy that
Combines all three techniques performs always better than the individual strategies.

In [15] the workload submitted to the NCSA Origin 2000 (O2K) system over two three-month periods is characterized. The paper studies the distributions of the job interarrival times and requested runtimes, the requested numbers of processors, the actual runtimes as a fraction of requested runtimes, the processor and memory utilizations, and the correlations among the different characteristics. It is shown that most requested runtimes are default values and that whether or not a default runtime is requested, over half the jobs have actual runtimes which are less than 20% of the requested values. Concerning the distribution of the requested numbers of processors it is concluded that there is a preference for small numbers and powers of two, which is in line with the results in [23].

In [20] the workload on a parallel system is characterized from an operating system perspective—the stream of jobs submitted to the system, their resource requirements and their behaviour—and the main challenges to generate a realistic workload are discussed. For the CTC SP2 workload considered in this paper it is observed that larger jobs tend to run longer and, similarly to other workloads, that small jobs and jobs with sizes powers of two occur more often.

In [24] measurements based on accounting traces of the 128-node iPSC/860 hypercube located at NASA Ames are presented. Jobs can be submitted to the system directly or through the NQS batch queuing facility. There were three types of jobs in the system: Unix commands, direct user jobs and NQS (batch) jobs. It was found that most of the user jobs (93.8%) where submitted directly. However, the 6.2% batch jobs accounted for 50.5% of the resource usage. During the workday the utilization approximately follows the submission rate, as the load is mainly composed of small jobs. During the night the utilization is significantly higher because fewer jobs are submitted, but they have a higher degree of parallelism and run longer on average. The average overall utilization obtained is 50.0%, lower than the 80% utilization measured for the Touchstone Delta, but higher than the 9% average utilization reported for workstations. There is a roughly uniform distribution of user jobs of different sizes, with the large ones consuming most of the resources. For a good utilization big jobs should be matched together since there are insufficient small jobs to fill the gaps. Both job runtimes and interarrival times have a high coefficient of variation, which suggests that a hyperexponential distribution may provide an appropriate model.

2.9.3 Common issues with NUMA architectures

Solutions to some of the problems occurring in multiclusters were inspired by solutions to similar problems in single clusters. An example is the necessity of splitting processors into pools in NUMA multiprocessors and the scheduling policies designed to manage the pools.

In [53], NUMA multiprocessors are split up into processor pools of equal sizes along architectural lines. The number of threads into which a job is split, and the number of pools—the ones with the lowest loads are chosen—across which it is spread—a parallel job incurring more overhead when it spans multiple pools—is
controlled with parameters. The main result is that using intermediate pool sizes and limiting the number of pools a job is allowed to span yields the lowest response times, as this entails the best locality.

The influence of splitting the processors into groups on the performance of the system was also studied in [12]. A technique for operating system schedulers called processor pool-based scheduling, designed to assign the processes of parallel applications in multiprogrammed, shared-memory NUMA multiprocessors, is presented and evaluated. It is assumed that a job starts as a single process, and that it may grow by starting additional processes. Different policies for the initial placement of the jobs and for the placement of its additional processes when it expands were studied. Since it was assumed that the number of processors required by each job is not known when the application starts, the best strategy for initial placement was found to be Worst Fit, because it leaves the largest room for the growth of jobs inside the pool. It was noted that when the number of processors required is known by the time of the arrival, the problem of choosing which processors to allocate is similar to a bin-packing problem with multiple bins. The importance of application parallelism in determining the pool size, and the influence of the architectural configuration were studied. The results showed that although application parallelism should be considered, the optimal pool size is a function of the system's architecture.

2.9.4 Non-FCFS queuing disciplines

An approach to improving the performance of scheduling rigid jobs by pure space sharing is to reorder the jobs in the queue, that is, to use non-FCFS policies. With Fit Processors First Served (FPFS), rather than leaving the processors idle when the job at the head of the queue does not fit, smaller jobs from the queue are allowed to overpass the jobs ahead of them. To avoid the starvation of big jobs, only a limited number of jobs are allowed to overpass a job that cannot be serviced and then processors are reserved and idled anyway. A more sophisticated approach, backfilling [40], requires users to estimate the runtime of their jobs. Using this information, only short jobs, expected to terminate in time, are allowed to overpass a waiting large job. Backfilling was developed for the IBM SP1 parallel supercomputer installed at Argonne National Lab as part of EASY (the Extensible Argonne Scheduling System). Users should provide accurate estimates, since a low estimation may lead to killing the job before it terminates and a high estimation may lead to a long wait time and CPU quota loss.

In [7] FCFS, FPFS, FPFS with job sorting (both decreasing and increasing) and backfilling were compared. In order to avoid starvation in FPFS and its variations, a time limit is used rather than the maximal number of times a job can be overtaken. It was concluded that FPFS and Fit Processors First Served (FPMPFS) give the best performance. Backfilling can improve performance as well, but it requires a-priori knowledge of the execution times of jobs, which [7] considers less practical.

A scheduler with backfilling has to support two conflicting goals: to move ahead
as many short jobs as possible, in order to improve utilization and responsiveness, and to avoid starvation for large jobs and to be able to predict when each job will run. Different versions of backfilling balance these objectives in different ways. The EASY backfilling algorithm only checks that jobs moved ahead in the queue do not delay the job at the head of the queue.

In [42] it is shown that this aggressive approach can lead to unbounded queuing delays for jobs in the queue, and prevents the system from making definite predictions as to when each job will run. However, in practice the job at the head of the queue only waits for currently running jobs, so if there is a limit on job runtimes, then the queuing delay for any job is bounded by the product of this limit and the job’s rank in the queue. In a conservative approach, short jobs are moved ahead and allowed to run only if they do not delay any job ahead of them in the queue. The authors ([42]) compare EASY and conservative backfilling for workloads measured on the IBM SP2 parallel supercomputer of Argonne National Lab. The performance metrics used are the average response time and the average bounded slowdown. For the workloads measured on SP2 systems the EASY algorithm provided better performance, so the added predictability of the conservative approach came at a cost. However, for workloads from other systems the algorithms had about the same performance. In such a case the conservative algorithm is preferable due to its improved predictability. The simulation results were found to depend on the workload and metric being used. It was also concluded that it is not so relevant how much backfilling is done, but which jobs are backfilled. Unlike previous studies on the influence of the workload when it was concluded that workloads can affect the quantitative results but not the qualitative results, this paper shows that the workloads dictate the results.

Checking the accuracy of the user estimates, [42] concludes that they are extremely unreliable. It can be noticed that users find the motivation to overestimate so that jobs will not be killed much stronger than the motivation to provide accurate estimates to enable the scheduler to perform better packing. In what the sensitivity to the accuracy of estimates is concerned, exaggerated estimates lead to better performance than tight estimates, since they leave more flexibility to the scheduler. This is in line with [34], where it was shown that systematically lengthening execution times effectively reduces slowdowns; techniques which combined sorting jobs by execution time and queue randomization were used to improve the performance.

In [47] an algorithm which improves the EASY backfilling algorithm reducing the average wait time by about 15% is presented. The algorithm supports both user selected and administrative priorities and guarantees a bounded wait time for all jobs.

In [32] the optimizations of sorting the job queue according to increasing numbers of processors requested, and of backfilling, are studied with simulations driven by traces derived from the logs of three supercomputer installations. On an SP2, for a sorted job queue, increasing the Maximum Allowable Skipping Count (MASC), a parameter with the same meaning as our MaxJumps, to a large value (simulations were presented with a MASC of 10, 100, 1000) yielded a considerable
decrease of the average turnaround time. However, on the Paragon, the sensitivity to the MASC was much smaller than on the SP2, so this sensitivity is very dependent on the workload.

2.9.5 Maximal utilization and bin-packing-related research

Whereas we approach the problem of the maximal utilization from a more theoretical perspective (see Chapter 3), experience with existing supercomputing installations [31] shows that FCFS results in a 40% - 60% utilization, that backfilling [40] gives an improvement of about 15%, and that reducing the maximal job size allowed increases the utilization. It has sometimes been suggested [7] to apply results for (online) bin packing [16] to scheduling in parallel systems. However, unless a system exhibits slotted behaviour, i.e., service times are deterministic and the jobs receiving service simultaneously start and finish at the same time, it is not obvious how to do this. In [17], such a slotted queue is studied for its stability, i.e., for when the maximal utilization is reached. The unit-size slots are filled with messages that have a geometric interarrival-time distribution and whose sizes are taken from some distribution on [0, 1], using the Next-Fit policy (which corresponds to FCFS). It turns out that when the message sizes are uniformly distributed on 1/N, 2/N, ..., (N-1)/N, the saturation point is at a utilization of (3/4)(1 + 1/(4N - 5)). A similar model is studied in [18], with type-i packets needing a fraction i/k of a single slot, i = 1, 2, ..., k, for some integer k > 0. It turns out that for First-Fit (which corresponds to FCFS with backfilling, when the arrival rates are symmetric (λ_i = λ_{k-i} for i = 1, 2, ..., k - 1), the stability condition is the best possible: \sum_i \lambda_i(i/k) < 1. Similarly, one might try to apply results for vector bin packing [14,33] to scheduling ordered jobs in multiclusters—how to do so for unordered jobs is however completely unclear.

In [38] multi-capacity aware bin-packing algorithms which use the additional capacity information when selecting items in the packing process are provided. Multi-capacity bin-packing is a generalization of the one-dimensional bin-packing problem in which the bin capacity and the item sizes are represented by multi-dimensional vectors. Past research focused on extending the single-capacity bin-packing to deal with the multiple capacities and they did not use the additional capacity information so they do not scale well with increased capacity counts. When an algorithm requires only that the item fits into the current bin, ignoring the component weights of the item and the current component capacities of the bin, a single capacity in a bin can fill up much sooner than the other capacities, resulting in a lower overall utilization. An improvement may be made selecting items to pack into a bin based on the relative weights of their capacities.

In [38] simulations of two offline algorithms for vector bin packing which search for items that will reduce the imbalance in the current bin are presented. In order to relate these algorithms to scheduling in multiclusters with deterministic service demands, the algorithms are also simulated for short item lists, with replacement of items before a new bin is started. This problem resembles scheduling ordered jobs without communication with deterministic service times. The heuristics used by
2.9. Related work

multi-capacity aware algorithms follows from the fact that at least one bin capacity is fuller than the others and if all the capacities are kept balanced, then more items are likely to fit in the bin. The multi-capacity aware algorithms provide improved performance over the previous algorithms. Their use in online job scheduling in the presence of multiple constraints is supported by the fact that they produce better packing from a short input list.

2.9.6 Research in the context of the DAS

Finally, let us mention some of the other research that is being performed in the context of the DAS. Whereas our work is at the operating systems level, other research on the DAS has concentrated on run-time systems [35, 36] and on applications [10, 44, 49].

In [36], a library is presented which optimizes the collective communication primitives of MPICH, a widely used version of MPI, in order to achieve fast communications in wide-area systems. Because collective communication algorithms are usually designed for LANs, they do not take into account the high latencies of wide-area links, which negatively influence their performance. The authors designed algorithms which are wide-area optimal in that an operation includes only one wide-area latency, and every data item is sent at most once across each wide-area link. They modified 14 collective operations of MPI and obtained substantial performance improvements over MPICH. As an example, in the case of the MPLBcast primitive, the completion time was reduced to 50% for 32 processors divided into 4 clusters and a message size of one byte. This was obtained by sending the message only once to each cluster over the wide-area links, and then broadcasting it inside each cluster on the fast local links.

In [10], several nontrivial algorithms on a multilevel communication structure (LAN clusters connected by a WAN, such as the DAS) were analyzed and several optimization techniques were used to improve their performance. The optimizations either reduced intercluster traffic or masked the effect of intercluster communications and caused a significant improvement. One of the optimized applications solves the Traveling Salesman Problem. It was improved by replacing the dynamic work distribution through a centralized queue with a static distribution over the clusters, each of them having its own local queue. For 32 processors divided into 4 clusters, the speedup was 24, compared to 15 for the unoptimized solution. The authors concluded that many medium-grained parallel applications can be optimized to run well on a multilevel, wide-area system. The same conclusion was reached in [44]. In [49], the performance of four parallel applications in wide-area systems is assessed by comparing the speedups of the original applications on a 64-cluster system with the speedups of versions of the applications optimized for wide-area execution on a 4 × 16 multicluster system.

In [48] a program aiming at the optimal integration of observed data in an oceanographic model describing the water transport phenomena in the area at the tip of South Africa is presented. Data assimilation techniques are used to study the dynamics of the system: all the information obtained from observations of
the system is combined with all the information on the dynamic evolution of the system. We ran the parallel implementation of this application on the DAS in different configurations with and without co-allocation, and measured the total execution times, which we used in simulations (see Chapter 7).
Chapter 3

The capacity loss in multicluster systems

In studies of scheduling in single clusters it has been shown that the achievable (maximal) utilization may be much less than 100%, even when enough load is offered to the system, because the jobs in execution may not leave enough processors idle for any eligible waiting job [31]. This problem may be aggravated in multicluster systems, where a job can be denied service even when there are enough idle processors in the system if the way the idle processors are spread among clusters does not suit the job requirements. In this chapter we study the maximal utilization when co-allocating jobs in multicluster systems, both with analytic means (we derive exact and approximate formulas when the service-time distribution is exponential), and with simulations with synthetic workloads and with workloads derived from the logs of actual systems.

The capacity loss and the utilization of high-performance computer systems have been major concerns [31]. From a practical point of view, a high utilization of real systems is often seen as a measure of successful (and profitable) operation. In [31] it is shown for real systems that FCFS yields 40 – 60% utilization, that backfilling increases it by 15 percentage points, and that reducing the maximum allowable job size further increases utilization. From a theoretical perspective, a very important problem in mathematical models of parallel job scheduling is to find the values of such parameters as the arrival rates, job sizes, and service times for which the system is stable. In this chapter we are interested in both points of view.

We first define the relation between the capacity loss and the utilization and discuss the reasons for capacity loss. Then we present an expression for the average maximal Multi-Programming Level (MPL) in multicluster systems with ordered requests, and deduce an approximation for the average maximal MPL in multiclusters with unordered requests and WF component placement, which we validate with simulations. We assess the capacity loss in multiclusters for ordered,
unordered and total requests depending on the job-size and service-time distribution, on the queuing discipline, and on the placement policy. Finally, we study the evolution of the capacity loss with the number of clusters in the system.

3.1 The maximal utilization and the capacity loss

In the model described in Chapter 2, processors may be idle while there are waiting jobs because the job at the head of the queue does not fit (FCFS) or because no job in the queue fits (FPFS). An important system-oriented performance metric of parallel systems is the maximal utilization. When \( \rho_m \) is the maximal utilization, that is, the average utilization of the system such that the system is unstable (saturated) at utilizations \( \rho \) with \( \rho > \rho_m \) and stable for \( \rho < \rho_m \), in general, we have \( \rho_m < 1 \).

In this chapter we do not explicitly consider the additional communication delays introduced by the relatively slow inter-cluster connections when jobs are spread across multiple clusters. As we define the (maximal) utilization based on the total time processors are allocated to jobs, these delays have hardly any effect on the utilization: If all jobs experience the same slowdown due to wide-area communication, then reducing the arrival rate by the same factor entails the same utilization (although, of course, a lower throughput). If we assume that the job service times in our model do include communication in a single cluster with a fast local network, we can get an indication of the performance of processor co-allocation when communication in multiclusters is taken into account in the following way. If the service times of all jobs are extended by (about) the same factor \( \alpha > 1 \) due to the relatively slow intercluster communication, multiplying our response-times curves (see for example Chapter 4, where we do not consider communication) by \( 1/\alpha \) and \( \alpha \) in the horizontal (utilization) and vertical (average response time) directions, respectively, yields the multicluster response times. In our model, one way of including wide-area communication is to extend the service time of jobs requiring co-allocation with an extension factor \( \alpha \) (see Section 2.4).

We define the capacity loss of the system (at the maximal utilization) as the average fraction of processors that are either idle or waiting for communication at the maximal utilization. In the absence of communication, so in this entire chapter, it can be expressed by the quantity \( l = 1 - \rho_m \). When there is communication inside an application, regardless whether it is local (intra-cluster) or global (inter-cluster) communication, the capacity loss increases. Of course, we expect that the extra component of the capacity loss introduced by the slower inter-cluster links is larger than the capacity loss due to the local communication.

We call gross utilization the utilization obtained for the system with extended service times. Since there is no preemption for communication, the processors are considered busy also when the jobs running on them are involved in inter-cluster communication. The net utilization takes into account only the processor usage for computations and fast local communication, i.e., the non-extended service times. Comparing the two utilizations in the same simulation settings we can estimate
the capacity loss due to the slow wide-area communication (see Chapter 7).

3.2 Reasons for capacity loss

In this section we discuss the reasons for capacity loss; at least four such reasons can be distinguished in multicluster systems with space sharing and rigid jobs.

First, it may be due to the structure of job requests. Ordered requests will obviously entail a higher capacity loss than unordered ones, which in turn yield a higher capacity loss than total requests. We will quantify this effect later in this chapter.

Second, it may be due to the distribution of the job-component sizes. With ordered requests, the capacity loss \( l \) can be very high, to the point that it approaches 1 for large cluster sizes and large numbers of clusters, as can be seen in the following somewhat pathological case. When in a multicluster with \( C \) clusters of size \( N \), all jobs have \( \lceil (N + 1)/2 \rceil \) tasks in cluster 1 and only one task in all the other clusters, \( l \) is close to \( (C - 0.5)/C \) for large \( N \), which is arbitrarily close to 1 for large \( C \).

Third, the queuing discipline employed may cause capacity loss. The job at the head of the queue may not fit, while some job further down the queue does fit, and so, a policy that deviates from the arrival order may have a lower capacity loss than FCFS.

A fourth reason for having \( \rho_m < 1 \) is that we are considering an on-line problem, taking scheduling decisions without knowledge of future job arrivals or service times. Such knowledge might be exploited by a policy that deviates from the arrival order.

3.3 Formulas for FCFS

In this section we first present an expression for the average maximal Multi-Programming Level (MPL), defined as the number of jobs simultaneously in service, in multicluster systems with the FCFS policy and with ordered requests, from which of course \( \rho_m \) can be derived. We also deduce an approximation for the average maximal MPL in multiclusters with FCFS, unordered requests, and WF component placement, which we validate with simulations in Section 3.5. In this section we assume that the service-time distribution is exponential.

3.3.1 Ordered requests

In this section we assume that requests are ordered. Let \( F \) be the (multidimensional) job-size distribution, which (allowing components of size zero) is defined on the set

\[
S_0 = \left( \prod_{i=1}^{C} \{0, 1, \ldots, N_i\} \right) \setminus \{(0, 0, \ldots, 0)\}.
\]
We have $F(\overline{n}) = P(\overline{\bar{n}} \leq \overline{n})$ for $\overline{n} \in S_0$, where $\leq$ denotes component-wise (in)equality. Let $f$ be the job-size density, so $f(\overline{n})$ is the probability of having a job of size $\overline{n} \in S_0$. Denoting by $G^{(i)}$ the $i$-th convolution of a distribution $G$ with itself, $F^{(i)}(\overline{N})$ and $F^{(i)}(\overline{N}) - F^{(i+1)}(\overline{N})$ are the probabilities that at least and exactly $i$ random jobs fit on the multicluster, respectively, where $\overline{N} = (N_1, N_2, \ldots, N_C)$. When the job-component sizes are mutually independent, we have $F^{(i)}(\overline{N}) = \prod_j F^{(i)}(N_j)$ for $i = 1, 2, \ldots$, with $F_j$ the distribution of the $j$-th components of jobs.

In our treatment of multiclusters with ordered requests below we follow [13]. There, a queuing model of a multiprocessor with $P$ processors and $B$ memory blocks is studied. The scheduling policy is First-Come-First-Loaded, in which a job is allowed from the head of the queue into the multiprogramming set when its memory requirements, taken from some discrete distribution $F$ on $[1, B]$, can be satisfied. When the number of jobs does not exceed $P$, every job gets a processor to itself; otherwise processor sharing is employed. The service-time distribution (on a complete processor) is exponential. When $P \geq B$, and so every job has a processor of its own, this model coincides with our single-cluster model with memory blocks assuming the role of processors. Under the assumption that there is always a sufficiently long queue, the Markov chain $V$ with state space $(z_1, \ldots, z_B)$, where the $z_i$'s are the memory sizes of the oldest $B$ jobs in the system, and the MPL, both immediately after a departure and the entailing job loadings, are studied. It turns out that the behaviour of $V$ is as if FIFO is used, and, by solving the balance equations, that the associated probabilities are as if the $z_i$ are independently drawn from $F$. In addition, the time-average maximal MPL is derived in terms of convolutions of $F$.

In our multicluster model, we also consider the sequence of the oldest jobs in the system such that it includes at least all jobs in service. Let $B$ be some upper bound of the number of jobs that can be simultaneously in service ($\sum_i N_i$ will certainly do). Let $\overline{Z} = (z_1, z_2, \ldots, z_B)$ be the processor state vector, which is the (left-to-right ordered) sequence of the sizes of the oldest jobs in the system. Some first part of $\overline{Z}$ describes the jobs in service, and the remainder the jobs at the head of the queue. When a job leaves, the new processor state vector is obtained by omitting the corresponding element from the current vector, shifting the rest one step to the left, and adding a new element at the end. Let $V$ be the set of processor state vectors.

Because the service-time distribution is exponential, for $v, w \in V$, the transition of the system from state $v$ to state $w$ only depends on $v$: each of the jobs in service has equal probability of completing first, and the job at the head of the queue to be added to the state is random. So in fact, $V$ is a Markov chain. The result of [13] explained above can be extended in a straightforward way to our situation—the important element is that the distribution $F$ simply determines which sets of jobs can constitute the multiprogramming set, but the underlying structure of a single or of multiple resources does not matter. So also now, the stationary probability
distribution \( \pi \) on \( V \) satisfies
\[
\pi(\overline{Z}) = \prod_{i=1}^{B} f(\overline{z}_i),
\]
which means that the distribution of the oldest \( B \) jobs in the system is as if they are independently drawn from \( F \). So, because the average length of the intervals with \( i \) jobs in service is inversely proportional to \( i \) due to the exponential service, we find for the average maximal MPL \( M \):
\[
M = \frac{\sum_{i=1}^{B} (F^{(i)}(\overline{N}) - F^{(i+1)}(\overline{N})) \cdot (1/i) \cdot i}{\sum_{i=1}^{B} (F^{(i)}(\overline{N}) - F^{(i+1)}(\overline{N})) \cdot (1/i)},
\]
which can be written as
\[
M = \frac{1}{1 - \sum_{i=2}^{B} \frac{F^{(i)}(\overline{N})}{i(i-1)}}. \tag{3.2}
\]
For single clusters this expression coincides with the formula in [13], p. 468. Denoting by \( s \) be the average total job size, the maximal utilization is given by
\[
\rho_m = \frac{M \cdot s}{\sum_i N_i}. \tag{3.3}
\]
There is one other case of interest in which the maximal utilization can be easily derived, namely when the service time is deterministic and the system has a slotted behaviour, which means that all jobs being served simultaneously start and finish at exactly the same time. Of course, if the latter happens once, it will happen forever. This behaviour can come about when there are two possible job sizes that exclude each other, that is, the sum of their sizes in some dimension \( i \) exceeds \( N_i \). Then there is a positive probability that two consecutive jobs have these sizes, so the second (and perhaps some other jobs) can only start when the first completes. From then onwards, in each slot the system is filled with random jobs until the next one does not fit. So the distribution of the sizes of the jobs in service corresponds to the general job-size distribution, and we find for the maximum time-average MPL \( M \):
\[
M = \sum_{i=1}^{B} \left( F^{(i)}(\overline{N}) - F^{(i+1)}(\overline{N}) \right) \cdot i. \tag{3.4}
\]
Of course, Eq. (3.3) also holds here.

3.3.2 Unordered requests

We now derive an approximation to the maximal utilization in multiclusters with unordered requests in case all clusters are of equal size \( N \). For job placement, WF
Chapter 3. The capacity loss in multic和平ter systems

is used. The job-size distribution $F$ is now defined on

$$
S_U = \{(s_1, s_2, \ldots, s_C) \mid 1 \leq s_1 \leq N, s_{i+1} \leq s_i, i = 1, 2, \ldots, C - 1\},
$$

that is, job requests are represented by a vector with non-increasing components. Compared to the case of ordered requests, the case of unordered requests presents some difficulty for two reasons. First, given a set of unordered jobs, it is not possible to say whether they simultaneously fit on a multic和平ter, because that depends also on the order of arrival of the jobs. For instance, if $C = 2$ and $N = 5$, then if three jobs of sizes $(2, 1), (3, 1), (2, 1)$ arrive in this order, they can all be accommodated, while if they arrive in the order $(2, 1), (2, 1), (3, 1)$, only the first two jobs can run simultaneously. Second, a departure can leave the system in a state that cannot occur when it is simply filled with jobs from the empty state. If with the first sequence of arrivals above the job of size $(3, 1)$ leaves, both $(2, 1)$-jobs will have their largest component in cluster 1. Our result below deals with the first problem—by running over all possible arrival sequences—but not with the second, which is why it is an approximation.

We now define the $i$-fold WF-convolution $F \ast G$ of two distributions $F, G$ on $S_U$ in the following way. Let for any $C$-vector $\bar{s} = (s_1, s_2, \ldots, s_C)$ the reversed vector $\text{rev}(\bar{s})$ be defined as $\text{rev}(\bar{s}) = (s_C, s_{C-1}, \ldots, s_1)$, and the ordered vector $\text{ord}(\bar{s})$ as the vector with the elements of $\bar{s}$ permuted such that they form a non-increasing sequence. Now if $f, g$ are the densities of $F$ and $G$, respectively, $F \ast G$ has density $h$ defined by:

$$
h(\bar{s}) = \sum_{\bar{t}, \bar{u} \in S_U, \bar{s} = \text{ord}(\bar{t} + \text{rev}(\bar{u}))} f(\bar{t}) \cdot g(\bar{u}).
$$

Then, putting $F^{[2]} = F \ast F$, we define inductively $F^{[i]} = F^{[i-1]} \ast F$ for $i = 3, 4, \ldots$. What this amounts to is that $F^{[i]}$ is the distribution of the non-increasingly ordered numbers of processors in the clusters of the multic和平ter that are in use when $i$ unordered requests are put on an initially empty system with WF. Now our approximation of the maximum average MPL $M$ is (cf. Eq. (3.2)):

$$
M = \frac{1}{1 - \sum_{i=2}^{B} F^{[i]}(N)/(i(i-1))},
$$

where again $B$ is some upper bound to the number of jobs that can simultaneously be served, from which an approximation of the maximal utilization (or the capacity loss) can be derived with Eq. (3.3).

Extending the results in this section to unequal cluster sizes is cumbersome, because then adding an unordered request to a multic和平ter depends on the numbers of idle rather than used processors. So one would have to replace the WF-convolution by a sort of convolution that depends on the cluster sizes.
3.4 Simulation methods

Unfortunately, when the job-component sizes are dependent, the computation of Eq. (3.2) for more than four clusters is very time-consuming; the same always holds for Eq. (3.5), whether job-component sizes are independent or not. In addition, we would like to validate our approximation of the maximal utilization for unordered jobs with the FCFS policy. Finally, in some cases, such as for non-exponential service times and for the FF placement policy of unordered jobs, our formulas and approximations do not apply. For these three reasons, we resort to (two types of) simulations.

In the first type of simulations, for FCFS with synthetic service-time distributions, we simulate the complete queuing model with Poisson arrivals, and we take the utilization in these simulations that yields an average response time of at least 1,500 time units (the average service time is 1 time unit). When simulating a queuing model close to its maximal utilization, it is very difficult to find out whether the simulation is still in its transient phase, and programming difficulties like running out of space for data structures such as job queues may arise. However, we have validated this approach in turn by running simulations for single clusters and for multiclusters with ordered requests, for which we have the exact solution based on Eq. (3.3) (see Section 3.5).

In the second type of simulations, when the scheduling policy is FCFS, we simulate the system in heavy traffic in that we suppose that the queue is always long enough when a job departs that we can schedule jobs until the next one does not fit. As we then only generate new jobs at departure times, and only one more than fits on the system, we do not encounter the difficulties mentioned above when simulating a queuing system close to its maximal utilization. Of course, we cannot use this method with FPFS because then we want to search the queue for jobs that fit. However, in an adaptation for FPFS of this method, we do model Poisson arrivals, and simulate the system for 100,000 arrivals. We then take the arrival rate (and so the utilization) for which the queue length does not hit either the value of 0 or of 1000 for the largest number of arrivals.

3.5 The accuracy of the approximation and the simulations

In this section we assess the accuracy of the approximation of Eqs. (3.5) and (3.3) of the capacity loss for unordered jobs, and of the simulation methods presented in Section 3.4.

We consider that all clusters are of size 32, all job-component sizes are non-zero and mutually independent, and the service-time distribution is exponential. The scheduling policy is FCFS and for unordered jobs the placement policy is WF.

In Tables 3.1 and 3.2 we show both exact (i.e., derived from Eq. (3.2)), approximation (i.e., derived from Eq. (3.5)), and simulation results for a single cluster and for multicluster systems with ordered, unordered, and total requests for dif-
Table 3.1: The capacity loss in a single cluster \((C = 1)\) of size 32 and in a multicluster with 4 clusters \((C = 4)\) of size 32 for ordered, unordered, and total requests, with job-component-size distribution \(U[n_1, n_2]\).

<table>
<thead>
<tr>
<th>job comp. size distr.</th>
<th>capacity loss</th>
<th>(C = 1)</th>
<th>(C = 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(n_1)</td>
<td>(n_2)</td>
<td>exact</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0.032</td>
<td>0.033</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0.043</td>
<td>0.044</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>0.139</td>
<td>0.139</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>0.169</td>
<td>0.169</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.051</td>
<td>0.052</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>0.145</td>
<td>0.145</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>0.174</td>
<td>0.175</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>0.149</td>
<td>0.150</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>0.177</td>
<td>0.178</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>0.094</td>
<td>0.095</td>
</tr>
</tbody>
</table>

Different distributions of the job-component sizes. The simulation results for a single cluster in both tables, and those for unordered jobs in the multicluster in Table 3.1, have been obtained with simulations of type one; the remainder have been obtained with simulations of the second type. The exact and simulation results for the single cluster and for the multicluster with ordered jobs agree extremely well; the match of the approximation and simulations for unordered jobs is quite reasonable.

As an aside, the results for multiclusters show the reduction in capacity loss when going from ordered to unordered to total requests.

### 3.6 Results

In this section we present results for the capacity loss as it depends on many parameters. The results in this section for ordered and total requests when the service time is exponential and the scheduling policy is FCFS, are obtained with the formulas of Section 3.3; all remaining results have been obtained with simulations of the second type as described in Section 3.4. Unless otherwise specified, all our results are for a multicluster system consisting of 4 clusters of 32 processors each (or, when considering total requests, for a single cluster of size 128), for independent non-zero job-component sizes, for exponential service times, for the FCFS policy, and for the WF placement policy for unordered jobs.
Table 3.2: The capacity loss in a single cluster \((C = 1)\) of size 32 and in a multicluster with 2 clusters \((C = 2)\) of size 32 for ordered, unordered, and total requests, with job-component-size distribution \(D(q)\) on \([1, 32]\).

<table>
<thead>
<tr>
<th>job comp. size distr.</th>
<th>capacity loss (C = 1)</th>
<th>(C = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ordered</td>
<td>simul.</td>
</tr>
<tr>
<td>(q)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.95</td>
<td>0.293</td>
<td>0.295</td>
</tr>
<tr>
<td>0.90</td>
<td>0.249</td>
<td>0.251</td>
</tr>
<tr>
<td>0.85</td>
<td>0.188</td>
<td>0.188</td>
</tr>
<tr>
<td>0.80</td>
<td>0.134</td>
<td>0.135</td>
</tr>
<tr>
<td>0.75</td>
<td>0.097</td>
<td>0.097</td>
</tr>
<tr>
<td>0.70</td>
<td>0.073</td>
<td>0.074</td>
</tr>
<tr>
<td>0.65</td>
<td>0.057</td>
<td>0.058</td>
</tr>
<tr>
<td>0.60</td>
<td>0.046</td>
<td>0.047</td>
</tr>
<tr>
<td>0.55</td>
<td>0.038</td>
<td>0.039</td>
</tr>
<tr>
<td>0.50</td>
<td>0.032</td>
<td>0.032</td>
</tr>
</tbody>
</table>

3.6.1 The influence of the job-size and service-time distribution

We consider the five distributions for the sizes of the job components described in Table 3.3. Note that the first three and the latter two have almost identical means.

Table 3.3: The means and the coefficients of variation of the distributions of the job-component sizes

<table>
<thead>
<tr>
<th>distribution</th>
<th>mean</th>
<th>cv</th>
</tr>
</thead>
<tbody>
<tr>
<td>(U[1, 7])</td>
<td>4.000</td>
<td>0.500</td>
</tr>
<tr>
<td>(D(0.9)) on ([1, 8])</td>
<td>3.996</td>
<td>0.569</td>
</tr>
<tr>
<td>(D(0.768)) on ([1, 32])</td>
<td>3.996</td>
<td>0.829</td>
</tr>
<tr>
<td>(U[1, 14])</td>
<td>7.500</td>
<td>0.537</td>
</tr>
<tr>
<td>(D(0.894)) on ([1, 32])</td>
<td>7.476</td>
<td>0.884</td>
</tr>
</tbody>
</table>

As expected, in Figure 3.1 we find that the capacity loss decreases when going from ordered to unordered (except for \(D(0.894)\), roughly speaking, the capacity loss is cut in half), and from unordered to total requests. In addition, when the mean, the coefficient of variation, or the maximum of the job-component-size distribution is larger (these are not independent), the performance is poorer.

In Figure 3.2 we compare the performance for unordered jobs with independent and dependent job-component sizes; in the latter case we first generate the total
job size as the sum of four copies of the component-size distribution, and then spread a job across the system as explained in Section 2.5. We find that when the maximum of the basic distribution used is low ($U[1,7]$, $D(0.9)$, and $U[1,14]$) or when large values hardly ever occur ($D(0.768)$), dependent component sizes lead to poorer performance. The reason for this is that in this case the job components are much larger than when they are independent. When the maximum of the basic distribution is higher and larger values are more common, as is the case for $D(0.894)$ on [1,32], this behaviour is reversed. In addition, for dependent component sizes, only the mean, and not the distribution, matters.

We varied the service-time distribution from deterministic through exponential to hyperexponential with a cv of 3 up to 10 (for the capacity loss, the mean is immaterial). We found that this distribution does not have a large impact on the capacity loss. For ordered (unordered) jobs it went from 0.239 (0.103) for deterministic to 0.262 (0.117) for hyperexponential with a cv of 10, which means about a 10% (14%) increase. The job-component-size distribution has an impact on the widths of holes in a schedule (i.e., the number of idle processors), while the service-time distribution has consequences for the lengths of these holes (i.e., the time processors are left idle). Apparently, the former is much more important than the latter.
3.6. Results

![Bar chart showing capacity loss for different job-component-size distributions.](chart)

Figure 3.2: The capacity loss depending on the job-component-size distribution for unordered jobs with (in)dependent job-component sizes

3.6.2 The influence of the queuing discipline and of the placement policy

In this section we first assess to what extent FPFS increases the maximal utilization compared to FCFS (see Figure 3.3). We consider unordered jobs and the two service-time distributions from Section 3.6.1 for which there is the most room for improvement. We find that a small value for MaxJumps (say 5) does not improve performance by much, but that a larger value (here 50) yields a decrease of capacity loss over FCFS of about 10 percentage points, an improvement similar to that found in practice from backfilling [31].

The results of comparing the placement policies FF and WF for unordered jobs are presented in Figure 3.4. The difference in performance is not very large; apparently, whereas the explicit aim of WF is to keep balanced loads in the clusters, FF achieves the same goal.

3.6.3 Using data from the logs

In Figure 3.5 we show the capacity loss for unordered and total requests when using both the sizes and the service times as they appear in the logs of the CTC (using only the jobs of sizes at most equal to 128) and the DAS1 (see Section 2.5). From the CTC log, we use the pairs consisting of the sizes and service times of the jobs as they appear in the log, while for the DAS we sample the distributions of the sizes and the service times as they appear in the log independently. For the total jobs, we simply use a single cluster of size 128 and the job sizes from the logs; for unordered requests, we split up jobs (with dependent component sizes)
Figure 3.3: The capacity loss when using FCFS or FPFS with different values for MaxJumps for unordered jobs.

Figure 3.4: The capacity loss depending on the placement policy for job components for unordered jobs.

as explained in Section 2.5. The capacity losses for the two logs are similar, even though the statistics of the logs are different (see Tables 2.1 and 2.4).

3.7 Asymptotic behaviour

In this section we study the behaviour of the capacity loss against the numbers of clusters used. We consider clusters of equal fixed sizes, and fixed job-component-
size distributions. Then we can either keep the number of (non-zero) job components equal to the number of clusters (which is somewhat unrealistic), or equal to some fixed value. In the latter case, the job-component sizes are not independent anymore. In the case of ordered jobs and equal numbers of clusters and job components, of course the average maximal MPL $M$ (and so $\rho_m$) is decreasing in the number of clusters, because the probability that large components of different jobs meet in the same cluster increases. Indeed, if $n_i$ is the largest possible size of the $i$-th job component, the asymptotic value $M_\infty$ of $M$ equals $\min_i [N_i/n_i]$, and the asymptotic capacity loss is $l_\infty = 1 - M_\infty s/\sum_i N_i$.

In Fig. 3.6, we plot the capacity loss as a function of the number of clusters (of equal size 32) for the job-component-size distributions from Table 3.3, with the number of job components equal to the number of clusters, or equal to 4. In the latter case, the non-zero components of an ordered request are chosen at random with equal probabilities. For Fig. 3.6.(a), we use Eqs. (3.2) and (3.3); the values of $l_\infty$ can of course easily be computed from the values in Table 3.3. For Fig. 3.6.(b)-(d), we use simulations of the second type. In what is probably in practice the most important of all of these cases, unordered jobs with a fixed number of components, the capacity loss decreases with the number of clusters.

Of course, the capacity loss in Fig. 3.6.(a) increases for an increasing number of clusters, because all jobs have a non-zero component in each cluster, and so the probability that a size conflict occurs in at least one cluster, and so a job cannot be accommodated, increases. In Fig. 3.6.(c), this effect is even more pronounced, which can be explained by the fact that here, the load in the clusters is more imbalanced due to the job components of size zero. Compared to Fig. 3.6.(a), the capacity loss in Fig. 3.6.(b) is lower because the load will be more balanced and any job component can go to any cluster, but when the largest component of a job fits nowhere, the job cannot be placed. Finally, in Fig. 3.6.(d) the capacity loss decreases, as there is an increasing freedom for placing jobs.
Figure 3.6: The capacity loss as a function of the number of clusters, which are all of size 32, with the number of job components equal to the number of clusters (a,b) or equal to 4 (c,d)

3.8 Conclusions

We have studied the capacity loss for co-allocation in multiclient systems with both analytic means and with simulations for a wide range of parameters. We have derived an exact (for ordered requests) and an approximate (for unordered requests) formula for the maximal utilization when the service times are exponential, and we have shown with simulations that the latter is quite accurate. Our conclusions are as follows.

- The extra capacity loss due to co-allocation can be quite large in some of the cases discussed in this chapter.

- However, for unordered jobs with independent job-component sizes co-allocation may be an attractive option, even when all jobs fit in a single cluster.

- The job-size distribution is more critical for the capacity loss than the service-time distribution.
Chapter 4

Co-allocation with GS without communication

In this chapter we study through simulations the performance of processor co-allocation in multiclusters depending on such parameters as the job-component-size and service-time distributions, the structure of job requests, the queuing discipline, the numbers and sizes of clusters, and the fractions of jobs with different numbers of components. The aim is to find out how much each of these parameters impact the performance and for which ranges of parameters co-allocation is worthwhile.

In order to isolate the aspects mentioned above, we ignore for now communication among the tasks of jobs. In this chapter, we consider the GS policy and various multicluster configurations with a total size of 128 nodes. Since communication is not considered, total requests have identical performance as flexible requests.

In the simulations in this chapter the mean service time is set to 1.0 time units, and the graphs showing the results depict the average response time in the same time unit. In this chapter, and in the rest of the thesis, the legends of the graphs representing the average response time as a function of the utilization always describe the curves in the same order, starting from high utilizations (from right to left).

4.1 The component-size and service-time distributions

In this section, we assess the performance of a system consisting of 4 clusters with 32 processors each, for flexible, unordered (with WF) and ordered requests depending on the distribution of the job-component sizes, on the service-time distribution, and on the request type.
4.1.1 The influence of the component-size distribution

We consider the five distributions for the sizes of the job components described in Table 3.3. Notice that $U[1,7]$, $D(0.9)$ and $D(0.768)$ have approximately the same mean 4, and that $U[1,14]$ and $D(0.894)$ have mean 7.5. In this section, the service-time distribution is exponential with mean 1.

![Graphs showing the influence of component-size distribution for ordered, unordered, and flexible requests](image)

Figure 4.1: The influence of the component-size distribution for ordered (top), unordered (middle) and flexible (bottom) requests, for exponential service times.

From Figure 4.1 we can conclude that independent of the component-size distribution, flexible requests generate the best performance and ordered requests
4.1. The component-size and service-time distributions

![Graphs showing average response time vs utilization for different distributions.](image)

**Figure 4.2:** The influence of the service-time distribution for ordered (top), unordered (middle) and flexible (bottom) requests, for job-component distribution $D(0.768)$ on $[1, 32]$.

**Figure 4.3:** The influence of the service-time distribution for ordered (top), unordered (middle) and flexible (bottom) requests, for job-component distribution $U[1, 14]$.

...the worst. For both the uniform distributions and the $D$ distributions, a larger mean implies a worse performance. For the same mean, uniform distributions for the job components yield better performance than the $D$ distributions, because they provide better packing due to a smaller maximum value for the components and a smaller coefficient of variation. For the most rigid type of requests, ordered requests, the mean of the distribution matters less than its range and variance, since although its mean is 7.5, $U[1, 14]$ has a better performance than $D(0.768)$...
which has mean 4. Similarly, even though \( D(0.9) \) on \([1, 8]\) and \( D(0.768) \) on \([1, 32]\) have both mean 4, the first distribution displays better performance, especially for ordered and unordered requests where not only the total size of the job request matters, but also the sizes of the components and the way they fit in the system.

**4.1.2 The influence of the service-time distribution**

We now consider three distributions for the service time, deterministic, exponential and hyperexponential (with coefficient of variation 3), all with mean 1. Figures 4.2 and 4.3 show the performance of the system for these distributions, for flexible, unordered and ordered requests, for component-size distributions \( D(0.768) \) on \([1, 32]\) and \( U[1, 14] \), respectively. The results are similar for the other distributions in Table 3.3. For all types of requests and for both component-size distributions, the best performance is obtained for deterministic service time and the worst for hyperexponential service time. The graphs indicate that for ordered requests the performance is better when components are from the uniform distribution, for all service time distributions, which is consistent with what we observed above for exponential service times.

In the rest of the chapter we will only use exponential distributions for the service times (with mean 1).

**4.1.3 The influence of the request type**

Comparing the performance for the different request types in Figures 4.1, 4.2, and 4.3 we can conclude that in all the cases considered the flexibility of the request has an important influence both on the maximal utilization and on the average response time. The most rigid request type defined in our model—ordered requests—provides the worst performance. The more flexible unordered requests show significant improvement and the flexible requests constantly yield the best results.

**4.2 The influence of the queuing discipline**

In this section we assess the performance of co-allocation in multiclusters for the two queuing disciplines introduced in Section 2.7. Before we can compare FCFS and FPFS, we have to choose the value of MaxJumps—the maximum number of times a job can be jumped over in FPFS. When MaxJumps is too large, the performance of individual jobs is negatively influenced, to the point that when MaxJumps \(\rightarrow \infty\), which amounts to FPFS without counters, starvation may occur under heavy loads. At the other extreme, with MaxJumps equal to 0 we return to FCFS. We have performed simulations for MaxJumps values in the range \(1 - 50\).

Figure 4.4 depicts the sensitivity of the response time to the value of MaxJumps for ordered, unordered and total requests, and for job-component-size distributions \( U[1, 7] \) and \( D(0.894) \). The figure shows that FPFS enhances the performance of the system for all request types and for both distributions. The improvement
is larger for the $D(0.894)$ distribution which generates larger jobs and has lower performance. The improvement is also higher for the more rigid request types. Ordered requests, which have the lowest performance for FCFS, benefit the most from FPFS. For total requests, which have very good performance, the gain from using FPFS is comparatively small, especially with job-component sizes from $U[1, 7]$. In addition, the influence of the structure and sizes of the requests observed for FCFS is maintained with FPFS.

Figure 4.4: The average response time as a function of the utilization for FCFS and for FPFS with several MaxJumps values. The results are for ordered (top), unordered (middle) and total (bottom) requests, and for job-component sizes from $U[1, 7]$ (left) and $D(0.894)$ (right).
4.3 Equal versus different cluster sizes

A question we would like to answer is whether a system with equal clusters provides a better performance than a system where nodes are unevenly divided among the clusters, for equal total system sizes. For this we look at two systems with 4 clusters, with total numbers of processors of 128: a system with equal clusters of size 32, and one with uneven clusters of sizes 64, 32, 16, 16. (FF uses the clusters in this order; see Section 4.4 for the reverse order.) Simulations were performed for ordered and unordered requests, and for total requests in a single 64-processor cluster to see whether only using the largest cluster in the uneven system can give better performance than co-allocation. For unordered requests both FF and WF were simulated; since the results were similar, only those for FF are shown.

![Graph](image)

Figure 4.5: The performance of ordered and unordered requests in a multicluster with equal clusters (32,32,32,32), and in a multicluster with unequal clusters (64,32,16,16).

Although the effect is less pronounced for unordered requests, for which the scheduler can deal well with unequal cluster sizes, Figure 4.5 shows that for both ordered and unordered requests, the system with equal clusters provides a much better performance. This can be explained by the fact that both types of requests need nodes from all the 4 clusters (the number of components is equal to the number of clusters) and once the smallest clusters get filled to the extent that they cannot accommodate components of a new job, the idle processors from the big clusters cannot be used either. Since the small clusters behave as a bottleneck for the utilization, we can expect that the performance deteriorates with the increase in the difference between the clusters' capacities. For ordered requests in uneven clusters the performance is even worse than for total requests in a system with half the capacity (64 processors, the utilization on the horizontal axis is then still relative to the whole 128-processor system), which means that we can ignore the smaller clusters and only use the largest.
4.4 Comparing systems with different numbers of clusters

For the same total size of its clusters, we can expect that the performance of a system varies with the number of clusters. We study this variation for systems with 128 nodes with at least 4 clusters, in the presence of unordered requests consisting of 4 components, which the scheduler can now put on any 4 different clusters. We compare the results for FF and WF.

Figure 4.6 compares two system configurations with clusters of equal sizes: a system with 4 clusters of 32 processors, and a system with 8 clusters of 16 nodes. Although in the second case the clusters are much smaller, the results show that the performance is similar at low utilizations and even slightly better at moderate and high utilizations, which suggests that the performance can be better when the number of clusters is larger than the number of request components. This can be explained by the fact that FF, which we use here, first fills up the first cluster with the largest job component. Then, when such a component does not fit in the first cluster anymore, FF in the 4-cluster case is still forced to use the first cluster for a job component. However, in the 8-cluster case FF may skip the first cluster(s) completely for a whole job once in a while, but will always try to put small job components on them. Apparently, this determines a better packing when the number of clusters is larger, even if those clusters are smaller.

![Figure 4.6: The influence of the number and sizes of clusters for equal clusters and unordered requests, with FF component placement.](image)

Figure 4.7 shows the performance of systems with different numbers of clusters of uneven sizes; the difference between the graphs is the order in which clusters are used by the scheduler. Taking as a reference the case (32, 32, 32, 32) in the two graphs, we notice that performance is better when the scheduler starts with the largest clusters. For a system where the user can express preference for certain clusters, this can be translated into the fact that preferring the larger clusters is a good option and improves both the response time and the maximal utilization. From Figure 4.7 (and Figure 4.6), we conclude that the performance is only good
in systems with clusters of equal size and, for both orders of using the clusters with FF, in systems with clusters of unequal sizes whose numbers of clusters are considerably larger than the number of job components.

Figure 4.7: The influence of the number and sizes of clusters for unordered requests; component placement uses FF and starts with the large clusters (top), and with the small clusters (bottom).

Figures 4.8 and 4.9 show the results for the same configurations, when WF is used for placing the jobs. We conclude that systems with large equal clusters give much better performance. For systems with unequal clusters, we should still prefer to have many (although relatively small) clusters, since the performance will be better than for few, large clusters.

Comparing WF with FF (large-small order), we notice that WF is much better for systems with few, equal, large clusters. Using WF instead of FF slightly improves the performance also for systems with unequal clusters, when the number of clusters is relatively small compared to the number of job components. However, comparing Figure 4.8 with Figure 4.6 we can see that for systems with small, equal clusters and with numbers of clusters considerably larger than the number of job components, WF has significantly worse performance than FF. This is also valid for systems with unequal clusters, as Figure 4.10 shows. Summing up, FF should
4.5. The fractions of jobs with different numbers of components

Figure 4.8: The influence of the number and sizes of clusters for equal clusters and unordered requests, with WF component placement.

Figure 4.9: The influence of the number and sizes of clusters for unordered requests with WF component placement.

be chosen as placement policy when dealing with systems with many but relatively small clusters, while WF is to be preferred for multiclusters with few, large clusters.

4.5 The fractions of jobs with different numbers of components

In this section, we study workloads consisting of jobs with different numbers of components. We consider a multicluster system with 4 clusters of 32 processors each and evaluate its performance for jobs having between 1 and 4 components. Mixes of jobs are also considered. This corresponds to a system in which co-allocation is combined with jobs requiring processors from a single-cluster, but both single cluster and multicluster jobs are submitted to the same central scheduler. We model unordered requests and two ways of generating components: either each component is obtained from the distribution $D(0.9)$, which means that jobs
Figure 4.10: Comparison between FF and WF for placing unordered requests, in systems with many, unequal clusters—(32, 32, 16, 16, 8, 8, 8, 8).

with fewer components are in general smaller, or jobs have the same total request size and their components are obtained as a sum of values from $D(0.9)$. In this second case a job with fewer components has in general larger components.

4.5.1 Jobs with components from the same distribution

Figure 4.11 compares the performance of the system for jobs with 1, 2, 3, and 4 components, and mixes of such jobs in different percentages. Each component is obtained from the distribution $D(0.9)$ and jobs are placed using FF. The performance is better for jobs with fewer components since they are smaller and also allow more ways of placement. For mixes the performance is somewhere between those of the participants in the mix, depending on the percentage of each of them. Figure 4.13 shows the performance for jobs with 1 and 4 components and for mixes of the two when WF is used. Compared to FF, WF brings visible improvements for jobs with 4 components and mixes with high percentages of such jobs, but has little effect on jobs with 1 component or on the mix with 75% jobs with a single component.

4.5.2 Jobs with the same total request size

In this section, each job has the same total request size. Figure 4.12 compares jobs with 1, 2, and 4 components and mixes of such jobs for the FF policy. For jobs with 2 components, each component is the sum of two values from $D(0.9)$. The worst performance is displayed by jobs with one component since that component is large compared with the clusters’ sizes. Significantly better performance is obtained for jobs with four components, although they have the disadvantage of placing a component in each cluster. For this reason, the best performance is shown by systems hosting jobs with two components since these components are still relatively small and need only two of the clusters (more possibilities for the placement). For mixes of 1 and 4 the performance of the system is even worse than
Figure 4.11: Different number of components, the same job-component size; FF placement.

Figure 4.12: Different number of components, the same total request size; FF placement.

Figure 4.13: Different number of components, the same job-component size; WF placement.

Figure 4.14: Different number of components, the same total request size; WF placement.

for jobs with 1 component since the requirements of jobs with 1 and 4 components are opposed to each other: Jobs with 4 components need to place one component in each cluster, while jobs with 1 component need enough room in one cluster to
place their large component and tend to fill in individual clusters. When the mix also contains jobs with 2 components the performance of the system improves, since they are easier to fit. Figure 4.14 shows the results for systems with jobs consisting of 1 component, 4 components, and mixes of 1 and 4 when WF is used. WF improves the performance for jobs with 4 components compared to FF, but leaves almost unchanged the results for jobs with 1 component and for mixes where such jobs are predominant.

4.6 Conclusions

In this chapter, we have studied the performance of processor co-allocation in multicluster systems for rigid multi-component jobs of different request types that impose various restrictions on their placement. From the results we derive the following conclusions:

- **Restrictive job request types yield worse performance.** Independent of the component-size and service-time distributions, when a large fraction of the workload consists of (un)ordered jobs (specifying the numbers of processors needed in distinct clusters, or, worse yet, in each cluster), the response time is higher.

- **Multiclusters with equal clusters provide better performance.** For the same number of clusters and total number of processors, ordered and unordered request types fit better in systems with equal clusters.

- **In general, a system with few, large clusters should be preferred.** For the same total number of processors and unordered requests, we can obtain a better performance if the system is only divided into a small number of clusters, especially if jobs are placed using WF.

- **For systems with few, large clusters WF should be used. For systems with many clusters, FF is better.** If a system has a small number of clusters compared to the number of job components, WF policy should be chosen for placing the jobs on clusters. On the contrary, for systems with many clusters FF gives better performance.

- **For ordered and unordered jobs, either the cluster sizes should be equal, or the number of clusters should be large relative to the number of job components.**

- **Large jobs or jobs with large components have low performance.** The performance is better when the jobs submitted to the system are small. For jobs with large (even if few) components compared to the cluster sizes the performance is worse than for jobs with more but smaller components. Spreading large single-component jobs over more clusters (co-allocation) can be better.

- **Jobs with many components yield poor performance.** Even when the cost of communication is not considered, the performance is lower for jobs with numbers of components close to the number of clusters in the system.
Chapter 5

Co-allocation with GS and communication

In multiclusters we may expect that the wide-area communication degrades the performance of co-allocation. The strength of this effect depends on the capacity of the wide-area links, on the amount of communication performed by jobs, and on the job-component sizes. In this chapter we study the influence of the slow wide-area communication on the performance of co-allocation depending on the job request type, the job-placement policy for flexible requests, and the mix of jobs of different request types. The communication is expressed by the communication-speed ratio, using the first method of including communication, as described in Section 2.4. In all the simulations in this chapter we consider the GS policy and FCFS. As a metric we use the mean response time as a function of the gross utilization.

5.1 Parameter settings

The simulations in this chapter are for systems with 4 clusters of 8 processors each, and for single clusters with 32 processors. The job-component-size distributions we use are $D(0.9)$ on $[1, 8]$ and $U[1, 4]$, with means 4 and 2.5, respectively.

For the structure of jobs we consider a parallel application as described in Section 2.4, with the tasks making up a job performing the same algorithm and alternating computation and communication steps. The time costs of the messages sent between tasks depend on whether the sender and the receiver are in the same cluster or not. The slowest task determines the job duration.

Due to communication, the service times of tasks are extended by the times needed for communication—the sum of the durations of all send operations to the other tasks. We only simulate a single iteration and assume the duration of the computation steps to be exponential with mean 1, and the time needed for a single local send operation to be 0.001.
With our parameter settings, and for a communication-speed ratio of 100, a task of an (un)ordered job with component-size distribution $D(0.9)$ on $[1,8]$ has an average total service time of about 2.2. This value is obtained by adding 1 for computation to 12 - 0.1 for the wide-area communication, where 12 is the average number of inter-cluster messages for a task. This average total service time is more than twice the mean time of a computation step. In [44], the results of running on the DAS a version of the Barnes-Hut algorithm for the $N$-body simulation problem are presented. The solution is optimized for wide-area systems and uses personalized all-to-all communication. It turns out that when the ratios of wide-area latency to local latency, and of local bandwidth to wide-area bandwidth are 100, the speedup of the algorithm in a multiclient with 4 clusters of size 8 relative to a single cluster of size 32 is about 0.45, which matches the extension by the factor of 2.2 of the service time derived from our parameters.

5.2 The influence of the communication-speed ratio

In this section we study the effect of the slow inter-cluster communication on the various request types. Figure 5.1 shows that for ordered requests, for unordered requests with WF component placement, and for flexible requests with the LB-S placement policy, the increase of the communication-speed ratio increases the response time. The performance is affected more for ordered and unordered requests than for flexible requests, which suggests that in those cases the average amount of inter-cluster communication per job is higher. The variation of the communication-speed ratio does not affect the results for total requests since in single clusters there is only intra-cluster communication, and those curves can be used as a reference in the graphs. Going from a communication-speed ratio of 10 to 100, the response times at a utilization of 0.4 are increased by a factor of about 1.8 for (un)ordered and 1.6 for flexible requests.

Table 5.2 contains the maximal utilizations and the coefficients of variation of the service time for the four request types and the communication-speed ratios from the graphs in Figure 5.1. For the three request types with co-allocation a higher communication speed ratio yields a higher maximal (gross) utilization and a lower coefficient of variation of the service time. The maximal utilization also increases with the flexibility of the request. Comparing the data for flexible and total requests we observe that flexible requests have a higher maximal utilization, which is due to the lower coefficient of variation.

5.3 The influence of the placement policies for flexible requests

In this section we discuss the effect on the performance of the three placement policies for flexible requests: CF, LB-S and LB-A, as defined in Section 2.6. The
Figure 5.1: The average response time as a function of the utilization for the four request types, for communication-speed ratios of 10 (top), 50 (middle), and 100 (bottom). The job-component-size distribution is $D(0.9)$.

interesting aspects to the placement policies for flexible requests when assessing their performance are the total number of wide-area messages and the maximum number of such messages per task, and the mixes of job request types.

To reduce inter-cluster communication, placement policies for flexible requests should not spread jobs unnecessarily over the clusters. Therefore, both CF and
Figure 5.2: The maximal (gross) utilization and the coefficient of variation of the service time for the four request types, depending on the communication-speed ratio.

<table>
<thead>
<tr>
<th>Communication speed ratio</th>
<th>Ordered</th>
<th></th>
<th></th>
<th>Unordered</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. util.</td>
<td>Cv of serv. time</td>
<td>Max. util.</td>
<td>Cv of serv. time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.534</td>
<td>0.872</td>
<td></td>
<td>0.560</td>
<td>0.872</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.548</td>
<td>0.597</td>
<td></td>
<td>0.575</td>
<td>0.597</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.555</td>
<td>0.448</td>
<td></td>
<td>0.582</td>
<td>0.448</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication speed ratio</th>
<th>Flexible (LB-S)</th>
<th></th>
<th></th>
<th>Total</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. util.</td>
<td>Cv of serv. time</td>
<td>Max. util.</td>
<td>Cv of serv. time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.699</td>
<td>0.861</td>
<td></td>
<td>0.692</td>
<td>0.980</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.714</td>
<td>0.570</td>
<td></td>
<td>0.692</td>
<td>0.980</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.720</td>
<td>0.420</td>
<td></td>
<td>0.692</td>
<td>0.980</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: The average response time as a function of the utilization for LB-S and CF for flexible requests, and for communication-speed ratios of 50 (top) and 100 (bottom). The job-component-size distribution is $D(0.9)$.

LB-S choose the minimal number of clusters, but whereas CF aims to reduce the total number of inter-cluster messages, LB-S may have a larger number of inter-cluster messages while balancing loads. Since LB-A potentially spreads jobs over more clusters, in smaller components, it may cause a still larger number of such messages.

When there are only flexible requests in the system, completely filling the clusters as in CF does not affect the performance and it seems that there is no need for using LB-S or LB-A. These latter placement policies become interesting when there are also (un)ordered requests, because then keeping an equilibrium between the loads of the clusters becomes essential. Ordered and unordered requests specify the number of processors required in distinct clusters and if any of the needed clusters is full, the job cannot be scheduled, even when the total number of idle processors is large enough. This yields an extra capacity loss that can be avoided
5.3. The influence of the placement policies for flexible requests

Figure 5.4: The average response time as a function of the utilization for LB-S and LB-A for flexible requests, and for a communication-speed ratio of 50. The job-component-size distribution is $D(0.9)$.

by an even loading of the clusters. We will investigate this further in Section 5.4.

Although there seems to be no direct benefit from balancing the load for flexible requests, Figure 5.3 shows that even for a system with only flexible requests LB-S proves to be better than CF. The explanation resides in the fact that although CF has fewer inter-cluster messages per job, the communication time is not evenly spread across the tasks, and on average, CF causes jobs to spend more time on communication.

As an example, consider the consecutive arrival of two flexible requests for 18 and 12 processors at an empty system with 4 clusters of 8 processors. For CF, LB-S and LB-A, the tuples of the numbers of tasks on the clusters for the first and the second job will be $(8, 8, 2, 0)$, $(6, 6, 6, 0)$, $(5, 5, 4, 4)$, and $(0, 0, 4, 8)$, $(0, 2, 2, 8)$, and $(3, 3, 3, 3)$, respectively. The total numbers of inter-cluster messages in a communication step for the first and the second job are 192, 216, 242, and 64, 72, 108, while the longest tasks in the two jobs have 16, 12, 14, and 8, 10, 9 such messages. This shows that CF does not entail more time spent in communication than LB-S for every job, although in Figure 5.3 it apparently on average does. The same can be noticed for LB-A.

When CF and LB-S are faced with a system with the same numbers of idle processors in the clusters, then for placing the same flexible request CF entails a higher maximal number of inter-cluster messages across all tasks in the job, but a lower total number of such messages than LB-S.

In a similar comparison of LB-S and LB-A, we find that at low utilizations, LB-A gives a slightly higher response time due to more communication, but at higher utilizations this difference attenuates, being within the confidence intervals (see Figure 5.4). The two policies have the same maximal utilization and at high utilizations the difference in response times cannot be noticed on the curves. For smaller jobs and higher speed ratios we expect this difference in performance to grow, implying that when only flexible jobs are admitted in the system, there is
no reason to choose LB-A.

5.4 Mixes of job request types

Real systems have to schedule multiple request types, and then the overall performance of the system is determined by the way the scheduler deals with all these types. The different request types in the system influence each other and results obtained for a system with just one type of jobs may not be valid also for a system where several types of requests are submitted. Therefore we study multicusters with both flexible and ordered requests, and compare the performance of using CF, LB-S and LB-A for flexible requests. We consider different ratios between flexible and ordered jobs, and we fix the communication-speed ratio at 50.

The influence of the ratio between ordered and flexible requests

In this section we assess the effect on performance of the ratio between flexible and ordered requests in the job mix. For flexible requests we compare the CF and the LB-S placement policies. Figure 5.5 shows that for both CF and LB-S the average response time grows with the increase of the percentage of ordered jobs: Going from a low to a high percentage of ordered requests, the performance is ever more decided by this type of requests.

![Figure 5.5: The average response time as a function of the utilization for a system with mixed requests, for different percentages of flexible and ordered requests, with LB-S (top) and CF (bottom). The job-component-size distribution is D(0.9).](image)

Load-balancing-smallest versus cluster-filling

We can expect that LB-S, and even more so CF, tend to obstruct ordered requests because they use the minimum number of clusters for flexible requests, and tend to fill up clusters. As Figure 5.5 indicates, for up to 50% ordered requests the system behaves slightly better with LB-S than with CF. This is due mostly to
5.4. Mixes of job request types

Figure 5.6: The average response time as a function of the utilization for LB-S and CF, and for a system with mixed requests and equal percentages of flexible and ordered requests. The job-component-size distribution is $U[1, 4]$.

the smaller amount of time spent in communication by the flexible requests and to their easier placement. Because jobs are large (the average job size is 16), on average there are at most two jobs simultaneously in service and there is little benefit for ordered jobs from load balancing. Assuming that there is an ordered job on the system and a flexible job is placed next, for an incoming ordered job it does not matter whether CF of LB-S was employed since the total size of the ordered job is probably too large to fit. Also when due to the total size only one job fits, the placement is irrelevant.

To show that the job sizes play a role here, we depict in Figure 5.6 the response time for a job mix of 50% ordered requests and 50% flexible requests, but now with job components obtained from $U[1, 4]$. Now the average job size is 10, as opposed to 16 previously. In this case the difference in performance is larger, so LB-S brings more benefit when jobs are smaller.

Load-balancing-smallest versus load-balancing-all

We concluded in Section 5.3 that with only flexible requests, there is no reason to use LB-A. However, we expect that in a system where flexible requests are mixed with ordered requests, LB-A can bring relevant performance improvements over LB-S.

In this section we compare the performance of a system with mixed requests for the two types of LB and conclude that because the job sizes are large compared to the clusters’ sizes, the benefit from using LB-A is insignificant. Even when 50% of the jobs are flexible (see Figure 5.7), the performance is dominated by the ordered requests, and LB-S and LB-A display similar results. Figure 5.8, where job-component sizes are obtained from $U[1, 4]$, shows a different situation. When jobs are small, the advantage of using LB-A is obvious, to the extent that LB-A with only 30% percent flexible requests performs better than LB-S with 50%
Figure 5.7: The average response time as a function of the utilization for LB-S and LB-A, and for a system with mixed requests and equal percentages of flexible and ordered requests. The job-component-size distribution is $D(0.9)$.

Figure 5.8: The average response time as a function of the utilization for LB-S and LB-A, and for a system with mixed requests and various ratios between ordered and flexible requests. The job-component-size distribution is $U[1, 4]$.

percent flexible requests.

5.5 Conclusions

In this chapter we have studied the performance of processor co-allocation in multicluster systems, in terms of response time as a function of the utilization, for multi-component jobs performing both intra-cluster and inter-cluster communication and having various restrictions on their placement. Our conclusions are as follows:

- **Slow inter-cluster communication deteriorates the performance for all request types.** A high communication-speed ratio between inter-cluster and local communication has a severe impact on performance. However, for ratios up to 50, and workloads with a large fraction of flexible requests co-allocation yields acceptable performance. The deterioration is rather small when compared to co-allocation for the same request type in the absence of communication.

- **The scheduling decisions for a request type should take into account the simultaneous presence in the system of other request types.** When the workload consists of flexible and ordered requests, the former should be placed in a balanced way across the clusters to keep room for the latter, even though this brings them an increased amount of inter-cluster communication.
Chapter 6

Scheduling policies for co-allocation

Two important issues related to co-allocation in multicluster systems are the structure of jobs and the way they are spread across the clusters, and the number of schedulers in the system and how they interfere. In multiclusters that allow co-allocation, jobs may be submitted with numbers of components that vary between one and the number of clusters. Whereas single-component jobs may be handled by local cluster schedulers, for the multi-component ones either a separate global scheduler has to be introduced—which may then also deal with single-component jobs—or the local schedulers have to be made aware of the whole multicluster system. Of course, using co-allocation does not mean that all jobs have to be split up into components and spread over the clusters, small jobs can still go to a single cluster. In general, there is in the system a mix of jobs with different numbers of job components.

In this chapter we assess the response times of jobs in multicluster systems for three queuing structures with different priority orders in which the associated schedulers are allowed to schedule jobs, and for several job-stream compositions defined by the fractions of jobs with different numbers of components. In Section 2.8 we have defined six scheduling policies, some of which have several variations: one with only a global queue (GS), one with only local queues (LS), and four with both (GP, LP, EQ and LQ). In the latter case, single-component jobs go to the local queues and multi-component jobs to the global queue. We first evaluate the different variations of the LS, LP and EQ policies and then we compare the six policies choosing the best variation for the three policies with multiple variations. We only consider unordered requests with WF placement.

We study the performance in terms of the average response time, displayed for each case at one or two fixed utilizations. When comparing the charts in this chapter to each other, one must be aware that the results are at different utilizations. In each chart, the utilization is chosen high enough so that at least
one of the policies is close to its maximum utilization or already saturated. When there are both local and global queues in the system we can expect that the performance of the policies differs between the global and local queues. This is why for GP, LP, EQ and LQ we depict besides the total average response time, also the average response times for the local queues and the global queue.

6.1 Parameter settings

All the simulations in this chapter are for a system with 4 clusters of 32 processors each, and the job-component sizes are generated from $D(0.9)$ on the interval [1, 8]. Jobs can have between 1 and 4 components, and the percentages of jobs with the different numbers of components influence the performance of the system. We express the job-stream composition as a tuple of four values representing, in this order, the percentages of 1-, 2-, 3- and 4-component jobs submitted to the system. For the distribution of service times we use an exponential distribution with mean 1.

We consider nine job-stream compositions depending on the percentages of jobs with different numbers of components. For all these compositions we consider first that local queues are balanced in the sense that they receive the same percentages of jobs submitted locally. For job-stream composition $(80, 0, 0, 20)$ we add a case with unbalanced local queues, where one queue receives 40% and the other three 20% of the jobs submitted locally. All the ten cases which result are presented in Table 6.1.

Table 6.1: The ten cases considered, depending on the job-stream compositions and the way jobs are spread among the local queues.

<table>
<thead>
<tr>
<th>% 1-comp.</th>
<th>% 2-comp.</th>
<th>% 3-comp.</th>
<th>% 4-comp.</th>
<th>balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>yes</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>yes</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>yes</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>0</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>80</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>yes</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>yes</td>
</tr>
<tr>
<td>80</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>no</td>
</tr>
</tbody>
</table>
6.2 Policies with multiple variations

In Section 2.8, depending on the order in which the local queues are enabled at job departures, we defined four variations of LS and three variations of LP and EQ. For these policies we may expect that this enabling order at job departures has a significant influence on the performance. In this section we assess and compare for each of the three policies the performance of the different variations.

6.2.1 The LS policy

In this section we compare the four variations of LS for nine of the ten cases introduced in Table 6.1. Because single-component jobs are restricted to their corresponding clusters, when there are only single-component jobs in the system the enabling order of the local queues does not influence the results, and all the variations of LS amount to the same. For this reason, the case with 100% 1-component jobs is not considered in this section. In Figure 6.1 the local queues are balanced, while in Figure 6.2 they are not.

In all the cases with balanced queues the LS-DO variation of the policy, where queues are enabled in the order in which they were disabled, displays the best performance. It treats all jobs and queues fairly, keeping in balance the numbers of jobs run from each queue, and since the queues receive equal percentages of the job stream, their lengths stay similar. When after a departure the queues are repeatedly visited, those which get disabled earlier managed to schedule fewer or at most the same number of jobs as the queues disabled later, so enabling them first is a good way to keep the queues balanced.

The worst performance in Figure 6.1a – g is shown by LS-OR, where the queues are enabled each time in the same order. This approach tends to favour the queues visited earlier (especially the first queue), while allowing the queues visited last to grow. When scheduling decisions are taken, there are up to four jobs (when no queue is empty) from which to choose one that fits in the system. The OR variation unbalances the lengths of the queues and, emptying the queues visited first, it also reduces the set of jobs among which the system searches for one that fits.

LS-RD displays very good performance in Figure 6.1a – h due to the fact that at each departure it randomly chooses the queue to be enabled first, which maintains in general balanced queue lengths. However, it does not take into account the job stream and it can delay large jobs that are hard to fit, decreasing the performance. Unlike LS-RD, LS-DO remembers the jobs that did not fit and enables the corresponding queues in the same order helping this way the large jobs to run: a queue repeatedly disabled because of the same (large) job advances in the visiting order if the other queues manage sooner to schedule their jobs, finally being enabled first at each departure until its job fits. This yields a better queue length balance when the DO variation is used, and as a consequence a better performance.

In the cases in Figure 6.1a – f, the performance of LS-RO is close to that of LS-RD. The RO variation of the policy is good for local jobs: When the load of
Figure 6.1: Response times for the four variations of LS, several job-stream compositions, and balanced local queues.

the system is high and a job releases its processors, there is a good chance that the clusters hosting larger components will have larger numbers of idle processors. Enabling first the queues corresponding to those clusters gives the single-component
jobs in the queues a better chance to run, before a multi-component job takes the processors away. Scheduling first the local jobs, restricted to their cluster, yields better performance since for the global jobs the components can be shuffled. This variation of the LS policy looks at the load of the system and not at the lengths of the queues. It assumes that keeping the load of the system balanced keeps the queues balanced as well.

When local jobs do not represent the majority, LS-RO displays good performance. However, for a high percentage of local jobs, as Figure 6.1g, h show, LS-RO is a bad choice. The explanation resides in the way the local and the global jobs interact there: At high loads, a queue that gets disabled with a multi-component job at its head while all the other queues have local jobs that fit has a very small chance to have its job scheduled before another multi-component job from another queue is started on the system, or some of the other queues become empty. When the local jobs scheduled from the other queues end, those queues will be visited first, leaving little room for the multi-component job.

Comparing the variations of LS for job-stream compositions with at least 80% local jobs, we find that DO balances the queue lengths (and also the load of the system, since all queues receive the same percentages of jobs with different numbers of components) adapting to the workload, RD balances the load but does not adapt to the jobs in the system, OR keeps the queue lengths unbalanced due to the way queues are visited, and RO causes the worst unbalance for the queues, taking the worst decisions by avoiding exactly the queues which have jobs that do not fit. LS-OR improves for a very low percentage of global jobs (see Figure 6.1h) because when there are just single-component jobs, it does not matter in which order the queues are visited.

![Graph showing response times for different LS policy variations](image)

Figure 6.2: Response times for the four variations of LS, job-stream composition (80, 0, 0, 20), and unbalanced local queues.

Figure 6.2 shows the average response time for the variations of the LS policy when 80% of jobs are single-component and the queues are unbalanced—one queue receives 40% of the jobs arriving to the system. OR displays the best performance because it gives priority exactly to the queue receiving more jobs, visiting it first after each job departure. DO, RD and RO have higher response times because
they treat all queues the same, ignoring the fact that one of them receives twice as many jobs and letting that queue grow. RO has the worst performance due to the high percentage of single-component jobs.

### 6.2.2 The LP policy

In this section we compare the three variations of the LP policy defined. Since with LP the local queues only get single-component jobs, which are restricted to the local clusters, the relative order in which they are enabled does not matter.

Only seven of the cases considered are relevant here: For 100% single-component jobs there are only local queues, while for 100% multi-component jobs there is only the global queue. In Figure 6.3 the local queues are balanced, while in Figure 6.4 the unbalanced case is assessed.

![Graphs showing response times for different variations of LP policy.](image)

**Figure 6.3:** Response times for the three variations of LP, several job-stream compositions, and balanced local queues.
6.2. Policies with multiple variations

All charts in Figure 6.3 display the best total performance, and the best performance for the global queue, for LP-GF, where at each job departure the global queue is enabled first. Looking at the local queues, we notice that enabling the global queue first rather than last deteriorates very little their performance; in most cases there is a very small increase in response time for the local queues compared to LP-LF. On the other hand, enabling first the local queues with LP-LF causes a large increase in response time for the global queue compared to LP-GF.

For LP-GF, the performance of the local queues gets worse compared to LP-LF when the global jobs have fewer components, since they fit better on the system and leave less room for the local jobs (see job-stream composition (50, 50, 0, 0)). When there is a high percentage of global jobs and they have many components, enabling the global queue first does not bother much the local jobs and has a good effect on the global jobs.

We can conclude that LP-GF is the best choice for the job-stream compositions in Figure 6.3, while having a higher total average response time and a very bad performance for the global jobs makes LP-LF the worst. In all the cases from Figure 6.3, LP-RD has a total performance and a performance for the global queue worse than those of LP-GF and better than those of LP-LF. This is due to the fact that LP-RD randomly chooses at each departure whether to enable first the global queue or the local queues, treating both types of queues equally.

![Graph](image)

Figure 6.4: Response times for the three variations of LP, job-stream composition (80, 0, 0, 20) and unbalanced local queues.

When the local queues are unbalanced (see Figure 6.4), LP-LF has the best total performance because always enabling the local queues first allows the queue with a higher load to fit its jobs without being bothered by the multi-component jobs from the global queue. No other choice can improve the situation for the most loaded queue as long as local jobs are restricted to the local cluster. On the negative side, LP-LF has a bad performance for the global queue. Here, LP-GF provides a high total average and a very high average response time for the local queues. LP-RD has a slightly worse total performance than LP-LF and a higher response time for the local queues, but a much lower response time for the global
queue. If we are interested in having a good total performance or a low response time for the local jobs, LP-LF is the best option in this case, but if we also want a low response time for the global queue, LP-RD should be chosen since its total performance and average response time for the local queues are not much worse than those of LP-LF, and it gives a much lower response time for the global jobs.

Figure 6.5: Response times for the three variations of EQ, several job-stream compositions, and balanced local queues.

6.2.3 The EQ policy
Like with the LP policy, for the EQ policy we defined three different variations; also in this case local queues only get single-component jobs and the order in which these queues are enabled does not matter. Again, only seven of the cases are relevant since for 100% single-component jobs there are only local queues, while when all jobs are multi-component there is only the global queue. In the
charts in Figure 6.5 the local queues are balanced, while in Figure 6.6 they are not.

Similar to the LP policy, all the charts in Figure 6.5 display the best total performance for the GF variation of the EQ policy; also the average response time for the global queue is the smallest for EQ-GF. In most cases, enabling first the global queue deteriorates very little the response time of the local queues compared to EQ-LF, while for the global queue enabling first the local queues with EQ-LF, causes a large increase in response time.

For EQ-GF, the average response time of the local queues gets worse compared to EQ-LF when the global jobs have fewer components, since they fit better on the system and leave less room for the local jobs (see composition (50, 50, 0, 0)). Compared to LP, for EQ this deterioration is significantly larger. While even the GF variation of LP gives priority to local jobs, EQ-GF does not. Here, when the global jobs have few components the response time is lower for the global queue than for the local queues, which can be explained by the fact that the local jobs are restricted to their clusters and global jobs can be scheduled on any clusters where they fit. For a high percentage of global jobs with many components, enabling the global queue first does not bother much the local jobs and has a very good effect on the global jobs. In such cases EQ-GF is the best option, while EQ-LF which has a higher total average response time, and a very bad performance for the global jobs is the worst.

Like with LP, in all these cases EQ-RD has a total performance worse than EQ-GF and better than EQ-LF, and the average response times for the global and the local queues have values situated between those displayed by the GF and LF variations. For job-stream composition (50, 50, 0, 0), EQ-RD can be the most appropriate choice because its total performance is only slightly worse than that of EQ-GF, it is almost as good as EQ-LF for the local queues, and much better for the global queue.

Similarly to LP, for unbalanced local queues EQ-LF has the best performance. On the other hand, EQ-LF has a rather bad performance for the global queue. Here, EQ-GF provides a much worse total performance and a very high average response time for the local queues. EQ-RD has a slightly worse total performance than EQ-LF and slightly higher response time for the global queue, but a much lower response time for the global queue. If we look for a good total performance or a low response time for the local jobs EQ-LF is the best option for this case, but if we also want a low response time for the global queue, EQ-RD should be chosen.

6.3 Performance comparison of the policies

In this section we compare the average response time for the six policies and all the ten cases in Table 6.1. Section 6.3.1 discusses the cases with both single- and multi-component jobs and balanced local queues, Section 6.3.2 the cases with only single-component or multi-component jobs, and Section 6.3.3 the case with
unbalanced local queues. When a policy has multiple variations, the one with the best total performance is depicted. In Section 6.3.1 (Figures 6.7—6.12) these are LS-DO, LP-GF and EQ-GF, while in Section 6.3.3 (Figure 6.16) LS-OR, LP-LF and EQ-LF are represented. In Section 6.3.2 (Figures 6.13—6.15) LS-DO was employed.

### 6.3.1 Dealing with both single- and multi-component jobs

In this section we compare the policies for the six job-stream compositions of Table 6.1 containing both single- and multi-component jobs and for balanced local queues. In all the figures in this section we depict the response times at two utilization values. The first utilization value allows us to compare the response times of all the policies, but is too low to differentiate among the policies with better performance. At the second value the system is already saturated for some of the policies (indicated by SAT; in such a case the height of the bar has no relevance).

Figure 6.7 compares the policies for a job stream containing 1-, 2-, 3- and 4-component jobs in equal fractions. The best performance is obtained for LS, where all jobs go to the local schedulers and all four schedulers are allowed to spread the multi-component jobs over the entire system. At any moment, the system tries to schedule up to four jobs (when no queue is empty), one from each of the four local queues, and the FCFS policy is transformed this way into a form of backfilling with a window of size 4. This explains why LS is better than the other policies. A disadvantage for LS compared to GS is that LS can place 1-component jobs only on the cluster where they were submitted, while the GS can choose from the four clusters one where the job fits. However, in the case in Figure 6.7, only 25% of jobs have one component, so their negative influence on the performance of LS is small.

GP, LP, EQ, and LQ try to schedule up to 5 jobs at a time, but since 75% of the jobs in the system are multi-component and they all go to the global queue,
6.3. Performance comparison of the policies

Figure 6.7: Response times for the scheduling policies for job-stream composition (25, 25, 25, 25).

and only the rest of 25% is distributed among the local queues, their performance is worse than that of LS.

GP displays the worst performance; it gives priority to the global scheduler and only allows the local schedulers to run jobs when the global queue is empty. Even if the job at the head of the global queue does not fit, the policy does not allow jobs from the local queues to run and this deteriorates the performance. The average response time for the global queue is the best from all the policies, but the average response time for the local queues is much worse than for the other policies. Since most of the jobs are multi-component, the global queue is the longest in most of the cases when a scheduling decision has to be taken and LQ behaves similarly to GP, its performance being the second worst. For the utilization value in the second chart, the system is saturated for both GS and LQ.

LP and EQ also run mostly jobs from the global queue, but they do not delay the jobs from the local queues when the job at the top of the global queue does not fit and this improves their performance. LP has a slightly better total average, and although it favours the local queues enabling the global queue only when at least one local queue is empty, it also has a better average response time for the global queue than EQ. Since there are few local jobs in the system, LP favouring them does not delay the multi-component jobs, on the contrary, imposing an order among queues and not randomly mixing jobs from the local and global queues allows jobs to fit better.

Figure 6.8: Response times for the scheduling policies for job-stream composition (50, 0, 0, 50).
Figures 6.8, 6.9 and 6.10 show that for GP the performance improves with the decrease of the percentages of jobs with 3 and 4 components: The local, global and total average response times for GP are smaller in Figure 6.10. Since jobs with more components cause a higher capacity loss, it is a bad choice not to allow the local schedulers to try to fit jobs from their own queues when the job at the head of the global queue does not fit. Waiting for enough idle processors in multiple clusters for that job results in a deterioration of the performance. This is shown also by the fact that LQ has worse performance when the percentage of multi-component jobs is higher.

The best performance in Figures 6.8 – 6.10 is displayed by LS followed by LP and EQ. The low response time of LP suggests that when none of the local queues is empty and at most 50% of jobs are local, delaying the global jobs to wait for the local jobs to fit does not deteriorate much the performance; this choice is an advantage for LP compared to EQ when the percentage of local jobs is smaller (see Figure 6.7).

The differences in performance are larger in Figure 6.8 where there are 50% 4-component jobs. GP and LQ reach saturation at lower utilizations than in the cases of Figures 6.9 and 6.10, where there are no 4-component jobs and all policies display more similar performance.

EQ performs well for all chosen job mixes because it tries to fit as many jobs as possible from all queues without taking into account the characteristics of the job
stream. Favouring the multi-component jobs by enabling the global queue first at job departures also has a positive influence on performance. In Figure 6.10 EQ is slightly better than LP due to the fact that there are many 1-component jobs and the 2-component jobs fit very well on the system.

![Figure 6.11: Response times for the scheduling policies for job-stream composition (80,0,0,20) and balanced local queues.](image)

![Figure 6.12: Response times for the scheduling policies for job-stream composition (90,0,0,10) and balanced local queues.](image)

When there is a higher percentage of local jobs (see Figures 6.11 and 6.12) EQ displays the best performance; here LP is worse because for 80% and 90% local jobs the policy significantly delays the global jobs. GS also shows good results for a high percentage of single-component jobs due to the fact that it does not restrict the local jobs to the corresponding cluster; at 90% local jobs its performance approaches that of EQ. LQ has a rather good performance in these two cases, balancing the lengths of the local and global queues.

Increasing the percentage of 1-component jobs would improve the performance of GS and deteriorate the performance of all the other policies (when there are 100% single-component jobs GP, LP, EQ and LQ all become LS). Increasing the percentage of multi-component jobs would improve the performance of LS, but worsen it for the rest (when there are only multi-component jobs GP, LP, EQ and LQ become GS).
6.3.2 Dealing with only single- or multi-component jobs

In this section we compare the policies for the three job-stream compositions of Table 6.1 in which either only single-component or only multi-component jobs are present. In all these cases the local queues are balanced. The system in Figure 6.13 contains only single-component jobs, so GP, LP, EQ and LQ are reduced to LS. In the other two cases there are only multi-component jobs, so GP, LP, EQ and LQ become GS. Therefore, Figures 6.13, 6.14 and 6.15 compare only the GS and LS policies.

We also used these cases to check our simulations and gain confidence in the results (for each of these job-stream compositions five policies have to display identical results).

![Figure 6.13: Response times for the scheduling policies for job-stream composition (100,0,0,0) (GP,LP,EQ,LQ coincide with LS).](image)

When there are only single-component jobs in the system (Figure 6.13), GS has better performance due to the fact that it chooses the clusters for the jobs (with WF), while with LS jobs can be scheduled only on the clusters they were submitted to. With single-component jobs GS does a sort of load balancing over the entire system while LS keeps the clusters in isolation.

In Figures 6.14 and 6.15 LS proves to be better because for multi-component jobs the local schedulers are not restricted to their own clusters and there are up to four jobs at a time from which to choose one that fits in the system.

6.3.3 The unbalanced case

We now deal with the single case of Table 6.1 with unbalanced local queues. For the policies defining local queues, Figure 6.16 compares the performance for a job-stream composition with 80% local jobs and unbalanced local queues, 40% of the local jobs going to one queue.

Although LS with its OR variation favours the local queue receiving the highest percentage of jobs, always enabling it first at job departures, its performance is worse than that of LP and EQ. Due to the separate global queue, LP and EQ look at the top of five queues for a job that fits, compared to only four queues
6.3. Performance comparison of the policies

Figure 6.14: Response times for the scheduling policies for job-stream composition (0, 0, 0, 100) (GP, LP, EQ, LQ coincide with GS).

Figure 6.15: Response times for the scheduling policies for job-stream composition (0, 50, 50, 0) (GP, LP, EQ, LQ coincide with GS).

Figure 6.16: Response times for the scheduling policies for job-stream composition (80, 0, 0, 20) and unbalanced local queues.

in the case of LS. Both LP and EQ favour the local queues at job departures (the LF variations) but LP gives slightly worse results because it delays more the multi-component jobs by not letting them run unless at least one local queue is empty.
6.3.4 Local versus global queues

In this section we discuss the performance of global and local jobs when there are both global and local queues, i.e., for GP, LP, EQ and LQ. We derive our conclusions from the results presented in the previous sections.

Considering the separate results for local and global queues, we notice that while LP provides the best results for local jobs, GP is the best for the global jobs. LP and EQ display a low response time for both the local and the global queues, with EQ showing better performance than LP for the global queue, while LP is better for the local queues.

If there is a high percentage of multi-component jobs LQ yields a smaller average response time for the global queue, while when there is a high percentage of single-component jobs it provides better response time for the local queues. LQ is fair to all jobs from the perspective that if there is a large job which is difficult to fit on the system, be it single- or multi-cluster, not only will LQ give that job a chance to run sooner than with other policies (unless they directly favour that type of jobs), but it will also limit the delay for the jobs behind it in the queue. In fact, LQ keeps the lengths of the queues balanced, switching its behaviour between GP and LP depending on the queue lengths. On the negative side, with LQ the performance of jobs of one type is more sensitive to the performance of jobs of the other type than for GP, LP or EQ.

When none of the local queues is empty the LP policy strongly favours the local schedulers by not letting the global scheduler run. However, when at least one local queue is empty and a job departs, the global scheduler is enabled first (LP-GF). This decision has a positive effect on the overall performance but slightly deteriorates the performance of the local queues and makes it dependent on the global jobs: The better the global jobs fit, the worse the performance of the local jobs is. This dependency is even stronger for EQ.

From the four policies discussed in this section the most practical would be either LP or EQ, since the other two delay the local jobs and it can be expected that the organizations owning the different clusters would not like their local jobs to be delayed in favour of the global, multi-component jobs. In most of the cases, for both LP and EQ the GF variation gave better results: At job departures, enabling first the global queue improves the total average response time and has little influence on the local jobs. Our results show that, for policies like LP and EQ, even a high percentage of global jobs in the system does not deteriorate the performance of the local jobs. However, users submitting multi-component jobs to a system implementing such a policy should be aware that the performance of their jobs is much influenced by the local jobs and can be significantly lower than the overall performance of the system.

6.3.5 Balanced versus unbalanced local queues

In this section we look at the way the unbalance of the local queues influences the performance of the system, for different levels of unbalance. Figure 6.17 presents
6.4. Conclusions

Figure 6.17: The total average response time for different levels of unbalance of the local queues, for the LS-DO policy and job-stream composition (80, 0, 0, 20).

the effect on performance of the way jobs are distributed among local queues for the LS-DO policy and job-stream composition (80, 0, 0, 20), comparing the case when the local queues are balanced with two cases of unbalance: One of the queues receives 40% (the last case of Table 6.1) or 70% (a case not included in Table 6.1) of the jobs submitted locally. The figure shows the values of the response time at two utilizations (although we did measurements for a larger number of utilization values) and proves that the more unbalanced the local queues are, the worse the performance is. This effect on performance becomes stronger when the utilization increases: At utilization 0.560 the performance of the system with balanced local queues is very similar to the performance of the system with lower unbalance (40% of locally submitted jobs at one queue), while at utilization 0.783 the difference is obvious. At the higher utilization the system with higher unbalance (70% of locally submitted jobs to one queue) is already saturated.

6.4 Conclusions

In this chapter we have evaluated the six scheduling policies for co-allocation in multicloud systems defined in Chapter 2, for unordered requests with WF placement. We conclude the following:

- When the job-stream composition contains a high percentage of single-component jobs, allowing them to run on any of the clusters, even if scheduled by a single global scheduler as with GS, proves to be a better choice than keeping them local to the cluster they were submitted to, as LS does.

- For multi-component jobs, having more schedulers in the system and distributing the jobs among them improves the performance; any of the jobs at the heads of the queues can be chosen to run if it fits, which entails a form of backfilling with a window equal to the number of queues in the system.

- For the LP and EQ policies, the presence of multi-component jobs that are co-allocated across the system does not impact much the response time of
local jobs.

- When there are separate queues for single- and multi-component jobs, favouring the multi-component jobs lowers the total performance. If the job at the head of the global queue does not fit, running jobs from the local queues improves the performance compared to waiting for enough free processors for the global job. The best performance is often obtained by giving priority to the local queues, but in a limited way by taking care that global jobs do get some chance to run (for example LP-GF and not LP-LF).

- LP-GF is to be preferred over LP-LF because it does not significantly worsen the performance for the local jobs while improving both the total performance and the performance of the global jobs. LP-LF would bring too little improvement for the local jobs to make up for the loss in total performance and global jobs' performance.

- When the clusters have different owners, a policy that favours the local jobs would probably be preferred. For this reason, although EQ-GF gives in most of the cases with balanced queues slightly better results, a variation of LP can be more suitable.

- When in a system with both local and global queues the former are unbalanced, the scheduling policy should give priority to the local queues. When there are only local queues (the global jobs are also submitted locally), the queue receiving a higher percentage of jobs should be favoured by the policy.
Chapter 7

Simulations based on real workloads

In this chapter we study the performance of co-allocation for the GS, LS and LP policies (one policy for each of the three queuing structures) with realistic workloads derived from the DAS system. From the policies with both local and global queues we chose LP because it is the most practical. These scheduling policies were also evaluated in Chapter 6, but there the workloads were based on synthetic distributions and we did not consider communication. In this chapter we use all three ways of including communication described in Section 2.4.

In Section 7.1 we evaluate the three multicluster scheduling policies for co-allocating unordered requests in multiclusters with a workload based on traces from the DAS. For multicluster executions we employ the second method of including communication, extending the single-cluster service times with a chosen factor (see Section 2.4). For comparison, we assess the performance of total requests in a single cluster of the same total size; the total numbers of processors needed is equal to the sum of the numbers required by unordered requests.

In Section 7.2 we describe two parallel applications modeling physical phenomena for which we perform simulations in Sections 7.4 and 7.5. In Section 7.3 we investigate and compare the total runtimes of single-cluster and multicluster execution of these two applications by carrying out measurements on the DAS. We have also performed detailed measurements of the time spent in communication of one of the two applications. Because they are not used in the simulations in this chapter, these communication measurements are included in Appendix A.

In Section 7.4 we present simulation results for unordered requests and the GS, LS and LP scheduling policies. The simulations are based on our measurements of the total runtime of the two applications, both on single clusters and on multicluster systems. The communication is included in the execution times of jobs, which is our third method of introducing communication.

In Section 7.5 we perform detailed measurements of the Poisson application
which we use in our simulations of ordered and total requests. Communication is included in an explicit and detailed way, which is our first method presented in Section 2.4.

To summarize, in this chapter we include data from the DAS in our simulations in three ways: By using distributions derived from traces of the DAS for both service times and total job sizes, by using measurement results from the DAS of the total execution times of applications, and by using the detailed measurements of the computation and communication steps, respectively.

7.1 Trace-based simulations

In this section we assess the performance of multicluster systems for the GS, LS and LP scheduling policies and for the DAS-t-900, DAS-s-128 and DAS-s-64 workload distributions (see Section 2.5). We consider unordered requests, several total job-size limits and job-component-size limits, and different ways of distributing the jobs across the local queues. We compare the results to those for a single cluster with FCFS.

The simulations are for a multicluster with 4 clusters of 32 processors each and for a single cluster with 128 processors. The extension factor to account for the wide-area communication, which has to be defined for the second method of including communication (see Section 2.4), is 1.25. For the policies where local queues are defined (LS and LP), we compare the balanced case (all local queues receive the same percentage of jobs) to the unbalanced case when one local queue receives 40% and the other three 20% of the jobs submitted locally.

In this section we only consider the gross utilization, and depict the response time as a function of this utilization because that is a fair basis for comparing the policies. Only in Section 7.1.4 we compare for all the policies and job-component-size limits the gross and the net utilizations. This comparison shows how efficient the global applications use the gross utilization offered. For SC (total requests with FCFS in a single cluster) there is no wide-area communication and the net utilization is equal to the gross utilization. Section 7.1.1 makes a general comparison of the policies. In Section 7.1.2 we study the effect of limiting the total job size. Section 7.1.3 discusses the impact on performance of the job-component-size limit.

7.1.1 Comparing the policies

In this section we compare the three policies defined for multiclusters to the FCFS policy in a single cluster. For multiclusters, with the maximum job-component size we vary the numbers and sizes of job components; it influences the performance by modifying the way jobs fit together and, through the percentage of single-component jobs, the percentage of jobs with extended service times.
Figure 7.1: The performance of the policies for job-component-size limits of 16, 24 and 32 (top to bottom); for LS and LP we depict results with balanced local queues (left) and unbalanced local queues (right).

Multiclusters versus single clusters

In this section we compare the four policies for each of the three job-component-size limits. For SC the size limit does not influence the results, so the SC curves can be used as a reference for the graphs in Figure 7.1. With the workload considered the performance is poor for all policies and all size limits. Even for total requests the maximal utilization is below 0.65. This seems to indicate that the bad performance is caused by the total sizes of the jobs; we further investigate this aspect in Section 7.1.2.
Even though it restricts single-component jobs to the local clusters and uses co-allocation to schedule its many multi-component jobs (48.7%) for which the service times are extended due to the global communication, LS performs much better than the other multicluster policies for a size limit of 16; it also provides a higher maximal utilization than SC, but a part of that utilization is spent waiting for the wide-area communication, so SC is still significantly better (see also Section 7.1.4). The main advantage of LS is that it distributes the multi-component jobs among the local queues allowing the local schedulers to spread them across the clusters; at each moment a job can be chosen from any of the local queues, which generates a form of backfilling with a window equal to the number of clusters. For size limits of 24 and 32, where there are only 26.2% and 22.0% multi-component jobs
respectively, the performance of LS is worse. In all the graphs LP displays the worst results because all the multi-component jobs are placed in a single global queue, and all the single-component jobs are restricted to the local clusters. Although GS has only a single global queue, it is consistently better than LP, and for a size limit of 32 (when there are many local jobs) it even approaches LS; this is due to the fact that it has the freedom to choose the clusters for the single-component jobs.

In itself, the fact that a multicluster policy can have similar performance to SC in terms of gross utilization makes co-allocation a good option, showing that the fragmentation caused by having multiple clusters can be overcome. Even when possible, admitting only single-component jobs would result in a lower utilization as our previous results have shown (see Chapter 4). Combining these results, it seems that a multicluster system that does not support co-allocation experiences more fragmentation. The internal loss of utilization caused by the slow global communication should be handled by structuring the applications to use the high utilization offered by some of the policies, while minimizing the amount of time spent with wide-area communication. An application with less global communication can benefit more from a policy that uses co-allocation.

**Balanced versus unbalanced local queues**

Comparing the balanced and unbalanced cases for LS and LP (see Figure 7.1) we notice that an unbalanced load for the local queues has a negative impact on performance. Since for both GS and SC all jobs go to the global queue, their curves can be used as a reference when comparing the pairs of balanced-unbalanced graphs for LS and LP.

The worsening of the performance is more pronounced for LS, especially for larger job-component-size limits, when there is a higher percentage of local jobs. One cause is that the least loaded queues get empty sooner, which decreases the backfilling window for the global jobs. The deterioration is stronger when there is a high percentage of local jobs which indicates a second cause: The local jobs are restricted to their corresponding clusters, and a higher percentage of (local) jobs in a queue means a higher load for the local cluster; besides, their cluster can be used with equal priority by global jobs from the other local queues. As a result, the more loaded local queue saturates faster than in the balanced case, decreasing this way the maximal utilization of the system. For a size limit of 32 and unbalanced local queues, LS performs worse than GS and similarly to LP.

For LP the performance deterioration due to the unbalance of the local queues is small for all size limits. All global jobs go to the global queue, and the local queues have priority on their local clusters as long as none of them is empty. When there are few local jobs, the loads of all local queues are low even in the unbalanced case (for a size limit of 16 the most loaded local queue receives 20.5% of all jobs); for a high percentage of local jobs the local queues access their clusters with priority, which reduces the impact of the unbalance. Even when a local queue gets empty and multi-component jobs from the global queue are started, the use
of WF insures that the least loaded clusters are chosen.

**LP approaching saturation**

For LP, we can expect that the performance differs between the global and local queues. In Figure 7.2 we depict for the LP policy, beside the total average response times, the average response times for the local queues and for the global queue. Beside the gross utilization we display in each chart the corresponding net utilization. The figure shows the same results as Figure 7.1, but for a small set of utilization values chosen so that at least one of the policies approaches saturation. This representation makes it easier to compare the response times of the policies but has the limitation of providing less information. At the chosen utilization values, LP, the policy with the worst performance, is approaching saturation—the global queue grows without bounds. As a consequence, we can easily see for LP the differences in response times, for example among the balanced and unbalanced cases, even though as the graphs in Figure 7.1 show these differences are small. On the other hand, if we compare the response times for the "better" policies like LS, the charts do not show the significant performance loss from the unbalanced case. At the utilizations from Figure 7.2, LS is on a very low point on its response-time curve and its performance is very good in both the balanced and the unbalanced cases. LP delays the large multi-component jobs much longer than LS and its performance bottleneck is the global queue and not the local ones. This is shown in Figure 7.2 where for LP the average response times for the global queue are much larger than those for the local queues. The response times for the global queue are displayed on the corresponding bars in the charts.

**Conclusions**

We conclude that for a system with a workload similar to that of the DAS, and an overhead due to the wide-area communication covered by the 1.25 extension factor, the LS policy is the best option. In all the cases the performance is very poor: For all job-component-size limits, both in the balanced and the unbalanced cases, and even for total requests. The main cause seems to be the total job-size distribution, and not the job-component-size limit, the global communication, the policies, or the extra fragmentation introduced by scheduling multi-component jobs in a multiclient system. We verify this assumption in the next section.

**7.1.2 Limiting the total job size**

In the job-size distribution derived from the DAS, only 2% of the jobs require more than 64 processors (out of which 1.2% need 128 processors to run, which is the entire system). Our assumption is that eliminating from the distribution this small percentage of very large jobs can bring important changes to our previous results.

Figure 7.3 compares the performance of the four policies discussed before, for the DAS-s-128 and the DAS-s-64 distributions, a job-component-size-limit equal
trace-based simulations

to 16, and balanced local queues. We choose this set of parameters because they previously provided the most interesting results (LS better than SC).

![Figure 7.3: The response times for maximal total job size 64 and 128 (job-component-size limit 16, balanced local queues).](image)

With DAS-s-64, the improvements in performance are large for LS, and even more so for SC. When a job requiring 128 (or a similarly large number of) processors is at the top of the queue, SC waits for the entire system to become empty, which yields a very low utilization. LS on the other hand can run jobs from the other queues and postpone the large job until either the other queues are empty, or they also have at the top large jobs that do not fit. For DAS-s-64 the largest jobs in the system require only half of the processors, which improves the utilization of SC and diminishes the advantage of LS.

LP and GS also perform better for DAS-s-64. Since it gives the local queues priority, with LP the very large jobs in the global queue are much postponed for DAS-s-128. For DAS-s-64 jobs are smaller, and LP outperforms GS because it can benefit from having more queues (also a form of backfilling but worse than in the case of LS since all global jobs go to the global queue).

The results in this section indicate that a very small percentage of very large jobs can significantly worsen the performance, and rather than designing a complicated policy to deal with such jobs, simply imposing a maximum size (in terms of numbers of processors) for the jobs submitted to the system brings more important improvements. Of course, for the users whose jobs are larger than the limit allowed, complying with this restriction translates into reconfiguring their jobs to use fewer processors and accepting the consequence of having longer service times. The extension of the service times is highly dependent on the specific application.

### 7.1.3 Setting the job-component-size limit

In this section we evaluate the performance of GS, LS and LP depending on the maximum size admitted for the job components. We compare three size limits:
Figure 7.4: The performance of LS, LP and GS (top-bottom) depending on the size limit of the job components. For LS and LP both the balanced (left) and unbalanced (right) cases are depicted.

16, 24 and 32 (see Figure 7.4), and use the DAS-s-128 distribution.

Having smaller job components improves the system's performance, but having many components relative to the number of clusters deteriorates the performance. A smaller size limit also means more multi-component jobs, so more jobs with extended service times, which also worsens the performance; for a size limit of 16 there are 26.7% more multi-component jobs than for a size limit of 32. Adding up all these factors, for GS a size limit of 32 provides slightly better results than 16.

For LS, due to the backfilling effect and to the fact that multi-component jobs can be spread across any of the clusters, while the single-component jobs
are restricted to the local clusters, a smaller size limit becomes an important advantage. For this policy a size limit of 16 is a much better choice than a size limit of 32. Comparing the balanced and unbalanced cases for LS we notice that the performance deteriorates more when the local queues are not balanced for a size limit of 32. The queue receiving extra load can place the multi-component jobs also on other clusters, but it has to keep local the single-component jobs, overloading the local cluster. It means that the more single-component jobs there are, the more will the system be affected by the unbalance.

For LP, all multi-component jobs go to the global queue and only the single-component jobs are spread among the local queues. Smaller local jobs fit better, and fewer local jobs means fewer jobs restricted to the local clusters. Fewer local jobs also means that the local queues empty faster and the global queue gets more often the chance to start jobs. On the other hand, the global queue has low priority when no local queue is empty, and the more global jobs, the higher the average response time for those jobs. As Figure 7.4 shows, LP performs better for a size limit of 16 than for 32, but the difference is small in both the balanced and the unbalanced cases.

For all the policies, the worst results are obtained for a job-component size limit of 24. Since all the factors discussed above would place a system with limit 24 in between the other two cases, the reason for this very large difference in performance should be in the way jobs fit together in the system, depending on the size limit of their components.

For a size limit of 24, the jobs split up differently are only those with sizes (integers) in the intervals [17, 24] and [33, 72] when compared to a size limit of 16, and the intervals [25, 32], [49, 64] and [73, 96] when compared to 32. Since for GS and LP the performance is very similar for size limits of 16 and 32, it makes sense to look for the reason of the much worse performance when the size limit is 24 in the interval [49, 64], which contains the jobs split differently compared to both the other limits. Checking the log, we found that the most relevant size in that interval is 64: 19.0% of the jobs in the log have this size. For a size limit of 16, the corresponding job request is (16, 16, 16, 16), for a size limit of 32 it is (32, 32, 0, 0) and for a limit of 24 we obtain (22, 21, 21, 0). Considering an empty system which receives a job of size 64, in the first two cases after the job is placed there are many jobs, with different numbers of components and sizes up to 64 that would still fit in the system. However, in the third case only single-component jobs with maximum sizes of 10 and 11 can fit in three of the clusters, and single-component jobs with a maximum size of 24 (due to the size limit) in the fourth, empty cluster. A second job with a size of 64 would also fit in the first two cases, but not in the third.

### 7.1.4 Gross versus net utilization

In Section 7.1 we have studied the average response time as a function of the gross utilization. In this section we discuss the difference between gross and net utilization, and quantify this difference for the cases considered in Section 7.1.
We have defined in Section 1.2 the net and the gross utilization based on the job service times in single clusters with fast local communication, and on the extended service times (at least, for multi-component jobs) to account for the slow wide-area communications, respectively. The difference between these utilizations is the capacity lost internally in multi-component jobs due to slow wide-area links. This internal capacity loss might be reduced by restructuring applications or by having them use (collective-) communication operations optimized for wide-area systems.

The performance of a multicluster policy may look good when considering the response time as a function of the gross utilization, but, when there is much internal capacity loss, the performance as a function of the net utilization (or of the throughput) may be poor. This "real" performance of a multicluster policy would improve with more efficient applications or with faster global communication (smaller extension factor). In the extreme case, if the global communication was as fast as the local communication, LS would sometimes provide even better performance than SC (see Figure 7.1).

In Figure 7.5 we depict the average response time for our three policies, for the DAS-s-128 job-size distribution, and for the three job-component-size limits as a function of both the gross and the net utilization. For LS and LP the local queues are balanced; the results for unbalanced local queues do not bring additional information. To assess the difference between the two utilizations for a specific policy and job-component-size limit at a certain response time, one should compare the graphs in the horizontal direction. Of course, for the same workload (defined by the arrival rate, and so, by the net utilization) and job-component-size limit, the difference between the gross and the net utilization is the same for all scheduling policies, albeit at possibly different response times.

In our model, the job sizes are independent of the job service times. This means that we can compute the ratio between the gross and the net utilization, independent of the scheduling policy, as the quotient of the weighted average total job size with single-component jobs having weight 1 and multi-component jobs having weight 1.25 (the extension factor) and the average total job size. For the DAS-s-128 job-size distribution we find for the job-component-size limits of 16, 24, and 32 ratios of gross and net utilization of 1.218, 1.173, and 1.159, respectively.

The difference between the gross and the net utilization grows with a decrease of the job-component-size limit—the more multi-component jobs the higher the amount of global, slow communication—and with the number of (multi-component) jobs that fit on the system—when jobs fit better, the workload can be higher, entailing more global communication. For these reasons, for all the policies a job-component-size limit of 16 yields a larger distance between the curves of the net and gross utilizations than 24 and 32. For a size limit of 24 there are 4.2% more multi-component jobs than for 32, but since jobs fit very poorly, the gross and net utilizations are closer together when the size limit is 24. Comparing the policies, we notice that for a size limit of 16, LS yields a much better utilization, and so also more wide-area communication, than the other policies, and as a consequence, it has the largest difference between the gross and the net utilization. For component-size limits of 24 and 32 the amount of utilization spent in global
Figure 7.5: The response time as a function of the gross and the net utilization for the LS, LP and GS policies and for the three job-component-size limits (balanced local queues for LS and LP).
Table 7.1: The maximal gross and net utilizations for different job-component-size limits for the GS policy.

<table>
<thead>
<tr>
<th>job-component-size limit</th>
<th>maximal utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gross</td>
</tr>
<tr>
<td>16</td>
<td>0.588</td>
</tr>
<tr>
<td>24</td>
<td>0.491</td>
</tr>
<tr>
<td>32</td>
<td>0.595</td>
</tr>
</tbody>
</table>

communication is rather similar for all three policies.

In Chapter 3 we have studied the maximal utilization (beyond which the system gets unstable) of co-allocation with analytic means (when the service times are exponential) and with simulations. Both methods only work when there is a single, global queue, so in the cases of GS and SC. In these simulations, we maintain a constant backlog and observe the time-average fraction of processors being busy, which yields the maximal gross utilization. For GS, the simulations produced the values for the maximal gross utilization as in Table 7.1; the maximal net utilizations are then computed with the ratios between the two types of utilization. These values are in very good agreement with the graphs for GS in Figure 7.5. For SC, our simulations gave us a maximal utilization of 0.625 (cf. the curve for SC in Figure 7.1).

### 7.1.5 Conclusions

In this section we have evaluated the performance of the GS, LS and LP policies for co-allocating unordered requests in multiclusters with a workload derived from traces of the DAS system. We conclude as follows:

- *The best policy proved to be LS*, with a performance comparable in some cases to using FCFS for total requests in a single cluster.

- Studying the job sizes, we have observed that a *small percentage of very large jobs with total sizes close to the size of the system can strongly impact on performance*. Although the choice of policy does matter for the performance, *when dealing with very large jobs* requiring (almost) the entire system to run, we found that by far the *largest improvement in performance can be obtained from simply limiting the total job size*.

- *When choosing the job-component-size limit*, care should be taken that the *jobs of the total sizes that occur most often* (in our log 19% of the jobs are of size 64!) are *split up in a way that makes them easy to fit in the system*. When both the clusters' sizes and the most popular total job sizes are powers of two, a limit that is also a power of two makes sense.

- In multicluster systems we have to deal with a significant amount of processor time spent waiting for the global communication. However, *co-allocation is*
7.2. Description of the applications

A viable option at least when the extension factor covering the duration of the global communication does not exceed 1.25.

- For some policies, such as LS, a higher amount of co-allocation—going from component size limit of 32 to 16—can significantly improve the performance, while for others (LP, GS) it does not. More co-allocation brings along more flexibility in spreading the jobs over the system, but also more global communication. For a policy that can take advantage of it, this flexibility can compensate the disadvantage of slower communication.

7.2 Description of the applications

In this section we describe the two applications for which we performed measurements on the DAS. The results of the measurements are discussed in Section 7.3 and used in the simulations in Section 7.4.

7.2.1 The Ensflow application

The Ensflow application [48] uses the data-assimilation technique to understand the evolution of streams and eddies in the ocean near the southern tip of Africa. In this technique, information from observations of the system is combined with information on the evolution of the system obtained from an implementation of the laws of physics. This can be done by using ensemble models that do not calculate the evolution of a single state but rather of a large number (an ensemble, typically 50-500) of different states (ensemble members). In our case there are 60 ensemble members that evolve for a period of 20 days with a time step of 24 hours. Every 240 hourly time steps, an analysis and an update of the ensemble members are done to obtain the optimal estimate for the past period. Each of the ensemble members evolves independently of the others during the time between analysis and update. The sequence of ensemble averages over time describes the development of the ocean’s currents best fitting the observations. The application has the following structure:

```
/*----------initialisation--------*/
initiate 60 ensembles;

/*----------start main loop--------*/
while time < stop_time
  /* computation */
  evolve the 60 ensembles;
  if (time = time_to_analyse)
    /* computation + communication */
    analyse and update ensembles;
  endif
endwhile
/*----------end main loop--------*/
```
The main loop is executed 20 times, with two data adjustments. Only during the data adjustment phase (analysis and update ensembles) data are exchanged (using MPI). The data of the ensemble members are local to the processors, and the ensemble members are distributed evenly over the processors. To avoid processors from being unnecessarily idle, we choose the number of processors such that the number of ensemble members is an exact multiple of it. In [48], the Ensflow application is described in more detail, and measurements of the total runtime on two multiprocessors are presented.

### 7.2.2 The Poisson application

Our Poisson application implements a parallel iterative algorithm to find a discrete approximation to the solution of the two-dimensional Poisson equation (a second-order differential equation governing steady-state heat flow in a two-dimensional domain) on the unit square. For the discretization, a uniform grid of points in the unit square with a constant step in both directions is considered. The application uses a red-black Gauss-Seidel scheme (see for instance [37], pp. 429–433), for which the grid is split up into "black" and "red" points, with every red point having only black neighbours and vice versa. In every iteration, each grid point has its value updated as a function of its previous value and the values of its neighbours, and all points of one colour are visited first followed by the ones of the other colour. The application, which is implemented in MPI, has the following structure:

```c
/******-initialisation---------*
if proc_index = 0
     read the initial data;
     /* communication */
     broadcast data to all the processes;
endif

/******-start main loop-------*
while global-error => limit
     /* computation */
     update black points;
     update red points;
     /* communication */
     exchange borders with neighbours;
     /* communication + synchronization */
     collect/distribute global error;
endwhile
/******-end main loop--------*/
```

The domain of the problem is split up into a two-dimensional pattern of rectangles of equal size among the participating processes. In our experiments, we assign only one process to a processor. A way of splitting up the domain is called a process(or) configuration, and is indicated by $h \times v$, with $h, v$ the numbers of
7.3. Runtime measurements on the DAS

Processes in the horizontal and vertical directions, respectively. In Section 7.3 we will consider the numbers of processors and the processor configurations as shown in Table 7.2.

Table 7.2: The processor configurations used in our measurements.

<table>
<thead>
<tr>
<th>total number of processors</th>
<th>processor configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4x2</td>
</tr>
<tr>
<td>16</td>
<td>4x4</td>
</tr>
<tr>
<td>32</td>
<td>8x4</td>
</tr>
<tr>
<td>64</td>
<td>8x8</td>
</tr>
</tbody>
</table>

Every process communicates with each of its neighbours in order to exchange the values of the grid points on the borders and to compute a global stopping criterion. Exchanging borders takes place in four consecutive steps; first all communication in the direction top is performed, and then in the directions bottom, left and right. The amount of communication depends on the size of the grid, the number of participating processes, and the initial data. When we execute the Poisson application on multiple clusters, the process grid is split up into adjacent vertical strips of equal width, with each cluster running an equal consecutive number of processes (we assume processes to be numbered in column-major order). For instance, for process configuration 4x4 and two clusters, the processes are split up as depicted in Figure 7.6. Here, processors 4–11 have to exchange border information with processors in the other cluster.

```
3 7 11 15
2 6 10 14
1 5 9 13
0 4 8 12
```

Figure 7.6: The process grid for the Poisson application for process configuration 4x4 divided over two clusters (left–right).

7.3 Runtime measurements on the DAS

In this section we present the results of the measurements of our two applications on the DAS. We use Globus for submitting multicomponent jobs to the DAS. In all of our experiments, the jobs always have components of equal size. Since at the time of the experiments Globus was unable to use the fast local DAS interconnect (Myrinet) but used the slower local Ethernet instead, we em-
ploy both PBS and Globus for running the applications in a single DAS cluster. The PBS measurements yield the best performance of single-cluster operation. Measurements with Globus on a system with $C$ clusters are labeled with Globus-$C$.

### 7.3.1 Total runtime of the Ensflow application

For an investigation of the total runtime we ran the Ensflow application once for different numbers of processors and clusters. The results of the measurements are presented in Figure 7.7.

![Bar chart showing total runtime of the Ensflow application](image)

**Figure 7.7:** The total runtime of the Ensflow application (in seconds) for different numbers of processors and clusters (No data when the number of processors is not a multiple of the number of clusters, and for 60 processors with Globus-1).

The gaps for 15 processors and Globus-2 and Globus-4, for 20 processors and Globus-3, and for 30 processors and Globus-4 are due to the fact that in these cases we cannot have equal-size job components. The gap for Globus-1 with 60 processors is caused by the limitation of 32 processors in a single cluster when using Globus. We find that the performance of multicluuster execution for all numbers of clusters considered compared to single-cluster execution is very good for this application. In addition, the speedup is quite reasonable. Relative to the 12-processor case, the efficiency slowly decreases to about 0.7 for 60 processors. The explanation of the good performance of multicluuster execution is that this application has a relatively small communication component.

### 7.3.2 Total runtime of the Poisson application

For a first investigation of the total runtime of the Poisson application, we ran the application once varying the grid size, the total number of processors (see Table 7.2 for the corresponding processor configurations), and the number of clusters. In addition to the total runtime, we also record the number of iterations needed to reach convergence. The results of the measurements are presented in Table 7.3, and graphically in Figure 7.8. (Because of the numbers of processors we consider,
we cannot use three clusters.) Again there are gaps for Globus-1, this time for 64 processors in a single cluster, for the same reason as above.

Table 7.3: The number of iterations and the total runtime (in seconds) of the Poisson application for different grid sizes and numbers of processors and clusters.

<table>
<thead>
<tr>
<th>grid size</th>
<th>total number of processors</th>
<th>number of iterations</th>
<th>PBS</th>
<th>Globus-1</th>
<th>Globus-2</th>
<th>Globus-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>8</td>
<td>2436</td>
<td>0.74</td>
<td>3.23</td>
<td>11.5</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2132</td>
<td>0.74</td>
<td>3.59</td>
<td>12.1</td>
<td>11.8</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2158</td>
<td>0.93</td>
<td>4.54</td>
<td>17.4</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2429</td>
<td>1.21</td>
<td>—</td>
<td>24.2</td>
<td>21.1</td>
</tr>
<tr>
<td>1000x1000</td>
<td>8</td>
<td>2630</td>
<td>70.9</td>
<td>86.6</td>
<td>109</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4347</td>
<td>60.2</td>
<td>78.6</td>
<td>119</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4356</td>
<td>34.3</td>
<td>46.7</td>
<td>68.8</td>
<td>67.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2650</td>
<td>8.1</td>
<td>—</td>
<td>30.7</td>
<td>31.7</td>
</tr>
<tr>
<td>2000x2000</td>
<td>8</td>
<td>2630</td>
<td>291</td>
<td>335</td>
<td>358</td>
<td>365</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4387</td>
<td>265</td>
<td>292</td>
<td>339</td>
<td>332</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4387</td>
<td>134</td>
<td>161</td>
<td>193</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2650</td>
<td>46.8</td>
<td>—</td>
<td>80.4</td>
<td>85.1</td>
</tr>
<tr>
<td>4000x4000</td>
<td>8</td>
<td>2630</td>
<td>1230</td>
<td>1277</td>
<td>1390</td>
<td>1463</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2644</td>
<td>649</td>
<td>725</td>
<td>766</td>
<td>767</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2651</td>
<td>357</td>
<td>371</td>
<td>402</td>
<td>440</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2650</td>
<td>188</td>
<td>—</td>
<td>231</td>
<td>251</td>
</tr>
</tbody>
</table>

We find that for a very small grid size, the runtime may increase considerably when using more clusters. However, for a large grid size, the performance of multicluster execution compared to single-cluster execution is quite reasonable. Since the processor configuration influences the number of iterations needed to reach convergence (which determines the total runtime), it is difficult to make a general statement about the speedup. In particular for grid sizes 1000x1000 and 2000x2000, the number of iterations to reach convergence is very variable. However, for grid size 4000x4000 the number of iterations is almost constant, and the speedup when going from 8 to 64 processors for PBS, Globus-2, and Globus-4 is 6.5, 6.0, and 5.8, respectively.

For a further investigation of the total runtime we now fix the processor configuration to 4x4, and we include a few additional grid sizes. For every set-up (grid size and number of clusters) we ran the application ten times. The results of the measurements (minimum, average, and maximum) are presented in Table 7.4. For a better comparison, we depict in Figure 7.9 for every grid size the (average) runtimes relative to the (average) single-cluster PBS runtimes (which are normalized to 1).

It is clear that for large grid sizes, this application is well suited for multicluster execution. The explanation is that the two major components of the total runtime,
Figure 7.9: The total runtime of the Poisson application (in seconds) for processor configuration 4x4 for different grid sizes and numbers of clusters, normalized with respect to PBS.

for a multicluster with 4 clusters of 32 processors each. In this section we only look at the gross utilization and depict the response time as a function of this utilization, because it is a fair basis for comparing the policies. After presenting the workloads in the simulations, we first discuss the influence of the numbers and sizes of the job components on the performance, and then the benefits and disadvantages of co-allocation, compared to a system without co-allocation. Finally, we make a general comparison of the policies. Section 7.4.2 compares for all the policies and workloads the gross and the net utilization, which shows how efficient the global applications use the gross utilization offered.

The workloads

Each of the jobs in the simulated workload is supposed to run one of our two applications; in the case of the Poisson application, we assume the grid size to be 4000x4000. We assess three cases: 100% of the jobs in the system run the Poisson application, 100% of the jobs run the Ensflow application, and each of the two applications is represented by 50% of the jobs in the system. Tables 7.5 and 7.6 display the execution times measured on the DAS for the two applications in the several configurations that we are using in the simulations. These values are the same as the ones depicted in Figure 7.7, and in Figure 7.8 for grid size 4000x4000; for a single cluster we use the PBS runtimes. We assume the interarrival times to be exponentially distributed.

Jobs are split up in different ways, but their components are always of equal size, and we also keep the percentages of jobs for each total size always equal. For the same total size, the various splitting choices admitted in the system receive equal probabilities.

We compare a no co-allocation case, when only single-component jobs are admitted, to several co-allocation cases. We define the following co-allocation rules:
7.4. Simulations based on runtime measurements

Table 7.5: The execution times (in seconds) for the Poisson application, depending on the total job size and the number of components, used in the simulations.

<table>
<thead>
<tr>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1230.0</td>
</tr>
<tr>
<td>16</td>
<td>649.0</td>
</tr>
<tr>
<td>32</td>
<td>357.0</td>
</tr>
</tbody>
</table>

Table 7.6: The execution times (in seconds) for the Enflow application, depending on the total job size and the number of components, used in the simulations.

<table>
<thead>
<tr>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3485.0</td>
</tr>
<tr>
<td>15</td>
<td>2836.0</td>
</tr>
<tr>
<td>20</td>
<td>1935.0</td>
</tr>
<tr>
<td>30</td>
<td>1563.0</td>
</tr>
</tbody>
</table>

1. [no] There are only single-component jobs, co-allocation is not allowed.

2. [co] Both single- and multi-component jobs are allowed, without restrictions on the sizes of job components and the numbers of components.

3. [rco] Both single- and multi-component jobs are allowed, but the job-component sizes are restricted to half of the clusters’ sizes.

4. [fco] Both single- and multi-component jobs. The job-component sizes are restricted to half of the clusters’ sizes, and only multi-component jobs with two components are allowed.

In Tables 7.7, 7.8, and 7.9 we show the resulting percentages of jobs for the numbers of components allowed for the Poisson application (here we disallow jobs of size 8 to be split into 4 components), for the Enflow application, and for an even mix of these, respectively.

The influence of the numbers and sizes of the job components

In Figure 7.10 we show the response time as a function of the (gross) utilization for the three job mixes, the three scheduling policies, and the four co-allocation rules. Because our two applications have very different service times, we assess the performance more in terms of the point where the system saturates (where the response-time curves rise very steeply) than in terms of the actual response times. The performance is the best for the Poisson application; a reason for this is that
Table 7.7: The percentages of jobs with different numbers of components for the four co-allocation rules for the Poisson application.

<table>
<thead>
<tr>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[no]</td>
</tr>
<tr>
<td>8</td>
<td>33.34%</td>
</tr>
<tr>
<td>16</td>
<td>33.33%</td>
</tr>
<tr>
<td>32</td>
<td>33.33%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[rco]</td>
</tr>
<tr>
<td>8</td>
<td>16.67%</td>
</tr>
<tr>
<td>16</td>
<td>11.11%</td>
</tr>
<tr>
<td>32</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

In that case all the job sizes are also powers of two, like the clusters' sizes, which makes them fit better in the system. For the Ensflow application the utilization achieved is worse because of the job sizes, which in most combinations add up in a way that leaves more idle processors in the system than in the case of the Poisson application. For all policies and co-allocation rules considered the worst performance is displayed by the mix of the two applications, where the different sizes of jobs are even more difficult to fit in an efficient way.

In all the graphs in Figure 7.10 the [co] co-allocation rule yields the poorest performance. This shows that although in general co-allocation provides more flexibility in placing jobs on the system, jobs with conflicting requirements can make the performance worse than that in the absence of co-allocation. The bad performance is due to the simultaneous presence in the system of large single-component jobs, using (almost) entire clusters, and of jobs with many components, even equal to the number of clusters in which case on each of the clusters there has to be enough room to accommodate a job component. Possible improvements are to restrict the maximum size of job components, and to limit the number of components of the multi-component jobs. The [rco] co-allocation rule includes the first restriction, while [fco] includes both. The graphs show that in all the cases considered imposing these restrictions significantly improves the performance. For LS, the performance for both the [rco] and [fco] cases proves to be much better than for the no co-allocation case. The same result holds for LP. When there are only single-component jobs LP becomes LS, and that is why for the no co-allocation case with LP the curve for LS is depicted. For GS, co-allocation does not enhance the performance, but maintains or only slightly improves it with the [fco] restrictions—large jobs are always split and only maximum two components are allowed—and deteriorates it in the other cases. For GS, the advantage of
Table 7.8: The percentages of jobs with different numbers of components for the four co-allocation rules for the Ensflow application.

<table>
<thead>
<tr>
<th>Total job size</th>
<th>[no]</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of job components</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>25.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>25.0%</td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>25.0%</td>
<td>0.0%</td>
<td></td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>25.0%</td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total job size</th>
<th>[rco]</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of job components</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>6.25%</td>
<td>6.25%</td>
<td>6.25%</td>
<td>6.25%</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12.5%</td>
<td></td>
<td>12.5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8.34%</td>
<td>8.33%</td>
<td></td>
<td>8.33%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>8.34%</td>
<td>8.33%</td>
<td>8.33%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total job size</th>
<th>[co]</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of job components</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>6.25%</td>
<td>6.25%</td>
<td>6.25%</td>
<td>6.25%</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12.5%</td>
<td></td>
<td>12.5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.0%</td>
<td>12.5%</td>
<td></td>
<td>12.5%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0.0%</td>
<td>12.5%</td>
<td>12.5%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total job size</th>
<th>[fco]</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of job components</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>12.5%</td>
<td>12.5%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>25.0%</td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.0%</td>
<td>25.0%</td>
<td></td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0.0%</td>
<td>25.0%</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

more flexibility brought by co-allocation does not compensate the disadvantage of longer service times due to the inter-cluster communication. The GS policy does not restrict single-component jobs to the local clusters, which makes the performance in the absence of communication rather good. Jobs are scheduled in a FCFS manner from the single queue and the more freedom in spreading the jobs on the clusters introduced by co-allocation is not used enough.
Figure 7.10: The performance of GS, LS and LP (top to bottom) for the Poisson application, the Enslow application, and a mix of the two in equal proportions (left to right), for the four co-allocation rules.
7.4. Simulations based on runtime measurements

Comparing the policies

In this section we compare the three policies defined for the three application mixes and the different co-allocation rules. From Figure 7.10 we conclude that independent of the application mix LS provides the best results for the co-allocation cases. When there are only single-component jobs the performance of GS is better. LP becomes LS when there are just single-component jobs, so the performance of the two policies is the same in the absence of co-allocation.

With the [rco] restrictions LS display much better results than LP, the difference between the two policies being that LP keep all multi-component jobs in a single queue. This relates to our observation for GS that when there is a single queue for multi-component jobs, those with many components are hard to fit and have a strong negative impact on performance. GS is better for the single-component jobs, but once multi-component jobs are allowed, the extra queue for the global jobs in LP and spreading the global jobs among the local queues in the case of LS bring enough benefits as to allow those policies to outperform GS.

Comparing all the cases considered, we concluded that the best results are displayed by LP and LS with the [fco] restrictions. The similar performance of LS and LP in that case shows that for those sizes and numbers for the job components, to have a separate queue for the multi-component jobs is enough and the backfilling effect with a window equal to the number of clusters induced by LS does not bring extra improvements.

7.4.2 Gross versus net utilization

In Section 7.4.1 we have studied the average response time as a function of the gross utilization. In this section we discuss the difference between gross and net utilization, and quantify this difference for the cases considered in Section 7.4.1. We have defined the net and the gross utilization based on the job service times in single clusters with fast local communication, and on the longer service times displayed by multi-component jobs running the same application on multiple clusters (thus using slow inter-cluster communication), respectively. The difference between these utilizations is the capacity lost internally in multi-component jobs due to slow wide-area links. This internal capacity loss might be reduced by restructuring applications [44] or by having them use (collective-) communication operations optimized for wide-area systems [36].

The performance of a multicluster policy may look good when considering the response time as a function of the gross utilization, but, when there is much internal capacity loss, the performance as a function of the net utilization (or of the throughput) may be poor. This "real" performance of a multicluster policy would improve with more efficient applications or with faster global communication.

In Figures 7.11, 7.12 and 7.13 we depict the average response time for our three policies, for the three application mixes and for the different ways of co-allocation studied, as a function of both the gross and the net utilization. To assess the difference between the two utilizations at a certain response time, one
Figure 7.11: The response time as a function of the gross and the net utilization for the GS policy, the three application mixes and the three co-allocation rules that allow co-allocation.
Figure 7.12: The response time as a function of the gross and the net utilization for the LS policy, the three application mixes and the three co-allocation rules that allow co-allocation.
Figure 7.13: The response time as a function of the gross and the net utilization for the LP policy, the three application mixes and the three co-allocation rules that allow co-allocation.
should compare the graphs in the horizontal direction. Of course, for the same workload (defined by the arrival rate, and so, by the net utilization), the difference between the gross and the net utilization is the same for all scheduling policies and co-allocation rules, albeit at possibly different response times.

The largest difference between the gross and the net utilizations is always displayed for the Poisson application. This is an expected consequence of the fact that this application requires the largest amount of communication. Spreading the jobs running this application on more clusters also results in more wide-area communication than for the Ensflow application, or the equal mix of the two applications.

For all the policies and job mixes, comparing the three co-allocation cases we observe that the largest amount of intercluster communication is shown for the [rco] restrictions, and the least for the [co] restrictions. By limiting the size of the single-component jobs, [rco] and [fco] increase the percentage of multi-component jobs which brings more wide-area communication. Since it limits the number of job components, [fco] yields a lower amount of intercluster communication compared to [rco]. These results are also valid for the Ensflow application, even though the differences are smaller since that application requires very little communication.

Despite the significant difference in performance, for the same application mix and the same restrictions imposed for co-allocation, all the three policies show very similar differences between the graphs for net and gross utilization. In general we could expect that policies with better performance would show more wide-area communication for the same set of jobs.

### 7.4.3 Conclusions

In this section we have performed measurements of two applications on our multicore DAS system, followed by simulations of three multicore scheduling policies incorporating co-allocation. Our results show that both applications, with appropriate parameter settings in one of them, are very well suited for multicore operation. We conclude the following.

- **The performance of multicore execution of the Ensflow application is very good,** which can be explained by its relatively small communication component.

- **Also the Poisson application is well suited for multicore execution,** at least for large grid sizes, **when the communication component becomes relatively small.**

- **The penalty for the slow multicore communication can be reduced by allowing the computation and communication parts of the processes of a multicore job to overlap.**

- **To be able to make a well-considered decision whether to submit an application to a single cluster or across multiple clusters,** it would be convenient to have a **synthetic application parameterized by the way it is split up across**
clusters and by its communication pattern, to simulate a range of possible applications.

- Our simulations of multicluster scheduling policies show that simply allowing co-allocation without any restrictions is not a very good idea, for none of the policies. In all cases, one should at least limit the job-component sizes, and preferably also the number of job components.

- Furthermore, we found that the policies with local queues (possibly with a global queue for multicomponent jobs) yield better performance than having only a single global queue.
Table 7.9: The percentages of jobs with different numbers of components for the four co-allocation rules and a mix of the Poisson and Ensflow applications in equal proportions.

<table>
<thead>
<tr>
<th>[no]</th>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>16.67%</td>
<td>0.0%</td>
</tr>
<tr>
<td>16</td>
<td>16.67%</td>
<td>0.0%</td>
</tr>
<tr>
<td>32</td>
<td>16.66%</td>
<td>0.0%</td>
</tr>
<tr>
<td>12</td>
<td>12.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>15</td>
<td>12.5%</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>12.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>30</td>
<td>12.5%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[rco]</th>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8.335%</td>
<td>8.335%</td>
</tr>
<tr>
<td>16</td>
<td>5.557%</td>
<td>5.557%</td>
</tr>
<tr>
<td>32</td>
<td>0.0%</td>
<td>8.33%</td>
</tr>
<tr>
<td>12</td>
<td>3.125%</td>
<td>3.125%</td>
</tr>
<tr>
<td>15</td>
<td>6.25%</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4.167%</td>
<td>4.167%</td>
</tr>
<tr>
<td>30</td>
<td>4.167%</td>
<td>4.167%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[co]</th>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8.335%</td>
<td>8.335%</td>
</tr>
<tr>
<td>16</td>
<td>5.557%</td>
<td>5.557%</td>
</tr>
<tr>
<td>32</td>
<td>5.554%</td>
<td>5.553%</td>
</tr>
<tr>
<td>12</td>
<td>3.125%</td>
<td>3.125%</td>
</tr>
<tr>
<td>15</td>
<td>6.25%</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.0%</td>
<td>6.25%</td>
</tr>
<tr>
<td>30</td>
<td>0.0%</td>
<td>6.25%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[fco]</th>
<th>total job size</th>
<th>number of job components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8.335%</td>
<td>8.335%</td>
</tr>
<tr>
<td>16</td>
<td>8.335%</td>
<td>8.335%</td>
</tr>
<tr>
<td>32</td>
<td>0.0%</td>
<td>16.66%</td>
</tr>
<tr>
<td>12</td>
<td>6.25%</td>
<td>6.25%</td>
</tr>
<tr>
<td>15</td>
<td>12.5%</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.0%</td>
<td>12.5%</td>
</tr>
<tr>
<td>30</td>
<td>0.0%</td>
<td>12.5%</td>
</tr>
</tbody>
</table>
7.5 Simulations based on detailed application measurements

In this section we present simulation results for ordered and total requests based on detailed measurement data obtained when running the Poisson application on the DAS. While in the previous section we measured and introduced in the simulations only the total execution times, here we perform detailed measurements of the runtime and communication time of the application. We include these measurements in our simulations, in which we explicitly model the structure of the application and the inter- and intra-cluster communication. In this section we apply our first way of including communication in the simulations, as described in Section 2.4.

7.5.1 Measurements

For the same grid size and the same initial data, we ran the Poisson application on a single DAS cluster with 8 processes (4x2 and 8x1 rectangles) and 16 processes (2x8 and 4x4), and on the 4-cluster DAS for configurations 4x2 and 4x4, scheduling an equal number of processes on each cluster. The grid size was 100x100. In Table 7.10 we report the number of iterations needed to reach convergence, the duration of the computation steps, and the total time needed for exchanging borders; the time cost of diffusing the local errors and computing the global error is always about 0.014 s. The measurements described in this section were performed on the DAS1 system.

<table>
<thead>
<tr>
<th>configuration</th>
<th>number of iterations</th>
<th>update (ms)</th>
<th>exchange borders (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>single cluster</td>
</tr>
<tr>
<td>4x2</td>
<td>2436</td>
<td>0.953-0.972</td>
<td>0.408-0.450</td>
</tr>
<tr>
<td>8x1</td>
<td>2418</td>
<td>0.970-0.994</td>
<td>0.260-0.315</td>
</tr>
<tr>
<td>4x4</td>
<td>2132</td>
<td>0.480-0.515</td>
<td>0.350-0.425</td>
</tr>
<tr>
<td>8x2</td>
<td>2466</td>
<td>0.470-0.525</td>
<td>0.337-0.487</td>
</tr>
</tbody>
</table>

7.5.2 Simulation results

We use the Poisson application structure and the measurement data obtained from the DAS for job sizes 8 and 16 (cases 4x2 and 4x4, respectively) in simulations of a single cluster of size 32, and a multic和平 with 4 clusters of size 8. The distribution of the job sizes is an equal mix of sizes 8 and 16, and of sizes (2,2,2,2) and (4,4,4,4), in the single cluster and the multic和平, respectively. The co-allocation on the DAS corresponds to the case of ordered requests in our simulations. Since
the job-component sizes are equal, identical performance would be obtained for unordered requests.

The results are presented in Figure 7.14 (we set the number of iterations to 10% of the values in Table 7.10). The response time values are depicted against the gross utilization. Since for the multiclient simulations the components of the requests are of equal sizes, which sum up to the two values of the request sizes used for the single-cluster case, and the total size of the multiclient is equal to the size of the single cluster, the two cases would have identical performance in the absence of communication. This implies that the worse performance displayed by the ordered requests compared to the total requests is caused by the slow intercluster communication and not by the rigidity of the ordered requests.

![Figure 7.14](image.png)

Figure 7.14: The average response time as a function of the utilization for ordered and total requests, with data obtained from the Poisson application on the DAS.

### 7.5.3 Conclusions

In this section we have shown our simulation results for a multiclient with data obtained from running the Poisson application on the DAS. Communication was directly introduced in simulations via detailed measurements. The grid size considered is small, so the fraction of the service time spent on communication is very large, which makes this case very unfavourable for co-allocation. We conclude that:

- **The amount of communication deteriorates the performance of co-allocation, when compared to a single-cluster system.** A way to improve the performance in this case is to reduce the fraction of the service time represented by communication, for example by running the application on fewer processors.

- Taking into account that the chosen setup is quite unfavourable to co-allocation, the increase in response time does not seem prohibitive. Therefore, **in practice the performance penalty of running multiclient jobs may not preclude co-allocation.**
Chapter 8

Co-allocation versus no co-allocation

We have previously assessed the performance of co-allocation in multiclusters depending on a variety of parameters and workloads. Of course, when the job size is larger than the cluster size, unless the restructuring of the job on fewer processors is desired, co-allocation is the only option. In such a case, the remaining question is how to perform co-allocation to make it the most advantageous for the job and for the system. The previous chapters provided answers to this question, but they also assessed the performance of co-allocation for cases when jobs could fit on a single cluster.

In this chapter we gather all our results of comparing co-allocation with no co-allocation. We only consider now the case when both options are valid, i.e., when jobs do fit on one cluster, to find out whether co-allocation can also be beneficial in such circumstances. We consider a multicluster system with 4 clusters of 32 processors each. The job-component-size distribution is $D(0.9)$ on $[1, 8]$ and the scheduling policy is GS with FCFS.

8.1 Unordered requests versus flexible requests

In this section, we compare the performance of flexible requests with that of unordered requests with different numbers of components. Each component is either from the distribution $D(0.9)$ on $[1, 8]$, or is a sum of values from that distribution, in such a way that the jobs generated for the different request types and numbers of components have the same total request sizes. For example, for the flexible requests the total request size is the sum of four copies of $D(0.9)$. The flexible jobs are placed with CF, and for the unordered jobs FF is used. For the results in this section communication was not considered.

Figure 8.3 shows that the best performance is obtained when the scheduler is allowed to choose the size of the components when co-allocating the jobs—flexible
requests and the worst when co-allocation is not allowed—the case of jobs with 1 component.

Since for many distributed applications the performance depends on the sizes of the components, users might sometimes be reluctant to submit their jobs as flexible requests. This is when co-allocation with unordered requests becomes an attractive option, combining the advantage of a good performance with the fact that it allows the users to choose the sizes of the components when submitting jobs. As the graph shows, the performance is slightly better for jobs with 2 components than for 4 components, which suggests that a number of components smaller than the number of clusters is to be preferred. Inter-cluster communication can deteriorate the performance of multi-component jobs, but, as our results in Chapter 3 have shown, for systems such as the DAS with reasonably fast inter-cluster links co-allocation remains a good choice in many cases.

![Graph](image)

Figure 8.1: The average response time as a function of the utilization for flexible requests, and unordered requests with different numbers of components.

## 8.2 The capacity loss

In this section we consider a multicluster with two cases of unordered jobs and WF placement. In either case, the (total) request sizes are obtained as the sum of 4 components generated from $D(0.9)$, but in one case, there are four job components, while in the other, the numbers are added to have a single job component. Here we use the second simulation method described in Section 3.4. In this method we suppose that the queue is always long enough when a job departs that we can schedule jobs until the next one does not fit. We only generate new jobs at departure times, and only one more than fits on the system.

As the results in Figure 8.2 indicate, rather than waiting for enough idle processors in one cluster, spreading requests over multiple clusters brings better performance. However, we do point out that with co-allocation, jobs will experience a slowdown due to wide-area communications, which is not accounted for here. For
8.3 Communication represented by the communication-speed ratio

A workload similar to ours, co-allocation is a viable choice for systems with fast inter-cluster communication and/or for applications with relatively little wide-area communication. The applications can waste about 14% of their service time on communication before co-allocation gets worse than no-co-allocation.

![Graph showing capacity loss comparison with and without co-allocation](image)

Figure 8.2: A comparison of the capacity loss when co-allocation is or is not employed; with co-allocation, jobs requests are unordered.

8.3 Communication represented by the communication-speed ratio

In this section we show that despite the extra communication, rather than waiting for enough idle processors in one cluster, spreading requests over clusters can bring better performance. Communication is included as an all-to-all personalized message exchange. In the co-allocation case we schedule flexible requests using LB-A; in the no-co-allocation case we have jobs consisting of a single component. The request sizes are obtained as the sum of 4 components generated from $D(0.9)$. Note that LB-A is not very favourable for co-allocation, as it may entail much time spent in inter-cluster communication (see Section 5.3).

Figure 8.3 (top) compares co-allocation with flexible requests with no co-allocation with single-component jobs using FF (the results for WF are slightly worse). At low utilizations avoiding co-allocation can bring benefits when the communication-speed ratio is large. However, for intermediate and high system loads, co-allocation is significantly better, even for communication-speed ratios larger than 50. We conclude that co-allocation is a good choice for systems with communication-speed ratios that are not extremely large. We only discuss here the gross utilization, and one should be aware that there is some extra utilization that is lost for the application due to the communication.

We studied further why co-allocation gives better results by looking at the way the scheduler works. The scheduler only makes an attempt at scheduling jobs when a job departs and the queue is not empty, and when a job arrives at an
empty system. We call a scheduling attempt unsuccessful when no job can be scheduled. Figure 8.3 (bottom) shows the percentages of unsuccessful scheduling attempts with and without co-allocation, with a communication-speed ratio of 50, as a function of the utilization (the graph stops just before the no-co-allocation case becomes unstable). Especially at average and high utilizations, in the absence of co-allocation, it occurs much more often that no job can be scheduled at a job departure.

Table 8.1 compares the maximal utilizations and the coefficients of variation of the service times for co-allocation with flexible requests and LB-A placement and no co-allocation with FF placement. The communication speed ratios are the same as those included in Figure 8.3. We conclude that the maximal net utilization for the communication and application structures considered is higher when co-allocation is employed for communication-speed ratios not higher than 19. For higher communication speed ratios the gross maximal utilization is still higher in the presence of co-allocation, but more and more of that utilization is
spent on communication.

Table 8.1: The maximal utilizations and the coefficients of variation of the service time for flexible requests with LB-A placement (top) and unordered single-component jobs with FF placement (bottom).

<table>
<thead>
<tr>
<th>communication speed ratio</th>
<th>max. util.</th>
<th>cv of serv. time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>net</td>
<td>gross</td>
</tr>
<tr>
<td>5</td>
<td>0.874</td>
<td>0.935</td>
</tr>
<tr>
<td>19.0</td>
<td>0.740</td>
<td>0.936</td>
</tr>
<tr>
<td>50.0</td>
<td>0.552</td>
<td>0.936</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>max. util.</th>
<th>cv of serv. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>gross</td>
</tr>
<tr>
<td>0.740</td>
<td>0.752</td>
</tr>
</tbody>
</table>

8.4 Communication included through runtime measurements

In this section we compare co-allocation with no co-allocation with the third way of including communication, as described in Section 2.4. As Figure 7.10 shows, in a large number of cases co-allocation can enhance the performance of a multicluster system, but it is necessary to avoid the simultaneous presence in the system of jobs with conflicting requirements. We have shown in Chapter 4 that large single-component jobs and jobs with many components deteriorate the system performance. Moreover, combining such jobs makes it even worse, which is also confirmed by Figure 7.10. For LS and LP it is enough to avoid large single-cluster jobs to make co-allocation worthwhile. Since LS stores multi-component jobs in all local queues, it provides (compared to the other policies) more flexibility and a larger choice—any of the jobs at the top of the queues—at each moment when a scheduling decision has to be taken. This is why avoiding jobs with many components does not influence the performance much. LP keeps all multi-component jobs in the global queue and the jobs with many components, which are more difficult to fit, impact the performance more. This can be concluded from the significant improvement brought by the [fco] restrictions compared to the [rco] ones. GS, as mentioned before, has good performance in the absence of communication due to the fact that it can run jobs from the single queue on any of the clusters. However, the same single queue makes co-allocation without restrictions ([fco]) perform poorly, and only when both the numbers and sizes of the components are restricted ([fco]) is co-allocation an advantage.
Chapter 8. Co-allocation versus no co-allocation
Appendix A

Communication-time measurements

In this appendix we measure the communication time needed for exchanging the values of border grid points in the Poisson application with processor configuration 4x4. All numbers presented below are averages over ten runs.

A.1 Per-process communication time for exchanging borders

We measure for each individual process(or) the total time (i.e., across all iterations) it spends exchanging borders with other processes. Figure A.1 contains the results for different numbers of clusters and grid sizes 100x100 and 4000x4000. As expected, for Globus-2, the processes at the edges of the clusters (i.e., the processes that need to communicate across cluster borders) need more time to communicate, although for some reason also some other processes take much time to communicate. This effect is relatively speaking much larger for grid size 100x100 than for grid size 4000x4000.

For Globus-2 with grid size 100x100 we also measure for each individual process, and in each direction, the total time it spends exchanging borders with other processes; the results are presented in Figure A.2. We see that in the cross-cluster directions left and right, receiving border information takes a relatively large amount of time.

A.2 Synchronized and non-synchronized operation

In our original Poisson application, we do not synchronize processes before they start their communication phases. So then, as soon as a process finishes its com-
putation in an iteration, it starts (trying) to communicate. We added an MPI command to our application in order to enable synchronized operation. Then, all processes synchronize in every iteration before the communication starts, so they all start communicating at (about) the same time. In both synchronized and non-synchronized operation, we measure the communication time in an iteration as the time elapsed between the last process finishing its computation phase and the last process finishing its communication phase. The cause of the difference between these communication times in these two modes of operation lies in the potential parallelism of computation and communication in non-synchronized operation. In general, in this mode of operation, the communication time is smaller, as we will indeed see in Section A.3.

A.3 Total communication time for exchanging borders

We define the total communication time for exchanging borders as the sum of the communication times of all iterations, both for synchronized and non-synchronized operation. In Table A.1, we show the minimum, average, and maximum (across ten runs) total communication time for synchronized operation; the variation is in general not very large. The difference between single-cluster and multicluster performance (and between PBS and Globus-1) is very large.

Table A.2 presents the (average) total communication times for exchanging borders when processes are or are not synchronized before communication. (This table contains the average results of Table A.1.) In Figure A.3 we depict the average total communication times from Table A.2 after normalization with respect to PBS. We find that with synchronized operation, communication in a single cluster is, depending on the grid size, 10–35 times faster than multicluster communication, while for non-synchronized operation (and realistic grid sizes), this factor is reduced to about 13. In addition, in Table A.2 we see that for large grid sizes the performance of multicluster communication strongly improves when the processes are not synchronized, but that this is not the case for a single cluster with PBS.

A.4 Data transfer rate of exchanging borders

We use the results of Table A.1 (with synchronized operation) to calculate the data transfer rate when exchanging borders. For this calculation we assume that the slowest communicating process is always an interior process with four borders to exchange. Since the processor configuration is 4x4, the number of grid points to communicate per iteration by an interior process (send and receive) is twice the side of the grid. For a grid point 8 bytes are reserved. In Figure A.4 we see that for PBS the data transfer rate strongly increases when the amount of data to be communicated increases. The highest data transfer rate for PBS is 35 Mbyte/s for grid size 4000x4000, while for multicluster execution (Globus-2 and Globus-4) the
highest data transfer rate of over 3 Mbyte/s is reached for a grid size of 2000x2000.

<table>
<thead>
<tr>
<th>Grid Size 100x100</th>
<th>Grid Size 4000x4000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PBS</strong></td>
<td><strong>Globus-1</strong></td>
</tr>
<tr>
<td>0.20 0.28 0.25 0.24</td>
<td>0.69 1.13 1.11 1.09</td>
</tr>
<tr>
<td>0.26 0.33 0.30 0.30</td>
<td>1.17 1.53 1.60 1.31</td>
</tr>
<tr>
<td>0.28 0.35 0.31 0.31</td>
<td>1.60 1.71 1.82 1.67</td>
</tr>
<tr>
<td>0.24 0.29 0.27 0.25</td>
<td>1.40 1.47 1.50 1.25</td>
</tr>
<tr>
<td><strong>Globus-2</strong></td>
<td><strong>Globus-4</strong></td>
</tr>
<tr>
<td>5.29 6.44 6.87 0.89</td>
<td>7.13 8.64 7.21 8.37</td>
</tr>
<tr>
<td>0.68 6.83 6.66 1.12</td>
<td>1.98 5.88 2.20 8.20</td>
</tr>
<tr>
<td>1.06 12.9 11.5 5.66</td>
<td>3.96 7.23 8.07 9.87</td>
</tr>
<tr>
<td>0.74 5.72 7.41 5.83</td>
<td>0.64 4.02 2.49 4.95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PBS</th>
<th>Globus-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.25 3.71 3.79 3.72</td>
<td>78 87 99 88</td>
</tr>
<tr>
<td>3.69 4.17 4.30 4.18</td>
<td>76 86 103 90</td>
</tr>
<tr>
<td>3.50 3.98 3.92 3.92</td>
<td>79 92 87 77</td>
</tr>
<tr>
<td>3.49 3.93 4.04 3.96</td>
<td>78 90 86 79</td>
</tr>
<tr>
<td><strong>Globus-2</strong></td>
<td><strong>Globus-4</strong></td>
</tr>
<tr>
<td>56 107 137 81</td>
<td>96 123 132 111</td>
</tr>
<tr>
<td>54 107 134 83</td>
<td>84 108 108 99</td>
</tr>
<tr>
<td>75 117 128 96</td>
<td>113 130 123 124</td>
</tr>
<tr>
<td>68 118 112 91</td>
<td>108 121 111 120</td>
</tr>
</tbody>
</table>

Figure A.1: The total per-process communication times for different grid sizes (in seconds).
Appendix A. Communication-time measurements

<table>
<thead>
<tr>
<th>Top</th>
<th></th>
<th></th>
<th></th>
<th>Bottom</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.31</td>
<td>0.11</td>
<td>0.15</td>
<td>0.11</td>
<td>0.13</td>
<td>0.12</td>
<td>0.11</td>
<td>0.13</td>
</tr>
<tr>
<td>0.23</td>
<td>0.27</td>
<td>0.44</td>
<td>0.29</td>
<td>0.22</td>
<td>0.20</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>0.75</td>
<td>5.34</td>
<td>4.83</td>
<td>0.21</td>
<td>0.20</td>
<td>0.14</td>
<td>0.13</td>
<td>0.51</td>
</tr>
<tr>
<td>0.14</td>
<td>0.11</td>
<td>0.12</td>
<td>0.12</td>
<td>0.33</td>
<td>0.25</td>
<td>0.33</td>
<td>0.25</td>
</tr>
<tr>
<td>Left</td>
<td></td>
<td></td>
<td></td>
<td>Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>6.30</td>
<td>0.40</td>
<td>0.09</td>
<td>0.10</td>
<td>0.54</td>
<td>6.12</td>
<td>0.56</td>
</tr>
<tr>
<td>0.02</td>
<td>6.66</td>
<td>0.27</td>
<td>0.09</td>
<td>0.12</td>
<td>0.53</td>
<td>6.30</td>
<td>0.59</td>
</tr>
<tr>
<td>0.06</td>
<td>5.98</td>
<td>0.11</td>
<td>0.09</td>
<td>0.10</td>
<td>0.13</td>
<td>6.95</td>
<td>4.51</td>
</tr>
<tr>
<td>0.07</td>
<td>6.03</td>
<td>0.13</td>
<td>0.09</td>
<td>0.14</td>
<td>0.15</td>
<td>6.98</td>
<td>4.97</td>
</tr>
</tbody>
</table>

Figure A.2: The total per-process communication times in each of the four directions for Globus-2 and grid size 100x100 (in seconds).

Table A.1: The total communication time for exchanging borders (in seconds) with synchronized operation for processor configuration 4x4.

<table>
<thead>
<tr>
<th>grid size</th>
<th>number of iterations</th>
<th>PBS</th>
<th></th>
<th></th>
<th>Globus-1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>min.</td>
<td>avg.</td>
<td>max.</td>
<td>min.</td>
<td>avg.</td>
<td>max.</td>
</tr>
<tr>
<td>50 x 50</td>
<td>865</td>
<td>0.15</td>
<td>0.16</td>
<td>0.19</td>
<td>0.52</td>
<td>0.62</td>
<td>0.87</td>
</tr>
<tr>
<td>100 x 100</td>
<td>2132</td>
<td>0.37</td>
<td>0.39</td>
<td>0.44</td>
<td>1.41</td>
<td>1.51</td>
<td>1.67</td>
</tr>
<tr>
<td>200 x 200</td>
<td>3570</td>
<td>0.69</td>
<td>0.72</td>
<td>0.74</td>
<td>2.90</td>
<td>3.04</td>
<td>3.56</td>
</tr>
<tr>
<td>400 x 400</td>
<td>3814</td>
<td>0.89</td>
<td>0.91</td>
<td>0.95</td>
<td>4.49</td>
<td>4.68</td>
<td>4.91</td>
</tr>
<tr>
<td>1000 x 1000</td>
<td>4347</td>
<td>2.09</td>
<td>2.17</td>
<td>2.31</td>
<td>9.49</td>
<td>10.2</td>
<td>11.8</td>
</tr>
<tr>
<td>2000 x 2000</td>
<td>4387</td>
<td>4.21</td>
<td>4.38</td>
<td>4.67</td>
<td>27.1</td>
<td>29.1</td>
<td>32.3</td>
</tr>
<tr>
<td>4000 x 4000</td>
<td>2644</td>
<td>4.50</td>
<td>4.88</td>
<td>5.47</td>
<td>127</td>
<td>134</td>
<td>139</td>
</tr>
<tr>
<td>10000 x 10000</td>
<td>2644</td>
<td>11.9</td>
<td>12.3</td>
<td>13.1</td>
<td>127</td>
<td>162</td>
<td>202</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grid size</th>
<th>number of iterations</th>
<th>Globus-2</th>
<th></th>
<th></th>
<th>Globus-4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>min.</td>
<td>avg.</td>
<td>max.</td>
<td>min.</td>
<td>avg.</td>
<td>max.</td>
</tr>
<tr>
<td>50 x 50</td>
<td>865</td>
<td>3.81</td>
<td>4.67</td>
<td>5.33</td>
<td>4.82</td>
<td>4.96</td>
<td>5.13</td>
</tr>
<tr>
<td>100 x 100</td>
<td>2132</td>
<td>9.25</td>
<td>11.8</td>
<td>15.1</td>
<td>12.0</td>
<td>12.2</td>
<td>13.7</td>
</tr>
<tr>
<td>200 x 200</td>
<td>3570</td>
<td>15.0</td>
<td>19.8</td>
<td>23.0</td>
<td>20.1</td>
<td>20.3</td>
<td>20.5</td>
</tr>
<tr>
<td>400 x 400</td>
<td>3814</td>
<td>18.0</td>
<td>22.5</td>
<td>29.4</td>
<td>17.5</td>
<td>20.8</td>
<td>22.6</td>
</tr>
<tr>
<td>1000 x 1000</td>
<td>4347</td>
<td>22.3</td>
<td>28.8</td>
<td>36.5</td>
<td>26.1</td>
<td>29.2</td>
<td>32.1</td>
</tr>
<tr>
<td>2000 x 2000</td>
<td>4387</td>
<td>41.3</td>
<td>45.1</td>
<td>49.6</td>
<td>41.2</td>
<td>42.8</td>
<td>45.8</td>
</tr>
<tr>
<td>4000 x 4000</td>
<td>2644</td>
<td>154</td>
<td>161</td>
<td>171</td>
<td>167</td>
<td>179</td>
<td>206</td>
</tr>
<tr>
<td>10000 x 10000</td>
<td>2644</td>
<td>186</td>
<td>267</td>
<td>398</td>
<td>290</td>
<td>358</td>
<td>427</td>
</tr>
</tbody>
</table>
Table A.2: The total communication time for exchanging borders (in seconds) with synchronized and non-synchronized operation for processor configuration 4x4.

<table>
<thead>
<tr>
<th>grid size</th>
<th>number of iterations</th>
<th>PBS synchronization</th>
<th>Globus-1 synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>50x50</td>
<td>865</td>
<td>0.16</td>
<td>0.11</td>
</tr>
<tr>
<td>100x100</td>
<td>2132</td>
<td>0.39</td>
<td>0.28</td>
</tr>
<tr>
<td>200x200</td>
<td>3570</td>
<td>0.72</td>
<td>0.54</td>
</tr>
<tr>
<td>400x400</td>
<td>3814</td>
<td>0.91</td>
<td>0.88</td>
</tr>
<tr>
<td>1000x1000</td>
<td>4347</td>
<td>2.17</td>
<td>2.00</td>
</tr>
<tr>
<td>2000x2000</td>
<td>4387</td>
<td>4.38</td>
<td>3.59</td>
</tr>
<tr>
<td>4000x4000</td>
<td>2644</td>
<td>4.88</td>
<td>5.26</td>
</tr>
<tr>
<td>10000x10000</td>
<td>2644</td>
<td>12.3</td>
<td>12.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>grid size</th>
<th>number of iterations</th>
<th>Globus-2 synchronization</th>
<th>Globus-4 synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>50x50</td>
<td>865</td>
<td>4.67</td>
<td>2.02</td>
</tr>
<tr>
<td>100x100</td>
<td>2132</td>
<td>11.8</td>
<td>4.16</td>
</tr>
<tr>
<td>200x200</td>
<td>3570</td>
<td>19.8</td>
<td>7.09</td>
</tr>
<tr>
<td>400x400</td>
<td>3814</td>
<td>22.5</td>
<td>9.30</td>
</tr>
<tr>
<td>1000x1000</td>
<td>4347</td>
<td>28.8</td>
<td>16.8</td>
</tr>
<tr>
<td>2000x2000</td>
<td>4387</td>
<td>45.1</td>
<td>18.5</td>
</tr>
<tr>
<td>4000x4000</td>
<td>2644</td>
<td>161</td>
<td>24.3</td>
</tr>
<tr>
<td>10000x10000</td>
<td>2644</td>
<td>267</td>
<td>62.5</td>
</tr>
</tbody>
</table>
Figure A.3: The total communication time for exchanging borders with synchronized (top) and non-synchronized (bottom) operation for processor configuration 4x4, normalized with respect to PBS.

Figure A.4: The data-transfer rate when exchanging borders.
Appendix B

Simulation details

In this appendix we provide implementation details for our simulations of the complete queuing model with Poisson arrivals.

The simulation programs were implemented using the CSIM simulation package [41]. In most of the graphs depicting the average response time as a function of the utilization of the system we include confidence intervals—an interval in which the exact value lies with a high probability, which is called the confidence level. All the confidence intervals in our simulations are at the 95%-level.

In our simulations we use synthetic distributions for the interarrival times, the job-component sizes, the service-times, and for selecting the request types of jobs. Each of those distributions employ the standard CSIM random number generator. In order to avoid correlations among the values generated for the four different variables, we choose a distinct non-zero seed for each of the corresponding distributions. In all our simulations we maintained the same seeds to make the results comparable.

The (pseudo-) random number generator implemented by CSIM is the UNIX random number generator, a mixed linear-congruential generator with the formula $x_n = (1, 103, 515, 245 \cdot x_{n-1} + 12, 345) \mod 2^{31}$. It has a full period of $2^{31}$ but displays a cyclic behaviour at the level of the low-order bits [30]. For full period generators all non-zero seeds are equally good, in the sense that they do not influence the period of the generator.
Bibliography


Acknowledgements

I would like to thank my supervisor Henk Sips, and my co-supervisor Dick Epema for their support and contribution to the elaboration of this thesis. I would also like to thank Steve Banen for his contribution to the application measurements.

Many thanks to Paulo Anita for promptly and patiently solving all my computer-support related problems, and to Richard Vdovjak and Jonne Zutt for taking the time to read chapters of my thesis, and for their many useful comments. Finally, I would also like to thank Richard Vdovjak for the original cover he designed for my thesis.
Summary
Performance Analysis of Processor Co-Allocation in Multicluster Systems
Anca Bucur

Building multicluster systems out of multiple, geographically distributed clusters interconnected by high-speed wide-area networks can provide access to a larger computational power and to a higher number and a wider range of resources. Instead of smaller groups of users with exclusive access to their single clusters, larger groups of users can share the multicluster. This potentially leads to lower job turnaround times and higher system utilizations, and makes larger job sizes possible by allowing jobs to use processors in multiple clusters simultaneously.

The possibility of creating multiclusters fits with the recent interest in computational and data grids, in which it is envisioned that applications can access resources (hardware resources such as processors, memory, and special instruments, but also data resources) in many different and widely dispersed locations at the same time to accomplish their goals. Applications running on multiclusters, and more generally, in grids, may then require (processor) co-allocation, i.e., the simultaneous allocation of resources (processors) in different clusters or subsystems of a grid.

There are two problems that may occur when employing processor co-allocation in multicluster systems. First, applications may not be well suited for multicluster execution because they deal poorly with the relatively slow wide-area links interconnecting the component clusters, which degrades their performance too much to justify such execution. Second, scheduling policies for such systems have to consider more restrictions than schedulers for single clusters in that components of single jobs may need to fit in separate clusters. In this thesis we focus on both problems, and assess with simulations the performance of scheduling policies employing processor co-allocation in multiclusters. We consider a wide variety of parameter settings, and design and evaluate six scheduling policies for co-allocation, some of which with multiple variations.

Our test-bed for studying co-allocation is the five-cluster 400-processor second-generation Distributed ASCI Supercomputer (DAS). In the most general setting, grids and grid resources are very heterogeneous; in this thesis we restrict ourselves to homogeneous multicluster systems such as the DAS and to processor co-allocation. Showing the viability of processor co-allocation in such systems may be regarded as a first step in assessing the benefit of co-allocation of various types of resources in more general grid environments.

In Chapter 2 we describe our model of multicluster systems, the workloads,
the queuing disciplines, and the scheduling policies. The system consists of a set of clusters of possibly different sizes with fast intra-cluster links and slow inter-cluster links. We assume jobs to be rigid and to be scheduled by policies employing space sharing. We allow different request types in that jobs may specify only the total number of processors they need—flexible requests—or the numbers of processors needed in the separate clusters—unordered requests—or in each of the clusters—ordered requests—of the multic和平 system. For comparison with the single-cluster case, we introduce total requests, which only specify the total number of processors needed, equal to the number required by (un)ordered requests, but this time in a single cluster. We design our scheduling policies for three different queuing structures: only a global queue, only local queues, and both global and local queues, in which case single-component jobs go to the local queues and multi-component ones to the global queue.

Chapter 3 assesses the maximal utilization when co-allocating jobs in multi-cluster systems with analytic means and with simulations, using both synthetic workloads and workloads obtained from the logs of actual systems.

In Chapter 4 we simulate multic和平 systems without communication, and evaluate the performance of co-allocation for different structures and sizes of job requests, different numbers and sizes of clusters, and two queuing disciplines. The scheduling policy employed in this chapter uses a single global queue.

In Chapter 5 we maintain the scheduling policy with a single global queue and introduce communication. We simulate multic和平 systems with the FCFS queuing discipline and with several policies for placing flexible requests, under workloads consisting of a single or of a mix of request types and for different communication-speed ratios.

In Chapter 6 we assess the response times of both single- and multi-component jobs in multic和平 systems for the six scheduling policies with different queuing structures defined in our model and for synthetic workloads. We vary the numbers of queues, the priority orders in which jobs are scheduled from multiple queues, and the fractions of jobs with different numbers of components.

In Chapter 7 we restrict ourselves to three representative scheduling policies for co-allocation in multic和平s, one for each queuing structure. We use workloads derived from a DAS log and from runtime measurements of single-cluster and multic和平 execution of two parallel applications on the DAS.

Chapter 8 compares systems with and without co-allocation, to derive sets of values for various parameters for which co-allocation is beneficial.

Our results allow us to conclude that although co-allocation makes scheduling more difficult and the wide-area communication critically impacts the performance, there is a wide range of realistic applications that may benefit from co-allocation. We also conclude that unrestricted co-allocation is not recommended and that a large number of parameters have to be taken into account in order to obtain good performance.
Samenvatting

Prestatie-analyse van Processorcoallocatie in Multiclustersystemen

Anca Bucur

Het creëren van multiclustersystemen bestaande uit meerdere geografisch gespreide clusters verbonden door snelle netwerken kan meer rekencapaciteit en grotere hoeveelheden en typen computer resources beschikbaar maken. In plaats van kleinere groepen gebruikers met ieder exclusieve toegang tot hun eigen cluster, kunnen grotere groepen gebruikers een multicluster delen. Dit kan leiden tot lagere responstijden van jobs en tot hogere systeembelastingen, en laat grotere jobs toe door ze tegelijkertijd processoren in verscheidene clusters te laten gebruiken.

De mogelijkheid van multiclasters past goed bij de recente belangstelling voor computationele en data-grids, waarin wordt voorzien dat applicaties resources (hardware-componenten zoals processoren, geheugens, en speciale instrumenten, maar ook gegevens) zullen gebruiken in vele verschillende, wijd verspreide locaties tegelijkertijd. Applicaties op multiclasters, en meer in het algemeen in grids, kunnen dan om (processor-)coallocatie verzoeken, d.w.z., om de gelijktijdige allocatie van resources (processoren) in verschillende clusters of deelstystemen van een grid.

Er kunnen zich twee problemen voordoen bij het gebruik van processorcoallocatie in multiclustersystemen. Ten eerste kunnen applicaties niet erg geschikt zijn voor uitvoering op multiclasters omdat ze slecht omgaan met de relatief langzame lange-afstandverbindingen tussen de clusters. Ten tweede moeten scheduling policies voor dergelijke systemen met meer beperkingen rekening houden dan schedulers voor één cluster, aangezien componenten van individuele jobs wellicht in verschillende clusters moeten passen. In dit proefschrift worden beide problemen behandeld, en wordt de prestatie geëvalueerd van scheduling policies die gebruik maken van processorcoallocatie in multiclusters.

Ons test-systeem voor het bestuderen van coallocatie is de uit vijf clusters en in totaal 400 processoren bestaande tweede-generatie Distributed ASCI Supercomputer (DAS). In hun meest algemene verschijningsvorm zijn grids en grid resources erg heterogeen; in dit proefschrift beperken we ons tot homogene multiclustersystemen zoals de DAS, en tot processorcoallocatie. Het aantonen van de levensvatbaarheid van processorcoallocatie in zulke systemen kan worden beschouwd als een eerste stap op weg naar het beoordelen van het nut van de coallocatie van verschillende typen resources in meer algemene grid-omgevingen.

In Hoofdstuk 2 beschrijven we ons model van multiclustersystemen, van de werklasten, van de wachtrij-disciplines, en van de scheduling policies. Het systeem bestaat uit een verzameling van clusters van mogelijkerwijs verschillende
Samenvatting

afmetingen met snelle interne, en langzame onderlinge verbindingen. We nemen aan dat jobs star zijn, en dat ze hun procesoren exclusief krijgen toegewezen. We ondertekenen verschillende typen verzoeken: jobs kunnen alleen het totale aantal benodigde procesoren specificeren—flexibele verzoeken—or de aantallen procesoren die ze nodig hebben in verschillende maar niet nader bepaalde clusters—ongeordende verzoeken—or in ieder van de clusters—geordende verzoeken—van de multicore.

De identifieking met het speciale geval van slechts één cluster introduceer we ook totale verzoeken, die alleen het totale aantal benodigde procesoren specifice ren, gelijk aan het door (on)geordende verzoeken gevraagde aantal, maar nu in slechts één cluster. Onze scheduling policies zijn ontworpen voor drie verschillende structuren van de wachtlijsten in het systeem: alleen een globale wachtrij, alleen lokale wachtrijen, en zowel een globale als lokale wacht rijken. In het laatste geval worden de jobs met één component naar de locale wachtrijen gestuurd, en de jobs met meerdere componenten naar de globale wachtrij.

In Hoofdstuk 3 wordt de maximale systeembelasting bij coallocatie van jobs in multicore-systemen onderzocht met analytische methoden en met simulaties, en met een werklast die ofwel synthetisch is, ofwel is verkregen uit observaties van bestaande systemen.

In Hoofdstuk 4 simuleren we multicore-systemen zonder communicatie, en evaluëren we coallocatie voor verschillende structuren en afmetingen van job-verzoeken, verschillende aantallen en afmetingen van clusters, en twee wachtrij-disciplines. De scheduling policy in dit hoofdstuk heeft alleen een globale wachtrij.

In Hoofdstuk 5 handhaven we de scheduling policy met alleen een globale wachtrij, en introduceren we communicatie. We simuleren multicore-systemen met de FCFS discipline en met verschillende manieren om flexibele verzoeken in het systeem te plaatsen, met werklasten die bestaan uit één, of uit een mengeling van typen verzoeken, en met verschillende relatieve snelheden van de lokale netwerken en het lange-afstandsnetwork.

In Hoofdstuk 6 gaan we de responstijd na van jobs met één of meerdere componenten voor de zes scheduling policies met de verschillende wachtrij-structuren die in ons model gedefinieerd zijn, en voor synthetische werklasten. We variëren de aantallen wachtrijen, de prioriteiten waarmee jobs in de verschillende wachtrijen worden bediend, en de fracties van jobs met verschillende aantallen componenten.

In Hoofdstuk 7 beperken we ons tot drie representatieve scheduling policies voor coallocatie in multicore-systemen. We gebruiken werklasten afgeleid van de DAS en van metingen van de verwerkingstijden van twee parallelle applicaties op één of meerdere DAS-clusters.

Hoofdstuk 8 vergelijkt systemen met en zonder coallocatie om waarden van parameters van het model te bepalen waarvoor coallocatie voordelig is.

Onze resultaten leiden ons tot de conclusie dat hoewel coallocatie scheduling moeilijk maakt en de communicatie tussen de clusters een kritieke invloed heeft op de prestatie, er een groot scala aan applicaties bestaat die hun voordeel kunnen doen met coallocatie. We concluderen ook dat onbeperkte coallocatie niet aan te bevelen is, en dat een groot aantal parameters in de beschouwing moet worden betrokken om een goede prestatie te verkrijgen.
List of publications


7th Workshop on Job Scheduling Strategies for Parallel Processing, volume 2221 of LNCS, pages 66-86, 2001.


Curriculum vitae

Anca Bucur was born in Bucharest, Romania, on March 27, 1973. She graduated from the Colegiul Național Șfântul Sava in 1991, and in the same year she started her studies in Computer Science at the Politehnica University of Bucharest, at the Faculty of Computer Science and Control Engineering. She graduated in 1996 as a licensed engineer, and received her Master's degree in 1997, at the same university. Between 1996 and 1998 she was a teaching assistant at the Faculty of Computer Science and Control Engineering of the Politehnica University of Bucharest. In 1999 Anca Bucur joined the Parallel and Distributed Systems Group of Delft University of Technology as a PhD student. Since October 2003 she is a scientist in the Software Architecture Group at Philips Research in Eindhoven.