Communication-aware job placement policies for the KOALA grid scheduler

DOI:
10.1109/E-SCIENCE.2006.261163

Communication-Aware Job Placement Policies for the KOALA Grid Scheduler

Ozan Sonmez, Hashim Mohamed, Dick Epema
Dept. of Computer Science, Delft University of Technology
e-mail: {O.O.Sonmez,H.H.Mohamed,D.H.J.Epema}@tudelft.nl

Abstract

In multicluster systems, and more generally, in grids, parallel applications may require co-allocation, i.e., the simultaneous allocation of resources such as processors in multiple clusters. Although co-allocation enables the allocation of more processors than available on a single cluster, depending on the applications' communication characteristics, it has the potential disadvantage of increased execution times due to relatively slow wide-area communication. In this paper, we present two job placement policies, the Cluster Minimization and the Flexible Cluster Minimization policies which take into account the wide-area communication overhead when co-allocating applications across the clusters. We have implemented these policies in our grid scheduler called KOALA *in order to serve different job request types. To assess the performance of the policies, we perform experiments in a real multicluster testbed using communication-intensive parallel applications.*

1. Introduction

Many of the parallel applications submitted to grids may benefit from the allocation of multiple resources such as processors in multiple sites simultaneously, i.e., they may require *processor co-allocation*. However, with processor co-allocation, the execution time of applications may be severely increased due to wide-area communication overhead. In this paper, we design, implement, and assess job placement policies that explicitly consider this overhead in the co-allocation of processors to parallel jobs.

With our KOALA [5] grid scheduler, which has been designed for such multicluster systems as the DAS [2] (see Section 5.3), it is our aim to research efficient placement policies that are specialized for different application types. In [18], we have introduced the *Close-to-Files* (CF) job placement policy for parallel applications that may need co-allocation. Basically, the CF policy tries to reduce the overhead of waiting in multiple clusters for input files to become available in the right locations. We have performed comparison experiments of CF with the *Worst-Fit* (WF) job-placement policy, which is the default policy of KOALA, whose aim is to keep the loads in the clusters balanced [7]. The results showed that the combination of the CF policy and file replication is very beneficial when jobs have large input files (i.e., larger than 2GB). However, neither CF nor WF targets to reduce the effects of wide-area communication overhead to the performance of parallel applications. In general, what we observe with KOALA is that users usually do not submit jobs with large input files, and even that jobs may not have any input file requirements. Therefore, the inter-cluster communication while the application runs induces the main overhead in most of the cases [11].

In this paper, we present two job placement policies, *Cluster Minimization* (CM) and *Flexible Cluster Minimization* (FCM), for jobs that may use co-allocation. The CM policy serves job requests in which jobs are divided into components each having a fixed processor requirement, whereas FCM serves job requests in which jobs are represented as a single component that is allowed to be divided into smaller components. The motivation behind the CM policy is to minimize the number of clusters to be combined for a given parallel job in order to reduce the number of inter-cluster messages. While FCM also aims at minimizing the number of clusters to be combined for parallel jobs, it is also expected to decrease the queue time of jobs, since it increases the placement chance by splitting the jobs into components according to the numbers of idle processors in the clusters. However, splitting a job into many components, which would be the case when the resource contention is high in the system, may increase the execution time of the job since more clusters would be combined, and consequently, more inter-cluster communication would take place. Studying this tradeoff is one of the aims of this paper. As stated in Section 2.2, our job model may not allow all parallel applications to be scheduled by FCM, since some applications dictate specific patterns for splitting up into components. Therefore, CM is our generic communication-aware policy.

The rest of the paper is organized as follows. First, communication-aware grid scheduling is discussed in Section 2. Then, Section 3 explains the main mechanisms of our KOALA grid scheduler. In Section 4, the job placement policies are presented. The experimental setup and the results are explained and discussed in Sections 5 and 6, respectively. Section 7 reviews related work. Finally, Section 8 makes some concluding remarks and points to future work.

2 Communication-Aware Grid Scheduling

In computational grids [13], various application types with different communication characteristics exist. Examples of these application types include data or process parallelism, divide and conquer, parameter sweep, workflows, and data-intensive applications. While some of these applications may need intensive communication, others such as parameter sweep applications may not. In order to increase the execution performance of the applications, a grid scheduler should have specialized policies for different application types.

2.1 Co-Allocation

Within the grid context, many parallel applications may benefit from co-allocation that enables the execution of the applications requiring more nodes simultaneously than available on a single execution site. However, co-allocation is not without cost; although it is expected to reduce the average job response time [12], it has the potential drawback of increased execution times of applications due to low network bandwidth and high latency over wide-area networks. Hence, the performance of co-allocation may be severely affected by the slow inter-cluster connections. In fact, this overhead depends on the communication pattern of an application and on the amount of data being communicated among its sub-tasks. For instance, a fine-grained parallel application is more likely to suffer from higher communication overhead than a coarse-grained parallel application.

2.2 The Job Model

In our model, a job comprises either one or multiple *components* that together execute a single application. A job component specifies its requirements and preferences such as the application it wants to run, the number of processors it needs, and the names of its input files. We assume jobs to be rigid, which means that the number of allocated processors for a job remains fixed during its execution. Job components may or may not specify an execution site where they want to run. Based on this distinction and on the distinction of whether a job does or does not indicate how it is split up into components, we consider three job request structures, *fixed* requests, *non-fixed* requests, and *flexible* requests.

In a fixed request, a job specifies the sizes of its components and the execution site from which the processors must be allocated for each component. On the other hand, in a non-fixed request, a job also specifies the sizes of its components, but it does not specify any execution site, leaving the selection of the execution sites, which may even be the same for some components, to the scheduler. In a flexible request, a job only specifies its total size and allows a scheduler to divide it into components (of the same total size) in order to fit the job on the available execution sites. With a flexible request, a user may impose restrictions on the number and sizes of the components. For instance, a user may want to specify for a job a *lower bound* on the component size or an *upper bound* on the number of components. By default, the lower bound is one, and the upper bound is equal to the number of execution sites in the system. Although it is up to the user to determine the number and sizes of the components of a job, some applications such as the Poisson equation solver in [8] can dictate specific patterns for splitting up the application into components, hence complete flexibility is not suitable in such a case. So, a user may specify a list of options of how a job can be split up, possibly ordered according to preference.

We believe that these request structures give users the opportunity of taking advantage of the system considering their applications' characteristics. For instance, a fixed job request can be submitted when the data or software libraries at different clusters mandate a specific way of splitting up an application. When there is no such affinity, users may want to leave the decision to the scheduler by submitting a non-fixed or a flexible job request.

Of course, for jobs with fixed requests, there is nothing a scheduler can do to schedule them optimally. So in this paper we concentrate on non-fixed and flexible requests.

3 The KOALA Grid Scheduler

It this section we present the KOALA grid scheduler that we have designed for such multicluster systems as DAS [2]. The main feature of KOALA is its support for processor co-allocation.

3.1 The Architecture of KOALA

The KOALA grid scheduler employs some of the Globus toolkit [3] services for job submission, file transfer, and security and authentication. Besides, it has its own mechanisms for data and processor co-allocation, resource monitoring, and fault tolerance. KOALA includes auxiliary components called *runners*, which provide users an interface for submitting their job requests and monitoring the progress of the job execution.

3.2 The Co-Allocation Process

We use the Globus Resource Specification Language (RSL) [3] as our job description language with the RSL "+" construct to aggregate job components to form multi-requests. Upon receiving a job request from a runner, the KOALA scheduler uses one of the placement policies (see Section 4) to try to place the job components. If the placement of the job succeeds and input files are required, the scheduler informs the runner to initiate the third-party file transfers from the selected file sites to the execution sites of the job components. To realize co-allocation, we use an atomic transaction approach [9] in which the job placement only succeeds if all resources required by the job are allocated.

If a placement try fails, KOALA places the job at the tail of a placement queue. This queue holds all the jobs that have not yet been successfully placed. The scheduler regularly scans the queue from head to tail to see whether any job is able to be placed. For each job in the queue we record its number of placement tries, and when this number exceeds a certain threshold value, the submission of that job fails.

3.3 Supported Application Libraries

Currently, KOALA is capable of scheduling jobs employing the Message Passing Interface (MPI) and Ibis [4] parallel communication libraries. The MPI jobs to be submitted to KOALA need to be compiled with the grid-enabled implementation of MPI called MPICH-G2 [6]. The MPICH-G2 allows us to combine multiple clusters to run a single MPI application by automatically handling both inter-cluster and intra-cluster messaging. Ibis is a Java-based communication library for grid computing which enables writing programs using multiple programming models, such as standard Java RMI, divide-and-conquer, message passing, and models which support group communication [4].

4 Job Placement Policies

In this section we will present our two communication-aware job placement policies, which are the Cluster Minimization policy for non-fixed job requests and the Flexible Cluster Minimization policy for flexible job requests. The default policy of KOALA is the Worst Fit (WF) job placement policy [18], which we use as a reference in our experiments (see Section 6). It places the job components in decreasing order of their sizes on the clusters with the largest (remaining) number of idle processors.

4.1 The Cluster Minimization Policy

The Cluster Minimization (CM) policy is designed to serve non-fixed job requests with the aim of minimizing the number of clusters across which parallel applications are spread in order to mitigate the overhead originating from the inter-cluster communication.

```
procedure ClusterMinimization()
1. given a job, order its components in
   decreasing order of size
2. order clusters in decreasing order of
   number of idle processors
/* order of clusters remains the same */
3. for each (job component j) do
4.   for each (cluster c) do
5.     if(c can accommodate j) then
6.       place j on c
7.     end if
8.   end for
9.   if(placement of j fails) then
10.    job submission failed, insert
       the job into the placement queue
11.  end if
12. end for
13. return;
```

Figure 1. Pseudo-code of the Cluster Minimization policy

When given a non-fixed job to place, the CM policy (see Figure 1) operates as follows. First, it sorts the clusters and the job components in decreasing order of number of idle processors and processor requirements (i.e., sizes), respectively (lines 1-2). The order of the clusters remains the same for the rest of the operations. Then it tries to place each component in order, on a cluster that can accommodate this component, traversing the ordered cluster sequence (lines 3-8). If a component cannot be placed, the job request fails and the job is put into the placement queue of the KOALA (lines 9-10). The decreasing order of the components is used to increase the chance of success for large components, and to place as many components as possible on a single cluster.

To illustrate the operation of CM and its difference with WF, let's assume that a non-fixed job request with three job components of size 8 each arrives at a system with three clusters, C_1, C_2, and C_3, which have 18, 15, and 12 idle processors, respectively. As Figure 2 shows, WF successively places the three components on the cluster that has the largest (remaining) number of available processors. This results in the placement of one component on each of the three clusters. However, CM tries to place as many components as possible on the emptiest cluster before considering the next one. Therefore, in this example, CM will place the components on only two clusters.

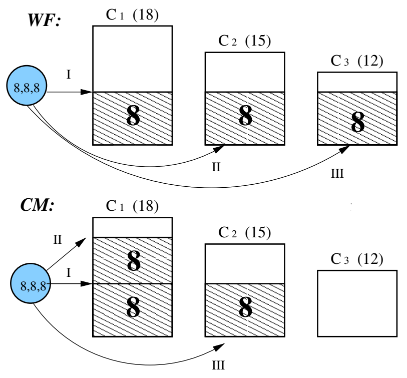

Figure 2. An example comparing the WF and the CM placement policies

It is of course possible to include a *quota* parameter into the policy that will prevent it from filling each of the clusters to more than a certain fraction, in order to avoid the local users of the clusters to suffer from resource starvation.

4.2 The Flexible Cluster Minimization Policy

The Flexible Cluster Minimization (FCM) policy is considered for users who desire a short queue time along with communication optimization for their applications, which do not necessitate a specific way of splitting up into components.

When given a job to place, FCM (see Figure 3) first sorts the clusters in decreasing order of the number of idle processors as in the CM policy (line 2). Then it tries to place a component with a size equal to the remaining number of processors requested which is initially set to the job's total number of processor request (line 4-9). If the placement succeeds, this means the total processors requested of the job is satisfied; hence the job is placed successfully. Otherwise, if the processor request is more than what is available in the cluster considered, it places a component with a size equal to the number of idle processors of that cluster, and subsequently, updates the number of processors requested for the remaining number of requested processors (line 10-14). After traversing the complete ordered cluster sequence, if the processor request cannot be satisfied, the job is put into the placement queue of KOALA (lines 16-18).

To illustrate the process of the policy, let's assume an example where the cluster availabilities are the same as in the example given for CM in Section 4.1, and where there is a flexible job request of 24 processors. In this case, WF and CM are not able to place this job since there are not enough idle processors in any cluster to accommodate the job. However, considering that the emptiest cluster, C_1, has 18 idle processors, FCM first creates a component of size 18. Then it considers the second emptiest cluster, C_2, which is able to meet the remaining request, and so it creates a component of size 6. As a consequence, it places the components onto the clusters as shown in Figure 4.

```
procedure FlexibleClusterMinimization()
1. given a job
2. order clusters in decreasing order of
   the number of idle processors
/* the order of clusters remains same */
3. $R$ = Size of processor request
/* $R$ is initially set to the job's total
   processor requirement */
4. for each (cluster $c$) do
5.   $S =$ # of idle processors of $c$
6.   if($S \geq R$) then
7.    create a job component $j$ with size $R$
8.    place $j$ on $c$
9.    return; // job submission succeeded
10.   else
11.    create a job component $j$ with size $S$
12.    place $j$ on $c$
13.    $R = R - S$;
14.  end if
15. end for
16. if($R > 0$) then
17.   job submission failed, insert the
      job to the placement queue
18. end if
```

Figure 3. Pseudo-code of the Flexible Cluster Minimization policy

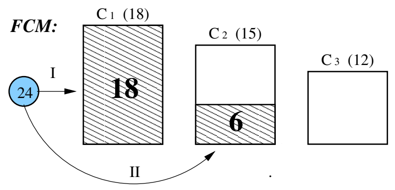

Figure 4. An example illustrating the placement process of the FCM policy

Similarly as in CM, a *quota* parameter can also be considered to prevent the policy from filling the clusters without leaving any space for local users.

5 Experimental Setup

In this section we describe the setup of the experiments we have conducted to evaluate the performance of the CM and FCM placement policies, and compare them with WF.

For the experiments, we use two communication-intensive MPI applications: The Laplace Equation Solver [14] and the 2D-Heat Equation Solver [1], which do not have any restrictions when scheduled by any of the policies. We set the interval between successive scans of the placement queue at 4 seconds, and we set the maximum placement try value to infinite.

5.1 The Laplace Equation Solver Application

The Laplace Equation Solver application [14] uses a finite-difference method to solve the Laplace equation for a square matrix distributed over a square (logical) processor topology. Each matrix element is updated based on the values of the four adjacent matrix elements. First, each process exchanges edge values with its four neighbors. Then, new values are calculated for the upper left and lower right corners of each process's matrix and the processes exchange values again. Finally, the upper right and lower left corners are calculated. This process is repeated until the data converge, i.e., the average change in any matrix element (compared to the value 20 iterations before) is smaller than a specified value. The number of processes, the matrix size, and the convergence factor can be adjusted in order to increase or decrease the execution time of the application.

5.2 The 2D-Heat Equation Solver Application

The 2D-Heat Equation Solver application [1] works with a master-worker paradigm. A two-dimensional domain is decomposed by the master process and then distributed by rows to the worker processes. At each time step, worker processes exchange border data with neighbors, since the current temperature of a point in the domain depends on its previous time step value plus the values of the adjacent points. Upon completion of all time steps, the worker processes return their results to the master process. The number of processes, the domain size, and the number of iterations affect the execution time of the application.

5.3 The Testbed

Our testbed is the Distributed ASCI Supercomputer (DAS) [2], which is a wide-area computer system in the Netherlands that is used for research on parallel, distributed, and grid computing. It consists of five clusters of 200 dual Pentium-III nodes (400 processors) in total. The distribution of the nodes over the clusters is given in Table 1. The clusters are interconnected by the Dutch university backbone (100 Mbit/s), and the workstations within a cluster are connected by Myrinet LAN (1200 Mbit/s). On each of the DAS clusters, the Sun N1 Grid Engine (SGE) [19] is used as the local resource manager. SGE has been configured to run applications on the nodes in an exclusive fashion, i.e., in space-shared mode.

Table 1. The distribution of the nodes over the DAS clusters

Cluster Location	Total number of nodes	Number of functioning nodes (during the experiments)
Vrije University	72	67
Uni. of Amsterdam	32	20
Delft University	32	26
Utrecht University	32	28
Leiden University	32	19

5.4 The Workloads

To evaluate the performance of the placement policies, we submit workloads of jobs to be co-allocated on the DAS that run the applications of Sections 5.1 and 5.2, in addition to the regular workload of the other users. For the Laplace Equation Solver application, we consider job sizes of 36 and 64, since it can run only on a square processor topology. For the 2D-Heat Equation Solver application, we consider total job sizes of 36, 64 and 72.

The component sizes are attained by dividing the job size by the number of components; hence, the components are of equal size within a particular job. For the experiments in which WF and CM are employed, the number of components are 2, 3 or 4; however, we restrict the component sizes to be not greater than 24. Therefore, the jobs of size 36 can have any of the three numbers of components, while those of size 64 have only 4, and those of size 72 have 3 or 4 components. In the experiments with FCM, all jobs are considered to be composed of a single component with the same total size as in the experiments with WF and CM. FCM can split up the jobs in any way, taking into account the number of available processors and the number of clusters in the system.

The average execution times of the applications, depending on the number of clusters combined and on the considered job sizes, are shown in Figure 5. The measurements have been done by submitting fixed job requests. Since the applications are communication-intensive, their execution time increases remarkably with multiple clusters.

The job arrival process is Poisson. The applications, their sizes, and the number of components are randomly chosen from a uniform distribution. We generate two workloads of 200 jobs, W1 and W2, with average interarrival times of 80 and 40 seconds, respectively. With these workloads, we intend to assess the policies in the circumstances

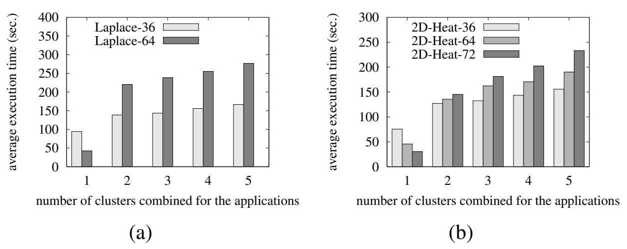

Figure 5. The average execution times of the co-allocated applications depending on the number of clusters combined and on the total number of processor

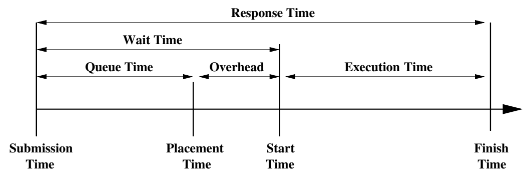

Figure 6. The timeline of a job submission in KOALA

where resource contention is low and high, respectively. We use the tools provided within the GrenchMark project [16] to ensure the correct submission of our workloads to the DAS.

5.5 The Background Load

One of the problems we have to deal with is that we do not have control over the background load due to the other users of the DAS. During the experiments, we monitor this background load and we try to maintain it at 15%–20%. This corresponds to the realistic load observed in the DAS system [15]. When this utilization drops below 15%, we inject dummy jobs to maintain the utilization within the given range. If the utilization rises above 20%, we kill the dummy jobs to lower the utilization. In case the background load rises above 20% and stays there for more than a minute without any of our dummy jobs, the experiment is declared void and is repeated.

6 Results

In this section, we present the results of the experiments described in Section 5.

6.1 Timeline of a job submission in KOALA

Before discussing our results, we illustrate the timeline of a job submission in KOALA (Figure 6). The time instant of the successful placement of a job is called its *placement time*. The *start time* of a job is the time instant when all components are ready to execute. The total time elapsed from the submission of a job to the start of its execution is the *wait time* of a job. The time interval between the submission and the placement shows the amount of time a job spends in the placement queue, i.e., the *queue time*. The time interval between the placement time and the start time is the startup overhead of the middleware (i.e., GRAM [3]). We will present the variation of this overhead depending on the number of components that a job is split up into. The difference between the start time and the time when a job's execution is finished (i.e., its *finish time*), is the *execution time*, and finally, the total time elapsed from submission to finish is the *response time* of a job.

6.2 Discussion of the Results

Figure 7 presents all the metrics that we consider in evaluating the performance of the policies. The average response time and average wait time (Figure 7(a) and Figure 7(b)) show that FCM performs consistently better than the other policies, and as one might expect, WF performs consistently worse than the others, both when the resource contention is low and high.

From Figure 7(c), we observe that for all policies, the average execution time of the applications increases when the resource contention is high due to the increase of the number of clusters combined for the applications. As Figures 7(d) and 7(e) show, FCM and CM are better in minimizing the number of clusters combined for the applications than WF, and therefore yield shorter execution times. WF schedules a job by distributing it as evenly as possible across the available clusters, and consequently causes more inter-cluster communication. It is also important to note that, although FCM tends to combine more clusters than CM when the resource contention is high, as a result of our experiments, it seems this does not have a serious effect on overall performance. FCM is able to utilize clusters better due to the flexibility mechanism, and in doing so, it achieves better cluster minimization.

In the low resource contention case, all policies have similar (low) queue times (Figure 7(f)); however, when the resource contention is high, FCM performs significantly better in terms of queue time, since it is allowed to split up jobs in any way it likes across the clusters. This substantial reduction in queue time eventually results in better overall performance in average response time.

From Figures 7(b) and 7(f), we find that the dominant factor for the wait time is the middleware overhead when the resource contention is low, and the queue time when the

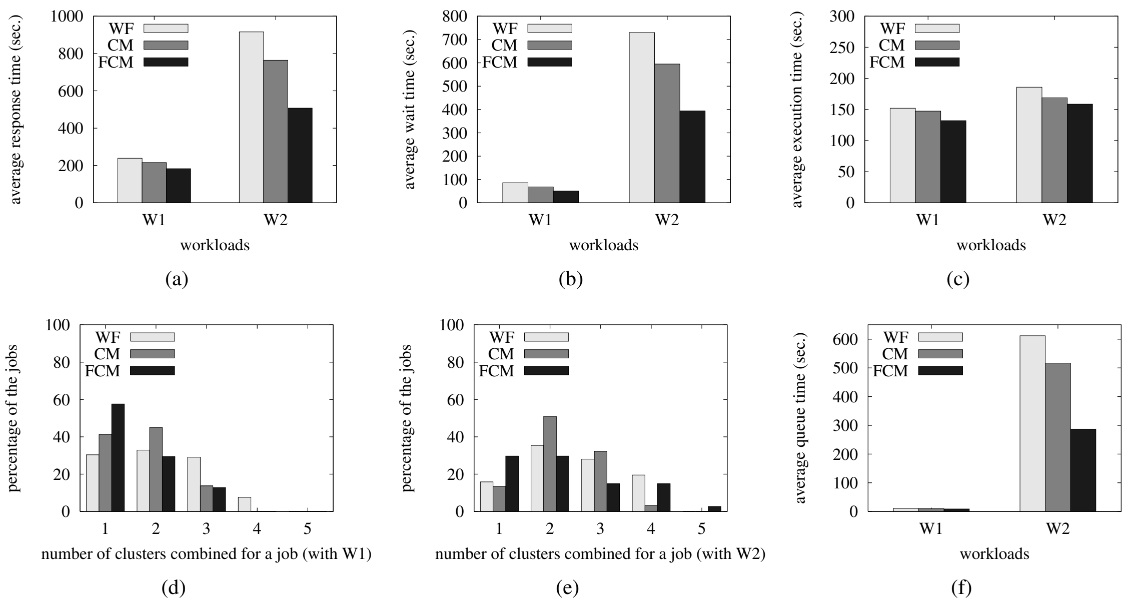

Figure 7. Performance results of the three policies with workloads W1 and W2

resource contention is high. Furthermore, the overhead increases when the number of components that an application is split up into, or when the resource contention increases (Figure 8).

Our experimental results show that splitting jobs into smaller components with the FCM policy to achieve a shorter placement time rather than having them wait in the queue for enough processors to become idle as CM policy does, yields better performance. However, it would be pretentious to generalize this statement since the results depend very much on the number of clusters, the number of processors in the system, and the execution time of the applications. Nevertheless, in our system where the execution time of an application is limited to 15 minutes as an administration policy, FCM seems to be the best policy to be used for communication-intensive applications. Although we observe that it would be really advantageous to schedule such applications on a single cluster from the perspective of execution time (Figure 5), users may prefer co-allocation when more processors are needed than available on a single cluster. For such cases, we have clearly demonstrated the advantage of minimizing the number of clusters to be combined.

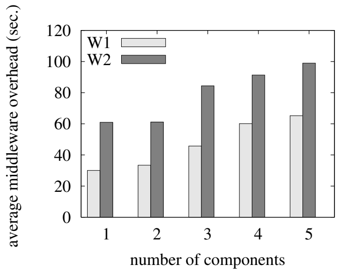

Figure 8. The average middleware overhead by the number of job components

7 Related Work

The Grid Application Development Software (GrADS) [10] enables co-allocation of grid resources for parallel applications that may have significant inter-process communication. Basically, for a given application, the resource selection mechanism of the GrADS tries to reduce the number of workstations to be considered according to their availabilities, local computational and memory capacities, network bandwidth and latency information. Then, among all possible scheduling solutions the one that gives the minimum estimated execution time is chosen for the application.

In [12], co-allocation is studied with simulations, taking the average weighted response time as the performance metric. The work presents an adaptive co-allocation algorithm that uses a simple decision rule whether to use co-allocation for a job considering the given parameters such as requested run time and the requested number of resources. The slow wide-area communication is considered by a parameter by which the total execution time of a job is multiplied. 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. The results show that considering the communication overhead, it pays to use co-allocation if the overhead is limited to about 40 % of the run time of a job.

In [17], several bandwidth-aware co-allocating meta-schedulers are presented for mini-grids. These schedulers consider network utilization to alleviate the slowdown associated with the communication of co-allocated jobs. For each job modeled, its computation time and average per-processor bandwidth requirement is assumed to be known. Besides, all jobs assumed to perform all-to-all global communication periodically. Several scheduling approaches are compared in a simulation environment consisting of clusters with globally homogeneous processors. The most significant result of the experiments performed is that co-allocating jobs when it is possible to allocate a large fraction (85%) on a single cluster provides the best performance in alleviating the slowdown impact due to inter-cluster communication.

8 Conclusions and Future Work

In this paper, we have presented two communication-aware job placement policies for parallel applications that we have implemented in our KOALA grid scheduler. Our experimental results show that the FCM policy is the best option to be used for communication optimization in KOALA, unless the application requires a specific way of splitting up into components. If so, one should prefer the CM policy.

The system in which we performed our experiments is homogeneous in terms of network connections. However, in a heterogeneous environment (a typical case in grids), bandwidth and latency between resources should be considered to achieve communication optimization. As future work, we aim to enhance our policies so that they will consider the impact of bandwidth and latency as resource selection criteria, and we aim to collaborate with other grid systems to test our policies in a heterogeneous environment.

Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ).

References

[1] 2d heat equation. http://www.mhpcc.edu/training/workshop/mpi/exercise.html.

[2] The distributed asci supercomputer. http://www.cs.vu.nl/das2/.

[3] The globus toolkit. http://www.globus.org/.

[4] Ibis: Efficient java-based grid computing. http://www.cs.vu.nl/ibis/index.html.

[5] Koala grid scheduler. http://www.st.ewi.tudelft.nl/koala/.

[6] Mpich-g2. http://www3.niu.edu/mpi/.

[7] A. I. D. Bucur and D. H. J. Epema. The maximal utilization of processor co-allocation in multicluster systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 60.1, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. I. D. Bucur and D. H. J. Epema. The performance of processor co-allocation in multicluster systems. In *CCGRID*, pages 302–309, 2003.

[9] K. Czajkowski, I. T. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *HPDC*, 1999.

[10] H. Dail, F. Berman, and H. Casanova. A decoupled scheduling approach for grid application development environments. *J. Parallel Distrib. Comput.*, 63(5):505–524, 2003.

[11] C. Dumitrescu, D. Epema, J. Dünnweber, and S. Gorlatch. User Transparent Scheduling of Structured Parallel Applications in Grid Environments. In *HPC-GECO/CompFrame Workshop held in Conjunction with HPDC'06*, 2006.

[12] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On advantages of grid computing for parallel job scheduling. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002)*, pages 39–46, Berlin, May 2002. IEEE Press.

[13] I. Foster and C. Kesselman. Computational grids. In *The Grid2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[14] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving problems on concurrent Processors, Volume 1: General Techniques and Regular Problems*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[15] A. Iosup, C. Dumitrescu, D. Epema, H. Li, and L. Wolters. How are real grids used? the analysis of four grid traces and its implications. In *The 7th IEEE/ACM International Conference on Grid Computing (Grid)*, September 28-29, 2006. To appear.

[16] A. Iosup and D. H. J. Epema. Grenchmark: A framework for analyzing, testing, and comparing grids. In *CCGRID*, pages 313–320, 2006.

[17] W. Jones, L. Pang, W. Ligon, and D. Stanzione. Bandwidth-aware co-allocating meta-schedulers for mini-grid architectures. *cluster*, 0:45–54, 2004.

[18] H. Mohamed and D. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *2004 IEEE International Conference on Cluster Computing*, pages 287–298. IEEE Society Press, 2004.

[19] Sun Grid Computing. http://wwws.sun.com/software/grid/.