Use of Run Time Predictions for Automatic Co-Allocation of Multi-Cluster Resources for Iterative Parallel Applications

Marco A. S. Netto[a,*], Christian Vecchiola[a], Michael Kirley[a], Carlos A. Varela[b], Rajkumar Buyya[a]

[a] *Cloud Computing and Distributed Systems (CLOUDS) Laboratory Department of Computer Science and Software Engineering The University of Melbourne, Australia*

[b] *Rensselaer Polytechnic Institute, USA*

Abstract

Metaschedulers co-allocate resources by requesting a fixed number of processors and usage time for each cluster. These static requests, defined by users, limit the initial scheduling and prevent rescheduling of applications to other resource sets. It is also difficult for users to estimate application execution times, especially on heterogeneous environments. To overcome these problems, metaschedulers can use performance predictions for automatic resource selection. This paper proposes a resource co-allocation technique with rescheduling support based on performance predictions for multi-cluster iterative parallel applications. Iterative applications have been used to solve a variety of problems in science and engineering, including large-scale computations based on the asynchronous model more recently. We performed experiments using an iterative parallel application, which consists of benchmark multiobjective problems, with both synchronous and asynchronous commu-

*Corresponding author: netto@csse.unimelb.edu.au

nication models on Grid'5000. The results show run time predictions with an average error of 7% and prevention of up to 35% and 57% of run time overestimations to support rescheduling for synchronous and asynchronous models, respectively. The performance predictions require no application source code access. One of the main findings is that as the asynchronous model masks communication and computation, it requires no network information to predict execution times. By using our co-allocation technique, metaschedulers become responsible for run time predictions, process mapping, and application rescheduling; releasing the user from these burden tasks.
Keywords: Rescheduling, resource co-allocation, grid computing, metascheduling, parallel computing, performance prediction, run time estimates, quality-of-service, asynchronous communication.

---

1. Introduction

Gathering the computing power of multiple clusters is fundamental for large-scale computations. Even users with small and medium size computations can aggregate resources from multiple clusters that would be otherwise wasted due to fragmentation in scheduling queues [1]. Applications with inter-process communication and workflows require coordinated access to these resources; a problem known as *resource co-allocation* [2]. Bag-of-tasks applications may also require co-allocation as users need the completion of all tasks to post-process or analyze the results [3].

Co-allocating resources from multiple clusters is difficult for users, especially when resources are heterogeneous. Users have to specify the number of processors and usage times for each cluster. Apart from being demand-

ing to estimate application run times, these static requests limit the initial scheduling due to the lack of resource options given by users to metaschedulers. In addition, static requests prevent rescheduling of applications to other resource sets; applications may be aborted when rescheduled to slower resources; unless users provide high run time overestimations. When applications are rescheduled to faster resources, backfilling [4] may not be explored if estimated run times are not reduced. Therefore, performance predictions play an important role for automatic scheduling and rescheduling.

The use of performance predictions for scheduling applications have been extensively studied [5, 6, 7, 8]. However, predictions have been used mostly for single-cluster applications and require access to the user application source code [9, 10, 11]. Parallel applications can also use multiple clusters and performance predictions can assist their deployment. One application model for multi-cluster environments is based on iterative algorithms, which has been used to solve a variety of problems in science and engineering [12, 13, 14, 15, 16, 17], and has also been used for large-scale computations through the asynchronous communication model [18, 19].

This paper proposes a resource co-allocation technique with rescheduling support based on performance predictions for multi-cluster iterative parallel applications. Iterative applications with regular execution steps (*i.e.* those with uniform computation workload in each iteration) can have run time predictions by observing their behavior with a short partial execution. This paper also proposes two scheduling algorithms for multi-cluster iterative parallel applications based on synchronous and asynchronous models. The algorithms can be utilized to co-allocate resources for iterative applications with

two heterogeneity levels: the computing power of cluster nodes and process computing requirements.

We performed experiments using an iterative parallel application, which consists of benchmark multiobjective problems, with both synchronous and asynchronous communication models on Grid'5000. The results using our case study application and seven resource sets show run time predictions with an average error of 7% and prevention of up to 35% and 57% of run time overestimations to support rescheduling for synchronous and asynchronous models, respectively. The performance predictions require no application source code access. In addition, an interesting result is that as the asynchronous model masks communication and computation, this model requires no network information to predict execution times. By using our co-allocation technique, metaschedulers become responsible for run time predictions, process mapping, and application rescheduling; releasing the user from these difficult tasks. The use of performance predictions presented here can also be applied when rescheduling single-cluster applications among multiple clusters.

2. Related Work

Extensive research has been carried out on scheduling/load balancing [20, 21, 22, 8], performance predictions [5, 6, 11, 7, 23], and resource co-allocation [24, 2, 25], which are the main areas related to our work.

Process remapping for iterative parallel applications have been investigated for dynamic load balancing purposes [21, 22]. The support for dynamic load balancing requires application modification, which is an approach that we avoid in this paper. Moreover, our work uses process remapping for appli-

cations waiting for resources in the scheduling queues. He *et al.* [8] addressed the problem of dynamic scheduling for parallel jobs in multi-cluster systems using performance predictions. In order to obtain run time predictions, they relied on the PACE (Performance Analysis and Characterization Environment) tool-kit [7], which requires application source code access. Berman *et al.* [10] from AppLeS project have also proposed the use predictions but for using in dynamic and adaptive scheduling decisions, and similar to the previous works, users have to modify the application source code.

Sadjadi *et al.* [9] proposed a modeling approach for estimating execution times of single-cluster long-running scientific applications for multi-cluster environments. Their approach relies on modeling resources, execution parallelism, and application input parameters, along with a set of previous application executions. The applications can access multiple resources, but those have to be homogeneous. Sanjay and Vadhiyar [6] developed a set of performance modeling strategies to predict execution times of parallel applications for single-cluster applications. As their target platform comprises non-dedicated clusters, their strategies require multiple and long application executions and rely on several configuration parameters. Yang *et al.* [11] introduced a performance translation method based on relative performance between two platforms for single-cluster applications. Predictions are generated based on a short execution of applications in each target platform and require source code access. Romanazzi and Jimack [5] proposed a prediction performance model for parallel numerical software systems on multi-cluster environments. Predictions for large-scale experiments are generated based on executing applications with fewer processors for short time periods, and

require source code modification. Tsafrir *et al.* [26] introduced an algorithm for generating run time predictions that require no application source code. However, it is for single-cluster applications and relies on previous application executions. Xu *et al.* [27] proposed a stochastic method for predicting execution times of bulk synchronous computations. Although their research focuses on a similar type of application presented in this paper, their evaluation is based on simulations and on homogeneous resources, whereas our work is evaluated on a large-scale environment with heterogeneous resources. Similar limitations were found in a work developed by Casanova *et al.* [28] who investigated performance predictions for iterative algorithms in distributed systems.

One of the resource co-allocation policies investigated by Bucur and Epema [24] considers meta-scheduler flexibility in defining the number of processors in each cluster automatically. However, the policy does not consider heterogeneous platforms, impact of run time estimates, and rescheduling. Czajkowski *et al.* [2] proposed fault tolerance mechanisms for resource co-allocation requests. Although one of their mechanisms considers process remapping when resources become unavailable, Czajkowski *et al.* have not considered application run time predictions and scheduling issues. Our previous work on resource co-allocation details the benefits of two rescheduling operations: start time shifting and process remapping [25]. We showed that rescheduling co-allocation requests increases system utilization and reduces user response time. In this paper, run time predictions can assist the metascheduler when remapping processes to new resource sets, avoiding user run time overestimations and application abortions due to underestimations.

The main difference between our work and existing ones is that we use performance predictions to co-allocate resources for multi-cluster applications and predictions are based on short partial executions requiring no access to the application source code. Most of existing work focuses on single-cluster applications and requires source code access or historical data of previous application executions to generate run time predictions. For resource co-allocation, most existing solutions rely on users defining the number of processors and run time estimates for each cluster and have no rescheduling support. Our mechanism leaves these responsibilities to the metascheduler and supports application rescheduling.

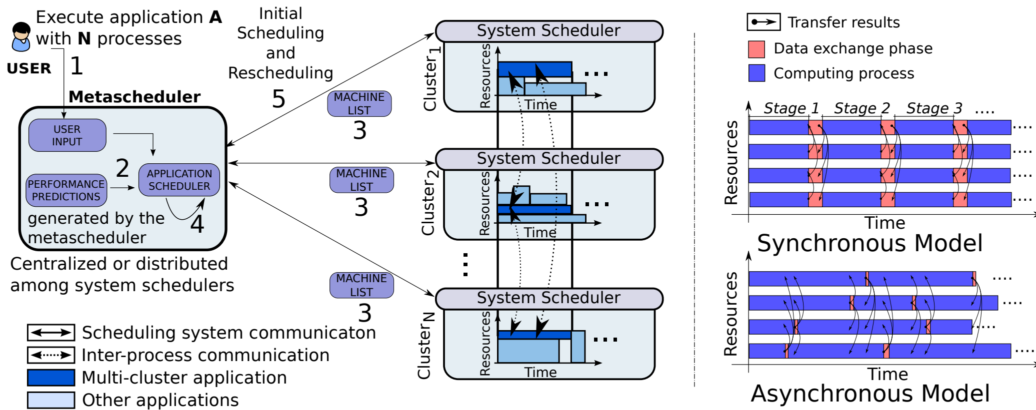


Figure 1: Deployment of iterative parallel applications in multiple clusters following synchronous and asynchronous models.

3. Iterative Parallel Applications

Iterative algorithms have been used in a large class of scientific and engineering problems, especially using optimization techniques such as genetic algorithms, particle swarm and ant colony optimization algorithms. These

algorithms consist of a set of computations inside a loop, which can be partitioned to execute in parallel.

We define an iterative parallel application as a set of processes p, each processing i iterations independently, that exchange information at time intervals called *stages*. The information exchange can follow synchronous or asynchronous communication models. The second model is becoming popular since it opens the opportunity for another parallel application model for large-scale systems [21, 18, 12, 19]. The reason is that asynchronous model can handle inter-process communication on high latency environments. The next sections present the two execution models.

3.1. Synchronous model

In this model, application processes are distributed to machines and results are merged once they have completed the number of iterations specified by the user. For this mode, a stage is fixed by the number of iterations defined by the user followed by a merging process or data exchange phase (Figure 1). When a stage finishes, its results are redistributed to the machines, which execute the next processes. The execution completes when all processes achieve the total number of iterations specified by the user. Processes may finish at different times due to heterogeneity in the system and application. In order to avoid idle processor time, depending on the application, it is possible to keep iterating all processes in a stage until they reach the minimum number of required iterations. For evolutionary-based optimization applications, keep iterating processes only improves the results, without negative side effects. Therefore, processes running on faster machines and/or using parameters that require less computing power iterate more than the

others.

There are several approaches to synchronize data among the application processes. One of them is to have a *master node* for each cluster involved in the execution, responsible for merging the results of nodes in its cluster. This master node can be randomly selected inside a cluster since all nodes have the same computing power. This node sends the results to the master node with better aggregate CPU, called *global coordinator*, which merges all the results and sends the merged result to all clusters. After that, processes start the new stage. Hierarchical data exchange inside a cluster can also be used to optimize the synchronization phase.

3.2. Asynchronous model

For this model, when a process finishes, it distributes its results to other processes asynchronously, merges its results with the last results from other processes, and continues its execution (Figure 1). Therefore, the stage in this model has a flexible number of iterations. This prevents any idle time, and provides better support for heterogeneous machines and processor fault tolerance. Note that as processes execute in multiple clusters, the impact of wide-area communication has to be minimized as much as possible [29]. Indeed, asynchrony masks communication and computation, and therefore minimizes this impact.

The application following the asynchronous model can use similar approaches from the synchronous model to distribute data among processes. The difference is that a process does not wait to receive data from other processes. The choice of the model depends on the application. Applications that have little data dependency can make use of the asynchronous model;

for instance, applications in which the more processes exchange data the better the final result is. In this case, if processes cannot exchange data several times, the final result will still be correct, but not better if more data were exchanged.

3.3. Importance of resource co-allocation

Resource co-allocation guarantees that all resources for executing the application are available at the same time. This is important for both models since: for the synchronous model, it prevents processes from being idle and thus completes the execution faster; whereas for the asynchronous model, it increases interaction among results of the application processes.

4. Resource Co-Allocation based on Performance Predictions

The metascheduler, responsible for co-allocating resources from multiple clusters, relies on four components to enable automatic process selection and rescheduling support. Here we present the sequence of steps to co-allocate resources using performance predictions and an overview of the metascheduler components (component details are presented in the next sections), as illustrated in Figure 1:

- **User input (Step 1):** users only need to specify the total number of processes and a script to determine application throughput on a resource automatically. Section 4.1 details how to provide the script.

- **Performance predictions (Step 2):** the metascheduler executes the script to collect application throughput on multiple resource configurations. Section 5 details an example of how to generate predictions.

- **Machine list (Step 3):** the metascheduler contacts system schedulers to obtain a list of resources that can be used by the user application. Section 4.3 describes the interaction between metascheduler and system schedulers.

- **Application scheduler (Step 4):** uses the machine list, performance predictions, and user preferences to generate a schedule of the application processes. This component generates a set of scripts used to deploy the application. Section 4.2 presents two application schedulers for iterative parallel applications.

A resource co-allocation request based on performance predictions can use different resource sets automatically. This is particularly important when rescheduling applications on multiple clusters. Note that rescheduling is allowed when requests are still in the waiting queues, and hence is different from migrating processes at run time. Moreover, once the metascheduler receives the performance predictions in a given set of resources, it does not need to allocate exactly the same resources for the actual application. This minimizes the chances of the actual application to wait in the job queue of clusters.

The **system scheduler** uses the application scheduler to obtain the application estimated run time. This estimation can be used for both the initial scheduling and the rescheduling (Step 5). As observed in Figure 1, there are two types of network communication, the *system communication*, which is the data transfer among the system components, and the *inter-process communication*, which is the data transfer among the application processes.

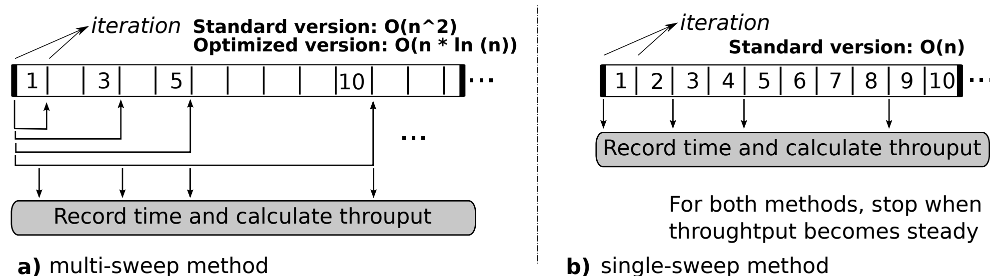


Figure 2: Methods for generating run time predictions without using application's source code.

4.1. Generation of run time predictions

Run time predictions are obtained automatically by executing an application process until the throughput (iterations/second) becomes steady. There are at least two methods to obtain the throughput of the application on a given resource (Figure 2).

Let n be the total number of iterations in an execution and k be the iteration i where the throughput becomes steady; in the worst case scenario, k=n. The first method requires a script that runs the application process from iteration $i = 0$ until $i = j$, where j is a counter from 0 to k. Counter j can be incremented by one for each sweep; which would require k sweeps of length k maximum. Thus, the execution time complexity for this algorithm is $O(n^2)$. However, as the difference between execution time of neighbour iterations may be minimum, it is possible to select a list of increments that follows a power increment. Therefore, it is possible to perform the multi-sweep method with a complexity of $O(n * ln(n))$.

The second method is to ask the application (if it supports) to write output files for each iteration executed and then check the time interval between iterations. The verification could be done for each iteration or for each

```
#!/usr/bin/python
...
iteration = 3
previousthroughput = 0
# 5% difference to be considered stabilized
differencefactor = 5
maxiteration = 1500
while iteration < maxiteration:
   initialtime = int(time.time())
   runCommand ("./myprog myargs -i:",iteration)
   totaltime = int(time.time()) - initialtime
   throughput = iteration/totaltime
   if throughputStabilized(previousthroughput, throughput,
                          differencefactor) == True:
      break
   iteration = int (iteration * 1.15 + iteration )
   previousthroughput = throughput
...
return throughput
```

Figure 3: Example of python script for generating run time predictions.

group of iterations; for both cases this method would require the processing of iterations just once; resulting in a complexity $O(n)$.

In a system composed of m resource configurations, the complexity of obtaining the throughputs are:

$O(m * n * ln(n))$ for the Multi-sweep method
$O(m * n)$ for the Single-sweep method

For both methods, no application source code is required. This is an interesting property of complex scientific applications as they usually generate intermediate output files, which can be used for understanding applications'

behavior without accessing internal data structures. In our case, we used these intermediate output files to predict run times, however they can also be used for application-level scheduling, as showed by one of the co-authors of this paper in a previous work [30].

The time to obtain the throughput depends on the application and its input parameters. The more node configurations the script is executed, the more schedule options can be granted. The metascheduler can ask system schedulers to use the otherwise wasted time of queue fragments to execute the script (similar strategy proposed by Netto *et al.* [31]) or submit it as a processor request to each cluster before the actual application execution. The evaluation section of this paper presents a detailed example of times to generate throughputs in comparison with total application execution time. Figure 3 illustrates an example of a multi-sweep based python script used to obtain the throughput of a machine for a given application. In this example, k is incremented by a factor of 1.15 for each attempt and the throughput is considered stabilized when the difference of the current and previous throughput is less than or equal to 5%.

Once the throughput is available, the application scheduler can calculate the execution time of each process by multiplying the throughput with the number of iterations specified by the user. The application scheduler determines the process locations and the overall execution time, which is described in the following section. Remark that, depending on user parameters, performance predictions can be reused for a given application, thus reducing prediction execution time.

Network cost. For the synchronous communication model, network cost

can also be included in the overall execution time. This cost can be estimated by the amount of data to be transferred among application processes, which comes from the user knowledge about the application, and the network latency among clusters, which comes from popular tools such as *ping* or *traceroute*, or more advance solutions such as King [32] and distributed binning [33].

4.2. Application schedulers

We developed two application schedulers, one for each communication model. These schedulers are frameworks for iterative parallel applications with two heterogeneity levels: computing power of cluster nodes and process computing requirements. The latter level is used for applications that have processes with different computing power requirements. The algorithms are based on the miminum completion time (MCT) algorithm [34], which performs well for heterogeneous tasks and resources. These algorithms are just an example on how the system can schedule tasks; *i.e.* other algorithms can be used for this purpose.

In order to map application processes to cluster resources, the scheduler considers the list of resources, number of iterations per process, number of processes, process requirements, performance predictions for computation and inter-process communication. For simplification reasons, we assumed that each CPU requirement contains the same number of processes.

Synchronous model's scheduler. For the synchronous model (Algorithm 1), the scheduler sorts resources by their computing power, which is determined during the performance prediction phase (Line 1) and assigns processes according to their process requirements (more CPU consuming ones

first) for each stage (Lines 6-12). The number of stages is determined by the total number of resources and total number of processes specified by the user (Line 3). As resources and process requirements make the process execution times vary, more iterations can be added to processes that would be waiting for slower processes (Lines 14-15).

Asynchronous model's scheduler. For the asynchronous model (Algorithm 2), the scheduler assigns one process of each process CPU requirement to the resource with the earliest completion time (Lines 3-6). The scheduler gives priority to longer processes (Line 1), however each group of processes with the same CPU requirement receives one resource per round (Line 4). Note that this algorithm is simpler than Algorithm 1 because the constraint of when the results of processes get mixtured is removed.

4.3. Scheduling and rescheduling

The application scheduler algorithms presented in previous section schedule processes. These processes require a certain amount of time to be executed on give set of resources. To be able to execute these processes, the application scheduler generates *requests* to the system scheduler, which represent the number of resources (*e.g.* machines) and the estimated usage time.

The interaction between system and application schedulers takes place during the initial scheduling and rescheduling of a request. The **initial scheduling** is triggered when a metascheduler requests for machines, whereas the **rescheduling** is triggered when a system scheduler wants to update its queue. For both cases, interaction between system and application's scheduler comprises three steps:

Algorithm 1: Pseudo-code for generating the schedule using the synchronous model.

```
1  Sort resources by decreasing order of computing power
2  Sort processes by decreasing order of CPU demand
3  numberOfStages ← numberOfProcesses / numberOfResources
4  numberOfProcesssesPerProcessRequirement ← numberOfResources /
   numberOfRequirements
5  maxCompletionTime ← 0
6  for 0 to numberOfStages do
7      r ← 0 (r: resource id)
8      for each processRequirement do
9          for 0 to numberOfProcessesPerProcessRequirement do
10             Schedule a process of this processRequirement to resource r
11             Update MaxCompletionTime
12             r ← r + 1
13     /* optional /*
14     for each resource r do
15         Make last process on this resource complete at
           MaxCompletionTime by increasing the number of iterations
```

1. The application scheduler asks the system scheduler for the earliest n machines available;
2. The application scheduler generates a schedule containing the application estimated execution time;
3. The metascheduler, or a system scheduler, verifies with the other system scheduler(s) whether it is possible to commit requests;
4. Step 1 is repeated if it is not possible to commit requests. A maximum number of trials can be specified.

Note that by using this algorithm, resource providers can keep their schedules private [35]. Alternatively, in Step 1, the application scheduler could ask

Algorithm 2: Pseudo-code for generating the schedule using the asynchronous model.

```
1 Sort processes by the decreasing order of their CPU requirement
2 n ← 0
3 while n < numberOfProcesses do
4     for each process CPU requirement do
5         Select resource r with earliest completion time
6         Schedule process to resource r
7     n ← n + numberOfProcessRequirements
```

system schedulers for all free time slots (*i.e.* available time intervals for resources) and then minimize interactions between the metascheduler and the system scheduler.

Details on rescheduling. Rescheduling of an application frequently occurs when other applications finish before the estimated time, since this event can generate fragments in the scheduling queue [25]. Rescheduling can be also necessary when resources fail or users modify/cancel requests. Whenever one of these events arise, the system scheduler triggers the rescheduling process. This process may change the number of resources used in each cluster, or simply change the starting time of the resource requests. For the first case, the partioning of iterations may be changed by the application scheduler, as it generates a new schedule. The requests in each scheduling queue of each cluster have to be configured to start at the same time. When rescheduling the requests on each cluster, the algorithm presented in this section optimizes the application starting time.

5. Evaluation

This section describes the experiments we performed to show how much time the metascheduler requires to generate application throughputs, the accuracy of prediction run times, and the impact of these predictions on the rescheduling.

We used an iterative parallel application based on evolutionary algorithms, which consists of benchmark multiobjective problems, with both synchronous and asynchronous communication models. We conducted the experiment in Grid'5000, which consists of clusters across France dedicated to large-scale experiments [36]. The clusters are spaced-shared machines shared by multiple applications. From these clusters, we selected seven resource sets to execute the application. These sets are examples of resources that are dynamically chosen by the metascheduler to execute our case study application. Experiments on how the metascheduler selects resource sets, when it reschedules multi-cluster applications, and the impact of rescheduling them can be found in our previous work [25]. Before describing the experiment, we provide an overview of the benchmark application.

5.1. Case study of iterative parallel application

EMO (Evolutionary Multi-objective Optimizer) is an iterative benchmark application based on Genetic Algorithms [37] and uses the concept of topology to drive the evolutionary process [38, 39]. A **topology** is a graph interconnecting individuals of a population and is characterized by: (i) the node degree representing the average number of connections for each individual; and (ii) the path line defining the number of hops to be crossed on average

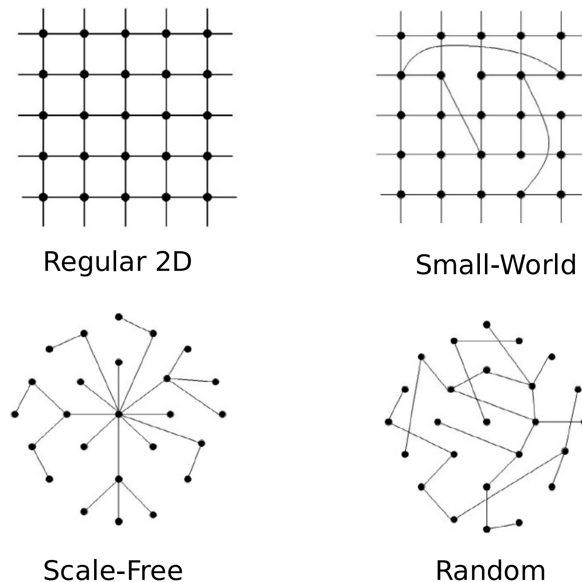


Figure 4: Topologies of EMO application.

to connect individuals. Individuals are chosen to exchange their information according to the topology links; process that determines how the current solutions of the approximation sets are selected to produce a new generation of solutions.

The use of topologies provides better solutions in general but requires a large number of elements in the approximation sets, which becomes compute intensive for non-trivial problems. Moreover, topologies have impact on the execution time: sparsely connected topologies imply faster execution times than fully connected ones, but propagate more slowly the updates in the approximation set. In this work we use four topologies: Regular 2D, Scale-Free, Small-World, and Random [38]. Figure 4 illustrates each topology. We chose these topologies since they represent the various ways the individuals can be interconnected.

EMO is an application that runs from the operating system shell and can be controlled by a set of 25 command line parameters. The main input parameters for our case study are: the topology used to produce the new solutions, the number of iterations, the size of the approximation set, and the optimization multi-objective function. As execution outcome, EMO produces: the final solution set (*Pareto Front*) and the approximation set that generated this solution. For the distributed version of EMO, we introduced a coordination layer that reiterates EMO processes by feeding them with updated information on the approximation set. An additional component, called EMOMerge, merges and partitions the approximation sets generated at each stage of the execution for synchronous model. For the asynchronous model, EMOMerge uses the last results received from the other processes.

Epsilon Indicator. Multi-objective functions identify a multi-dimensional space whose properties are difficult to visualize effectively. It is then necessary to adopt synthetic measures that generally aggregate information about the quality of a solution into one number called indicator. In our case study, we use a quality indicator called *Epsilon* [40], which is based on the distance between a reference solution and the *Pareto front*.

5.2. Experimental configuration

We used seven clusters located in three cities in France with heterogeneous computing capabilities. Table 1 presents an overview of the node configurations for these clusters[1]. Machines in the same location share the same file system, which simplifies file transfer between nodes of clusters in

[1]More details about the machines in Grid'5000 can be found at https://www.grid5000.fr

the same site. The inter-site communication is over a 1Gbps Ethernet network, whereas the intra-site communication varies for each site (Myrinet and 10 Gbps Ethernet).

Table 1: Overview of the node configurations.

Cluster	Location	CPUs' Configuration
paradent	Rennes	Intel Xeon L5420 2.5 GHz
paramount	Rennes	Intel Xeon 5148 2.33 GHz
paraquad	Rennes	Intel Xeon 5148 2.33 GHz
sol	Sophia	AMD Opteron 2218 2.6 GHz
bordemer	Bordeaux	AMD Opteron 248 2.2 GHz
azur	Sophia	AMD Opteron 246 2.0 GHz
bordeplage	Bordeaux	Intel Xeon EM64T 3 GHz

Table 2: Seven resource sets, from the clusters in Grid'5000, selected by the metascheduler to execute user applications.

Clusters/Resource Sets	1	2	3	4	5	6	7
paradent	32						
paramount	04		20			04	
paraquad	04	20					
sol		12	12	20	20	12	
bordemer		02	08	20	08	06	20
azur		06			06	10	
bordeplage					06	08	20

Application configuration. In order to test the performance and behaviour of multiobjective evolutionary algorithms (MOEA), Deb *et al.* [41] have proposed a set of functions designed to evaluate different aspects of these algorithms. The functions are named DLTZn where n ranges from 1 to 9. These functions are designed incrementally and are useful tests for

checking the ability of algorithms to converge to a hyperplane (*DLTZ1*), to scale up their performance in a large number of objectives (*DLTZ2*), to converge to the Pareto optimal front (*DLTZ3*), and so on. We decided to use the *DLTZ6* function that both evaluates the ability to converge to a curve and constitutes a good test for the computational complexity of a MOEA. We configured EMO to solve the DTLZ6 function with a setup of 10 objectives as suggested by Deb *et al.* We have used four topologies and 1024 individuals (also known as candidate solutions) for each process with a minimum of 200 iterations for process. The application was deployed on 40 cores using 480 EMO instances, i.e. 120 instances per topology in order to find the best solutions for DLTZ6.

Resource sets. We configured the metascheduler to access seven resource sets in Grid'5000. Table 2 presents the list of clusters and number of cores used in each resource set. These resource sets are examples of resources chosen dynamically by the metascheduler for the EMO application. The clusters are space-shared machines, and hence the resource sets are dedicated to the application, which is a common set up for existing HPC infrastructures.

Inter-process communication overhead. Communication is based on file transfer between processes during the merging phases. The files transferred among the sites are 500 Kbytes on average. Therefore, the cost of transferring the files is minimum compared to the total application execution time, which takes minutes. However, file transfer, performed by the metascheduler, relies on secure copy (*scp*) command, which requires authentication. Therefore, we used inter-site file transfer as 800ms (in a 1Gbps Ethernet network), which is taken into account by the metascheduler to calculate the run time prediction.

Metrics. To evaluate co-allocation based on performance predictions and their importance on rescheduling, we measured the *time to generate predictions* and analyze the *difference between the actual and the predicted execution times.* The prediction for each resource set assists schedulers to know whether they can reschedule the new sub requests into the scheduling queues of other schedulers. Therefore, we measured the impact of predictions for the application rescheduling. To understand the application output on different resource sets, we also measured *execution times* and the *Epsilon indicator*, which indicates the quality of the solutions inside the application, for both synchronous and asynchronous models.

5.3. Results and Analysis

Performance predictions generation. The metascheduler executed an independent process in six nodes with different computing power and the throughputs became steady before 250 iterations for all processes. Table 3 presents the throughputs for each cluster and topology. We observe that sparsely connected networks, such as the Regular 2D and the Scale-Free networks, imply a faster iteration time than more connected networks, such as the Random topology. For the Random topology, the value of the path line was around five times smaller than the path line of the Regular 2D. This means that, on average, the selection of the individual to exchange information requires traversing a list five times smaller than for the Random topologies; and this reflects the execution time difference. Table 4 shows the time spent to obtain the throughput for each node configuration and topology. Most of the throughput values took less than one minute to be obtained. The total CPU time to generate the predictions is only 5% and 10% of the

overall CPU time of the longest and shortest experiment respectively. As the predictions can be re-utilized or used by longer executions or executions with more processes, the cost for generating predictions tends to be zero. It is important to remark that users with long experiments have more benefits than those with short ones due to the trade-off between total application execution time and performance prediction time.

Table 3: Throughput (iterations/sec.) for each machine configuration and topology.

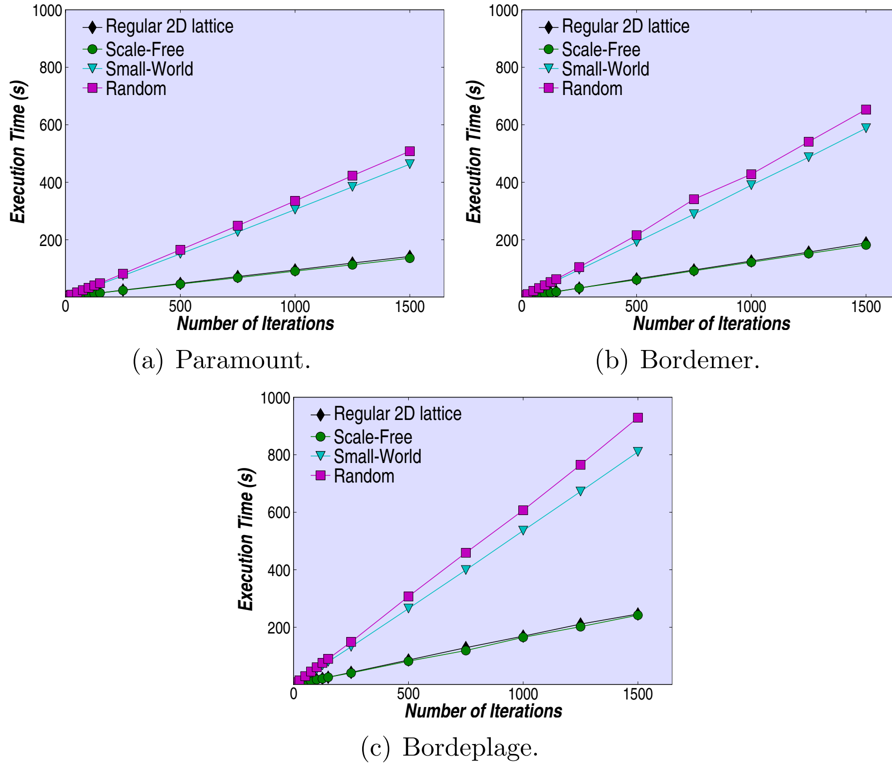
Cluster/Top.	Regular 2D	Scale-Free	Small-World	Random
paradent	10.87	11.36	3.57	3.29
paramount	10.00	10.42	3.33	3.05
paraquad	10.00	10.42	3.33	3.05
sol	9.26	9.62	3.09	2.81
bordemer	7.81	7.81	2.60	2.38
azur	7.14	7.35	2.34	2.14
bordeplage	5.81	6.10	1.89	1.68

Table 4: Time in seconds to obtain the throughputs for each machine configuration and topology.

Cluster/Top.	Regular 2D	Scale-Free	Small-World	Random
paradent	23	14	42	46
paramount	25	15	45	49
paraquad	25	15	45	49
sol	27	16	49	54
bordemer	32	19	58	63
azur	35	21	64	70
bordeplage	43	26	80	90

Regular behaviour. In order to show the regularity of the throughput, we configured the meta-scheduler to collect the throughput data until 1500 iterations. As we can see in Figure 5, which shows the execution times as a function of the topologies and number of iterations for three machine config-

urations in Grid'5000 (for exemplification purposes, we included only three configurations), EMO has a regular throughput over the iterations. This happens because EMO processes the similar amount of work in each iteration, which is common for several iterative applications. Figure 6 represents the throughput (iterations/second) of an EMO execution using Regular 2D and Random topologies on a single core of seven machine configurations as a function of number of iterations. We obtained similar results for the other two topologies.



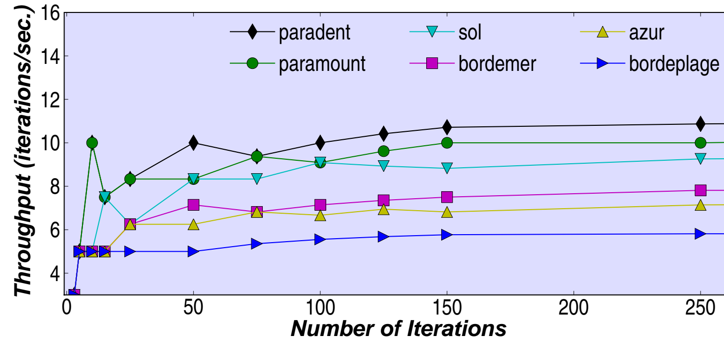
(a) Paramount.

(b) Bordemer.

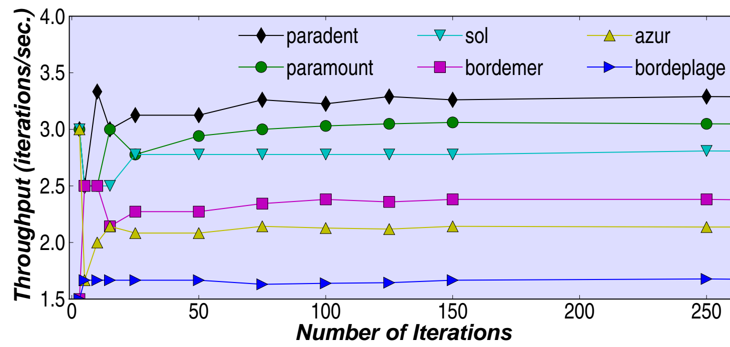
(c) Bordeplage.

Figure 5: Execution times as a function of the topologies for three machine configurations in Grid'5000.

Accuracy of predictions. Figure 7 presents the predicted and actual



(a) Regular 2D topology.



(b) Random topology.

Figure 6: Throughput for Regular 2D and Random topologies on each machine configuration.

execution times for synchronous and asynchronous models. Actual execution times are averages of five executions for each resource set. We observe that the execution time for the asynchronous model is shorter than the synchronous model for all resource sets. For the asynchronous model, all EMO processes execute the minimum required number of iterations, whereas for the synchronous model, EMO processes may execute more iterations in order to wait for processes that take longer. In addition, the difference between actual and predicted execution is on average 8.5% for synchronous and 7.3%

for the asynchronous model. These results highlight that it is possible to reschedule processes on multiple clusters since schedulers can predict the execution time for different resource sets. Note that the predictions for the asynchronous model is slightly better than for the synchronous model. This reason is that the asynchronous model requires less accurate inter-process communication predictions than the synchronous model since the network overhead impact in the first model is minimum.

For the quality of the predictions (Figure 7), resource sets 2 and 3 present more accurate predictions (distance between predicted and actual execution times) compared to the other sets for the synchronous model. This happens because the merging phase is split by sites (locations in France). For these sets, three sites are used, and therefore the load for merging results is well balanced. Resource set 6 also comprises three sites, but only four resources in the site that has the best computing power compared to the other two. Thus there in an unbalance in processing power among the sites generating a bottleneck in the merging phase. For the asynchronous model, the worst prediction is for resource set 7 since 20 resources from the worst cluster (*bordeplage*) are used, which makes the merging process slower.

Importance of predictions to rescheduling. When remapping processes from one resource set to another, the application run time may remain the same, increase or decrease. When it remains the same, schedulers just have to redefine the number of resources in each cluster; which can be performed by the metascheduler or by the system schedulers themselves. This is the case for remapping processes from, for example, resource set 1 to 2 and 2 to 3 or 4 for synchronous and asynchronous model respectively. When the run

time increases, the prediction generated by the application-scheduler may avoid the application to be aborted due to underestimations. A rescheduling from a shorter to longer execution time is desired when the application can start earlier than the initial schedule predicted. This is the case for remapping processes from resource set 1 to 7. Figure 8 shows that overestimation is required to avoid the application to be killed when rescheduling from resource set 1 to the other resource sets without the use of predictions (the higher the value the more useful is our metascheduler based on predictions). For the synchronous model, 35% of overestimation is required, whereas for the asynchronous model 57%. When rescheduling a request from a longer to shorter run time, predictions assist schedulers to increase the chances of backfilling sub requests [4]. This happens because longer jobs tend not to fill the fragments in the scheduling queues [42]. This is the case when rescheduling processes from resource set 7 to 1 for synchronous and asynchronous model.

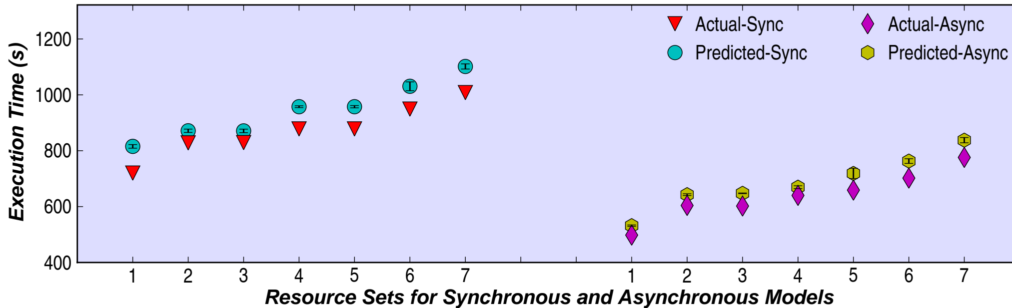


Figure 7: Predicted and actual execution times for synchronous and asynchronous models.

Synchronous versus asynchronous models. In order to understand the output produced by the application, we have also compared the quality of the optimization results between synchronous and asynchronous models. Figure 9 shows the Epsilon indicator (the lower the better) for synchronous and

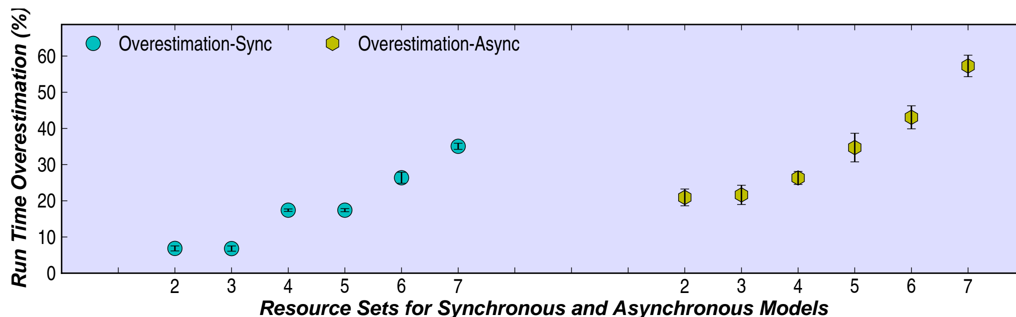
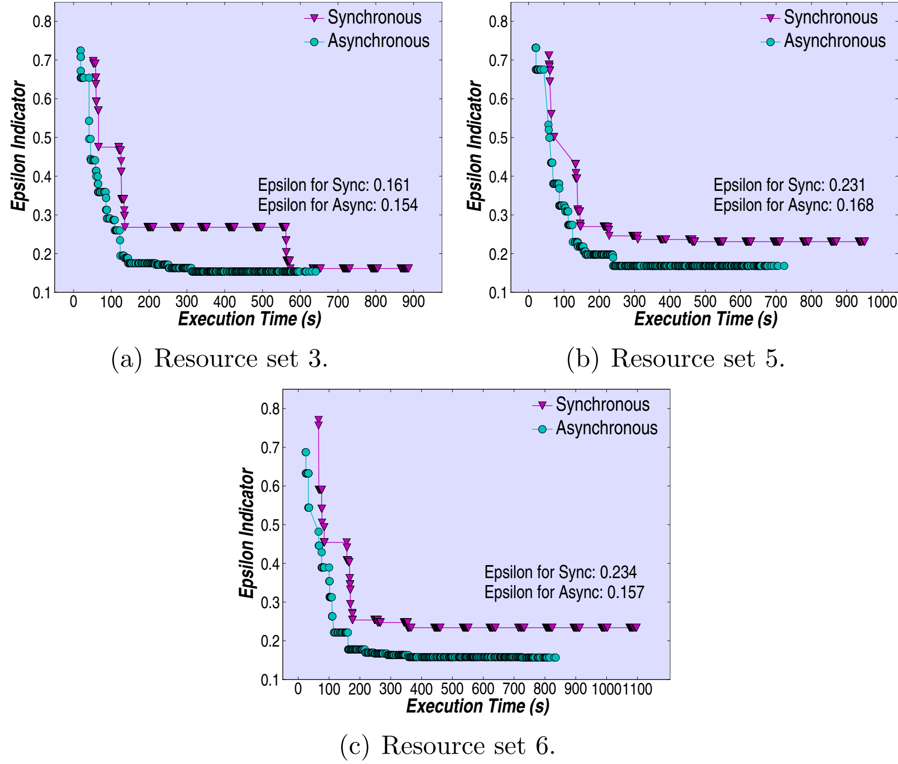


Figure 8: Overestimation required to avoid application being aborted due to rescheduling from resource set 1 to other sets without co-allocation based on performance predictions.

asynchronous models under three resource sets. The asynchronous model converges faster and produces better results than the synchronous model. This happens because the asynchronous model is able to mix more results from different EMO processes, which might have different topologies, in relation to the synchronous model. For resource set 3, the synchronous model produces similar result for the Epsilon indicator as the asynchronous model, but the Epsilon values get closer after a considerable execution time. Figure 10 illustrates the importance of mixing results from different topologies. The results show that although Random, which is the most CPU consuming topology, has the greatest impact on the Epsilon indicator, the less CPU consuming Scale-Free topology contributes to the function optimization. Moreover, even for one-topology executions, asynchronous produces better optimization results and it converges faster than its synchronous counterpart. Similar results were obtained for the other resource sets but are not included due to space constraints. The comparison results between synchronous and asynchronous model showed here corroborate the results presented by Desell *et al.* [19] with their application in the astronomy field; *i.e.* asynchronous

model has better convergence rates, especially when heterogeneous resources are in place.



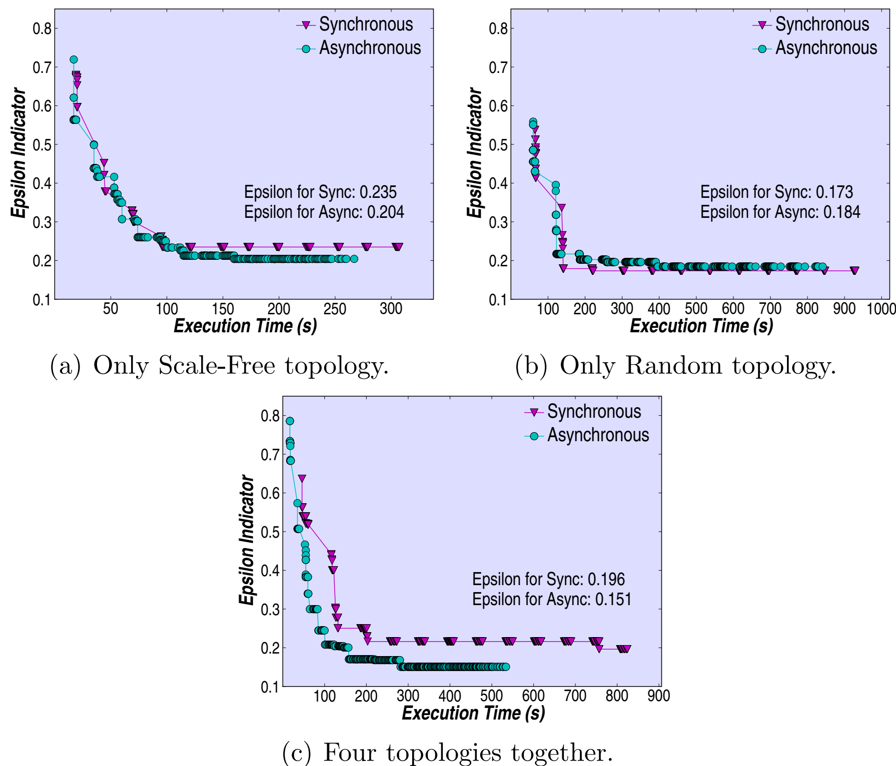
(a) Resource set 3.

(b) Resource set 5.

(c) Resource set 6.

Figure 9: Epsilon indicator for three resource sets on both communication models.

Other examples of iterative applications. The results presented so far are from a single application using both synchronous and asynchronous execution models. Here we show examples of other three applications that also present regular execution times. The first application is PEPS (Performance Evaluation Of Parallel Systems) [13], which aimed at solving numerically very large Markov Chains. The second application is ABC toolbox [14], which implements Approximate Bayesian Computations (ABC) algorithms



(a) Only Scale-Free topology.

(b) Only Random topology.

(c) Four topologies together.

Figure 10: Epsilon indicator for resource set 1 showing the importance of mixing topologies.

and is used for analysing evolutionary history of biological especies. The third application is Amber [15], which is a set of molecular mechanical force fields for the simulation of biomolecules and a package of molecular simulation programs. All these three applications are iterative and can be executed in parallel. We performed experiments using three inputs for each application, varying the Markov Chain, sampling size, and number of interactions of atoms for PEPS, ABC toolbox, and Amber, respectively. Figure 11 presents the throughput behaviour for the three applications for each input parameter. We observed that for the PEPS application, the throughput for each input

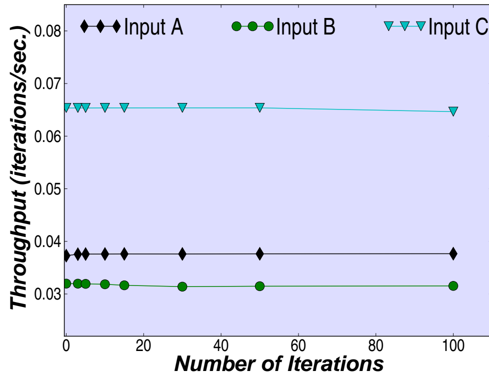
is steady from the beginning of the executions. For the ABC toolbox and Amber applications, the throughput becames steady after a few iterations complete, which is a similar behaviour of EMO (Figure 6). We executed the applications using 300, 5000, and 5000 for PEPS, ABC toolbox, and Amber respectively. As our aim is to be able to predict the execution time of each application using a few iterations, we used 5, 30, and 30 iterations to execute the applications. Table 5 presents the predicted and actual execution times using these number of iterations. Apart from the inputs A and B of ABC toolbox, all the other executions provided accurate predicted execution times, showing the viability of the sampling method for four iterative applications: EMO, PEPS, ABC toolbox, and Amber. For these applications we were able to use the single-sweep method.

Table 5: Predicted execution time (in seconds) for PEPS, ABC toolbox, and Amber applications. The total number of iterations are 300, 5000, and 5000 for PEPS, ABC toolbox, and Amber respectively. The number of iterations used to define the throughput are 5, 30, and 30 for these three applications.

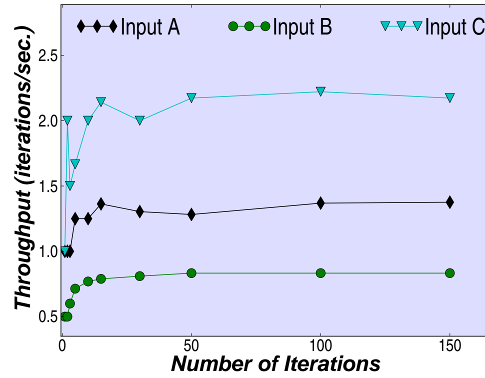
Applications	**PEPS**			**ABC toolbox**			**Amber**		
Inputs	A	B	C	A	B	C	A	B	C
Predic. Time	7959	9350	4574	6410	4165	10865	3965	2475	8330
Actual Time	7968	9479	4713	4820	5445	10850	4085	2485	8655

6. Conclusions

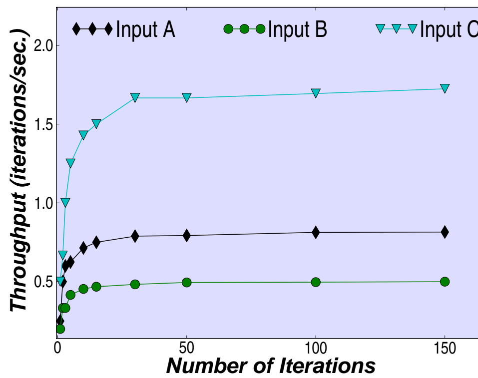
Resource co-allocation ensures that applications access processors from multiple clusters in a coordinated manner. Current co-allocation techniques mostly depend on users to specify the number of processors and usage time for each cluster, which is particularly difficult due to heterogeneity of the computing environment.



(a) PEPS.



(b) ABC toolbox.



(c) Amber.

Figure 11: Throughput analysis of three iterative applications with three input parameters.

This paper presented a resource co-allocation technique with rescheduling support based on performance predictions for multi-cluster iterative parallel applications. Due to the regular nature of these applications, a simple and effective performance prediction strategy can be used to determine the execution time of application processes. The metascheduler can generate the

application performance model without requiring access to the application source code, but by observing the throughput of a process in each resource configuration using a short partial execution. Predictions also enable automatic rescheduling of parallel applications; in particular they prevent applications from being aborted due to run time underestimations and increase backfilling chances when rescheduled to faster resources.

From our experiments using an iterative benchmark parallel application with both synchronous and asynchronous communication models on Grid'5000, we observed run time predictions with an average error of 7% and prevention of up to 35% and 57% of run time overestimations for synchronous and asynchronous models, respectively. A relevant remark is that the asynchronous model requires no network information to predict execution times since this model masks communication and computation. The results are encouraging since automatic co-allocation with rescheduling support is fundamental for multi-cluster iterative parallel applications; in particular because these applications, based on asynchronous communication model, are used to solve problems in large-scale systems.

Acknowledgments

We thank Alexandre di Costanzo, Marcos Dias de Assunção, Mukaddim Pathan, and the anonymous reviewers for their valuable comments on this paper. We also would like to thank Antonio Lima and Raquel Dias for their assistance with the PEPS, ABC toolbox, and Amber applications. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN devel-

opment action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Dept. of Innovation, Industry, Science and Research (DIISR). It is also partially supported by U.S.A. NSF CAREER CNS Grant No: 0448407.

References

[1] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, A. Streit, On advantages of grid computing for parallel job scheduling, in: Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid'02), Berlin, Germany, 2002, pp. 39–46.

[2] K. Czajkowski, I. Foster, C. Kesselman, Resource co-allocation in computational grids, in: Proceedings of the 8th International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, USA, 1999, pp. 219–228. doi:10.1109/HPDC.1999.805301.

[3] M. A. S. Netto, R. Buyya, Offer-based scheduling of deadline-constrained bag-of-tasks applications for utility computing systems, in: Proceedings of the 18th International Heterogeneity in Computing Workshop (HCW'09), in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), Rome, Italy, 2009.

[4] A. W. Mu'alem, D. G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling,

IEEE Transactions on Parallel and Distributed Systems 12 (6) (2001) 529–543.

[5] G. Romanazzi, P. K. Jimack, Parallel performance prediction for numerical codes in a multi-cluster environment, in: Proceedings of the 2008 International Multiconference on Comp. Science and Information Technology (IMCSIT'08), Wisla, Poland, 2008.

[6] H. A. Sanjay, S. S. Vadhiyar, Performance modeling of parallel applications for grid scheduling, Journal of Parallel and Distributed Computing 68 (8) (2008) 1135–1145.

[7] S. A. Jarvis, D. P. Spooner, H. N. L. C. Keung, J. Cao, S. Saini, G. R. Nudd, Performance prediction and its use in parallel and distributed computing systems, Future Generation Computer Systems 22 (7) (2006) 745–754.

[8] L. He, S. A. Jarvis, D. P. Spooner, X. Chen, G. R. Nudd, Dynamic scheduling of parallel jobs with QoS demands in multiclusters and grids, in: Proceedings of the International Conference on Grid Computing (GRID'04), 2004.

[9] S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, X. J. Collazo-Mojica, A modeling approach for estimating execution time of long-running scientific applications, in: Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08), 2008.

[10] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. M. Figueira, J. Hayes, G. Obertelli, J. M. Schopf, G. Shao, S. Smallen, N. T. Spring, A. Su, D. Zagorodnov, Adaptive computing on the grid using AppLeS, IEEE Transactions on Parallel and Distributed Systems 14 (4) (2003) 369–382.

[11] L. T. Yang, X. Ma, F. Mueller, Cross-platform performance prediction of parallel applications using partial execution, in: Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC'05), 2005.

[12] J. M. Bahi, S. Contassot-Vivier, R. Couturier, Performance comparison of parallel programming environments for implementing AIAC algorithms, The Journal of Supercomputing 35 (3) (2006) 227–244.

[13] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, W. Stewart, The PEPS software tool, Computer Performance (2003) 98–115.

[14] D. Wegmann, C. Leuenberger, S. Neuenschwander, L. Excoffier, ABC-toolbox: a versatile toolkit for approximate Bayesian computations, BMC bioinformatics 11 (1) (2010) 116.

[15] D. Case, T. Cheatham III, T. Darden, H. Gohlke, R. Luo, K. Merz Jr, A. Onufriev, C. Simmerling, B. Wang, R. Woods, The Amber biomolecular simulation programs, Journal of computational chemistry 26 (16) (2005) 1668–1688.

[16] G. Morris, D. Goodsell, R. Huey, A. Olson, Distributed automated dock-

ing of flexible ligands to proteins: parallel applications of AutoDock 2.4, Journal of Computer-Aided Molecular Design 10 (4) (1996) 293–304.

[17] B. Brooks, R. Bruccoleri, B. Olafson, et al., CHARMM: A program for macromolecular energy, minimization, and dynamics calculations, Journal of computational chemistry 4 (2) (1983) 187–217.

[18] Z. Li, M. Parashar, A decentralized computational infrastructure for grid-based parallel asynchronous iterative applications, Journal of Grid Computing 4 (4) (2006) 355–372.

[19] T. J. Desell, B. K. Szymanski, C. A. Varela, Asynchronous genetic search for scientific modeling on large-scale heterogeneous environments, in: Proceedings of the 17th Heterogeneity in Computing Workshop (HCW'08), in conjunction with 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08), 2008.

[20] C. A. Bohn, G. B. Lamont, Load balancing for heterogeneous clusters of PCs, Future Generation Computer Systems 18 (3) (2002) 389 – 400.

[21] O. Sievert, H. Casanova, A simple MPI process swapping architecture for iterative applications, International Journal of High Performance Computing Applications 18 (3) (2004) 341–352.

[22] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, C. A. Varela, Malleable iterative MPI applications, Concurrency and Computation: Practice and Experience 21 (3) (2009) 393–413.

[23] V. W. Mak, S. F. Lundstrom, Predicting performance

of parallel computations, IEEE Transactions on Parallel and Distributed Systems 1 (3) (1990) 257–270. doi:http://dx.doi.org.ezp.lib.unimelb.edu.au/10.1109/71.80155.

[24] A. I. D. Bucur, D. H. J. Epema, Scheduling policies for processor coallocation in multicluster systems, IEEE Transactions on Parallel and Distributed Systems 18 (7) (2007) 958–972.

[25] M. A. S. Netto, R. Buyya, Rescheduling co-allocation requests based on flexible advance reservations and processor remapping, in: Proceedings of 9th IEEE/ACM International Conference on Grid Computing (GRID'08), Tsukuba, Japan, 2008.

[26] D. Tsafrir, Y. Etsion, D. Feitelson, Backfilling using system-generated predictions rather than user runtime estimates, IEEE Transactions on Parallel and Distributed Systems 18 (6) (2007) 789–803.

[27] C. Xu, L. Wang, N. Fong, Stochastic prediction of execution time for dynamic bulk synchronous computations, The Journal of Supercomputing 21 (1) (2002) 91–103.

[28] H. Casanova, M. Thomason, J. Dongarra, Stochastic performance prediction for iterative algorithms in distributed environments, Journal of Parallel and Distributed Computing 58 (1) (1999) 68–91.

[29] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, R. F. H. Hofman, Optimizing parallel applications for wide-area clusters, in: Proceedings of the 12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98), 1998.

[30] M. A. S. Netto, , A. Breda, O. N. de Souza, Scheduling complex computer simulations on heterogeneous non-dedicated machines: a case study in structural bioinformatics, in: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05), Vol. 2, IEEE, 2005, pp. 768–775.

[31] M. A. S. Netto, R. N. Calheiros, R. K. S. Silva, C. A. F. D. Rose, C. Northfleet, W. Cirne, Transparent resource allocation to exploit idle cluster nodes in computational grids, in: Proceedings of 1st International Conference on e-Science and Grid Technologies (e-Science'05), IEEE Computer Society, 2005, pp. 238–245.

[32] P. K. Gummadi, S. Saroiu, S. D. Gribble, King: estimating latency between arbitrary internet end hosts, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement (IMW'02), ACM, 2002, pp. 5–18.

[33] S. Ratnasamy, M. Handley, R. M. Karp, S. Shenker, Topologically-aware overlay construction and server selection, in: IEEE INFOCOM, 2002.

[34] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, R. F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[35] E. Elmroth, J. Tordsson, A standards-based grid resource brokering service supporting advance reservations, coallocation, and cross-grid inter-

operability, Concurrency and Computation: Practice and Experience 21 (18) (2009) 2298–2335.

[36] R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, I. Touche, Grid'5000: a large scale and highly reconfigurable experimental grid testbed, International Journal of High Performance Computing Applications 20 (4) (2006) 481.

[37] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, John Wiley & Sons, 2001.

[38] M. Kirley, R. Stewart, Multiobjective evolutionary algorithms on complex networks, in: Proceedings of 4th International Conference Evolutionary Multi-Criterion Optimization (EMO'07), Lecture Notes Computer Science 4403, Matsushima, Japan, 2007.

[39] C. Vecchiola, M. Kirley, R. Buyya, Multi-objective problem solving with offspring on enterprise clouds, in: Proceedings of the 10th International Conf. on High-Performance Computing in Asia-Pacific Region (HPC Asia'09), 2009.

[40] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. da Fonseca, Performance assessment of multiobjective optimizers: An analysis and review, IEEE Transactions on Evolutionary Computation 7 (2) (2003) 117–132.

[41] K. Deb, L. Thiele, M. Laumanns, E. Zitzler, Scalable test problems for evolutionary multiobjective optimization, Evolutonary Multiobjective Optmization.

[42] D. Tsafrir, D. G. Feitelson, The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help, in: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'06), San Jose,USA, 2006.