An RMS Architecture for Efficiently Supporting Complex-Moldable Applications

Cristian Klein, Christian Pérez

An RMS Architecture for Efficiently Supporting Complex-Moldable Applications

Cristian KLEIN and Christian PÉREZ
INRIA/LIP
ENS de Lyon, France
Email: {cristian.klein, christian.perez}@inria.fr

***Abstract*—High-performance scientific applications are becoming increasingly complex, in particular because of the coupling of parallel codes. This results in applications having a complex structure, characterized by multiple deploy-time parameters, such as the number of processes of each code. In order to optimize the performance of these applications, the parameters have to be carefully chosen, a process which is highly resource dependent. However, the abstractions provided by current Resource Management Systems (RMS) — either submitting rigid jobs or enumerating a list of moldable configurations — are insufficient to efficiently select resources for such applications. This paper introduces CooRM, an RMS architecture that delegates resource selection to applications while still keeping control over the resources. The proposed architecture is evaluated using a simulator which is then validated with a proof-of-concept implementation on Grid'5000. Results show that such a system is feasible and performs well with respect to scalability and fairness.**

I. Introduction

Background

High-performance computing is characterized by increasingly complex applications, as they attempt to more accurately model physical phenomena. For example, the Salome framework [22] allows applications to be developed by coupling several parallel codes for numerical simulations. At deployment the individual codes are instantiated and mapped onto target resources. Choosing the parameters (e.g., the number of processes of each parallel code) to transform the application from an abstract description to a concrete one, so as to optimize its performance, is highly dependent on the available resources: not only may codes have different speed-up functions, but they may exhibit different behaviors from one machine to another.

To run these applications on HPC resources, such as clusters, supercomputers or grids, a resource allocation has to be requested from a *Resource Management System* (RMS). RMSs strive to offer simple interfaces to satisfy the needs of all users. Generally, two types of resource request are supported: (i) *rigid jobs*, for which the resource requirements are fixed by the user; (ii) enumerating a list of *moldable configurations* and let the RMS choose a configuration depending on the state of the resources. The latter is used for efficiently scheduling **simple-moldable** applications, which are characterized by a single deploy-time parameter, the number of processes. Since the number of distinct values the parameter can take is small, all moldable configurations can be enumerated.

But code coupling applications are **complex-moldable**, being characterized by a complex structure, with multiple deploy-time parameters. Therefore, for selecting the resources that enable an efficient execution, application-specific resource selection algorithms are required. However, employing such algorithms in practice is difficult due to three main reasons. First, RMSs are providing limited information about the availability of the resources, which is required as input to resource selection algorithms.

Second, updating a resource request to adapt to any resource change can only be done by cancelling the request and resubmitting a new one, which will be scheduled after all other requests. On a loaded platform, this can greatly increase the end-time of the application. Using a list of moldable configurations does not solve the problem, as the number of resource configurations that would have to be enumerated is too large.

Third, in many computing centers, multiple HPC clusters are used as each upgrade adds a new hardware generation. These resources are commonly managed through separate queues making it difficult to launch applications which span multiple clusters [20].

Contribution

To solve the problem of efficiently supporting complex-moldable applications, we propose and evaluate a new RMS architecture that delegates resource selection to applications (more precisely their launchers). The RMS treats applications as *moldable* [14], which means that they can be launched on a variety of resource configurations. However, the resources allocated to an application cannot change after deployment. Therefore, an application cannot negotiate resources during its execution.

The proposed RMS assumes centralized control, targeting any system where such a control can be enforced such as supercomputing centers, enterprise grids or HPC clouds. Our approach is especially suited for computation centers with multiple clusters. Large scale platforms with

multiple administrative domains such as grids are outside the scope of this paper.

The contribution of this paper is threefold. First, it presents CooRM, an RMS architecture that allows applications, in particular complex-moldable applications, to efficiently employ their specialized resource selection algorithm. An example of such an application is presented in Section II. Second, it proposes an implementation using a simple RMS policy. Third, it shows that CooRM behaves well with respect to potential issues, such as scalability and fairness (see definitions in Section II).

The remaining of the paper is organized as follows. Section II motivates the work by presenting a complex-moldable application with a specialized resource selection algorithm. Section III proposes CooRM, a novel RMS architecture, which is evaluated in Section IV both using a simulator and a proof-of-concept implementation. Section V analyzes some features of the proposition while Section VI discusses related work. Finally, Section VII concludes the paper and opens up perspectives.

II. Problem Statement through a Motivating Example

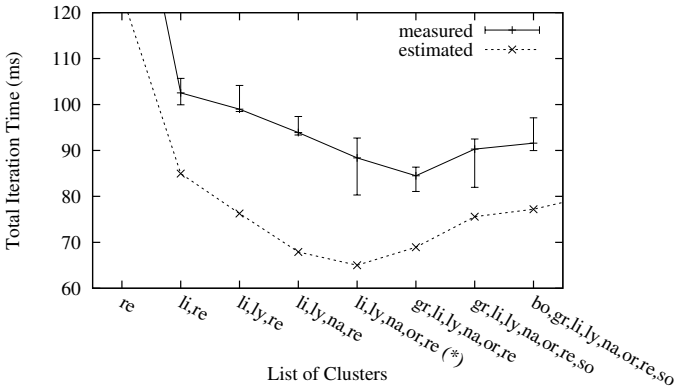
Motivating Example

Computational ElectroMagnetics (CEM) is a Finite-Element Method (FEM) which provides solutions to many problems, such as antenna performance. FEM works on a discretization of space (i.e., a mesh) over which it applies an iterative algorithm. Increasing the precision of the result is done by using a more refined mesh, which in turn increases the computation-time and the required amount of memory. Such applications are usually run on a single cluster, in order to take advantage of the high-speed interconnect. However, the increasing need for more precision is pushing towards a multi-cluster execution.

As part of the French ANR DiscoGrid project[1] a CEM application was ported to allow multi-cluster execution. The resulting application uses TCP to efficiently couple MPI codes running on multiple clusters. Thus, the application has a complex structure, the parameters being the number of processes to launch on each cluster.

Experiments showed that multi-cluster execution of the studied application can indeed lead to a smaller execution time, provided the resources are carefully chosen. The speed-up depends both on cluster metrics (node computing power, SAN latency and bandwidth), as well as the inter-cluster WAN latency and bandwidth (see Figure 1). The complexity of determining the exact number of clusters and nodes per cluster that minimize execution-time is exponential. However, it turns out that it was not so difficult to design an efficient resource selection heuristic based on the application's performance model and the resource metrics [10].

[1]ANR DiscoGrid, 2005–2009, http://www-sop.inria.fr/nachos/team_members/Stephane.Lanteri/DiscoGrid/



Bordeaux, **Gr**enoble, **Li**lle, **Ly**on, **Na**ncy, **Or**say, **Re**nnes, **So**phia

(*) Optimal solution found by our resource selection algorithm.

Fig. 1. Performance of a CEM application for various cluster sets of Grid'5000.

Unfortunately, using such an algorithm in practice is difficult, because current RMSs do not offer an adequate interface. Even RMSs that have been specifically designed to support moldable applications are not satisfactory, because the number of resource configurations to enumerate is exponential (see Section VI).

Problem Statement

Seeing that application-specific resource selection algorithms can improve performance, this paper focuses on the following question: **What is the interface that an RMS should provide, to allow each application to employ its specialized resource selection algorithm?**

How such resource selection algorithms should be written is application-specific and outside the scope of this paper. However, if a clean and simple RMS interface exists, developers could provide application-specific launchers (which implement a resource selection algorithm) with their applications, so as to improve the response time experienced by the end users. As an example, the above CEM application gives an idea of the effort and benefits of developing such a launcher.

Since these selection algorithms might take some time, we are especially concerned with two issues:

a) Scalability: The selection algorithm should be called only when necessary, e.g., exhaustively iterating over the whole resource space is not practical. Since the result of the selection only depends on the available resources and not on the application's internal state (we treat applications as moldable, see Section I), *memoization*[2] can be used. Therefore, the system needs to reduce the number of unique inputs for which the selection algorithm is invoked, a metric that we shall call the number of **computed configurations**.

[2]Def.: save (memoize) a computed answer for later reuse, rather than recomputing it.

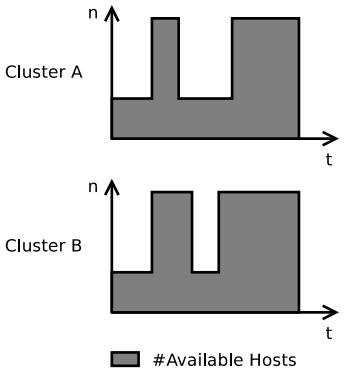


Fig. 2. Example of a view sent to an application.

b) Fairness: For applications with intelligent resource selection, a new fairness issue arises. An application A with a lengthy selection (e.g., ten seconds) should not be delayed (i.e., its end-time is increased) by an application B with a quicker selection (e.g., less than a second) submitted just after A. Since A has been submitted before, A should have a higher priority, therefore its resource selection should not be impacted by applications submitted after it. Thus, the RMS should make sure that applications are not delayed, as long as they have "reasonable" lengthy resource selection algorithms.

III. The CooRM Architecture

This section introduces CooRM, an RMS architecture which delegates resource selection to applications. First, the rationale of the architecture is given. Then, interactions between applications and the RMS are detailed. Finally, an implementation of the RMS policy is proposed.

A. Rationale

CooRM is inspired by a batch-like approach, where the RMS launches one application after the other as resources become available. Batch schedulers work by periodically running an algorithm which loops through the list of applications and computes for each one a start-time based on their resource requests.

Similarly, in CooRM each application i sends one **request**[3] r_i, containing the number of hosts to allocate on each cluster and the estimated runtime for which the allocation should take place. The RMS shall eventually allocate resources to the application, guaranteeing exclusive, non-preempted access for the given duration. Requests never fail, their start-time is arbitrarily delayed until their allocation can be fulfilled[4]. The application is allowed to terminate earlier, however, the RMS shall kill an application exceeding its allocation.

[3]One request is sufficient for an application which cannot change its resource allocation during execution, a simplification that we have made in Section I.

[4]Abusing this concept, if an application requests more resources than exist on the platform, the start-time is set to infinity.

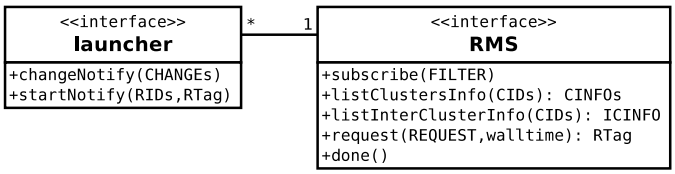


Fig. 3. Application Callbacks and RMS Interface in CooRM

In order to optimize an internal criterion, applications need to adapt their requests to the availability of the resources. To this end, CooRM presents each application i a **view** V_i which stores the estimated availability of resources. A view associates to each cluster a **Cluster Availability Profile** (`CAP`), which is a step function presenting the estimated number of available hosts as a function of time, as illustrated in Figure 2. Some hosts might not be available during a certain period, either because of the request of another, higher-priority application, or because of policy-specific decisions (e.g., applications may not run overnight). The views represent the currently available information which aids applications in optimizing their requests. For example, an application i can use its view V_i to compute a request r_i which would most likely minimize its completion time.

One more issue remains. When an event occurs (e.g., an application finishes earlier than it estimated), the views of applications might change and previously computed requests might be sub-optimal. For example, if a new resource is added to the system, an application that has not yet started might want to update its request in order to take advantage of this new resource. Therefore, until its start, each application i receives an up-to-date view V_i', so that it can send an updated request r_i' to the RMS.

The above approach can be considered a dynamic, distributed scheduling algorithm. The RMS is responsible for applying a **policy**, which decides how to multiplex resources among applications, while the **selection** of resources, which decides how to optimize the application's structure to the available resources, is handled by the applications themselves.

B. Actors and Interactions

The system consists of one or more applications, their launchers (which contain the application-specific resource selection algorithm) and the RMS. Since the interactions between the launcher and the application itself are programming-model dependent and do not involve the RMS, this section focuses on the interactions between the launcher and the RMS (Figure 3).

An application negotiates the resources it will run on through its launcher, which could run, either collocated with the RMS on a front-end, or on a distinct host (e.g., the user's computer). Before describing the RMS-launcher protocol, let us first define some data types:

- `FILTER` is a JSDL-like [6] filter to select candidate

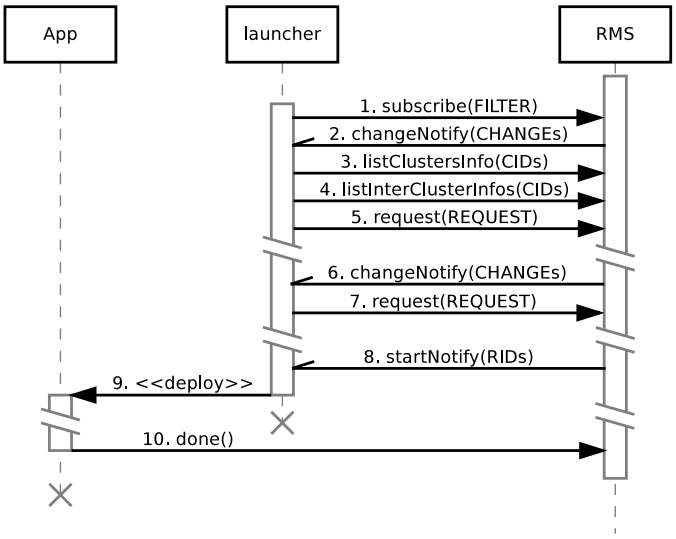


Fig. 4. Example of interactions between an RMS, an application and its launcher.

clusters. It specifies the minimum number of hosts, per-host RAM, total RAM, scratch space, etc.

- `CID` (cluster ID) uniquely identifies a cluster.
- `CINFO` (cluster info) stores the cluster's properties, e.g., the number of hosts, the number of CPU cores per host, size of RAM, size of scratch space, etc.
- `ICINFO` (inter-cluster info) stores information about the interconnection of one or more clusters, e.g., network topology, bandwidth and latency.
- `CAP` (see Section III-A) stores the number of available hosts as a function of time.
- `REQUEST` describes a resource request. It contains the number of hosts to allocate on each cluster and the duration of the allocation.
- `RID` (resource ID) uniquely identifies a host, e.g., by hostname.
- `CHANGE` represents a change event for a cluster. It is composed of the tuple { `CID`, `type`, `CAP` }, where `type` specifies whether cluster information, inter-cluster information or the availability has changed. In the first two cases, the launcher shall pull the information it requires using the interface provided by the RMS. In the latter case, the new `CAP` is contained in the message.
- Plurals are used to denote "set of" (e.g., `CIDs` means "set of `CID`").

Figure 4 presents a typical example of interaction between a single application (through its launcher) and the RMS: (1) The launcher `subscribe`s to the resources it is interested in. Depending on the input of the application, the launcher might use the filter to eliminate unfit resources like hosts with too little memory or unsupported architectures. (2) The RMS registers the application in its database and sends a `changeNotify` message with the relevant clusters and their `CAPs`; the launcher uses this data to update its local view of the resources. (3) Since the launcher has no previous knowledge about the clusters, it has to pull the `CINFOs` by calling `listClustersInfo` and (4) the `ICINFO` by calling `listInterClusterInfo`. (5) The launcher executes the resource selection algorithm, computes a resource request and sends it to the RMS. (6) Until these resources become available and the application can start, the RMS keeps the application informed by sending `changeNotify` messages every time information regarding the resources or their estimated availability changes. (7) The launcher re-runs the selection and updates its request, if necessary. (8) When the requested resources become available, the RMS sends a `startNotify` message, containing the `RIDs` that the application may use. (9) The launcher deploys the application. (10) Finally, when the application has finished, it informs the RMS that the resources have been freed by sending a `done` message.

For multiple applications, each launcher creates a separate communication session with the RMS. No communication occurs between the launchers. It is the task of the RMS to compute for each of them a view, so that the goals of the system are met.

C. A Simple RMS Policy

This section presents an example of an RMS policy implementation (as defined in Section III-A).

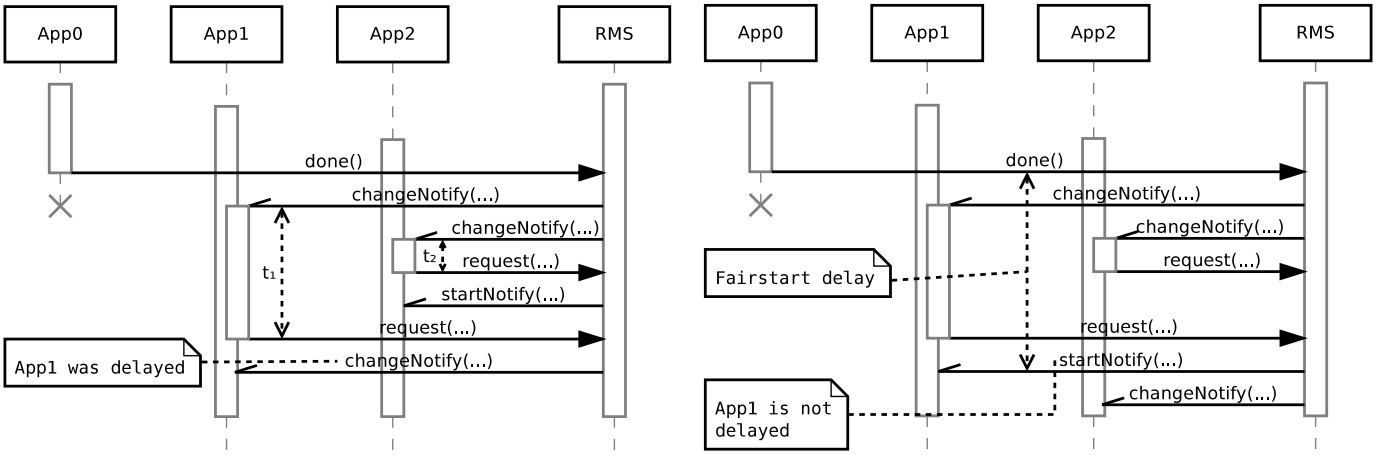
The policy is triggered whenever the RMS receives a `request` or a `done` message, similar to how rigid-job RMSs run their scheduling algorithm when a job is received or a job ends. In order to coalesce messages coming from multiple applications at the same time and reduce system load, the policy is run at most once every **re-policy interval**, a parameter of the system. The choice of this parameter is briefly discussed in Section IV.

The policy algorithm is similar to first-come-first-serve with repeated, conservative back-filling (CBF) [21]. For each cluster, a profile with the expected resource usage at future times is maintained, which is initialized to exclude the resources used by running applications. Then, for each waiting application, ordered by the time-stamps of their `subscribe` messages (which serve as an implicit priority), the algorithm executes the following actions:

1) Set the view of the current application to the current profiles.
2) Find the first "hole", where its request fits and store the found start-time.
3) Update the profiles to reflect the allocation of resources to this application.

If a stored start-time is equal to the current time, `RIDs` are allocated from the pool of free hosts and `startNotify` is sent to the corresponding application.

Fairness: Let us come back to the fairness issue (see the definition in Section II) and explain how CooRM solves it. Let us assume that three applications are in the system. `App0` is already running, while `App1` and `App2` have sent requests to the RMS and are waiting for the



(a) without fair-start delay

(b) with fair-start delay

Fig. 5. Fairness issue for adaptable applications

`startNotify` message. Also, let `App1`'s resource selection algorithm take d_1 seconds, while `App2`'s d_2 seconds, with $d_1 > d_2$. According to our goal of fairness, since `App1` arrived before `App2`, the latter should not be able to delay the former.

Figure 5(a) shows how `App1` could be delayed by `App2`. When `App0` sends the `done` message, both applications want to take advantage of the newly freed resources. Since `App2` has a quicker resource selection than `App1`, without any mechanism in place, the RMS would launch it immediately. `App1`, which has a slower resource selection, could not take advantage of these resources, despite the fact that it had priority over `App2`.

Therefore, in order to ensure fairness for applications with a lengthy resource selection, resources allocated to an application are not immediately released after a `done` message, but are artificially marked as occupied for a **fair-start** amount of time (see Figure 5(b)), which is an administrator-chosen parameter. This additional delay, allows high-priority applications with intelligent resource selection enough time to adapt and request the resources that will be freed after the fair-start delay expires. How to choose this parameter is discussed in the next section.

IV. Experiments and Results

This section evaluates CooRM. First, an overview of the experiments is given, then the simulation results are analyzed. Finally, the simulations are validated, by comparing the results with values obtained using a real implementation.

A. Overview

The purpose of CooRM is to allow applications with complex structure (we call them *complex-moldable*) to employ their own resource selection algorithms, while ensuring scalability and fairness as defined in Section II. In our experiments, a multi-cluster CEM application is used as a complex-moldable application. However, it is unlikely that a platform will be reserved for running only applications with a complex structure. Therefore, the testing workloads also need to include single-cluster applications with a simple structure (we call them *simple-moldable*), submitted, as it is currently done, either as *rigid* jobs (the host-count is fixed by the user) or using a list of *moldable configurations* (the host-count is chosen from several values, but cannot be changed once the job started).

Comparison to an existing system is difficult, since there are no RMSs that offer the exactly same services. Thus, CooRM is compared to an RMS that gives equivalent schedules, so as to focus on the overhead that CooRM may bring. To this end, we have chosen OAR [9] both because it uses the CBF scheduling algorithm and due to its support for moldable jobs. In essence, each job is submitted with a list of host-count, wall-time pairs and OAR greedily chooses the configuration that minimizes the job's completion-time. Exhaustively enumarating all the configurations is at most linear for simple-moldable applications: for n_C clusters with n_H host, one has to enumerate at most $n_C \times (n_H - 1)$ configurations. However, the number of configurations can be exponential for complex-moldable applications: for n_C clusters with n_H hosts per cluster, the CEM application presented as motivation has $(n_H + 1)^{n_C} - 1$ configurations. Clearly, it is impractical to enumerate them all.

Simulator: We have written a discrete-event simulator to study the behaviour of CooRM, which includes a "mock" implementation of OAR (we shall call it OAR-sim). For CooRM, the re-policy interval has been set to 1 second since we want a very reactive system. The fair-start has been set to 5 seconds, a good value to allow the applications we target enough time to run their resource selection algorithms. This parameter will be further discussed in Section IV-C.

For CooRM, applications receive their views (Figure 2) and "instantly" (i.e., the simulation time is not advanced) send a new request aimed at minimizing their completion-time. For OARsim, applications enumerate all the possible configurations and let the RMS choose which one to execute.

Resource Model: Resources are made of n_C clusters, each having $n_H = 128$ hosts. To add heterogeneity, the i^{th} cluster ($i \in [2, n_C]$) is considered $1+0.1\times(i-1)$ times faster than the 1^{st} cluster. The clusters are interconnected using a hierarchical model of a WAN, with latencies typically found over the Internet.

Application Model: We generated two types of workloads. First, workloads W_0 include only "legacy" applications to compare CooRM with OARsim. They were generated by taking packs of 200 consecutive jobs from the LLNL-Atlas-2006-1.1-cln trace from the parallel workload archive [1]. Since traces do not contain enough information to reconstruct the moldability of applications, we consider that 20% of the applications are simple-moldable, having an Amdahl's-law speed-up. The remaining 80% of the applications are considered rigid, with host-counts and execution-times found in the traces, subject to the speed-up of our resource model. The job arrival rate is set to 1 application per second, as the issues we are interested in appear when the system is under high load.

Second, we generate workloads which include complex-moldable applications. Since, the number of configurations to submit to OARsim for CEM is exponential in n_C (see above why), we only test CooRM with these workloads. We only present a subset of setups, so as to focus on the most interesting ones. Workloads W_1 have been generated starting from W_0 in which one instance of the CEM application (see Section II) has been inserted. These workloads act as a reasonable case, with a realistic mix of applications. Workloads W_2 have been generated starting from W_0 by replacing 50% of the jobs with instances of the CEM application. These workloads are the worst-case for CooRM that we have found during all the experiments. Workloads W_3 contain 100% CEM application, to test the scalability in an extreme case, with only complex-structure applications.

B. Scalability

We are interested in the following metrics: the number of **computed configurations** (defined in Section II) and the **simulation time** (measured on a single-core AMD Opteron™ 250 at 2.4 GHz) which gives us the CPU-time consumed on the front-end to schedule all applications in the workload. For the case where the launchers and the RMS are run on separate hosts, we also need to measure the network traffic. Therefore, we measure the **total size of the messages**, by encoding CooRM messages similarly to Corba's CDR. For each of these metrics, Figure 6 plots the minimum, maximum and quartiles.

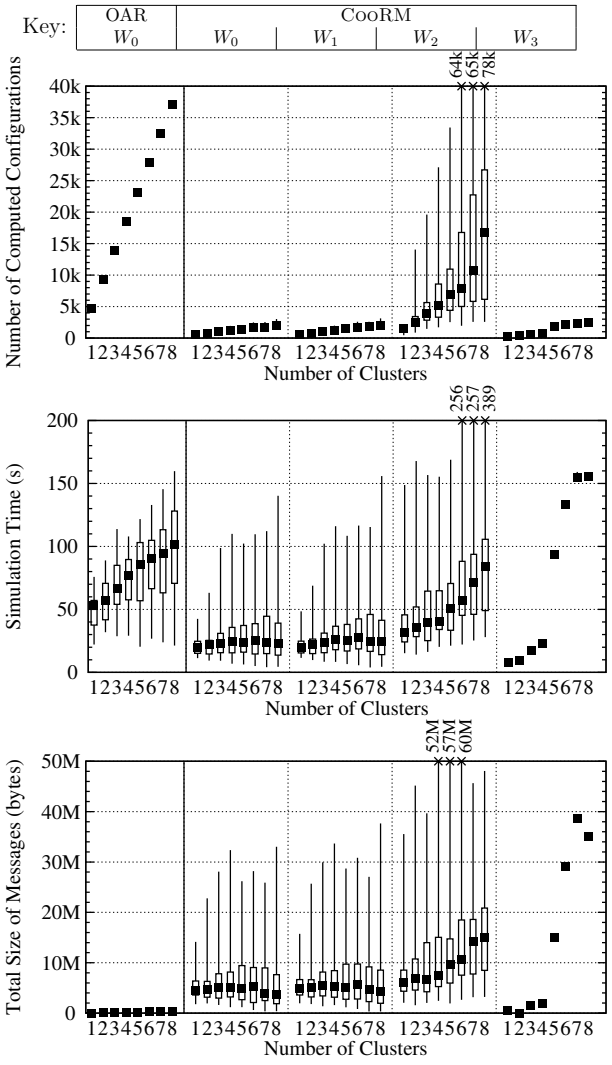


Fig. 6. Simulation results for OAR (W_0) and CooRM (W_{0-3}) for 1 to 8 clusters.

For simple-moldable applications (W_0), the results show that CooRM outperforms OARsim both regarding the number of computed configurations and the simulation time. We remind the reader that the obtained schedules are equivalent. For CooRM, more data needs to be transferred between the RMS and the applications, nevertheless, the total size of the messages is below 35 MB for 200 applications, in average 175 KB per application, which is a relatively low value for today's systems.

For complex-moldable applications, the results show that introducing one single CEM application does not significantly influence the metrics (W_1). Workloads W_2, which are the worst case, increase the values of the metrics, nevertheless, the scalability of the system holds. The network traffic is below 60 MB, in average 300 KB per application, a value which can be handled by today's systems. The time to schedule all applications is below 400 s, usually below 100 s, which is quite low, considering

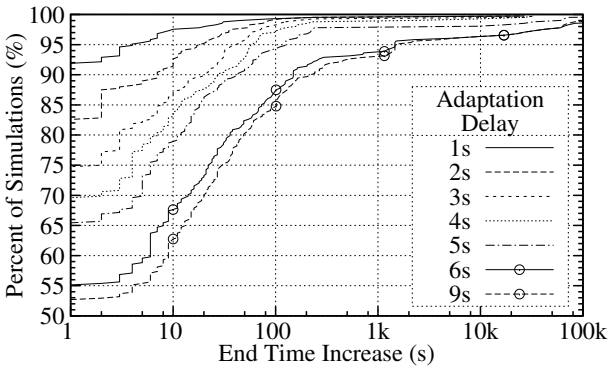
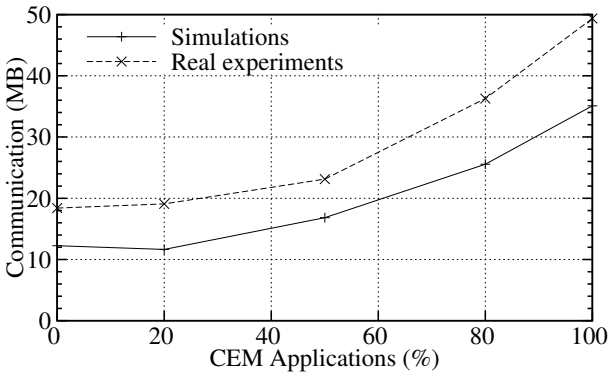
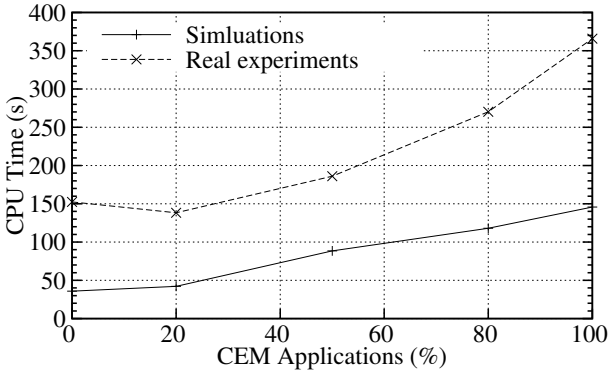


Fig. 7. Unfairness caused by insufficient fair-start delay.



(a) Network Usage



(b) CPU Usage

Fig. 8. Comparison between simulations and real experiments.

that it includes the time taken by the CEM resource selection algorithm.

For W_3, an extreme case where only CEM applications are present, the metrics tend to have smaller values. This is due to the fact that the resource selection algorithm tries to select all the hosts from a cluster, thus applications are better "packed". Therefore, the number of computed configurations is significantly reduced.

To sum up, results show that CooRM scales well both for existing workloads and workloads in which the number of complex-moldable applications is large.

C. Fairness

Let us now discuss the importance of the fair-start parameter (see Section III-C). To simulate applications with lengthy selections, we took workloads W_1 (see Sect. IV-A) and added to the CEM application an **adaptation delay**: when receiving a new view, the launcher waits for a timeout to expire, before sending a new request. Thus we are interested in how much the application ends later compared to an "instant" response.

Figure 7 shows the empirical cumulative distribution of the end-time increase, which is the end-time for the plotted adaptation delay minus the end-time when the adaptation-delay is zero, for an otherwise identical experiment instance. We note that the higher the adaptation delay, the more the CEM application is delayed. In particular, when the adaptation-delay is higher than the fair-start delay, the application is significantly impacted; in up to 5% of the cases the application was delayed more than an hour. Therefore, the fair-start delay should be set to a high-enough value, so that complex-moldable applications have time to compute a new request.

Note that a high fair-start delay makes resources idle longer, thus increasing resource waste. Depending on the average job execution time, a high fair-start delay might not be acceptable. For example, the traces used in the experiments have an average job runtime of 1.5 h. Thus, 0.5% of the resources would be wasted, for a fair-start delay of 30 s, which is quite small. In contrast, if the average job runtime were 5 min, 10% of the resources would be wasted. Thus, an administrator has to reach a compromise between resource waste and fairness.

D. Validation

We have developed a proof-of-concept implementation of CooRM in Python to validate the values obtained by simulations. The communication protocol has been ported to Corba as it was straightforward. Since we focus on the RMS-launchers interactions (which are functionally equivalent to the simulations), launchers are only deploying a sleep payload.

For these experiments, we used an instance of workload W_0 in which we replaced a percentage of applications with instances of the CEM application, similarly to W_2 and W_3 (see Section IV-A). We used the same resource model as during simulations, with $n_C = 8$. The RMS was run on the first processor, while all the launchers were run on the second processor of a system with two single-core AMD Opteron^{TM} 250 processors, running at 2.4 GHz.

Figure 8(a) compares the TCP traffic generated in practice to the simulation results. Values obtained in practice are up to 50% higher than those obtained by simulations. This is caused mainly because of Corba's IIOP overhead, but also because we neglected some messages in the simulations. However, the generated traffic is of the same order of magnitude, therefore we argue that the scalability from

the network perspective is validated, even if the RMS and the launchers run on separate hosts.

Figure 8(b) compares the simulation time to the CPU-time consumed by the whole system (RMS and launchers). Practical values are up to 3.3 times higher than simulations. However, this is to be expected, since data needs to be marshalled/unmarshalled to/from CORBA. Also, the measured CPU-time includes starting up the launchers which, due to the need of loading the Python executable and compiling the byte-code, is non-negligible. Nevertheless, the load on the CPU is quite low, which shows that CooRM scales well as the number of applications with intelligent resource selection increases.

To sum up, the differences between theoretical and practical results are implementation specific. A different implementation (e.g., another middleware than CORBA) might show values closer to the simulations. Therefore, we argue that the conclusions drawn in Section IV-B and IV-C are of practical value.

V. Discussions

The RMS policy proposed in Section III-C does not attempt to do any global, inter-application optimizations. We deliberately chose a simple approach, as, in real-life, access to HPC resources is either paid or limited by a quota. This forces users to choose the right trade-off between efficiency and completion-time. CooRM is flexible enough to allow applications to consider quota-related criteria when selecting resources. Nevertheless, should a better RMS policy be found, it can readily be used by changing only a few implementation details.

Simple-moldable applications (characterized by a single integer, i.e., the number of processors) can be reshaped [24], so as to improve system throughput. Such an approach might be extended and used as an alternative CooRM policy; however, whether this benefits complex-moldable applications needs to be studied.

An alternative design could have allowed applications to submit *utility functions* [19], then let the RMS optimize global utility. We have refrained from adopting such a solution, because we do not believe that utility can be translated into comparable values for applications with a complex structure.

VI. Related Work

We group related work into three categories: theoretical studies of moldability, RMS support for moldability and application-level scheduling.

A. Theoretical Studies of Moldability

Let us first review articles that have studied moldability from a theoretical perspective.

To our knowledge, Feitelson [14] was the first to introduce the word *moldable* in the context of parallel job scheduling, defining it as a job for which the scheduler can choose the number of processors at launch. We extend this definition and differentiate between *simple-moldable* applications, whose simple structure allows the scheduler to decide how to launch it, and *complex-moldable*, whose complex structure requires an application-specific resource selection algorithm.

The benefits of treating jobs as moldable has been highlighted in [17]. However, the cited work assumes that applications have constant area (i.e., doubling the number of hosts halves the execution time), which does not hold for many applications. Other works assume more realistic moldable application models [13] and study how to efficiently schedule them on super-computers [25], [24].

In contrast, our approach assumes that resources are not homogeneous. Applications include several codes, which can be launched on multiple clusters. The performance models of such applications become complex and exhaustively enumerating all configurations to the RMS, as done in cited works, is impractical. Indeed, if a multi-cluster application can choose any number of hosts between 0 and n_H on n_C clusters, then the number of resource configurations the application can run on is $(n_H+1)^{n_C}-1$, which is exponential.

B. RMS Support for Moldability

This section reviews how moldable applications are supported in current Resource Management Systems.

High-performance computing systems, such as clusters, federation of clusters or supercomputers, are managed by RMSs which offer the same base functionality: submitting jobs, which are launched once the requested resources are available, or making advance reservations, which have a fixed start-time [15]. The resources are chosen by the RMS based on a user-submitted Resource Specification Language (RSL).

Let us review some RSLs in increasing order of their expressiveness. Globus' RSL [16], which aims at targeting common features found in all cluster schedulers, specifies requirements like the number of hosts and wall-time, thus jobs are rigid from the RMS's point-of-view. For example, Oracle Grid Engine (former Sun Grid Engine) does not allow the user to express moldability [2]. OGF's JSDL [6] and SLURM [18] improve on this, allowing ranges (minimum and maximum) to be used for the host-count, thus supporting a basic form of moldability. However, there is a single wall-time, which cannot be described as a function of the allocated resources. This reduces back-filling opportunities, as an application cannot express the fact that it frees resources earlier if more hosts are allocated to it.

An improved support for moldability can be found in OAR [9]. The user gives a list of host-counts and wall-times, then the RMS chooses the configuration which minimizes the job's completion time. This approach allows more flexibility in describing the resources that an application can run on, however, exhaustively describing the whole set of configurations (limited to the resources

available to the RMS) may be very expensive. Nevertheless, this approach seems compelling if the number of configurations is small, which is why we chose to compare against it.

The Moab Workload Manager [3] supports a similar way of scheduling moldable jobs (which are call "malleable jobs" in the Administrator's Guide). The user provides a list of host-count and wall-times (called "task request list"), from which the RMS will chose the resource configuration that minimizes both job completion time and maximizes job utilization. Unfortunately, we have been unable to find more details about the employed algorithm, which is why we refrained from comparing against Moab. At any rate, one would have to enumerate all configurations, similarly to using OAR, which would only solve our problem if the number of configurations is small.

C. Application-level Scheduling

To optimize scheduling of individual applications, *brokers* (also called *application-level schedulers*) have been proposed which first gather information about the system, then use advance reservations [26] or *redundant requests*.

Advance reservations are used as they offer a solution to the co-allocation problem [20], however, they also fragment resources [23], thus leading to inefficient resource utilisation. Redundant requests are multiple jobs targeted at individual clusters: when one of these jobs starts, the others are cancelled. Redundant requests are harmful as they worsen estimated start times and create unfairness towards applications which cannot use them, by hindering back-filling opportunities [12]. The impact of using redundant requests for emulating moldable jobs has not been studied, however, we expect them to be at least as harmful.

To retrieve information about the available resources, grids expose information through the GLUE schema [5], which presents both current resource availability and scheduled jobs. However, the future estimated resource occupation (as computed by the RMS) is not provided. Estimating when a request might be served is cumbersome, since the application has to emulate the scheduling algorithm of the RMS. Even if the approach were theoretically possible, jobs are usually not exposed in grids, due to privacy concerns.

Similarly to GLUE, resource- and job-related information can be accessed through the WIKI Interface [3]. Besides the drawbacks highlighted above, this interface has been designed for system-level consumption, between a site-level scheduler and a cluster-level scheduler, and applications cannot connect to it.

In contrast to the above solutions, CooRM allows applications to select the resources they need without resorting to workarounds. Co-allocation in a single administrative domain is provided without fragmenting resources. For selecting the desired resources, applications are provided with the best information that the RMS has about the current and future availability of the resources.

Pilot jobs [11] (i.e., submitting container jobs inside which tasks are executed) are used either to reduce RMS overhead, or to allow on-the-fly resource acquisition. For the first case, pilot jobs can be readily used with CooRM. For the latter, CooRM does not deal with dynamic resource usage, which has been left as future work.

The AppLeS project [7] offers a framework for Application-Level Scheduling. The moldable-application scheduler (called "SA") was designed to improve scheduling on a single super-computer. SA gets a list of host-count and wall-times from the user, then chooses a job size which would reduce turn-around time. The choice is made at submit-time, which, since it does not use the most up-to-date information about the state of the resources, does not always achieve the best results.

While we share some of the concerns of the AppLeS project, our focus is to define a clean and simple application-RMS interaction, which would allow effective usage of multi-cluster platforms. For example, CooRM notifies the applications when the state of the resources changed and, through it's fair-start mechanism, ensures that the state of the resources is somewhat more stable. Therefore, applications have better information about the state of the resources and can take better decisions.

VII. Conclusion

This paper proposes CooRM, an RMS which allows complex-moldable applications to employ their specialized resource selection algorithms. This allows such applications to select resources so as to optimize their structure, thus enabling an efficient execution.

When used in multi-cluster environments, applications can easily span multiple clusters. Such platforms are common in today's computing centers, as several generations of hardware coexist.

In the end, CooRM offers a clean and simple interface for resource selection, so as to allow developers to easily create application-specific launchers. Both simulations and real experiments using the resource selection algorithm of a multi-cluster CEM application have been presented. They have shown that the approach is feasible and performs well with respect to scalability and fairness.

Future work can be divided in two directions. First, CooRM assumes that clusters are perfectly homogeneous and that the choice of host IDs inside a cluster can be left to the RMS. However, large clusters and supercomputers feature non-homogeneous networks (such as a 3D torus [4]) with varying latencies and bandwidths between hosts. While research is still in progress, in future, applications might want to optimize their resource selection for an efficient placement inside non-homogeneous clusters. For example, resources might be selected so that codes which communicate often are placed close, while those who communicate rarely are placed on distant hosts.

Second, current HPC RMSs lack support for applications with dynamic resource requirements. For example, Adaptive Mesh Refinement (AMR) simulations [8] unpredictably change their resource requirements during execution. Currently, the users of such applications are forced to reserve enough resources to meet the peak demand of the application, which is inefficient. In future, we aim at extending CooRM with runtime resource negotiation, so as to efficiently support such applications.

VIII. Acknowledgments

This work was supported by the French ANR COOP project, n°ANR-09-COSI-001-02.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (https://www.grid5000.fr).

References

[1] Parallel workload archive. www.cs.huji.ac.il/labs/parallel/workload.

[2] qsub man page – Sun Grid Engine. http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html.

[3] Adaptive Computing Enterprises, Inc. Moab workload manager administrator guide, version 6.0.2. http://www.adaptivecomputing.com/resources/docs/mwm/index.php.

[4] N. R. Adiga, M. A. Blumrich, et al. Blue Gene/L torus interconnection network. *IBM J. Res. Dev.*, 49:265–276, 2005.

[5] S. Andreozzi et al. GLUE Specification v. 2.0. Report GDF.147, OGF, 2009.

[6] A. Anjomshoaa, F. Brisard, et al. JSDL v. 1.0. Report GDF.136, OGF, 2008.

[7] F. Berman, R. Wolski, H. Casanova, W. Cirne, et al. Adaptive computing on the grid using AppLeS. *IEEE Trans. Parallel Distrib. Syst.*, 14:369–382, 2003.

[8] G. L. Bryan, T. Abel, and M. L. Norman. Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation. In *SC '01*.

[9] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, et al. A batch scheduler with high level components. *CoRR*, abs/cs/0506006, 2005.

[10] E. Caron, C. Klein, and C. Pérez. Efficient grid resource selection for a CEM application. In *RenPar'19*, Toulouse, France, 2009.

[11] A. Casajus, R. Graciani, et al. DIRAC pilot framework and the DIRAC workload management system. *Journal of Physics: Conference Series*, 219(6), 2010.

[12] H. Casanova. Benefits and drawbacks of redundant batch requests. *Journal of Grid Computing*, 5(2):235–250, 2007.

[13] A. B. Downey. A model for speedup of parallel programs. Technical report, 1997.

[14] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, London, UK, 1996. Springer-Verlag.

[15] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. In *Job Scheduling Strategies for Parallel Processing*, 2004.

[16] Globus Alliance. Resource specification language. www.globus.org/toolkit/docs/5.0/5.0.2/.

[17] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213, Washington, DC, USA, 2004. IEEE Computer Society.

[18] M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[19] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Węglarz. A multicriteria approach to two-level hierarchy scheduling in grids. *Journal of Scheduling*, 11(5):371–379, 2008.

[20] H. Mohamed and D. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *CCGrid'05*, volume 2, pages 784–791. IEEE CS Press, 2005.

[21] A. W. Mu'alem and 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):529–543, 2001.

[22] A. Ribes and C. Caremoli. Salome platform component model for numerical simulation. *COMPSAC*, 2:553–564, 2007.

[23] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proc. of IPDPS'00*, pages 127–132, 2000.

[24] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. *CLUSTER'03*, page 92, 2003.

[25] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *HiPC'02*, volume 2552 of *LNCS*, pages 174–183. Springer Berlin / Heidelberg, 2002.

[26] A. Sulistio, W. Schiffmann, and R. Buyya. Advanced reservation-based scheduling of task graphs on clusters. In *Proc. of HiPC'06*, 2006.