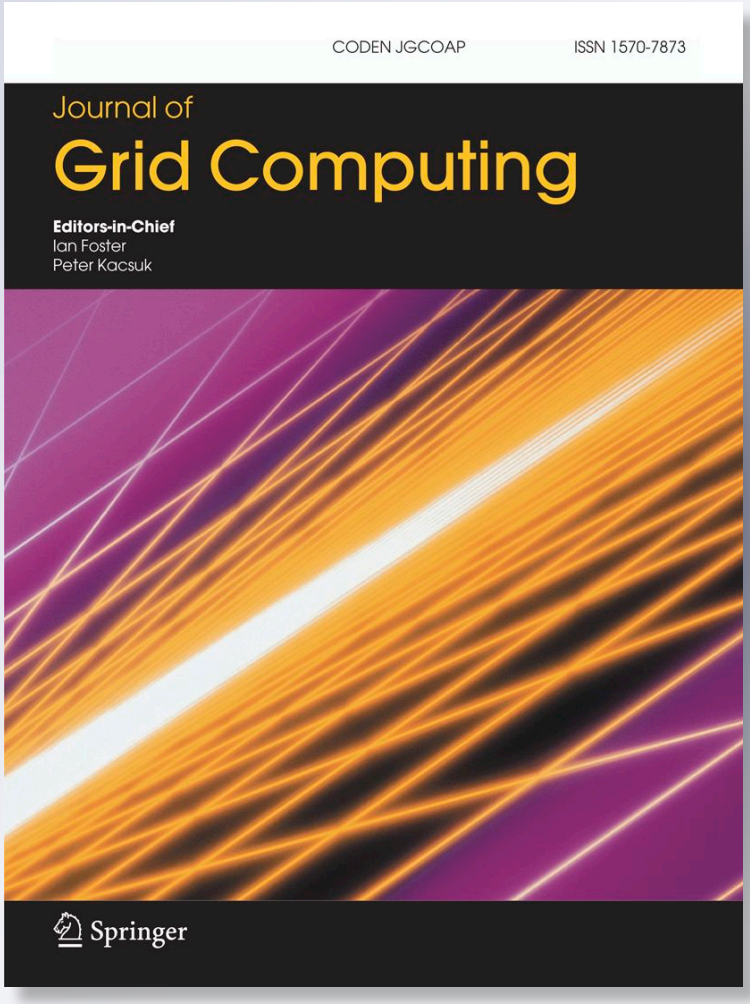
Adaptive Executions of Multi-Physics Coupled Applications on Batch Grids

Sivagama Sundari Murugavel, Sathish S Vadhiyar & Ravi S Nanjundiah

Journal of Grid Computing

ISSN 1570-7873
Volume 9
Number 4

J Grid Computing (2011) 9:455-478
DOI 10.1007/s10723-011-9197-9



Springer

J Grid Computing (2011) 9:455–478
DOI 10.1007/s10723-011-9197-9

Adaptive Executions of Multi-Physics Coupled Applications on Batch Grids

Sivagama Sundari Murugavel · Sathish S Vadhiyar · Ravi S Nanjundiah

Received: 6 May 2011 / Accepted: 22 September 2011 / Published online: 11 October 2011

Abstract Long running multi-physics coupled parallel applications have gained prominence in recent years. The high computational requirements and long durations of simulations of these applications necessitate the use of multiple systems of a Grid for execution. In this paper, we have built an adaptive middleware framework for execution of long running multi-physics coupled applications across multiple batch systems of a Grid. Our framework, apart from coordinating the executions of the component jobs of an application on different batch systems, also automatically resubmits the jobs multiple times to the batch queues to continue and sustain long running executions. As the set of active batch systems available for execution changes, our framework performs migration and rescheduling of components using a robust rescheduling decision algorithm. We have used our framework for improving the application throughput of a foremost long running multi-component application for climate modeling, the Community Climate System Model (CCSM). Our real multi-site experiments with CCSM indicate that Grid executions can lead to improved application throughput for climate models.

Keywords Adaptive framework · Batch systems · Climate models · Multi-component applications · Rescheduling

This work is supported partly by Ministry of Information Technology, India, project ref. no. DIT/R&D/C-DAC/2(10)/2006 DT.30/04/07 and partly by Department of Science and Technology, India, project ref no. SR/S3/EECE/59/2005/8.6.06.

S. S. Murugavel (✉) · S. S. Vadhiyar
Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India
e-mail: m.shivi84@gmail.com

S. S. Vadhiyar
e-mail: vss@serc.iisc.ernet.in

R. S. Nanjundiah
Centre for Atmospheric & Oceanic Sciences, Indian Institute of Science, Bangalore, India
e-mail: ravi@caos.iisc.ernet.in

1 Introduction

Multi-physics coupled MPMD applications are large scientific applications comprising of interacting component applications. Such applications have become prevalent since multi-disciplinary multi-component models are used to accurately model interacting physical processes or phenomena in the areas of climate, space weather, solid rockets, fluid structure interaction, heart disease and cancer studies [1–3]. Many of these applications, used to model evolution of physical systems with time, are also long running. For example, assessment of climate change needs to

be done far into the future and requires long simulations. Such simulations would require computational time running of the order of weeks (if not months). Such long running executions can be benefited by the ability to run on a collection of small scale HPC systems/clusters. Hence it is essential to execute such long running simulations on multiple parallel systems of a Grid. Integrating such applications in Grids is important for large scale deployment and use of scientific applications on Grid systems by the scientific community.

Many current large scale Grid frameworks [4, 5] consist of multiple distributed sites with each site having one or more batch systems with corresponding batch scheduling and queuing policies. Executing long running coupled applications on such Grids with multiple powerful batch systems is highly essential for providing high performance for the applications. However, execution of the components of a long running coupled multi-component application (MCA) on different queues of a batch Grid is challenging due to different factors. One practical challenge is to coordinate the concurrent executions of the different components in different batch queues. Component jobs submitted to different queues can have different startup times. However, the coupled nature of the applications necessitates the components to execute concurrently. Another challenge is associated with the maximum execution time limits imposed on individual jobs in most batch systems. Due to long running characteristics of the MCAs, the batch jobs will not be able to complete execution within the execution time limits. Component jobs will have to be checkpointed before reaching the execution time limits on the respective batch queues, resubmitted and continued from the previous executions on possibly a different set of batch systems. Thus, the set of *active batch systems* available for simultaneous execution of the components of a long running MCA can vary at different points of execution due to the different startup times of the components and due to jobs reaching the execution time limits.

We have developed **Morco** (**M**iddleware framework for l**o**ng **r**unning multi-**co**mponent applications), a framework for execution of long running coupled multi-component applications (MCAs) on multiple batch systems of a batch Grid. Our framework effectively addresses the challenges related to coordinating the executions of the components, and sustaining long running executions using multiple submissions of the component jobs to the systems. As the number of active batch systems available for execution can vary during execution, our framework performs selective migration and rescheduling of components, and provides dynamic resource allocation. We have also developed a robust rescheduling decision algorithm that considers various dynamic parameters including the availabilities of the batch systems, and execution rates to decide whether and where to reschedule when the available set of active batch systems changes. The rescheduling decisions involve the use of a genetic algorithm for component mapping, performance models for performance estimation of different schedules, and dynamic predictions of batch queue dynamics. Climate change studies need studies of a coupled ocean, atmosphere and biogeophysical system. The submodules for each of these components need to evolve in an interactive fashion. CCSM (Community Climate System Model) [6, 7] is a foremost long running multi-component climate model consisting of submodules for the climate components. Using **Morco**, we performed execution of CCSM for 8 days of wallclock time on a Grid consisting of three parallel systems and four batch queues and show that multi-site executions can provide better application throughput with climate simulations.

The primary unique and novel feature of Morco is that it is the first Grid middleware framework, to our knowledge, that supports coordinated execution of components of long running coupled multi-component applications using multiple submissions of batch jobs on multiple independently administered batch systems of a Grid. While the concepts of checkpointing and rollback used in our framework are not new and have been applied for single component applications and mostly on interactive platforms, ours is the first work that performs coordinated scheduling, rescheduling and migration of components of multi-component applications on batch systems considering dynamic batch queue events. Our framework is

intended for long running applications whose execution times are significantly greater than the execution time limits associated with the batch systems.

Section 2 explains the execution model used for executing different components of a multi-component application in multiple batch systems. Section 3 describes in detail the various components of our Grid framework and the component interactions. In Section 4, we explain our robust rescheduling decision algorithm. Section 5 describes the genetic algorithm we have used for component scheduling. Section 6 presents our experiment setup and various results related to long executions of a coupled climate model on multiple batch systems and Section 7 discusses applicability of our framework to generic multi-component applications. In Section 8, we briefly compare our work with other existing efforts for coallocation on batch systems. Section 9 summarizes our work and lists our future plans.

2 Execution Model

Coupled multi-physics or multi-component applications (MCAs) can be executed across multiple batch systems or queues by submitting a job with a subset of components to each queue. However, since the components are coupled, i.e. since they involve multiple dependencies, they cannot execute unless they are all allocated resources simultaneously. Simultaneous start or availability of resources on independently administered batch systems cannot be guaranteed. Moreover, batch queue systems are associated with limits for execution time for a job. Hence, we follow an execution model in which we compose a dynamically reconfiguring job script that can be made to execute any set of components as demanded at runtime and submit this to all queues. The configuration, i.e. mapping of the MCA component to processors on multiple systems, is dynamically determined at runtime, depending upon the set of available/active queues. The job script on a queue is resubmitted at the end of its active (execution) time limit on the queue.

The execution model followed for MCA execution across multiple batch systems is illustrated in Fig. 1 with an example of a two-component application executing on three batch sites.

In our execution model, when an MCA is executed on B batch systems or queues, B job submissions are made to the queues with different

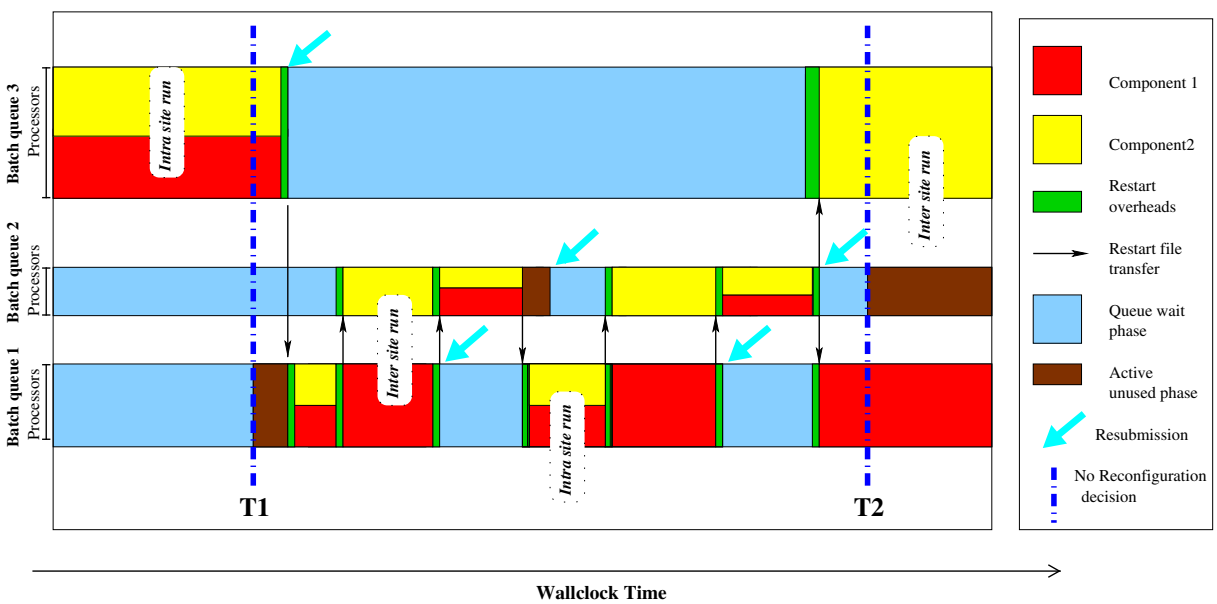


Fig. 1 Execution model: execution of a two-component application on three queues

processor requirements or request sizes. When some subset of submissions made to the batch systems, $B_A \subseteq B$, becomes active (when they are no longer queued and are ready for execution), the components of MCA are executed on a subset, $B_A\_running \subseteq B_A$, of this set of active batch systems, based on a scheduling/rescheduling strategy.

When a submission on a batch system reaches the execution time limit of the corresponding queue, the batch system, *sys*, becomes unavailable for components that were executing on the system, and is hence removed from B_A. If the MCA is not executing on this system, i.e., $sys \notin B_A\_running$, then the current execution is not affected and the system is removed from set B_A. However, if some MCA components are executing on this system, i.e., $sys \in B_A\_running$, all the components of MCA are made to create checkpoints of their states, remapped or rescheduled[1] to the updated set of active batch systems, B_A, and continued execution from the checkpoints of their previous states. Also, a job submission is made to *sys*. When a batch system *sys* becomes active, a decision of whether to reschedule or not is made and if the decision is to reschedule, the MCA executing on the already active systems is made to create checkpoints, the set B_A is updated to include the system that became newly active, components are remapped to new/updated B_A and continued execution from the checkpoints.

For rescheduling and continuing executions on a new configuration, we use the application level checkpointing mechanisms supported in MCAs. Coupled multi-component applications are typically executed for long durations to model long term physical phenomena. To support such long running simulations, an MCA contains *restart* facilities where the application can be made to store its execution state as *restart dumps* and simulations for an execution can be continued from the previous executions using the *restart dumps* of the previous executions. For example, the Community Climate System Model (CCSM) [6], a coupled multi-component application consisting of atmosphere, ocean, land and ice and coupler components, provides application level checkpointing mechanisms by which values of a set of primary variables are stored/checkpointed by the application to stable storage in the form of restart dumps [8]. During restart, CCSM reads the variable values from the restart dumps and continues execution from the previous time step. The time steps at which CCSM should checkpoint restart dumps can be specified by the user using input configuration files.

In summary, a decision of whether to reschedule is made, whenever the set B_A changes, either due to some batch systems becoming active or due to one of the active batch systems becoming unavailable upon the corresponding submission reaching its execution time limit on the system. This process continues until the MCA completes its required computations. In the case of CCSM, the process is continued until CCSM completes simulation of a certain number of simulated years.

In Fig. 1, the job script is submitted to all the three batch queues. Batch queue 3 becomes active first, and the two components of the MCA are executed on this batch queue. At T1, batch queue 1 becomes active. However, a decision is made not to reschedule the MCA components to this queue, and the application is continued on batch queue 3. After then, the job on batch queue 3 reaches its execution time limit, checkpoint/restart files are transferred from batch queue 3 to the remaining active queue, batch queue 1, and components are migrated/rescheduled to batch queue 1. Also, the job is resubmitted to batch queue 3. At some point, batch queue 2 becomes active, a scheduling decision is made, one of the components (marked yellow) is migrated to batch queue 2, while the other component (marked red) is made to execute on larger number of processors on batch queue 1. This results in an inter-site run involving batch queues 1 and 2. This process continues until the application finishes execution.

3 Morco: Grid Middleware Framework

The architecture of our framework is illustrated in Fig. 2. The following subsections describe the

[1] In this paper, the terms "rescheduling", "remapping" and "reconfiguration" refer to the same event of rescheduling application components to processors, and hence are used interchangeably.

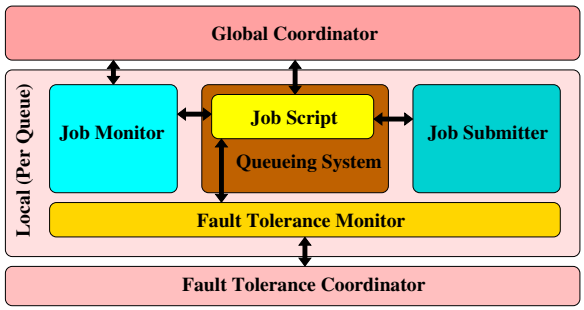


Fig. 2 Morco framework: components and interactions

various components, their primary functions and the interactions between the components.

3.1 Global Coordinator

The global coordinator daemon is the most significant server daemon and is executed on a location that is accessible from the front end nodes of all the systems. It contains a record of all the global static and dynamic information pertaining to the multi-component executions, and is informed of any event occurring on any site by the local job monitors. It also makes important decisions related to scheduling, rescheduling and processor allocation for components and instructs other components of the framework to implement these decisions. The rescheduling decisions followed by the global coordinator are explained in Section 4.

3.2 Fault Tolerance Coordinator

The fault tolerance coordinator is used for making fault tolerant decisions for the entire multi-physics application execution across all batch systems. It runs on the same location as the global coordinator, and interfaces with the fault tolerance monitors running on the batch systems. On receiving fault related events from a fault tolerance monitor of a batch system, the fault tolerance coordinator coordinates the cleanup of executions on the other batch systems, and notifies the global coordinator for relaunching the application components on the available batch systems and continuation from the previous execution.

3.3 Job Monitor

The job monitor daemons track the local behavior of the MCA jobs on the batch systems and interfaces with the global coordinator. A job monitor daemon is started for each queue used in the framework on the front end node of the respective system. The monitor notifies the global coordinator of various events including:

(1) sending a *START* message when the batch system becomes active due to the start of a MCA job in the system,
(2) sending a *STOP* message when a submitted MCA job nears the execution time limit of the queue, and
(3) sending a *STOPPED* message when the current execution has been stopped or a *TRIGGER* is received from the global coordinator, and the MCA job is ready for reconfiguration. The job monitor also processes and communicates the configuration information to the job scripts.

3.4 Fault Tolerance Monitor

The fault tolerance monitor daemons track the failures on the individual batch systems and interfaces with the fault tolerance coordinator. A fault tolerance monitor daemon is started for each queue on the front end node. It monitors the application progress on the systems using the output files generated by the application. In case of discrepancies, the monitor considers the execution status as a failure and reports the failure of the execution on the local system to the fault tolerance coordinator.

3.5 Job Submitter

Job submitter is another daemon that is started for each queue and runs on the front end node of the respective batch system. Its main functionality is to iteratively submit the *MCA job script*. It monitors the status of a submitted job, waiting for it to complete before submitting the next MCA job script such that only one MCA job exists in the system, in the queued or execution state, at a given time.

3.6 MCA Job Script

The *MCA job script* submitted to each queue supports dynamic reconfigurations of the application at runtime. At a given rescheduling event, the job scripts on the active running batch queues synchronize using the global coordinator[2] and simultaneously launch multiple MPI applications, one for each active queue, that coordinate to form a single MPI world. Existing middleware mechanisms including PACX-MPI [9] and MPICH-GX [10] can be used to coordinate the MPI components executed on different batch systems to form a single MPI world. These mechanisms can be used for communications between application components started on different batch systems.

After submitting a MPI application with a set of MCA's components on a set of processors, the job script waits for the MPI application to complete due to rescheduling or reconfiguration of the multi-component application by the coordinator. Upon completion due to reconfiguration, it waits for a new configuration from the coordinator and launches the corresponding new MPI components. The MCA job script exits execution if the execution time limit is reached for the job.

The general outline of the *MCA job script* is shown in Algorithm 1.

Algorithm 1 Outline of submit-script.sh

Algorithm:MCA Job script

```
repeat
    while component-config file and transfer-complete not
    present wait ;
    MyComponents = get components on MyQueue from
    component-config file ;
    repeat
        unpack restart files for MyComponents ;
        execute MyComponents on MyQueue using MPI ;
        /* Execution stopped due to
           reconfiguration                    */
        pack restart files for MyComponents ;
        RunStatus = status of the last run ;
        /* for fault tolerance                */
    until RunStatus is PASS ;
until exec time limit reached ;
```

[2]For brevity, we refer to the global coordinator simply as the coordinator for the rest of the paper.

3.7 Component Interactions: States, Transitions and Lifecycle of Components and Batch Systems

A batch system primarily has four states: *ACTIVE-RUNNING*, *ACTIVE-SILENT*, *INACTIVE*, and *STOP*. A batch system enters an ACTIVE-RUNNING state if the MCA job submitted to it has been allocated for execution and has started executing the MCA components on the system. An ACTIVE-SILENT state is similar to an ACTIVE-RUNNING state where the MCA job submitted to it has been allocated for execution. However, this active system is not chosen for execution of components, and hence waits (or remains silent) till it is chosen for execution. INACTIVE state is one in which the MCA job that is either in ACTIVE-RUNNING or ACTIVE-SILENT state reaches the execution time limit on the system and hence is aborted from execution. Finally, a STOP state is a temporary state in which the MCA job executing in a batch system in the ACTIVE-RUNNING state stops and waits for a new scheduling configuration from the coordinator.

The coordinator is a persistent daemon and has four states: *LISTENING*, *SIGNALING*, *WAIT-FOR-STOP* and *DECISION-MAKING*. In the LISTENING state, the coordinator waits for requests from the job monitor daemons running on the batch systems. Upon receiving a notification from a job monitor about a batch system entering the ACTIVE-RUNNING state or the INACTIVE state from an ACTIVE-RUNNING state, the coordinator enters the SIGNALING state where it sends signals to the other batch systems that are in their ACTIVE-RUNNING states to stop their running MCA jobs. It also sends signals to the batch systems that are in their ACTIVE-SILENT states. It then enters the WAIT-FOR-STOP state. In this state, it waits for acknowledgment from the job monitors of these active systems about stopping the MCA jobs. Upon receiving the acknowledgments, the coordinator enters the DECISION-MAKING state where it derives a new resource allocation for the MCA components to the subset of current batch systems in ACTIVE-RUNNING and ACTIVE-SILENT states, and sends the schedule to the

corresponding job monitors. If the new schedule does not include an active batch system, it informs the corresponding job monitor, and marks the batch system as entering the ACTIVE-SILENT state. The coordinator then goes back to the LISTENING state.

The job monitor daemon is also a persistent daemon running on a batch system and consists of *MONITOR*, *NOTIFY-START*, *NOTIFY-INACTIVE*, *NOTIFY-STOPPED*, and *WAIT-FOR-RESPONSE* states. It has a MONITOR state that monitors the status of the MCA job that is submitted and queued. It also monitors the execution progress of the MCA job that has been executing. If the executing MCA job reaches the INACTIVE state, it enters the NOTIFY-INACTIVE state, where it notifies the coordinator about the system entering the INACTIVE state, and goes back to the MONITOR state where it monitors the status of the new MCA job that been submitted to the system. When the batch system has allocated resources for MCA job execution, the job monitor enters the NOTIFY-START state where it notifies the coordinator of the system becoming active, and enters the WAIT-FOR-RESPONSE state. In this state, it waits for a new configuration from the coordinator. If the configuration contains this batch system, the monitor marks the batch system as ACTIVE-RUNNING, communicates with the job script to use the configuration and then goes back to the MONITOR state, where it tracks the progress of the MCA job. If the configuration does not contain this batch system, the monitor marks this batch system as entering the ACTIVE-SILENT state. When the batch system enters a STOP state due to the executing MCA job stopped by the coordinator to make a scheduling decision, the job monitor enters the NOTIFY-STOPPED state to inform the coordinator about the STOP status of the batch system.

The job submitter is a continuously-running service in a batch system and has *SUBMIT* and *WAIT* states. In the SUBMIT state, it submits a MCA job with a request size and component configuration specified in the configuration file sent by the coordinator. When the batch system enters the ACTIVE-RUNNING state, the job submitter enters the WAIT state where it waits for the batch system to enter the INACTIVE state. When the batch system enters the INACTIVE state, the submitter goes back to the SUBMIT state. The states and transitions of the job script are illustrated in the pseudocode of Algorithm 1.

A multi-component application (MCA) job script is submitted to each of the batch systems with a request for a specific number of processors by the job submitter. When a job on a batch system is active, its job monitor sends *START* to the coordinator. The coordinator decides whether to reschedule; if the decision is to not reschedule, it marks the system as ACTIVE-SILENT, informs the job monitor to not execute, and continues listening for the next event. If decision is to reschedule, it stops existing run (sends *TRIGGER* to job monitors of ACTIVE-SILENT batch systems), waits for *NOTIFY-STOPPED* from all active job monitors, determines new schedule, performs the restart file transfers and sends the new configuration information to all active job monitors. The job monitors, upon receiving this information, communicate with the job script to launch the next MPI execution. When a job on the batch system is close to time out, its job monitor sends *NOTIFY-INACTIVE* to the coordinator. The subsequent steps are the same as above, except that if this leaving system is an ACTIVE-SILENT system, a reconfiguration is not needed. If it is an ACTIVE-RUNNING system, a reconfiguration is mandatory with the new configuration not involving this system. The rescheduling decisions are described in detail in the next section.

4 Rescheduling Decisions

In this section, we first describe the types of rescheduling and their impact on the states of the components and the queues. We then give the motivations of rescheduling, and our rescheduling algorithm.

4.1 Mandatory and Optional Rescheduling

Rescheduling is of two types: *mandatory* and *optional*. Rescheduling becomes mandatory when a MCA job executing on a batch system, *sys*,

reaches the execution time limit on the system/queue. In this case, the coordinator has to derive a new schedule not taking into account the batch system, *sys*. The batch queue enters the INACTIVE state, the job monitor enters the NOTIFY-INACTIVE state, notifying the coordinator, and the coordinator subsequently enters the SIGNALING, WAIT-FOR-STOP and DECISION-MAKING states as described in Section 3.7.

When a MCA job submitted to a batch system, *sys*, or queue becomes ready for execution, rescheduling becomes *optional*. During this event, a new best schedule is determined by the coordinator. If the new schedule contains this batch system, *sys*, the batch queue enters the ACTIVE-RUNNING state, and the MCA application executing on other active batch systems, i.e., other batch systems in ACTIVE-RUNNING states, is rescheduled to a new configuration containing the batch system, *sys*. The new schedule may not contain some of the other ACTIVE-RUNNING batch systems. Those systems enter the ACTIVE-SILENT states. If the new best schedule is the same as the old schedule and does not contain this batch system, *sys*, the batch system enters the ACTIVE-SILENT state. In both cases of the new schedule containing and not containing the batch system, *sys*, the job monitor on the batch system, *sys*, enters the NOTIFY-START, notifying the coordinator about the system, *sys*, becoming active, and subsequently enters the WAIT-FOR-RESPONSE state. The coordinator in turn enters the SIGNALING, WAIT-FOR-STOP and DECISION-MAKING states as described in Section 3.7.

4.2 Motivations for Rescheduling

As described in the earlier sections, the number of active batch systems available for execution of MCA jobs can change during execution. At a given point of execution, when the available set of active systems changes, the coordinator has to decide whether to continue the MCA with the current set of active systems used for execution or to reschedule to a new set. If the coordinator decides to reschedule, it has to determine the best schedule or set of active systems for execution.

To make the rescheduling decisions, the coordinator has to compare different candidate schedules with the current configuration. The most important parameter for comparison between two sets of active systems is the execution rates of the application on the two sets. The execution rate denotes the rate of simulations of a long running MCA. For CCSM, it is the simulated climate time per unit wallclock time. For a total of M available active systems or queues, we construct a lookup table with $2^M - 1$ entries corresponding to all possible subsets of queues (except the null set). For each subset, the table contains the best configuration and the corresponding execution rate. A configuration for a subset of queues specifies the mapping of the MCA components to the queues in the subsets, and the processor allocation for each component.

We use a combination of the genetic algorithm described in Section 5 and real profiling runs to construct the lookup table. We first use the genetic algorithm that uses application performance models to determine the best estimated configuration and resource allocation for each subset of active systems. For each of these configurations, we perform real application profiling runs and obtain the actual execution rates. The actual execution rates of the configurations are stored in the lookup table and used for comparisons between different configurations. Thus, while the estimated execution rates by the performance models are used in the genetic algorithm to obtain relative rankings between the configurations, the actual execution rates obtained by the profiling runs are used in our rescheduling decisions, leading to overall accuracy of the rescheduling policies. The profiling experiments were MCA runs executed for short durations (for CCSM, 4 days of climate simulations) on each configuration with the same experimental setup as for the longer runs. The 4-day CCSM simulation was timestamped and execution times were measured to estimate the execution rate, in terms of number of climate days simulated per wallclock day, as well as the initialization and restart overheads. Note that we had constructed and used a lookup table as above to improve upon the results of the application performance model by performing a reasonable number of real experiments. For small number of queues and processor configurations, the size of the lookup table and

the resulting time to construct the lookup table are manageable. For larger number of queues, it becomes imperative to use the results of the application performance model directly.

In addition to the execution rates, the coordinator has to also consider other factors or parameters for comparison of configurations. A configuration with the best execution rate may contain a queue that can become inactive soon, i.e. the MCA job on the queue can near its execution time limit. Rescheduling to this configuration will lead to a situation where the application will have to be rescheduled again after very little progress in execution. Frequent rescheduling can result in high rescheduling overheads and cause overall loss in application performance. The coordinator also has to consider the times to next events on the configurations for its rescheduling decisions. The next event on a configuration of active queues can either be one of the queues in the configuration becoming inactive or an inactive queue outside the configuration becoming active.

Our rescheduling algorithm considers all these different factors and has a two-fold objective, (a) to minimize number of rescheduling events to minimize overheads, and (b) to use the best possible configuration across the set of available queues. It is based on a single-step look-ahead strategy, i.e., the current decision is based on selecting the option with best execution progress until the next reconfiguration event.

4.3 Rescheduling Algorithm

The algorithm is invoked by the coordinator when the set of active batch systems changes. The input to the algorithm is the set of active systems, S. The algorithm first obtains a list of best configurations on all subsets of S and sorts these configurations in the decreasing order of execution rates. It then initializes a *base configuration*, b. The base configuration is initialized to the current configuration of active systems used for execution if rescheduling is not mandatory. Rescheduling is mandatory when one of the active systems in the current configuration becomes inactive due to the MCA job reaching the execution time limit on the queue. In such cases, the base configuration is initialized to the configuration on top of the sorted list. Next, the algorithm compares each configuration, c, in the list with the current base configuration, b, and updates the base configuration, b, to c if c is evaluated to be better. After all configurations in the list have been thus evaluated, the base configuration gives the best configuration. If this configuration is the same as the current configuration used for execution, the coordinator decides to not reschedule. Else, the coordinator decides to reschedule the executing application to the best configuration. The *base configuration* in our algorithm is compared with not only the top configuration in the lookup table with the best execution rate, but with the set of good configurations in the table, since a configuration with a lower execution rate can be a better choice if its time to next event is higher, as illustrated later with sample cases (Figs. 4 and 5). The pseudo code of the algorithm is given in Algorithm 2.

Algorithm 2 Rescheduling decision algorithm

```
Algorithm:GetBestSchedule
S = Get_Subset_Configurations(); /* configuration
    with highest execution rate for each
    subset                                    */
S = Sort_decreasing(S) ; /* Decreasing sort of
    execution rate                            */
if rescheduling mandatory then
    best_conf = b = S(0);
    Remove S(0) from S;
    Base_rescheduling = TRUE;
else
    best_conf = b = existing configuration;
    Base_no_rescheduling = TRUE;
end
foreach schedule c in S do
    tb = next event on base configuration (b);
    tc = next event on schedule (c);
    tcb = time to make same progress in c as by b in tb;
    tbc = time to make same progress in b as by c in tc;
    if Base_no_rescheduling then
        T = set_crossover_point ;
        best_conf=compBaseNoRes(b,c,tb,tc,tcb,tbc,T,OH)
        if best_conf == c then
            b=c; Base_rescheduling= TRUE;
        end
    else
        best_conf=compBaseRes(b,c,tb,tc,tcb,tbc,OH) ;
        if best_conf == c then
            b=c
        end
    end
end
return best_conf ;
```

An important step in the algorithm is the comparison of a base configuration, b, with a new configuration, c. This comparison is based on the following conditionalities: (1) when b is the current configuration (i.e. default with no rescheduling) and (2) when the b is not the current configuration and is one of the schedules from the set S. These two kinds of comparisons are made by invoking the functions, *compBaseNoRes* and *compBaseRes*, in Algorithm 2.

If the base configuration, b, is the current configuration, and its execution rate is higher than the other configuration c, the coordinator continues executing the application with the current configuration. In other cases, following are the different parameters used for the comparison of b and c configurations:

1. T is the time to crossover point beyond which execution on the new configuration c will result in higher execution or simulation progress than execution on the base configuration. It can be observed that this crossover point exists only when the base configuration is the current configuration of execution, i.e. in *compBaseNoRes*.
2. tb and tc are the times to next events on b and c configurations respectively. The next event can be one the active queues in the configurations becoming inactive or an inactive queue outside the configurations becoming active.
3. tcb is the time that will be taken by the application when executed on configuration c to achieve the same simulation or execution progress that can be achieved if executed on configuration b in time tb, i.e. before the next event happens on configuration b. This is important for cases when the next event on b configuration happens earlier than the next event on c configuration. In such cases, the coordinator has to decide if executing application initially on b configuration and then rescheduling to c configuration after the next event happens with b at time tb will give an overall advantage to the application. Similarly, tbc is the time that will be taken by the application when executed on configuration b to achieve the same simulation or execution progress that can be achieved if executed on configuration c in time tc, i.e. before the next event happens on configuration c.
4. OH is the rescheduling overhead.

The parameter T is determined using the execution rates of configurations b and c. tbc and tcb are determined using the execution rates and the times to next events, tc and tb. Determination of tb and tc depends on the next event on the corresponding configurations. The next event on a configuration is either one of the active queues becoming inactive or one of the inactive queues becoming active. The time to the former event can be estimated by the coordinator at any point since it maintains a record of the time at which each queue last became active, and the execution time limit for a batch queue is anyway known a priori. For determining the time for an inactive queue becoming active event, the queue waiting time of the MCA job has to be estimated for the queue. For our current work, we predict the queue waiting time of an MCA job as the mean waiting time of jobs with similar request sizes from the queue history. This strategy can give only crude approximations and at best can only differentiate configurations with very low and very high queue waiting times. Predicting queue waiting time is challenging and accurate point predictions cannot be made [11, 12]. In future, we plan to use bounds on queue waiting times instead of point predictions, like the QBETS effort by Brevik et al. [12], in which upper bounds of queue waiting times are predicted. We then plan to use the lower and upper bounds of queue waiting time estimates for the two configurations b and c for MCA execution. This will result in a four-way comparison (e.g., MCA execution on configuration b with lower bound queue waiting time and on configuration c with upper bound queue waiting time) leading to four decisions. We can then analyze the four decisions by obtaining the worst case loss in performance due to wrong decisions, and make the most conservative decision.

All possible comparison cases and the resulting decisions are shown using a decision tree in Fig. 3. There are a total of fifteen cases of comparisons corresponding to the fifteen leaf nodes of the decision tree. The color of each leaf node indicates the decision applicable in that particular case.

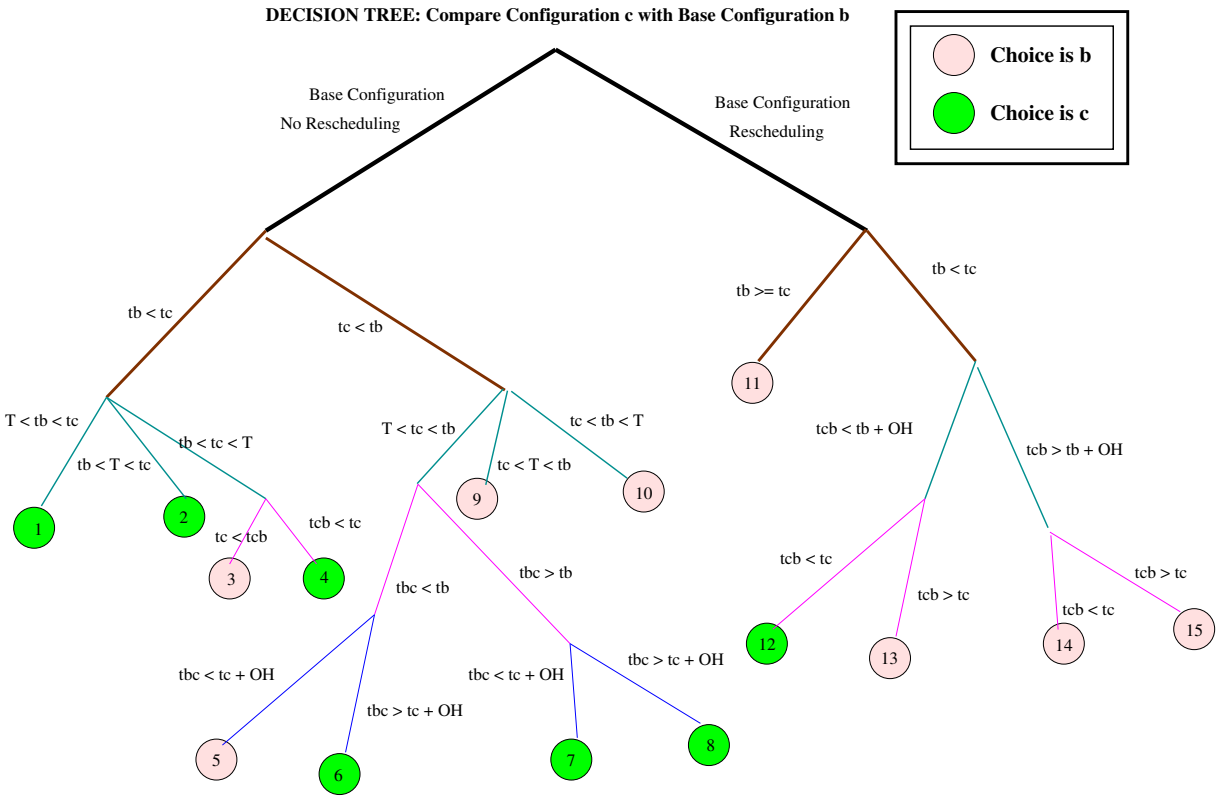


Fig. 3 Rescheduling decision tree

In the following, we describe two representative cases of comparisons. Similar reasoning behind the decisions can be applied for the other cases. P is a point in execution where rescheduling decision is made.

Case 5 This is a case, shown in Fig. 4, where continuation of the application execution without rescheduling on the current configuration, b, is compared with rescheduling to a new configuration, c. The rescheduling decision for this case primarily depends on the duration for which each configuration can last. In this case, $T < tc < tb$, i.e., time to next event with configuration b is greater than the time to next event on configuration c, and both are greater than the crossover point T, beyond which c gives higher application progress than b. Also tb is greater than tbc ($tbc < tb$), the time when the application when executed on configuration b can achieve the same progress that can be achieved if executed

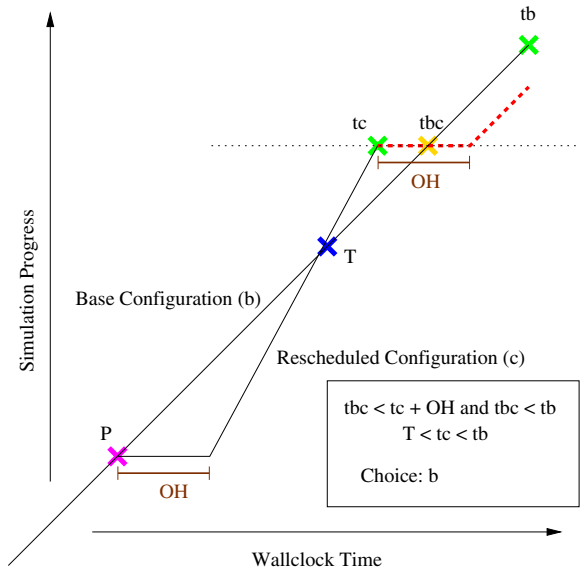


Fig. 4 Rescheduling decision case 5

on configuration c in time tc. Thus, at some point tbc before tb, choice b makes as much progress as choice c makes by tc. Also, in this case, $tbc < tc + OH$. This means that though at tc, choice c performs better than choice b, rescheduling at tc would result in a progress made at $tc + OH$ to be equal to the progress made by b at an earlier point, tbc ($< tc + OH$). Thus the application will make better progress if continued on the current configuration, b, than to reschedule on the configuration c, execute till tc, and then reschedule back to b, incurring a rescheduling overhead of OH, and continue execution after $tc + OH$, as shown by the dotted line in the figure. Hence, the choice for this case is b, i.e. continue execution on the current configuration.

Case 12 This is a case, shown in Fig. 5, where both configurations b and c involve rescheduling and hence rescheduling overheads. In this case, though configuration c gives lower execution rate than b, it has a higher time to next event, i.e., $tc > tb$. tc is also greater than tcb, the time when the application if executed on c will achieve the same progress that can be achieved if executed on b in time tb. Moreover, in this case, $tcb < tb + OH$. Beyond tcb, choice c outperforms choice b as indicated by the dotted line showing progress beyond tb with choice b. Thus, the application will make better progress if rescheduled to configuration c than rescheduling to configuration b, execute till tb, then reschedule to c and continue execution after $tb + OH$. Hence, the choice for this case is c.

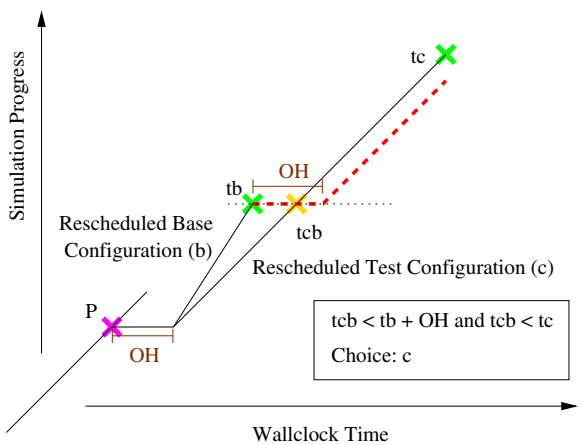


Fig. 5 Rescheduling decision case 12

5 Scheduling of Components

We have developed a genetic algorithm for scheduling of components to available queues. The sizes and locations of the active queues are inputs to the algorithm. Each chromosome is modeled as a string of length equal to the number of components, with each value indicating the processor sizes for each component. We then evaluate a chromosome for scheduling on a given set of active systems. For evaluating a chromosome that specifies the component sizes, we explore all possible distributions of the components, with the specific sizes, on the set of active systems. Thus, we try all possible component mappings to the active queues for a given processor allocation specified by the chromosome. For each mapping, we use an *application specific fitness function* to calculate a *fitness value for the mapping*. The maximum fitness value for all component mappings for a given chromosome is used as the *fitness value of the chromosome*. For example, for CCSM, we calculated the fitness value for a component mapping as the expected number of climate days that can be simulated within the maximum execution time limit of the batch systems.

The *application specific fitness function* used by the genetic algorithm is a multi-site execution performance model function that simulates the execution of the multi-component application on a set of active batch systems. Various application characteristics including amount of computations in components, pattern and amount of communications between the components etc. are embedded completely in the performance model and are not known to the components of our Morco framework, namely the scheduler and coordinator. Using this design, the Morco middleware is made generic to integration of any multi-component application.

For CCSM, to estimate the number of climate days, we use a multi-site execution performance model of CCSM that considers intra and inter site bandwidths and processor speeds to model the

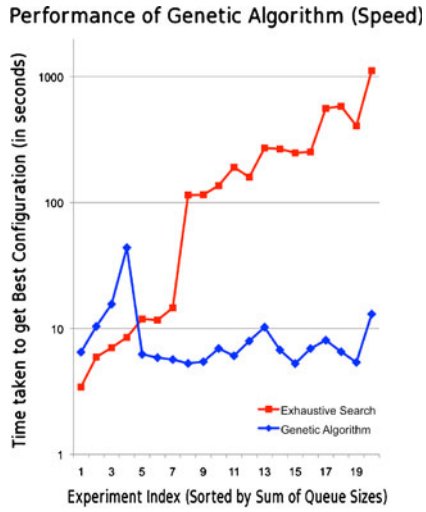
times for initialization, component computation and communication and restart transfers. These times along with the workflow pattern of the component execution are used in an event based simulator to compute the number of climate days that can be simulated within the maximum execution time limit. The workflow pattern of component execution in CCSM is as follows. Each component communicates the data processed by it to the coupler at periodic intervals. This interval of communication, *coupling period (CP)*, can be different for different components. Within each CP, a component performs some computations of its local data, receives data from the other components through the coupler, performs some computations of this received data and sends its processed data to the other components through the coupler. These phases are denoted as *send-to-recv computations (S-R)*, *recv communications (Re)*, *recv-to-send computations (R-S)* and *send communications (Se)*, respectively.

The performance model of CCSM takes as input, the total wall-clock time for CCSM execution, the CPs for the four components, the number of clusters, the inter-cluster bandwidth, and the allocation of processors in the clusters for the components. The model then models the execution workflow of CCSM to obtain the total number of simulated days or application throughput for the total wall-clock time available for CCSM execution. In order to model the execution flow and predict the number of simulated days for a given execution time (simulation rate), the performance model uses models for the different phases, namely, *S-R*, *Re*, *R-S*, and *Se* for each component. These *phase models* predict the execution times for the phases for a given number of processors. To construct these *phase models*, we conducted many experiments by executing CCSM with a medium resolution. The experiments were conducted across two AMD Opteron clusters, one with 16 cores and another with 8 cores, with different application and system configurations. For each experiment, the head nodes of the two clusters were connected to each other by one of 10 Mbps, 100 Mbps and 1 Gbps switched Ethernet. CCSM was then executed with a given distribution of the components to the processors and with a given allocation of processors to each component, and the times for the different phases were observed.

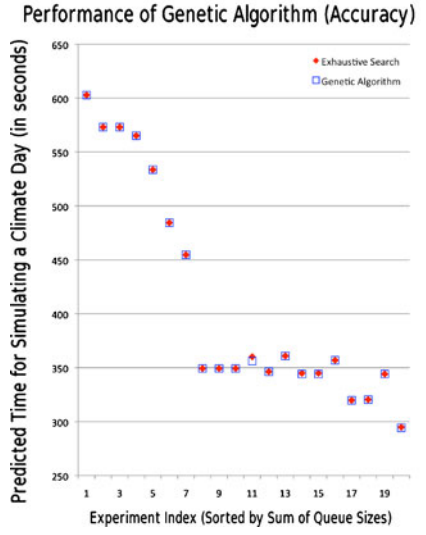
We used a simple equation, $computeTime = a + b/componentSize$, for modeling each of the computation phases of a component. *componentSize* is the number of processors allocated for the component and *computeTime* is the execution time corresponding to the computation phase. a and b denote the model coefficients and were obtained by linear regression using the observed execution times corresponding to the actual experiments across the two clusters. For modeling a communication phase for a given inter-cluster bandwidth, we used the average of the observed communication times for the phase corresponding to the actual experiments across the two clusters. Thus the *phase models* for a given inter-cluster bandwidth can be used for predicting the simulation rate of CCSM for any number of processors allocated to the components for an inter-cluster bandwidth of 10 Mbps, 100 Mbps or 1 Gbps.

The four standard steps of any genetic algorithm are initialization, selection, mutation and crossover. We use a population size of 200 chromosomes and initialize each chromosome with random valid component sizes. The chromosomes are evaluated based on the fitness functions and are arranged in the descending order of fitness values. We use elitism where the first half of the chromosomes in this order with high fitness values are retained for the next generation. Of these selected chromosomes, we use a normalized fitness function as the probability distribution function to select candidate pairs for single point crossover. Each child chromosome is mutated at a random point of mutation with a probability of 0.2. The algorithm is continued until the fitness value of the fittest chromosome does not change for 20 generations, or until a maximum of 1,000 generations.

We evaluated the genetic algorithm by comparing the schedules generated by the genetic algorithm with the schedules by an exhaustive search approach that evaluates all possible schedules to determine the best schedule. Figure 6 shows the times taken for determining the schedules and the predicted execution times for simulating a climate day in CCSM using the schedules generated by the genetic algorithm and the exhaustive search approach. The graphs show the



(a) Performance of Genetic Algorithm (Speed). Times taken by the algorithms shown in log-scale.



(b) Performance of Genetic Algorithm (Accuracy). Cost of Best Configurations.

Fig. 6 Performance of genetic algorithm

results for 20 different CCSM and batch system configurations arranged in the order of the sum of the queue sizes. As the figures show, with the increase in sizes of the queues, the time taken by the exhaustive search method increases by large amounts while the genetic algorithm almost takes constant time to generate the best schedule. The second graph in the figure shows that the schedules generated by the genetic algorithm are competitive when compared to the optimal schedules by the exhaustive search method. The mean percentage difference in predicted execution times of the schedules generated by the exhaustive search method and the genetic algorithm is only 0.065%, with the standard deviation of 0.1237.

6 Experiments and Results

We have used our framework for execution of a foremost long running multi-component parallel application, CCSM (Community Climate System Model) [6], a global climate system model from National Center for Atmospheric Research (NCAR) [13]. CCSM is a MPMD application consisting of five components, namely, atmosphere, ocean, land and ice and a coupler component which transforms data and coordinates the exchange of information across the other model components. Intra component communications in CCSM involve ten times larger amount of data and are three times more frequent than inter component communications. In our work, we use only one processor for executing coupler since parallelization of coupler does not significantly improve performance. Hence, we consider a maximum of four batch systems or queues for executing the non-coupler components. Since our framework is intended for long running multi-component applications, we focused on three main experiments corresponding to one single-site and two multi-site runs, each with execution duration of about 6–8 days, thus totaling more than 3 weeks of experiments. One multi-site run was performed with the adaptive rescheduling policy discussed in this paper and another run was performed with a greedy "use all" rescheduling policy, described later in this section. The greedy rescheduling run was performed to gauge the efficacy of our rescheduling policy.

For the single-site experiment, the largest queue, *queue-64* was used. For this case, a job corresponding to MCA is submitted requesting for 64 processors. When the job nears the execution time limit, the restart dumps of the execution are created in the application, and the application is exited. The job submitter submits a new 64-processor job to the system. Once the job becomes

active, the job continues its execution from the previous restart dumps, and continues execution. Thus MCA jobs with request sizes of 64 processors are submitted successively one after the other to *queue-64* after the execution time limits corresponding to previous job executions.

The multi-site runs involved execution of CCSM across four batch queues in three clusters, namely, *fire-16*, a AMD Opteron cluster with 8 dual core 2.21 GHz processors, *fire-48*, another AMD Opteron cluster with 12x2 dual core 2.64 GHz processors, and *varun*, an Intel Xeon cluster with 13 8-core 2.66 GHz processors. Four queues were configured on these systems with OpenPBS: one queue, *queue-14*, of size 14 on *fire-16*, one queue, *queue-48*, of size 48 on *fire-48*, two queues, *queue-32* and *queue-64*, of sizes 32 and 64, respectively, on *varun*. The AMD clusters are located at the Supercomputer Education and Research Centre and the Intel Xeon cluster is located at the Centre for Atmospheric and Oceanic Sciences, and are connected through a campus network with a bandwidth of around 500 Kbps. The AMD clusters are connected to each other with Gigabit ethernet switches. The connections within the three clusters are using switched Gigabit Ethernet.

External loads were simulated by submitting synthetic MPI jobs to the queuing systems based on the workload model developed by Lublin and Feitelson [14]. The same synthetic external workload trace was used for the single-site and multi-site runs. The maximum execution time limit for all jobs on all queues was set to 12 hours. The execution time limits on the queues of our department range from 8 h to 256 h in multiples of 2. Our choice of 12 h is close to the least limit. Larger limits will reduce the number of rescheduling events occurring in a simulation period, reduce the overheads, and thus increase the benefits due to multi-site executions with our framework. Thus, by using a small limit of 12 h, the migration capability of the framework can be adequately tested, and benefits due to multi-site executions can be analyzed in a scenario with frequent rescheduling. The CCSM MPMD application was submitted with MPICH2 using the "-configfile" option. The coordinator was started on the front end node on *fire-16*. A job monitor and a job submitter corresponding to each queue were started on the front end of its cluster.

While the 8-day single-site run on *queue-64* performed climate simulations of 6 years, 10 months and 21 days, the 8-day multi-site run with adaptive rescheduling performed climate simulations of 7 years, 1 month and 24 days. This involved 187,848 computational time steps with multiple inter process communications involved in each time step. As the jobs on each of the four queues became active and inactive, the CCSM runs were automatically reconfigured and restarted by our framework. The execution profile of CCSM on the various queues during this multi-site execution is shown in Fig. 7.

The figure shows the location of execution of various CCSM components along the execution timeline as the configurations change. The figure comprises of four subplots corresponding to the four queues in our experiment, as indicated by the labels at their top right corners. The x axis shows the experiment timeline in hours, while the y axis has the total number of processors available in each queue. The colored regions correspond to the execution of CCSM, while the white regions correspond to processor periods that are either unused or used by other jobs in the queue. Each color in the figure corresponds to a single component. For any given x axis value corresponding to a given time instant, the components executing in each queue and the number of processes used by each component are indicated by the component colors and the height of each color, respectively.

For example, during the 12th–18th hour of execution, the ocean component, represented by the light blue bar, is executed on *queue-32* and the atmosphere component, represented by the dark blue bar, executed on *queue-64*. The execution begins on the first queue that begins active, *queue-14*. However, within an hour, when *queue-48* becomes active, all components migrate to the larger *queue-48* and continue execution. At the 7^{th} hour, *queue-32* becomes active and all components migrate from *queue-48* to *queue-32*. Note that the migration happens from a larger (and hence faster) configuration to a smaller configuration. At the 12th hour, when *queue-64* becomes active, the rescheduler decides to use both active queues, with ocean continuing on *queue-32* with a larger

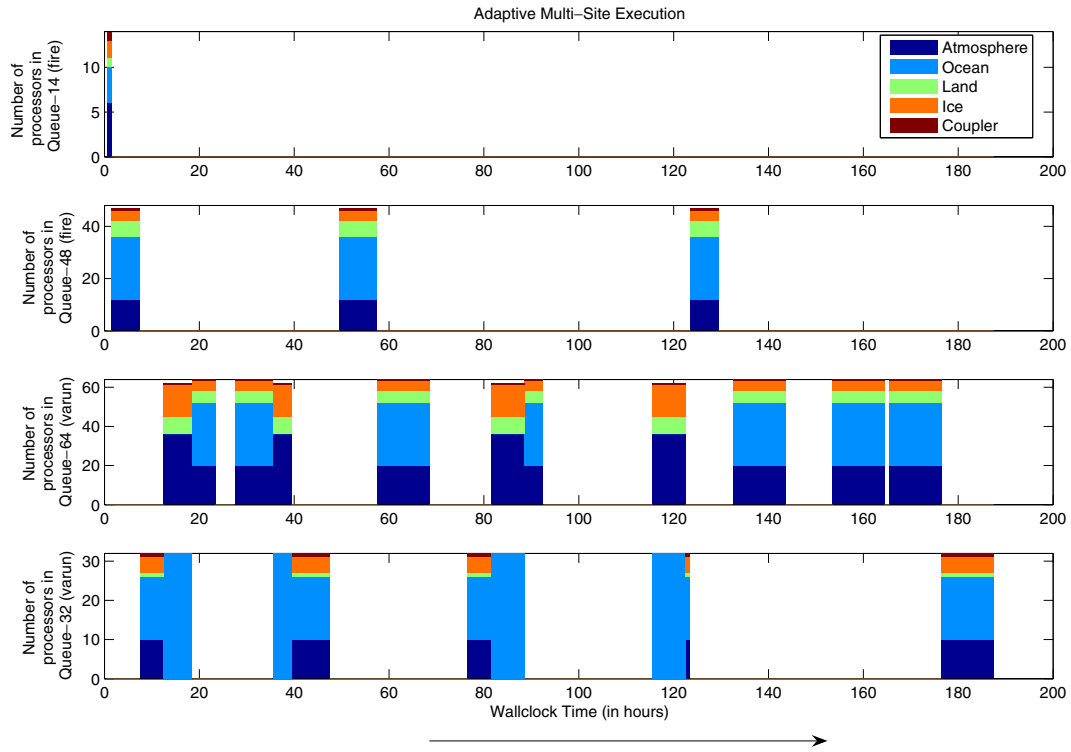


Fig. 7 Execution profile of the application components on multiple sites

number of processors and other components migrating to *queue-64*. Thus, a variety of decisions, migrations and executions were observed during the 8-day run.

Note that the *queue-14* was not involved in the executions because of the very small number of processors it contributes which do not sufficiently offset the inter site communication overheads or the rescheduling overheads. Most of the rescheduling decisions corresponding to events in *queue-14* resulted in no-rescheduling decisions. This is indicated by the blue bars in Fig. 8.

Figure 8 shows the points during the multi-site execution where events resulted in reconfigurations (red bars) and no reconfigurations (blue bars). The experiment involved a total number of 34 rescheduling decisions and 20 reconfigurations of CCSM components performed automatically by our framework, involving non trivial complex coordinations.

Thus, whenever new batch systems become active or active systems reach execution time limit, our Morco framework automatically decides whether to reschedule, (if yes) stops the execution

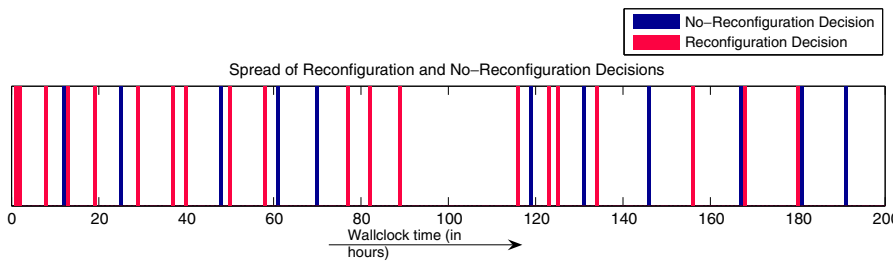


Fig. 8 Rescheduling and no-rescheduling decision points

on the current configuration, calculates a new configuration with different component sizes and different locations, reconfigures the CCSM components to the new configuration and continues the execution.

Figure 9 compares the execution progress of CCSM on three different runs: multi-site run with adaptive rescheduling as discussed above, single-site run and a multi-site run with greedy rescheduling. Each point in the figure corresponds to a restart point in the experiments. The flat regions of the single-site execution curve correspond to the longer queue waiting times of the CCSM batch jobs. Comparing the first two curves, i.e., multi-site run with adaptive rescheduling and the single-site run, we note that the non executing (flat) periods total around 118 hours for the single-site run and only 53 h for the multi-site run with adaptive rescheduling. However, this large benefit in active duration is not translated into large gains in the execution progress of multi-site run mainly because of the limited scalability of CCSM to processor sizes beyond that of the largest queue, which was also the queue used in the single-site run. The gain is further dampened by other factors such as the uneven load distribution across the CCSM components and non linear relationship between progress and the different component sizes. In the figure, we find that the multi-site Grid execution gives comparable and even better overall progress of executions than the single-site executions, in spite of the above factors and various implementation overheads related to multi-site executions including restart overheads, multiple rescheduling, inter site communication, reconfiguration and rebuilding overheads.

While the application related factors discussed above are unavoidable, the rescheduling related overheads can be minimized by better policies such as the one presented in this paper. The third curve shows multi-site execution without using our adaptive rescheduling algorithm. This greedy policy involves rescheduling at "every" event to the best configuration involving "all" the active queues. As can be seen in the figure, the run with adaptive rescheduling significantly outperforms the one with the greedy approach. The average throughput improvement using the adaptive policy over the greedy policy is 18%.

Figure 10 shows the percentage of time spent with different number of active queues during the multi-site runs.

While the multi-site run with the greedy policy had a large percentage of time spent in configurations comprising of processors from all the four queues, the multi-site run with adaptive rescheduling had only spanned a maximum of two queues, namely, *queue-64* and *queue-32*, located

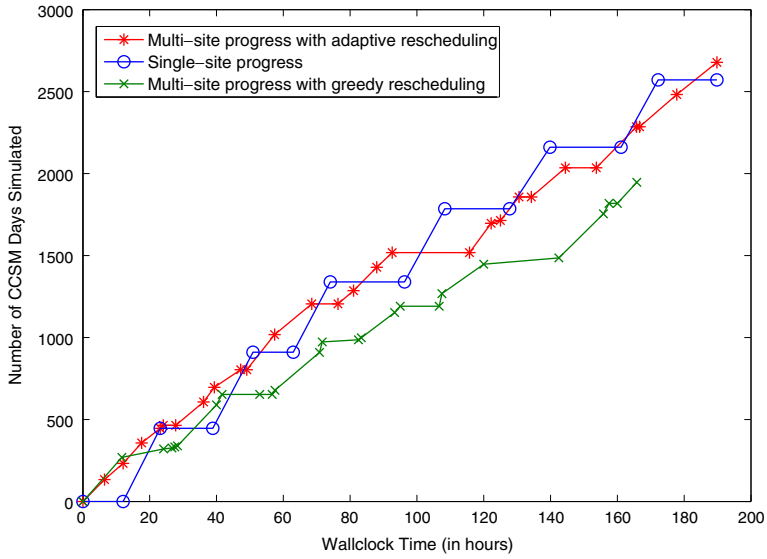


Fig. 9 Comparison of application progress with single-site execution

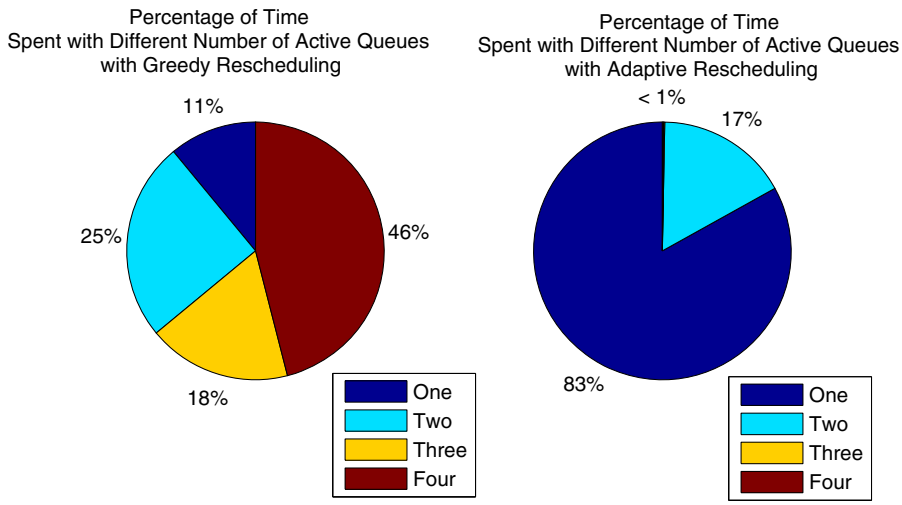


Fig. 10 Percentage of time spent in different number of active queues in the multi-site executions with greedy rescheduling and adaptive rescheduling

in a single department. Application execution on more than two queues will involve the use of slow and shared campus network connecting the machines of two departments. Hence the application execution rate does not increase substantially when involving processors of more than two queues as shown in Fig. 11. Our rescheduling algorithm does not involve execution on more than two queues since the gains in execution rates will be offset by the rescheduling overheads.

Although the adaptive rescheduling run mostly did not involve simultaneous use of multiple queues as described above, it did span across the different queues at different points of time ensuring a smooth and continuous progress. Figure 12 shows the percentage of time spent in various queues during the multi-site execution with adaptive rescheduling. We note that almost 40% of the execution time is spent on queues with fewer processors than that used for the single-site execution. These results indicate that for CCSM, the benefit of multi-site execution is mostly due to the increased duration of availability (as shown by the smaller flat periods for adaptive rescheduling curve in Fig. 9) than due to the use of larger number of processors. Thus our results demonstrate

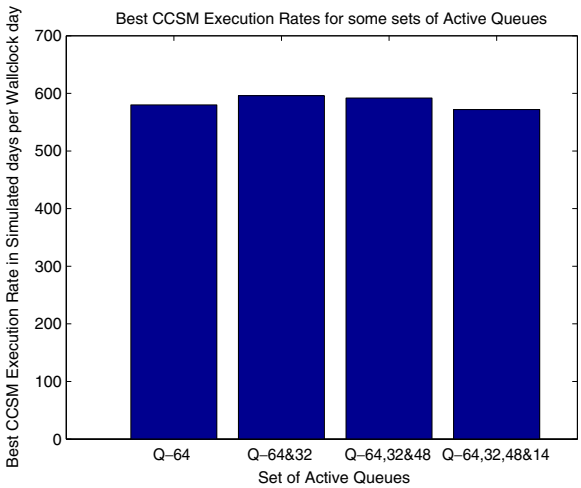


Fig. 11 Best execution rates for CCSM when executed across certain sets of active queues

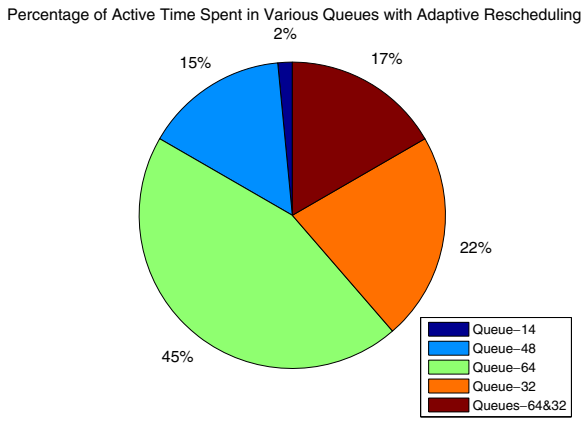


Fig. 12 Percentage of time spent in different queues in the multi-site execution with adaptive rescheduling

the use of Grids for performance improvement of a significant scientific application like CCSM mostly without increasing the number processors used for single cluster executions, while most of the existing work on Grid employs larger number of processors for performance improvement. Hence Grids can be powerful paradigms even for applications with low to moderate scalability.

Multi-site executions will provide greater benefits for CCSM on batch Grids where the average queue wait times are much higher than the job execution time limits, i.e. when the queues are heavily loaded with other external jobs. In such cases, the effect of continuous progress will be more substantial. Applications with significant better scalability than CCSM will obtain larger benefits with multi-site executions.

Within the coordinator, the overhead due to reconfiguration was around 7 to 8 min when a decision to reconfigure was made, of which 5 to 6 min were spent in restart file transfers and backups for fault tolerance. A no-reconfiguration decision was generally arrived at in less than a minute with no loss of execution progress since the current application run is not stopped. Figure 13 gives the times consumed by various phases of the single and multi-site executions including the overheads. Idle time refers to the time when the CCSM jobs were not executing and were waiting in the queues. The remaining times correspond to executions of CCSM jobs when some of the systems become active. Some fraction of this active time is consumed by the multi-site execution overheads, while the remaining time is spent for useful CCSM computations. The most significant overhead is in the startup which includes MPI initializations across CCSM components, restart file reads, and initialization of various CCSM components. There are also noticeable overheads involved in compilation and preprocessing of components, as well as in writing and packing of restart files. A small overhead is also incurred by the transfer of restart files.

As the figure indicates, the multi-site executions have lower percentages of idle time than the single-site execution. The single-site executions as expected have very low overheads. The percentage of time spent in overheads decreases from 6% in multi-site executions with greedy rescheduling to 4% in multi-site executions with adaptive rescheduling. This is due to the fewer number of reconfigurations performed with the adaptive rescheduling. There is also an increase in percentage of idle time and decrease in percentage of useful computation time with multi-site executions with adaptive rescheduling. This is expected because the greedy rescheduling policy makes use of all available resources while adaptive rescheduling policy uses available resources only when it is expected to improve the performance.

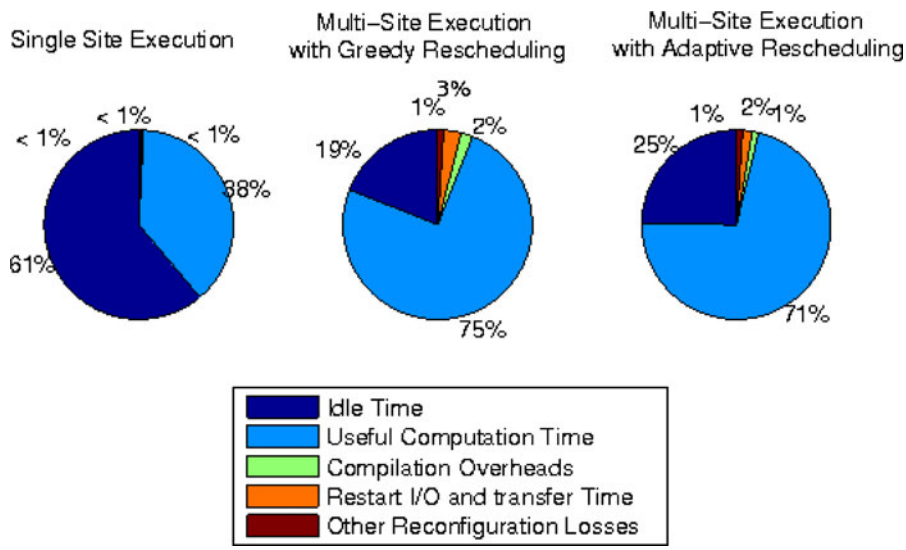


Fig. 13 Percentage of time spent in idling, computing and overheads

Although useful computations are performed for 71% of the total time in the adaptive run and 38% of the total time in the single-site run as shown in Fig. 13, the speedup due to multi-site executions is not as huge when compared to single-site executions as shown in Fig. 9. This is because the percentage of total time when the optimal number of processors (64 processors) is active/available in multi-site runs is only slightly higher than in single-site runs. As shown in Fig. 12, percentage of active time spent in 64-processor queue is 62% (45 + 17). Hence the percentage of total time when 64-processor queue is active is only 44% (62% of 71%) in multi-site runs, and 38% in single-site runs leading to lesser benefits with multi-site runs. About 27% of the total time is used in computations on fewer than 64 processors in multi-site runs.

Thus, multi-site executions of CCSM using our Morco framework ensure continuous progress and regular updates of long running climate simulations, with our adaptive rescheduling process outperforming the greedy rescheduling approach. Multi-site executions also provide the added benefits of using the available systems and not relying on a single system, as is the case with single-site executions.

7 Discussion

While we have demonstrated the Morco framework with CCSM, the framework is generic and can support any long running multi-component MPMD application with very few modifications. The various daemons of our framework, including the coordinator, job monitor and submitter, and the job script respond to generic events including components starts, stops and restarts, and can be used without modifications for other applications. The primary application specific components are the performance model used by the genetic algorithm to evaluate the fitness of the schedules, and the restart facilities in the application to dump the restart files during application reconfigurations by the coordinator. Performance modeling is a common approach for predicting the execution times of parallel applications and many performance modeling strategies exist for characterizing executions of parallel applications on heterogeneous clusters and networks [15, 16]. In the absence of the performance models, sample profiling runs of the MCAs can be used to approximate the execution times on multiple batch systems.

Most of the multi-component applications are long running and have inbuilt checkpoint and restart facilities developed by the model developers in anticipation of system failures and for execution on supercomputing sites with limits on the wallclock time per job submission. Many climate and weather forecasting models [17, 18], and long running applications in CFD and molecular dynamics [19, 20] perform application specific checkpointing and restarting for fault tolerant simulations. These applications can be made to create checkpoints during the reconfigurations by the coordinator. For other applications, the checkpoint dumps can be created dynamically on the occurrence of an event by using a checkpoint library [21, 22].

In our current framework, we use CCSM specific inputs, namely, the names and locations of the components and restart files, the scripts for building CCSM, and macros for executions, to our framework components. However, we can trivially generalize these using the standard naming schemes as in WSRF [23] and XML.

Our framework requires execution of daemons including coordinator, job submitter and job monitor on the front end nodes of the batch system. The coordinator daemon should be executed on one of the front end nodes or an external node that is accessible by all the front end nodes. Executing daemons on front end nodes to coordinate executions across batch system is a common approach in many Grid middleware frameworks [24, 25]. Typically, the front end nodes of batch systems have public IPs and are accessible by front end nodes of other batch systems.

The adaptive rescheduling policy as currently implemented leads to resource wastage by idling allocated resources. This can be addressed by dynamically releasing idling processors back to the resource pool if they are unlikely to be used in a preset future time window or the execution time limit of their queue. Also, queues like *queue-14* which are almost never used can be identified

based on event history and submission to such queues can be stopped without any performance deterioration.

8 Related Work

Most of the existing efforts deal with rescheduling and migration of single component applications on interactive systems. Sarkar et al. [26] proposed an integrated multi-agent system (MAS) with autonomous agents and an adaptive execution scheme for achieving guaranteed performance on the basis of the SLAs. They consider adaptive execution of a batch of independent jobs on a Grid with interactive systems. The work proposes algorithms for migration of executing tasks based on dynamic resource characteristics and application performance properties. Luccheze et al. [27] have developed generational scheduling with task replication (GSTR) algorithm for adaptive load balancing and rescheduling executing tasks by replication. The algorithm considers both the tasks ready for execution and the tasks currently executing. They applied the algorithm for parallel application with task graphs.

There has been increasing interest in coallocating parallel applications across multiple clusters [28–31]. Casanova analyzed the impact of redundant submissions on the other jobs in the system [30]. In this work, a job submitted to a system is redundantly submitted to other batch systems. When the job starts execution in one of the systems, the redundant jobs submitted to the other systems are canceled. The author concluded that while redundant tasks decrease average turnaround times of jobs and helps load balancing across clusters, they can cause heavy load in the systems and unfairness to the users who do not use such redundant jobs. In our work, we do not replicate jobs on multiple batch queues. We decompose a single job into multiple sub-jobs and submit these sub-jobs to many batch queues. There have been few efforts related to adaptivity for workflow applications. Yu and Buyya [32] present taxonomy of workflow scheduling based on various categories including workflow design, structure, composition system, QoS constraints, information retrieval, workflow scheduling, planning scheme, performance estimation, fault tolerance and data movement. The work by Nurmi et al. [33] deals with execution of workflow applications on different batch systems of a Grid. In their work, they schedule different tasks of a workflow application to different batch systems of a Grid based on predictions of execution times of the tasks on the systems and the queue waiting times in the systems [12]. In our work, we consider multi-component applications that contain periodic communications between different batch systems unlike the workflow applications.

Bucur and Epema have extensively studied the benefits of coallocation of processors from different clusters in a Grid for job executions [28]. In their work, they analyze the impact of using different scheduling policies, component sizes and number of components on coallocation. Our work is complementary to their efforts since we analyze the benefits of coscheduling multiple components of a specific application on multiple clusters of a Grid. However, the efforts by Bucur and Epema consider execution of short jobs where the different components of a job are submitted to the different batch queues only once and the components complete executions within the execution time limits associated with the batch queues. In some of their scheduling policies, they assume the existence of a global queue for submission of multi-component jobs and also assume the use of the same job execution policy (FCFS) on all the local queues of the clusters. They also assume that all clusters become simultaneously available for execution of components.

Buisson et al. [29] in their work on scheduling malleable applications in multi-cluster systems, have developed a middleware framework called DYNACO for their application runner, MRunner, to execute malleable applications. Our framework, though similar, is designed for multi-submission executions on generic batch scheduling systems. Markatchev et al. [25] have developed a middleware framework for checkpointing, migration and reconfiguration for execution of traditional long running applications. While they consider batch systems and execution time limits of the systems, and perform migration of batch jobs before reaching the time limits, unlike us, they do not perform simultaneous

execution of an application across jobs on multiple queues.

Ko et al. [31] have presented a solution for coupled multiphysics applications across multiple queues. Their solution also includes initial co-scheduling, dynamic resource allocation, load balancing and handling different queue wait times. They investigate their solution with a coupled Computational Fluid Dynamics (CFD) and Molecular Dynamics (MD) code. Their strategy uses the PilotJob/BigJob framework based on the Simple API for Grid Applications (SAGA). BigJob is a container job consisting of a number of subtasks. They explore the problem of load imbalance across the two components in the times to reach their synchronization steps. They perform load balancing for efficient utilization of resources by dynamic readjustment of allocated resources to each component. They iteratively apply their load balancing solution until the resource allocation for the components reach the steady state solution. They compare different scenarios including allocation of a single big job for the entire coupled code, allocating two big jobs for each component on a single machine and two machines. Their work however does not handle continuous execution of long running applications. Our framework also solves problems due to different execution time limits and queues becoming inactive during execution, and includes robust rescheduling policies.

The work by Kim et al. [34] has built a programming and runtime framework for management of application workflows. The framework includes dynamic resource provisioning and management including adapting to application and resource dynamics satisfying deadline and budget constraints. The primary objectives they consider are application acceleration or time to completion, resource conservation or using minimal amount of resources, and resilience to failures. Their work is based on performance model of application for initial allocation and dynamically updating the performance model based on dynamics. Their framework uses CometCloud computing engine for cloud computing. They demonstrate their work on a workflow for modeling oil reservoir on hybrid infrastructure consisting of TeraGrid resources and Amazon EC2 resources. Our rescheduling strategies are confined to traditional HPC environments due to batch queue dynamics.

9 Conclusions and Future Work

In this work, we have developed Morco, middleware for execution of multi-component applications on independently administered batch Grids consisting of multiple batch systems. The framework, which is generic and non intrusive, requires no special administrative privileges or coallocated global scheduling. The framework, with robust adaptive rescheduling decisions, dynamic resource allocation and fault tolerance, supports continuing execution across multiple time-distributed submissions on each queue. With an experiment involving 8 day execution of a complex multi-component application, CCSM, on a batch Grid with four batch queues on three systems, we have shown that our framework enables multi-site executions yielding better application throughput.

While our framework currently does not use Grid specific details, our execution model and middleware framework is practical for deployment, use and obtaining benefits for long running multi-component applications like CCSM on multiple batch systems. The framework is generic, non intrusive and does not require special administrative privileges, coallocated global scheduling, or batch queues with fixed queuing policies. In future, we plan to deploy our framework on practical Grid infrastructures, particularly, integrating our components with Grid information service like WebMDS for monitoring and discovery [35] and for obtaining dynamic resource properties. The information service will have to be augmented with aggregate statistics on batch queues including queue waiting times, performance characteristics of the application, and periodic updates on the availability and failures of the batch systems. The coordinator in our framework will use these information from the Grid information service to make scheduling and rescheduling decisions.

While our current work involves over provisioning of resources, as future work we plan to

incorporate some additional features in Morco such as variable request sizes and dynamic release of resources to improve the system resource utilization. We also plan to use our framework for performing very long duration runs for multi-century climate simulations with CCSM and investigate long term climate phenomena.

References

1. Coveney, P., Fabritiis, G.D., Harvey, M., Pickles, S., Porter, A.: On steering coupled models. In: e-Science All Hands Meeting (2005)
2. Larson, J., Jacob, R., Ong, E.: The model coupling toolkit: a new Fortran90 toolkit for building multiphysics parallel coupled models. Int. J. High Perform. Comput. Appl. **19**, 277–292 (2005)
3. Delgado-Buscalioni, R., Coveney, P., Riley, G., Ford, R.: Hybrid molecular-continuum fluid models: implementation within a general coupling framework. Philos. Trans. R. Soc. Lond. A **363**, 1833 (2005)
4. TeraGrid: http://www.teragrid.org. Accessed Sept 2011
5. UK e-Science: http://www.rcuk.ac.uk/escience/default.htm. Accessed Sept 2011
6. Community Climate System Model (CCSM): http://www.ccsm.ucar.edu. Accessed Sept 2011
7. Collins, W., Bitz, C., Blackmon, L., Bonan, G., Bretherton, C., Carton, J., Chang, P., Doney, S., Hack, J., Henderson, T., Kiehl, J., Large, W., McKenna, D., Santer, B., Smith, R.: The community climate system model version 3: CCSM3. J. Climate **19**(11), 2122–2143 (2006)
8. Ccsm user guide: http://www.cesm.ucar.edu/models/ccsm3.0/ccsm/doc/UsersGuide/UsersGuide.pdf. Accessed Sept 2011
9. Gabriel, E., Resch, M., Beisel, T., Keller, R.: Distributed computing in a heterogenous computing environment. In: EuroPVMMPI'98 (1998)
10. Park, K., Park, S., Kwon, O., Park, H.: MPICH-GP: a private-IP-enabled MPI over Grid environments. In: Proc. of Second International Symposium on Parallel and Distributed Processing and Applications, ISPA04, Hong Kong, China, pp. 469–473 (2004)
11. Smith, W., Taylor, V., Foster, I.: Using run-time predictions to estimate queue wait times and improve scheduler performance. In: Job Scheduling Strategies for Parallel Processing (JSSPP), pp. 202–219 (1999)
12. Brevik, J., Nurmi, D., Wolski, R.: Predicting bounds on queuing delay for batch-scheduled parallel machines. In: PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 110–118 (2006)
13. The National Center for Atmospheric Research (NCAR): http://www.ncar.ucar.edu. Accessed Sept 2011
14. Lublin, U., Feitelson, D.: The workload on parallel supercomputers: modeling the characteristics of rigid jobs. J. Parallel Distrib. Comput. **63**(11), 1105–1122 (2003)
15. Lee, B., Brooks, D., de Supinski, B., Schulz, M., Singh, K., McKee, S.: Methods of inference and learning for performance modeling of parallel applications. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Jose, CA (2007)
16. Yang, L., Ma, X., Mueller, F.: Cross-platform performance prediction of parallel applications using partial execution. In: SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 40 (2005)
17. Parallel Climate Model (PCM): http://www.cgd.ucar.edu/pcm. Accessed Sept 2011
18. Skamarock, W., Klemp, J., Dudhia, J., Gill, D., Barker, D., Wang, W., Powers, J.: A description of the advanced research WRF version 2. NCAR, Tech. Rep. Technical Note (2005)
19. Lefantzi, S., Ray, J.: A component-based scientific toolkit for reacting flows. In: Proc. Second MIT Conference on Computational Fluid and Solid Mechanics, pp. 1401–1405 (2003)
20. ANSYS FLUENT: http://www.ansys.com/products/fluid-dynamics/fluent/default.asp. Accessed Sept 2011
21. Vadhiyar, S., Dongarra, J.: SRS—a framework for developing malleable and migratableparallel applications for distributed systems. Parallel Process. Lett. **13**(2), 291–312 (2003)
22. Fernandes, R., Pingali, K., Stodghill, P.: Mobile MPI programs in computational Grids. In: PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 22–31 (2006)
23. WS Resource Framework: http://www.globus.org/wsrf. Accessed Sept 2011
24. Czajkowski, K., Foster, I., Kesselman, C.: Agreement-based resource management. Proc. IEEE **93**(3), 631–643 (2005)
25. Markatchev, N., Kiddle, C., Simmonds, R.: A framework for executing long running jobs in Grid environments. In: HPCS '08: Proceedings of the 22nd International Symposium on High Performance Computing Systems and Applications, pp. 69–75 (2008)
26. Sarkar, A.D., Roy, S., Ghosh, D., Mukhopadhyay, R., Mukherjee, N.: An adaptive execution scheme for achieving guaranteed performance in computational Grids. J. Grid Computing **8**(1), 109–131 (2010)
27. de O. Lucchese, F., Yero, E., Sambatti, F., Henriques, M.: An adaptive scheduler for Grids. J. Grid Computing **4**(1), 1–17 (2006)
28. Bucur, A., Epema, D.: Scheduling policies for processor coallocation in multicluster systems. IEEE Trans. Parallel Distrib. Syst. **18**(7), 958–972 (2007)
29. Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., Epema, D.: Scheduling malleable applications in multicluster systems. In: CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing, pp. 372–381 (2007)

30. Casanova, H.: Benefits and drawbacks of redundant batch requests. J. Grid Computing **5**(2), 235–250 (2007)
31. Ko, S.-H., Kim, N., Kim, J., Thota, A., Jha, S.: Efficient runtime environment for coupled multi-physics simulations: dynamic resource allocation and load-balancing. In: CCGRID 2010: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 349–358 (2010)
32. Yu, J., Buyya, R.: A taxonomy of workflow management systems for Grid computing. J. Grid Computing **3**(3–4), 171–200 (2005)
33. Nurmi, D., Mandal, A., Brevik, J., Koelbel, C., Wolski, R., Kennedy, K.: Evaluation of a workflow scheduler using integrated performance modelling and batch queue wait time prediction. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 119 (2006)
34. Kim, H., el-Khamra, Y., Rodero, I., Jha, S., Parashar, M.: Autonomic management of application workflows on hybrid computing infrastructure. Sci. Program. **19**(2–3), 75–89 (2011)
35. Zhang, X., Freschl, J., Schopf, J.: A performance study of monitoring and information services for distributed systems. In: HPDC '03: Proceedings of the 12th IEEE International Symposiumon High Performance Distributed Computing, p. 270 (2003)