2014 43rd International Conference on Parallel Processing

A Batch System with Fair Scheduling for Evolving Applications

Suraj Prabhakaran [1,a,b], Mohsin Iqbal [2,b], Sebastian Rinke [3,a,b], Christian Windisch [4,a,b], Felix Wolf [5,a,b]

[a]German Research School for Simulation Sciences, Aachen, Germany
[b]RWTH Aachen University, Aachen, Germany
{[1]s.prabhakaran, [3]s.rinke, [4]c.windisch, [5]f.wolf}@grs-sim.de
[2]mohsin.iqbal@rwth-aachen.de

***Abstract*—Cluster batch systems usually support only static allocation of resources to applications before job start. After job start, applications cannot increase or decrease their resource set. However, some applications unpredictably evolve during execution and thus may require additional resources. If the extra resources cannot be delivered during runtime, those applications may have to run longer to finish, or are not even able to finish when their job's time slice expires. Likewise, a job may have to end without additional resources due to hardware limits being reached, such as the memory available to the compute node. To avoid such scenarios, users have to make large static allocations to account for a potential demand for resources. This leads to wastage of resources as they idle before they might actually be used at an unknown point. In this paper, we propose a batch system with dynamic allocation facilities to support on-the-fly resource allocation to unpredictably evolving jobs based on demand. We present a novel dynamic resource allocation strategy that also accounts for a fair assignment of resources between the usual rigid jobs and the evolving jobs. The results for a CFD production application and a mixed workload of rigid and evolving jobs (based on the widely used ESP benchmark) show that our system not only reduces the job waiting and job turnaround times, but also increases system utilization and system throughput.**

***Keywords*—dynamic resource management; dynamic scheduling; batch systems**

I. Introduction

The batch system is an essential middleware for managing supercomputing resources. A cluster batch system consists of a resource manager (RMS) and a scheduler which work together to efficiently schedule resources to jobs. Resources are mapped to jobs based on their requirements, which include the number of processors, memory, accelerators and software requirements. The allocation mechanism depends on the type of job under consideration. As defined by Feitelson and Rudolph in [1], jobs can be classified in four categories based on their flexibility. The most common type are *rigid* jobs which require a fixed number of processors and the batch system must allocate all the resources requested. The second class of jobs are *moldable* jobs, in which a requirement can be *molded* or *modified* by the batch system before starting the job. In such a case, the application must be prepared to run on less or more processors than required. For rigid as well as moldable jobs, the allocation is made before job start and hence is *static* in nature. The third class of jobs, known as *malleable* jobs, allow the batch system to shrink or grow the job's resource allocation during the application execution. Since the grow or shrink operation is initiated by the batch system, this is an effective strategy to improve cluster utilization and job waiting times. However, shrinking the resource set may lead to larger turnaround times for these jobs. The final class of jobs, known as *evolving* jobs, may also grow or shrink its reservation during runtime. However, in contrast to malleable jobs, the grow or shrink operation is initiated by the application itself. The application may request more resources or release a part of its existing allocation during execution. The scheduler must handle such circumstances appropriately. Due to the expansion and contraction of existing allocations, they are called as *dynamic* allocations.

Traditionally, most batch systems support only static allocations, the reason primarily attributed to the rigid nature of most parallel applications. However, due to increasing computational complexity, adaptive or evolving behavior is exhibited by many applications. Applications using mesh adaptation techniques such as multiscale analysis [2] or the adaptive mesh refinement (AMR) [3] often exhibit evolving behavior. For example, the CFD flow solver Quadflow [4], which uses the multiscale analysis technique, may unpredictably evolve for different problems due to an increase of the grid size during a grid adaptation phase. Such evolving applications require additional resources to distribute work evenly across processors. In some cases, the growth in data size may exceed the memory limit of the node and additional nodes are required to redistribute data and continue execution.

Dynamic allocation is also essential for applications that have different resource requirements for different computational phases. Some applications also unexpectedly entail simultaneous execution of an analyzer of intermediate results or other additional smaller simulations alongside the main simulation. They may also require additional resources in order to leave the main simulation unaffected. Weather simulations that require simultaneous execution of nested simulations to track multiple weather phenomena are an example [5]. In task-parallel applications, new tasks emerging as a result of intermediate computations can be offloaded to new resources without having to steal resources from the main program. This is more relevant in heterogeneous architectures consisting of accelerators such as the DEEP cluster [6]. In this ongoing project, the architecture consists of a cluster part and a booster part, with booster nodes designed to run computationally intensive parallel kernels. They can be statically or dynamically allocated to applications running on cluster nodes, depending on application demands.

Dynamic allocations are not only beneficial for the appli-

0190-3918/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.2014.44

cations but also for overall system utilization. By acquiring additional resources only at the required computational phase, they need not be preallocated to the job and hence may be used by other jobs. When an executing application no longer requires some resources, they can be released before the application terminates, thus making them available for other jobs in the queue. Such a flexible allocation mechanism can improve the system throughput, the waiting time and the turnaround time for jobs. Dynamic allocations also help during node failures by allocating spare nodes to affected jobs, thus improving fault tolerance. Due to its essential advantages, dynamic provisioning of traditional resources is considered an important aspect towards reaching exascale [7]. However, the lack of dynamic allocation facilities today means the system and applications do not have the stated benefits.

Enabling these facilities poses multiple challenges. Since malleability decisions are made single handedly by the scheduler, remapping of resources to jobs can be performed at any time at the scheduler's discretion. But in contrast to malleable jobs, evolving jobs introduce the challenge of scheduling unexpected dynamic requests along with static job submission requests, and raise questions of how they must be treated by the scheduler. From the user perspective, satisfying all dynamic requests favors the evolving jobs but may lead to an unfair resource starvation scenario for users submitting rigid jobs. From the system perspective, dynamic allocations may be counter productive to throughput and system utilization without an effective dynamic scheduling strategy. Therefore, handling such requests while maintaining fairness and system efficiency is the most challenging requirement of supporting unpredictably evolving jobs in a cluster environment.

This paper advances the state of the art in scheduling and resource management by enabling dynamic allocations for unpredictably evolving jobs. We propose a dynamic batch system for unpredictably evolving applications that integrates fairness considerations in the scheduling process. Our main contributions include:

- An extended Torque/Maui batch system that allows dynamic allocations
- A dynamic fairness strategy implemented in the Maui scheduler to efficiently service dynamic and static allocations
- Showing the benefits of enabling dynamic allocations, thereby strongly motivating such architectures for future systems

We use the Quadflow application to demonstrate the benefits of dynamic allocations for evolving applications and evaluate our dynamic scheduling strategy with a dynamic ESP benchmark (modified to include evolving jobs) to highlight the various user-centric and system-centric advantages. Results show better system utilization, throughput and reduced waiting times.

The remainder of this paper is organized as follows. Section II elaborates on evolving jobs and dynamic allocation mechanisms and sketches the approach taken in this paper. Section III presents our dynamic scheduling strategy and its implementation in the Torque/Maui batch system. In Section IV, we present the evaluation of our work. Section V compares our approach with other related work. Finally, we present our conclusions in Section VI and discuss future work.

II. Scheduling Evolving Jobs

In this section, we discuss the problem of scheduling evolving jobs in detail. We briefly describe the evolving behavior of Quadflow and review the different strategies for scheduling evolving jobs by identifying their challenges, feasibility and consequences. By that, we define the focus of our work and outline our approach.

A. Job Evolution in Quadflow

As described in Section I, a job can evolve for various reasons. A common reason for evolution is the increase of data size and, as a consequence, computational demand during execution. This characteristic is observed in Quadflow [4], which solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method on locally refined grids. The computational grids are represented by block-structured parametric B-Spline patches to deal with complex geometries. The simulation performs a grid adaptation in each iteration which may or may not affect the size of the grid. For certain real-world problems, the size of the grid may rapidly grow during an adaptation phase which enlarges the computations. The evolution of data and computation cannot be predicted due to the multitude of factors that govern the computation for different problems. Even if the evolution can be predicted for a particular problem, the number of cells that will be produced and therefore the number of additional processors that will be required cannot be foretold. Such a pattern can also be observed in several AMR applications [8], [9]. If such applications were able to request additional resources according to their evolution, longer execution times and the risk of job abortion could be avoided.

B. Dynamic Resource Allocation

Scheduling unexpected dynamic resource-allocation requests from running jobs introduces many challenges. A dynamic request could be served in several ways:

- Allocating the idle resources
- Allocating resources from a separate partition maintained specifically to serve dynamic requests
- Stealing resources from malleable jobs
- Stealing resources from preemptive jobs

However, even by exercising all four options, a dynamic resource allocation request cannot always be satisfied. Since the primary goal of batch job schedulers is to increase throughput and resource utilization, they aim to accommodate as many jobs as possible and maintain the highest possible utilization of the system. Therefore, there may not be enough idle resources in the cluster to serve the evolving job. The separate partition could also be in use by other evolving jobs. Furthermore, resource stealing is not feasible when there are no malleable or preemptive jobs at the time of the dynamic request.

On the other hand, allocating the idle resources to unexpected requests raises another issue. In practice, production clusters run with a well mixed workload of small and large jobs which often leaves an incomplete cluster utilization. This gives an opportunity to allocate idle resources to many dynamic requests. However, it may cause unfair delays to high priority

reservations of queued jobs causing an extended waiting time. As illustrated in Figure 1, consider a cluster system with six nodes in which job A is executing on nodes 0 and 1 for a time slice of 8 hours. Job B acquires nodes 2 and 3, and is scheduled for to run for 4 hours. Queued job C requires 4 nodes and the earliest time it can start is after 4 hours when job B has terminated. Then it could run on nodes 2 to 5. However, if A dynamically acquires the idle nodes 4 and 5 before B terminates, job C will be delayed by an additional 4 hours. Hence, allocating idle resources to dynamic requests can improve system utilization but possibly at the expense of fair resource access for evolving and rigid jobs. At the same time, improving availability through the stated methods still cannot guarantee resources to uninformed dynamic requests without prior knowledge of their evolution.

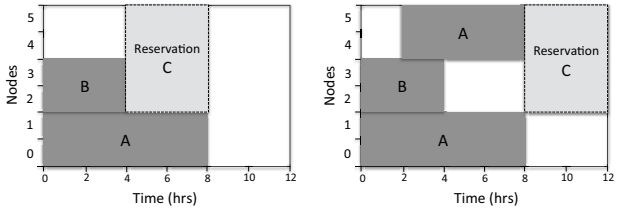
However, with an indication of the minimum and maximum amount of resources that may be required by an evolving job, the dynamic resources can be preallocated to this job. This approach grants all dynamic requests made by the job. To assure their availability, these resources cannot be used by rigid jobs. Only malleable or preemptive jobs could be assigned to these resources so that they can be withdrawn when required by the evolving job. Unfortunately, predicting the appropriate size of preallocation is error prone and may result in too few or no resources being used by the evolving job, but still charged to the user's account. Evolving jobs can block a considerable amount of extra resources and for a workload dominated by rigid jobs (which is still typical today), this may cause starvation for rigid jobs and wastage of resources.

Thus, designing a dynamic scheduler with the primary goal of guaranteeing resources to all dynamic requests by evolving jobs cannot provide good system utilization and may result in users having to pay for unused resources as well (*guaranteeing approach*). On the other hand, designing a dynamic scheduler with the primary goal of improving system performance cannot guarantee resources to all dynamic requests and a dynamic allocation may result in unfair resource usage scenarios, as illustrated in Figure 1 (*non-guaranteeing approach*). Performance of a cluster system not only depends on the scheduler but also on the workload. However, the guaranteeing approach depends to a great extent on the workload to achieve good system utilization. The disadvantages resulting as the consequence of absence of malleable jobs are far greater compared to the non-guaranteeing approach.

Considering the hard-to-predict nature of evolving applications like Quadflow and the dominance of rigid jobs in today's supercomputers, our design is based on the non-guaranteeing approach where evolving jobs can be allocated with available idle resources. Resource availability for dynamic requests can be enhanced by having a separate partition or by preempting low priority jobs. Our dynamic assignment strategy (discussed in the next section) ensures a fair administrator-configurable allocation of resources between dynamic and static requests. Thus, our strategy can be used to accomplish a desired balance in supporting evolving jobs, improving system performance and fair access to resources.

III. The Dynamic Batch System

In this section, we describe the implementation of our dynamic allocation facilities in the Torque/Maui batch system. We start with an overview of the Torque/Maui batch system and discuss the extensions to (i) Torque for dynamic allocation and (ii) Maui for dynamic scheduling including the fairness mechanism.



(a) State before job A's evolution (b) State after job A's evolution

Fig. 1. Effect of dynamic allocation of job A on static reservation of job C.

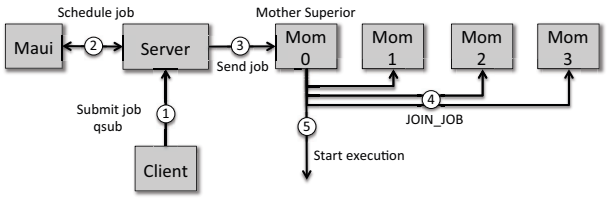


Fig. 2. Workflow of the Torque/Maui batch system. Circled numbers indicate the sequence of steps.

A. Overview of Torque/Maui Batch System

The Torque/Maui batch system is one of the most commonly used middleware for batch job control. The Torque resource manager [10] is based on the PBS project [11] extended to improve scalability and fault tolerance and is currently maintained by Adaptive Computing. Torque is usually integrated with sophisticated schedulers such as Maui [12], which provides advanced scheduling features such as job prioritization, fairshare and backfill scheduling. The widespread use of the Torque/Maui batch system was also one of the principal reasons for choosing this ensemble to implement our dynamic scheduling facilities.

A Torque/Maui cluster consists of a headnode, a frontend, and many compute nodes. The headnode runs the `pbs_server` daemon (*server*) and the Maui scheduler daemon. The compute nodes run the `pbs_mom` daemon (*mom*). Users are provided with a number of client commands to communicate with the server for tasks such as job submission, alteration and checking the status of a job. They are installed on the frontend. Figure 2 illustrates the typical workflow of the Torque/Maui batch system. The client submits a job through the `qsub` command by specifying the number of nodes, the number of processors per node, the duration for which resources are required (*walltime* of the job), and other software or hardware requirements. The job is then queued at the server. When resources are allocated for this job by the Maui scheduler, the server sends the job to one of the nodes allocated for this job (called *mother superior*) and updates the state of this job in the queue as *running*. The mother-superior node and the other allocated nodes perform a *join* operation, after which the user application starts execution. The TM interface in Torque allows applications to interact with its local moms. For example, it is used by MPI to spawn processes on other hosts.

Algorithm 1 Maui Iteration

1: **while** TRUE **do**
2: Obtain resource information from Torque
3: Obtain workload information from Torque
4: Update statistics
5: Refresh reservations
6: Select jobs eligible for priority scheduling
7: Prioritize eligible jobs
8: Schedule the jobs in priority order and create reservations
9: Backfill jobs
10: **end while**

The Maui scheduler communicates with the server and schedules jobs iteratively. A scheduling iteration is followed by a period of sleeping or processing external commands. Maui will instantly start a new iteration when (i) a job or resource state change occurs, (ii) a reservation boundary event occurs, (iii) an external command to resume scheduling is issued or (iv) a configurable timer expires. The steps of a scheduling iteration are detailed in Algorithm 1.

During each iteration, Maui obtains the most recent information about resources and jobs from Torque and updates the historical statistics and usage information of all the jobs. Then, jobs meeting a minimum scheduling criterion, based on throttling policies and job states, are selected and considered for scheduling. The selected jobs are prioritized according to various policies and scheduled in the order of their priorities. When a lack of resources prevents the idle job with the highest priority from starting, the earliest time when the resources are available for this job is determined and a reservation is created. Maui continues to create reservations for N such highest priority jobs where N can be configured using the `ReservationDepth` parameter. Jobs that are not reserved are then backfilled out of order. Backfilling is a strategy of increasing resource utilization by running low priority jobs out of order as long as they do not disturb the high priority reservations. A higher `ReservationDepth` leads to a more conservative backfilling while a lower `ReservationDepth` allows more jobs to be backfilled.

Maui considers various aspects for job prioritization such as the priority weight of a user, resource requirements, and waiting time of a job. Furthermore, fairshare policies are used to ensure fair resource access among different users. A detailed description of these features is available in [12].

B. A Resource Management System for Evolving Jobs

Given the structure of the Torque/Maui batch system, the following requirements for supporting dynamic (de)allocation for evolving jobs are imperative.

- An interface through which applications can request/release resources at runtime
- Functionality to queue dynamic requests for scheduling at the server
- Functionality to associate/disassociate nodes and jobs dynamically during job runtime

We extended Torque by adding the above features and the workflow of a dynamic allocation is illustrated in Figure 3. Applications can use the `tm_dynget()` function of the extended TM interface by specifying the number of nodes and processors per node. The mother superior forwards the request to the server which changes the job to a special *dynqueued* state. This triggers a new scheduling cycle and additional resources are allocated for this request. The server forwards the new hostlist to the mother superior and changes the job state back to *running*. The hosts from the existing allocation and the dynamically allocated hosts perform a *dyn_join* operation which expands the resource allocation for the job. The mother superior then responds to `tm_dynget()` with the dynamically allocated hostlist. MPI applications can use the MPI-2 dynamic process management facilities to spawn new processes on the additionally allocated nodes. MPI implementations offer a "host" or "add-host" parameter to the MPI_Info argument to specify a newly allocated hostlist. Similarly, the function `tm_dynfree()` can be called to release nodes by passing the list of nodes to be released as a parameter (illustrated in Figure 4).

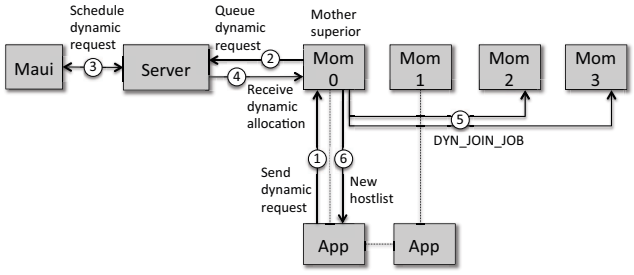


Fig. 3. Dynamic allocation of nodes 2 and 3. Circled numbers indicate the sequence of steps.

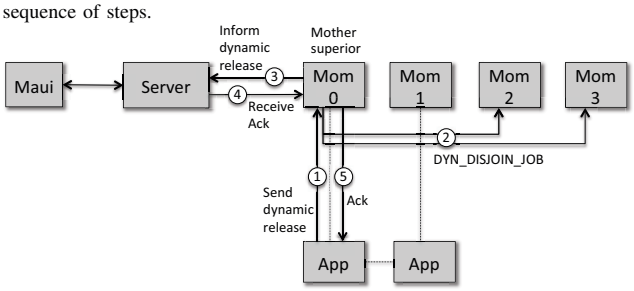


Fig. 4. Dynamic deallocation of unused nodes 2 and 3. Circled numbers indicate the sequence of steps.

The call `tm_dynfree()` usually returns true, as a release operation is rarely unsuccessful. During dynamic deallocation, the moms perform a *dyn_disjoin* operation with the nodes to be released and the server is informed of the deallocation. Finally, the server updates the freed node's states internally, after which they can be allocated to other jobs.

Basically, any process from any host of the parallel job can call `tm_dynget()` to request new resources through its local mom. However, to ensure that only one dynamic request from the same job is pending at the server at a time, the dynamic requests are always forwarded to the server through the mother superior. This simple API consisting of two functions is sufficient for dynamic resource (de)allocation.

C. Dynamic Scheduling with Maui

By design, the Maui scheduler supports scheduling of rigid jobs only. In our work, the Maui scheduler was extended to schedule dynamic requests by:

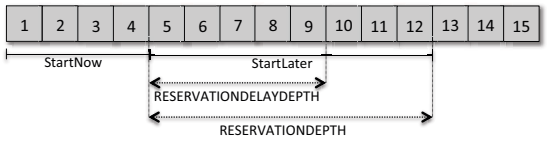


Fig. 5. The number of *StartNow* and *StartLater* jobs in a queue. `ReservationDepth` is longer than `ReservationDelayDepth`.

- Enriching Maui's iteration with a scheduling algorithm that also supports dynamic requests
- Enhancing the resource allocation mechanism to allocate resources for dynamic requests
- Implementing a dynamic fairness scheme to ensure fairness between dynamic and static requests

The extended Maui iteration is detailed in Algorithm 2. The algorithm prioritizes a list of eligible static jobs and dynamic requests separately. While the static jobs are prioritized according to normal priority factors, the dynamic requests are prioritized in FIFO order. The static jobs are then scheduled and the necessary reservations are created but the jobs are not started immediately. The reserved static jobs can be classified in two categories: (i) *StartNow*: jobs that can be started immediately, and (ii) *StartLater*: jobs that can be started only at a later point of time. In the next step, for each dynamic request in the queue, the scheduler tries to allocate the idle resources and measures the delays that may be caused to the *StartNow* and *StartLater* jobs.

In each iteration, the number of *StartNow* jobs varies and is determined based on the number of considered jobs and available resources. In the original Maui iteration, the number of *StartLater* jobs is determined based on the `ReservationDepth` parameter, mainly to control backfilling. In the extended algorithm, the number of *StartLater* jobs is determined by the maximum of `ReservationDepth` and `ReservationDelayDepth`, where `ReservationDelayDepth` is a configurable parameter to control the number of *StartLater* jobs for which delays need to be measured (Figure 5). Therefore, for delay computations, the reservations of `ReservationDelayDepth` number of jobs will be considered and for backfilling, reservations of `ReservationDepth` number of jobs will be considered. This allows delays to be computed for a controlled number of jobs irrespective of whether a conservative backfilling with large `ReservationDepth` is deployed or optimistic backfilling with lower `ReservationDepth` is deployed. Similar to `ReservationDepth`, a proper choice of `ReservationDelayDepth` for a site depends on its workload characteristics.

Once the delays to the static jobs are measured, the dynamic fairness policies are invoked to determine whether the allocation is fair and can be allowed. The dynamic fairness policies are site-configurable parameters and are described in Section III-D. If the reservation is allowed, the dynamic fairness statistics are updated and the job allocation is expanded. If not, the dynamic request is rejected. The algorithm continues processing the same for every dynamic request and when all dynamic requests have either been satisfied or rejected, the static jobs are scheduled and started in the priority order. In this step, the number of jobs started may be different than the number of *StartNow* jobs in the previous step due to resources allocated to dynamic requests. Thereafter, low priority jobs are backfilled out-of-order.

Algorithm 2 Extended Maui Iteration

```
1:  while TRUE do
2:    Obtain resource information from Torque
3:    Obtain workload information from Torque
4:    Update statistics
5:    Refresh reservations
6:    Select static jobs eligible for priority scheduling
7:    Select dynamic requests eligible for priority scheduling
8:    Prioritize eligible static jobs
9:    Prioritize eligible dynamic requests
10:   Schedule static jobs in priority order and create reservations (without job start)
11:   for each dynamic request in the queue do
12:     Try to allocate resources for dynamic request (from idle before preemptible resources)
13:     if resources are available for the job then
14:       Check dynamic fairness policies to determine if job is allowed to get resources
15:       if job is allowed then
16:         Continue dynamic job with expanded resource allocation
17:         Update dynamic fairshare statistics
18:       else
19:         Reject the dynamic request
20:       end if
21:     else
22:       Reject the dynamic request
23:     end if
24:   end for
25:   Schedule the static jobs in priority order and create reservations (with job start)
26:   Backfill static jobs
27: end while
```

Strategies for allocating resources in response to dynamic requests can be controlled by site-specific parameters. For example, a dynamic request may obtain resources by preempting (when enabled) other running low priority or backfilled jobs. Existing Maui parameters can be used for this purpose. In the current version, due to the simple dynamic (de)allocation protocol, applications that cannot continue without an expanded set of resources must request for resources again at a later point in time if rejected. In contrast, leaving the dynamic request queued at the server and blocking the application until resources are obtained is not the best choice for evolving jobs that can continue execution but would have to run longer without more resources. An efficient negotiation mechanism where the application can specify a timeout for obtaining resources and where the batch system can indicate the time of availability of resources would be beneficial, and is one of our future goals.

D. Dynamic Fairness Policies

Fair sharing of resources between users is a compulsory responsibility of a site and is realized through job, user, and resource accounting. In static scheduling, fairness policies play a decisive role in the prioritization of jobs at most sites, as supercomputing resources are shared by an extensive group

of users. The Maui scheduler's fairshare policies, configurable through a set of administrator parameters, allow fine-tuned control of resource sharing among different users, groups, accounts, classes and quality of service [12]. However, when extending support for evolving jobs, they cannot be used to control the ill effects of resource stealing by an unpredictably evolving job.

For the dynamic scenario, we introduce two types of fairness policies dictated by the new parameter `DFSPolicy`: (i) `DFSSingleJobDelay` and (ii) `DFSTargetDelay`. The `DFSSingleJobDelay` simply imposes a limit on how long each queued job of a particular user can be delayed due to dynamic allocations to evolving jobs. The limit can be different for every user and can be set by the `DFSSingleDelayTime` parameter.

On the other hand, the `DFSTargetDelay` policy limits the cumulative delay caused to users over a configurable interval. The delay is set with the `DFSTargetDelayTime` parameter and the interval with the `DFSInterval` parameter, both in total seconds or `HH:MM:SS` format. The dynamic fairness setting can also be configured to combine both policies or disabled by setting the `DFSPolicy` to `DFSSingleTargetDelay` or `NONE`, respectively. When disabled, the dynamic requests will have the highest priority over the static jobs and the delay caused to static jobs will be ignored. Furthermore, the `DFSDynDelayPerm` parameter (1: allow, default ; 0: disallow) specifies whether a particular user's job can be delayed or not due to dynamic requests. Thus, a dynamic allocation will be unsuccessful if it would delay a job that is not authorized to be delayed. Also, when the evolving job and the static job are from the same user, the delay is not considered.

After each interval, the current delays are rolled back according to the `DFSDecay` parameter. This parameter indicates how much the current delay should decay at the end of an interval. For example, if the limit of delay for a user is 4800 seconds for an interval and if the current delay at the end of the interval is 3600 seconds, then a `DFSDecay` of 0.2 will result in the current delay of the next interval being initialized by 20% of 3600 seconds, which is 720. Therefore, the user's jobs can be delayed for a maximum of 4080 seconds in the new interval. This parameter allows historical delays to be considered.

Figure 6 shows a configuration of the `DFSSingleAndTargetDelay` policy over an interval of 6 hours with a decay of 0.4. Basically, the above delay permission and time settings can be set not only for users but also for groups, accounts, job classes and quality of service of the jobs. In an interval, assuming the current delay to be 0 for all users and groups, `user01`'s jobs can be delayed for any amount of time but cumulatively `user01` may experience only a maximum of an hour's delay. On the other hand, `user03` has no limit on the cumulative delay but each of `user03`'s jobs can only be delayed by a maximum of half an hour. `User04`'s limits combines both methods where the user can only experience up to 2 hours of cumulative delay but each job may only be delayed by 15 minutes at most. The `group05`'s configuration limits the cumulative delay experienced by all the users belonging to the group to a maximum of 4 hours. When user and group limits are specified for a user and his group, the most restrictive limits are used. Finally, jobs of `user02` and users

```
DFSPOLICY           DFSSINGLEANDTARGETDELAY
DFSINTERVAL         06:00:00
DFSDECAY            0.4

USERCFG[user01]     DFSDYNDELAYPERM=1 DFSTARGETDELAYTIME=3600 \
                    DFSSINGLEDELAYTIME=0
USERCFG[user02]     DFSDYNDELAYPERM=0
USERCFG[user03]     DFSDYNDELAYPERM=1  DFSTARGETDELAYTIME=0 \
                    DFSSINGLEDELAYTIME=00:30:00
USERCFG[user04]     DFSDYNDELAYPERM=1 DFSTARGETDELAYTIME=02:00:00 \
                    DFSSINGLEDELAYTIME=00:15:00
GROUPCFG[group05]   DFSTARGETDELAYTIME=04:00:00
GROUPCFG[group06]   DFSDYNDELAYPERM=0
```

Fig. 6. An example of dynamic fairness configuration.

of `group06` are not allowed to be delayed due to dynamic allocations. These simple parameters easily enable the desired dynamic configuration for a site according to its job mix. The parameters can be used to effectively avoid starvation of static jobs.

An aspect to be taken into consideration for a careful choice of delay limits is the effect of walltime and actual execution time of evolving jobs. Since dynamic reservations are also made until the rest of the walltime of the evolving job, delay limits are checked based on this time. However, users choose walltimes that are usually greater than the actual execution time of the application. Furthermore, the evolving application may finish earlier with additional dynamic resources. Thus the delay calculated during the dynamic allocation may be longer than the actual delay that will occur. Therefore, the delay limits should be configured with moderately higher values than intended to handle such instances. This enables a more accurate fairness measure. Also, user's attempts to take advantage of the system by submitting a small job (in order to get higher priority) and expanding after job start will negatively affect the user by affecting the fairshare priority for the user's next job. In our approach, the delay accumulated by the queued job does not depend on the walltime of the queued job. This may provide another level of optimization, but also adds another level of complexity to administrator's effort in establishing an optimal setting.

IV. Experimental Evaluation

In this section, we evaluate our proposed batch system. We use Quadflow as a proof of concept application and show the benefits that dynamic allocation can deliver for certain groups of production applications. We further present an analysis of a dynamic workload from both the user and the system perspective, using the ESP benchmark suite modified to contain evolving jobs.

Our evaluation consists of real experiments and is not based on simulations. All the experiments were conducted on a 15-node cluster system equipped with 2 Intel Xeon X5570 processors per node running at 2.93 GHz (8 cores per node). A separate 16th node was used as the headnode running the modified Torque version 4.1.0 and Maui version 3.3.1. The same node was also used as the frontend. As MPI implementation, we used Open MPI version 1.7.3.

A. Quadflow

As described earlier, the MPI-based CFD flow solver Quadflow solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method on locally refined grids. Starting on the coarsest grid level, the computational grid of the investigated flow configuration

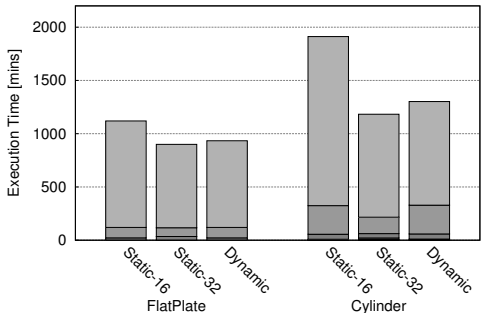


Fig. 7. Execution times of static and dynamic Quadflow test cases broken down by adaptation phase. Same shades denote the same phase.

is successively refined until the final grid level is reached. The local refinement of the grid leads to high computational efficiency. However, since the areas in need of refinement can only be identified during the solution process, no prior knowledge is available on the development of the number of grid cells. Two generic test cases are investigated in the following: (i) The laminar boundary layer flow over a flat plate in a supersonic flow field at Mach 2.6 is a pertinent example of a generic validation test case [13]. The boundary layer requires a high local resolution, whereas large parts of the flow domain can be kept quite coarse. (ii) The supersonic flow around a 2D Cylinder at Mach 5.28 is a typical example of a high-enthalpy stagnation point problem. Such flow fields are characterized by strong bow shocks, which need to be captured again with high local resolution. However, the exact location and size of these shocks is not known apriori which makes it difficult to predict the required number of grid cells in advance. Realistic scenarios frequently involve shock-shock interactions [14], in which the aforementioned problems become even more severe.

Figure 7 shows the execution times of the two cases in a static scenario with 16 and 32 cores (8 process per node), and a dynamic scenario where the execution is started with 16 cores/processes and expanded to 32 cores/processes at a threshold point. After each grid adaptation, the next computation phase is shaded lighter than the previous one. Technically, both cases use different numerical methods and the computational intensity of the FlatPlate case with one cell is equivalent to the Cylinder case with 4-5 cells. In the dynamic scenario, the dynamic allocation was done when a grid adaptation step led to more than 3000 cells per process for the FlatPlate and 15000 cells per process for the Cylinder test case. The application performed a total of 2 and 5 adaptations for the FlatPlate and the Cylinder test case, respectively. The threshold for the number of cells per process was exceeded in the final grid adaptation phase in both cases. That is, a dynamic request was issued after the last grid adaptation.

We can observe that by expanding its allocation to twice the number of allocated cores, the Cylinder test was faster by 33% (saving 10 hours) and the FlatPlate by 17% (saving 3 hours). The applications could also have been started with a larger allocation of 32 cores to obtain the speedup displayed without any dynamic allocation. However, this is only possible if a user can predict the threshold-exceeding growth of cells per process. A larger static allocation may also lead to under-loaded resources with too few cells per process as can be seen in our example. For instance, for the FlatPlate case, we can see that the time taken until the final grid adaptation level is identical when executed with 16 or 32 cores. This implies that starting the execution with 32 cores (i.e., with an extra 16 cores) has no effect as long as the number of cells stay within the threshold. By using resources only when required, such applications can obtain a similar speedup compared to starting the execution with a larger allocation. This not only reduces the usage costs for the user but also allows unused resources to be allocated to other jobs, thereby improving system utilization and throughput. These aspects are studied in the next section.

TABLE I. THE VARIOUS JOB TYPES, RESOURCE REQUIREMENTS AND THEIR STATIC EXECUTION TIME (SET) AND DYNAMIC EXECUTION TIME (DET) OF THE DYNAMIC ESP BENCHMARK.

Job type	User	Size	Count	SET [secs]	DET [secs]
A	user01	0.03125	75	267	-
B	user02	0.06250	9	322	-
C	user03	0.50000	3	534	-
D	user04	0.25000	3	616	-
E	user05	0.50000	3	315	-
F	user06	0.06250	9	1846	1230
G	user06	0.12500	6	1334	1067
H	user06	0.15820	6	1067	896
I	user06	0.03125	24	1432	716
J	user06	0.06250	24	725	483
K	user07	0.09570	15	487	-
L	user08	0.12500	36	366	-
M	user09	0.25000	15	187	-
Z	user10	1.00000	2	100	-

B. Dynamic ESP Benchmarks

A meaningful inference of scheduling performance can be obtained by only analyzing the scheduling outcome of a given workload. Given the scope of this work, a workload consisting of rigid and evolving jobs is necessary to evaluate the proposed batch system. Common scheduler evaluation benchmark workloads contain only rigid jobs. We are not aware of any benchmark with evolving jobs that is capable of assessing dynamic scheduling quality. Therefore, we modified the well-known ESP benchmark [15] so that it consists of both evolving and rigid jobs for our workload. Considering applications like Quadflow, we mainly focus on the dynamic allocation rather than dynamic deallocation.

The original ESP benchmark is composed of 230 jobs with 14 different job types running the same synthetic application. Each job type has a unique fixed execution time and uses a fraction of the total resources. The benchmark was modified to contain 30% evolving jobs and 70% rigid jobs (totaling to 69 evolving and 161 rigid jobs). Each rigid job type was considered to be run by a unique user and the evolving jobs were considered to be executed by the same user as listed in Table I. Job types F, G, H, I and J are considered as evolving jobs and the time at which the dynamic request is sent is modeled as in the Cylinder case of Quadflow. From the complete static and dynamic run of the Cylinder test case, it can be derviced that a dynamic allocation is needed after 16% of the total static execution time. Therefore, F, G, H, I and J jobs request 4 additional cores each after 16% of their total static execution time according to the ESP benchmark. When resources are not available at that point, the job continues and requests resources again after 25% of the total static run time as a second chance to obtain resources. If both attempts fail, the job continues with the current allocation. If the dynamic allocation is successful, a linear reduction of the execution time for the evolving job is assumed. Jobs are submitted in

TABLE II. PERFORMANCE COMPARISON OF THE EVALUATION CONFIGURATIONS

Config	Time [mins]	Satisfied Dyn Jobs	Util [%]	Throughput [Jobs/min]	Throughput [% Increase]
Static	265.78	0	77.45	0.86	-
Dyn-HP	238.78	43	85.02	0.96	11.3
Dyn-500	248.85	20	82.26	0.92	6.8
Dyn-600	241.06	27	83.57	0.95	10.2

a particular order with the first 50 jobs submitted instantly. Thereafter, jobs are submitted one by one with an interval of 30 seconds between each job submission. The workload consists of 2 special Z type jobs which use the complete cluster. After submitting the other 228 jobs, the Z jobs are submitted 30 minutes after the last job submission. As defined by the ESP benchmark, once the Z jobs are submitted, they receive the highest priority in the queue and no other low priority job can be executed. Backfilling is also disabled for the period that a Z job is queued. Evolving jobs that are already running may still obtain resources dynamically during this phase. The corresponding static execution time (SET) and dynamic execution time (DET) are also listed in Table I.

Four configurations were used for our evaluations. First, a static workload where F, G, H, I and J do not acquire any dynamic resources. Second, a dynamic workload with dynamic fairness disabled, thus giving dynamic requests highest priority (Dynamic-HP). In the third configuration, a dynamic fairness policy limited the cumulative delay for each static user's jobs by 500 seconds in an interval of 1 hour (Dynamic-500). Similarly, the fourth configuration limited the cumulative delay for each static user's jobs by 600 seconds (Dynamic-600). In all the configurations, the `ReservationDepth` and `ReservationDelayDepth` parameters were set to 5. Table II lists the results for various performance characteristics of the four configurations of the workload.

The first two columns of Table II show the total execution time of the workload (in minutes) and the number of evolving jobs that succeeded with their dynamic requests. The highest priority configuration (Dyn-HP) achieves the best overall system performance. 43 out of 69 evolving jobs obtained dynamic resources and the workload execution time was 10% faster (27 minutes). The system utilization increased to 85% as compared to 77% in the static setting and the throughput (TP) increased by 11.3%. Although the configuration improves the overall performance, it does not consider the delays caused to other static jobs.

Figure 8 shows the effect of such a configuration. It compares the waiting time of jobs (in the order of job submission) in the static workload with the dynamic workload in the dynamic highest priority configuration. It is evident that due to better resource utilization and earlier completion of evolving jobs, the overall waiting time of several jobs is reduced. However, we can see that many jobs with job IDs between 70 and 125 experience longer waiting times as compared to the static scenario. This unfairly affects the users who submitted jobs in this range. This can be observed in Figure 9, which compares the waiting time of type L jobs in the order of their submission. Half of the type L jobs are affected by longer waiting times. For other large production workloads consisting of long running jobs, these delays are more severe for certain users. Thus, obtaining the highest performance leads to such undesirable consequences.

The dynamic fairness policies address such issues. Figure 10 compares the waiting times in the Static, Dynamic-HP and Dynamic-500 configurations. The waiting time of jobs can be observed to be more uniform with respect to the static scenario. Figure 9 also shows the considerable improvement that type L jobs obtain due to this strategy. However, the configuration satisfied only 20 of the 69 evolving jobs, which reduced the throughput and the system utilization (Table II) compared to the highest priority configuration. This is a natural consequence of enabling a restrictive fairness scheme. However, moderate fairness policies can also enable a balance between system performance and user fairness. Figure 11 compares the waiting time of the Static and Dynamic-HP with the Dynamic-600 configuration. We can observe that with a little less restriction the number of successful dynamic requests increased to 27 and a system utilization and throughput close to that of the Dynamic-HP configuration is realized (Table II).

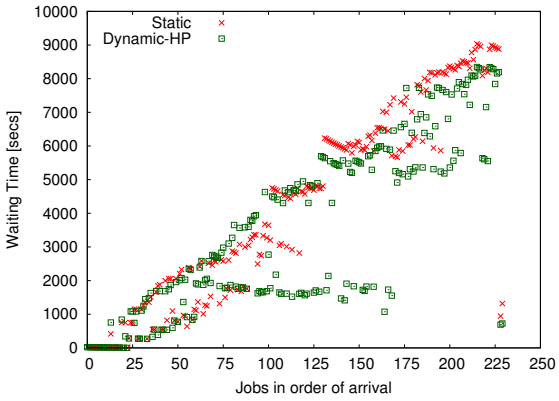


Fig. 8. Comparison of waiting times of jobs in the static and dynamic workload where highest priority is used for dynamic requests.

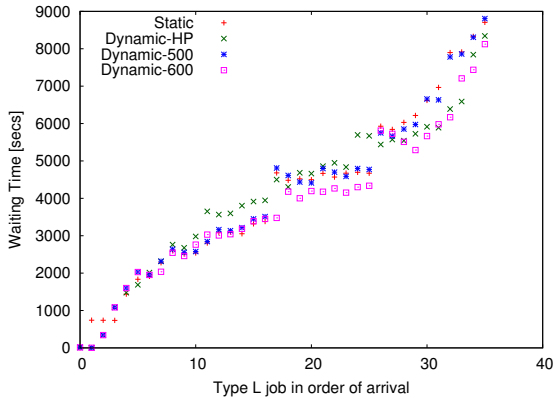


Fig. 9. Comparison of waiting times of type L jobs in all four configurations.

An important aspect that leads to this result is also the backfilling strategy. Our dynamic scheduling algorithm prefers to allocate idle resources to dynamic requests over backfilling the resources for smaller low priority jobs (as long as the dynamic request satisfies the fairness condition). This may give the impression that fewer jobs are backfilled in such dynamic environments. Our results show the opposite. There may be idle resources or resources may become idle shortly after responding to all dynamic requests. These may not be enough to service the high priority job in the queue (delayed due to dynamic request). However, it allows more smaller

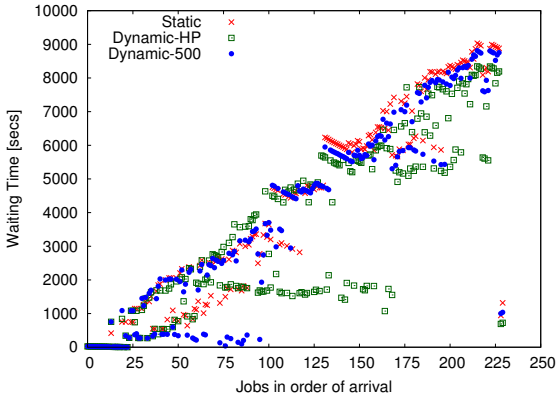


Fig. 10. Comparison of waiting times of jobs in Static, Dynamic-HP and Dynamic-500 configurations.

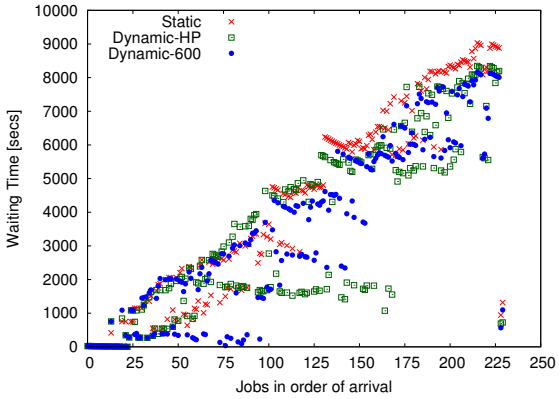


Fig. 11. Comparison of waiting times of jobs in Static, Dynamic-HP and Dynamic-600 configurations.

jobs to be backfilled, which leads to higher throughput. In Figures 8, 10 and 11, the backfilled jobs are the ones with considerably lower waiting times in the mid range of the job IDs. The Dynamic-HP configuration backfills the greatest number of jobs, followed by the Dynamic-600 and Dynamic-500 configurations. That is, the larger the number of successful dynamic requests, the greater was the backfilling ability and the higher was the throughput (refer Table II). Nevertheless, this pattern largely depends on the workload and may vary for a prioritized workload which maintains a fully utilized system. Thus, in the scenario of scheduling unpredictably evolving jobs, the results show that our approach provides a robust and flexible way to obtain a good balance between system performance and fairness.

C. Dynamic Allocation Overhead

The gain for an evolving application also depends on the overhead of the dynamic scheduling mechanism. Figure 12 shows the overhead of allocating from 1 to 10 nodes dynamically from a job running on one statically allocated node. Two scenarios are compared: (i) dynamic allocation without any workload at the batch system and (ii) with a workload of rigid jobs and a `ReservationDelayDepth` parameter of 5. It can be observed that the overhead for a dynamic allocation for as many as 10 nodes lies only in the sub-second range, which is negligible for a real-world application.

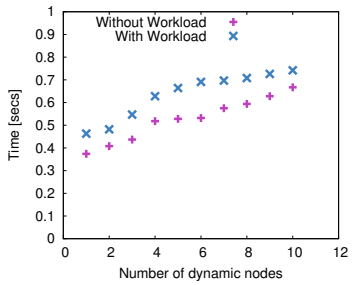


Fig. 12. Time taken for dynamic allocation of 1 to 10 nodes from a job using one statically allocated node.

V. Related Work

Efficient resource management and scheduling for cluster systems for rigid and moldable jobs is a well studied topic that has seen substantial advancement. The growing complexity of applications and their adaptive nature has motivated many researchers to seek dynamic resource management and scheduling solutions. Most of the work, however, pertains to supporting malleable jobs as a promising way of improving system performance. Notable are the RMS systems KOALA [16] and OAR [17], which show a clear benefit on supporting malleable jobs but agree that providing support for evolving jobs is rather challenging. Other work is mostly theoretical or based on simulation.

One of the early works in scheduling evolving jobs was performed by Boon-Ping and Shell-Ying [18] where dynamic scheduling is based on genetic algorithms. The approach was evaluated with simulators. However, whether the approach can be used with complex scheduling aspects such as prioritization, backfilling and fairshare, was not addressed. Investigating the RMS requirements for evolving jobs, Ghafoor et al. [19] proposed protocols for supporting evolving jobs and implemented a prototypical RMS to analyze the dynamic allocation overhead. However, the question of scheduling was not considered.

The challenge of scheduling unpredictably evolving jobs was investigated by Klein et al. with the CooRMv2 RMS [20]. In their approach, along with the general job requirements, the number of resources that may be dynamically required during execution must be indicated at job submission. The additional resources are preallocated for the job and may be used only by malleable or preemptive jobs. The authors evaluate their approach with a workload of rigid and malleable jobs and prove the benefits. However, as already discussed in Section II, this leads to performance degradation with workloads of evolving and rigid jobs like those used in our approach.

Support for dynamic allocation on demand can also be found in the Moab Workload Manager [21] and the SLURM resource manager [22]. Moab supports resource expansion and shrinkage for evolving jobs by regularly querying each application about its load. However, this is available for interactive workloads only. The SLURM resource manager supports expand/shrink operations by allowing a running job to submit a new job with a *dependency* indicator and then merging the allocations. By submitting a new job, the existing static fairshare mechanism is used to prioritize the dynamic request. In our approach, however, we distinguish the dynamic and static requests and introduce new fairness schemes for scheduling. Moreover, the approach in SLURM demands that

when dynamically releasing resources, the job must release all the nodes that came with a single dynamic request. Our approach provides more flexibility without such a restriction. Jobs may dynamically deallocate any subset of their current allocation.

From a slightly different angle, the fairness problem was addressed by Dinesh Kumar et al. [23] where jobs expand their walltime rather than consuming more resources. Their approach is based on extensions to a lookahead optimizing scheduler (LOS), which finds the best combination of jobs to be run simultaneously with the highest resource utilization. However, using the approach for dynamic allocations is not feasible.

Our approach enables efficient dynamic allocation as well as effective dynamic fairness strategies which, to our knowledge, have not been studied before at the depth presented in this paper.

VI. Conclusion and Outlook

Dynamic resource management facilities are key to serve the needs of the growing complexity of applications, improve fault tolerance and increase overall system performance. However, many challenges need to be overcome in order to create a fully-fledged batch system which can efficiently schedule all type of jobs and achieve good system performance. In our approach, we provide a solution to some of the many issues in dynamic resource management as a step towards developing advanced batch systems for the future.

Our proposed batch system enables on-the-fly resource allocation for evolving jobs based on runtime requests while ensuring fair access to resources for rigid and evolving jobs. The approach allows jobs to use additional available resources during unforeseen evolution. The batch system can expand and shrink resource allocations to jobs with little overhead. Our results show that applications can reduce their turnaround time and waiting time while increasing system utilization and throughput. Moreover, the dynamic fairness policies provide a simple set of parameters to configure fairness metrics according to site-specific requirements.

In the future, we intend to improve the batch system with better negotiation protocols between applications and the batch system and a fair prioritization mechanism between dynamic requests. We also plan to enable efficient scheduling for malleable jobs and address their fairness issues to develop a full-fledged batch system.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n° 287530. We also thank Adrian Spona from the German Research School for Simulation Sciences for valuable advice.

References

[1] D. G. Feitelson and L. Rudolph, "Towards convergence in job schedulers for parallel supercomputers," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996.

[2] S. Müller, *Adaptive Multiscale Schemes for Conservation Laws.*, ser. Lecture Notes in Computational Science and Engineering. Springer, 2003.

[3] V. W. Tomasz Plewa, Timur Linde, "Adaptive mesh refinement: Theory and applications." Springer, 2003.

[4] F. Bramkamp, P. Lamby, and S. Müller, "An adaptive multiscale finite volume solver for unsteady and steady state flow computations," *J. Comput. Phys.*, Jul. 2004.

[5] P. Malakar, V. Natarajan, S. S. Vadhiyar, and R. S. Nanjundiah, "A diffusion-based processor reallocation strategy for tracking multiple dynamically varying weather phenomena," in *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013.

[6] N. Eicker, T. Lippert, T. Moschny, and E. Suarez, "The deep project - pursuing cluster-computing in the many-core era," in *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013.

[7] J. Dongarra and et al., "The international exascale software project roadmap," *International Journal on High Performance Computing Applications*, 2011.

[8] D. Martin, P. Colella, M. Anghel, and F. Alexander, "Adaptive mesh refinement for multiscale nonequilibrium physics," *Computing in Science Engineering*, May 2005.

[9] G. L. Bryan, T. Abel, and M. L. Norman, "Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. New York, USA: ACM, 2001.

[10] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006.

[11] R. L. Henderson, "Job scheduling under the portable batch system," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1995.

[12] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 2001.

[13] C. Windisch, B. Reinartz, and S. Müller, *Numerical Simulation of Coolant Variation in Laminar Supersonic Film Cooling*. American Institute of Aeronautics and Astronautics, February 2012.

[14] C. Windisch, B. Reinartz and S. Müller, "H-adaptive simulation of hypersonic flows in thermochemical nonequilibrium." American Institute of Aeronautics and Astronautics, 2012.

[15] A. T. Wong, L. Oliker, W. T. C. Kramer, T. L. Kaltz, and D. H. Bailey, "Esp: A system utilization benchmark," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[16] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling malleable applications in multicluster systems," in *2007 IEEE International Conference on Cluster Computing*, Sept 2007.

[17] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic mpi," in *Proceedings of the 11th International Conference on Distributed Computing and Networking*. Springer-Verlag, 2010.

[18] B.-P. Gan and S.-Y. Huang, "Scheduling dynamically evolving parallel programs using the genetic approach," in *The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, May 2000.

[19] S. Ghafoor, T. Haupt, I. Banicescu, R. Carino, and N. Ammari, "A resource management system for adaptive parallel applications in cluster environments," in *In Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution 2005*, April 2005.

[20] C. Klein and C. Perez, "An rms for non-predictably evolving applications," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2011.

[21] "Moab hpc basic edition," http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.

[22] 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)*. Springer-Verlag, 2003.

[23] D. Kumar, Z.-Y. Shae, and H. Jamjoom, "Scheduling batch and heterogeneous jobs with runtime elasticity in a parallel processing environment," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012.