A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications

Suraj Prabhakaran*†, Marcel Neumann*†, Sebastian Rinke*†, Felix Wolf *†, Abhishek Gupta‡, Laxmikant V. Kale‡
*German Research School for Simulation Sciences, Aachen, Germany
Email: {s.prabhakaran, m.neumann, s.rinke}@grs-sim.de.com
†Technische Universität Darmstadt, Darmstadt, Germany
Email: wolf@cs.tu-darmstadt.de
‡Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
Email: {gupta59, kale}@illinois.edu

***Abstract*—The throughput of supercomputers depends not only on efficient job scheduling but also on the type of jobs that form the workload. Malleable jobs are most favorable for a cluster as they can dynamically adapt to a changing allocation of resources. The batch system can expand or shrink a running malleable job to improve system utilization, throughput, and response times. In the past, however, the rigid nature of commonly used programming models like MPI made writing malleable applications a daunting task, which is why it remained largely unrealized. This is now changing. To improve fault tolerance, load imbalance, and energy efficiency in emerging exascale systems, more adaptive programming paradigms such as Charm++ enter the scene. Although they offer better support for malleability, current batch systems still lack management facilities for malleable jobs and are therefore incapable of leveraging their potential. In this paper, we present an extension of the Torque/Maui batch system for malleability. We propose a novel malleable job scheduling strategy and show the first batch system capable of efficiently managing rigid, malleable, and evolving jobs together. We demonstrate that our strategy achieves consistently superior performance in comparison to every other state-of-the-art malleable job scheduling strategy under varying dynamics of the workload.**

***Keywords*-malleable jobs; evolving jobs; adaptive scheduling; adaptive resource management; batch systems**

I. Introduction

A batch system is an important middleware for managing supercomputing resources. It consists of a scheduler and a resource manager which work together to run parallel jobs on a cluster. In addition to just executing jobs, it is also responsible for efficient job management such as maintaining high system utilization and throughput (system perspective) while also ensuring faster response times and fairness among the jobs (user perspective). However, achieving the stated performance not only depends on efficient job scheduling, but also on the workload and the types of jobs it comprises.

As defined by Feitelson and Rudolph [1], jobs can be classified in four categories based on their flexibility. The first and the most common type is the *rigid* job, which requires a fixed number of processors throughout its execution. The second type is called the *moldable* job, whose resource set can be *molded* or *modified* by the batch system before starting the job (e.g., to effectively fit alongside other rigid jobs). Once started, its resource set cannot be changed anymore. Since the allocation of rigid and moldable jobs must be finalized before the job starts, it is termed *static* allocation. The third type is called the *evolving* job, which requests a resource allocation expansion or shrinkage during the job execution. Applications that use multiscale analysis [2] like Quadflow [3] or adaptive mesh refinement (AMR) [4] often exhibit evolving behavior typically due to unexpected increases in computations or having reached hardware limits (e.g., memory) on a node. Finally, the fourth class of jobs is called the *malleable* job. In contrast to evolving jobs, the expansion and shrinking of resources on a malleable job are initiated by the batch system. The application adapts itself to the changing resource set. This property of expanding or shrinking evolving and malleable jobs (together termed *adaptive jobs*) at runtime is called *dynamic* allocation.

Malleable jobs hold a strong potential to obtain high system performance. Batch systems can substantially improve the system utilization, throughput and response times with efficient shrink/expand strategies for malleable jobs. Similarly, applications also profit when expanded with additional resources as this can increase application speedup and improve load balance across the job's resource set. Enabling malleable jobs in cluster systems requires three major components: (i) a parallel runtime that is able to adapt to a changing resource set, (ii) a batch system with dynamic allocation facilities, and (iii) a communication mechanism between the two. Traditionally, all batch systems supported only static allocations. This is primarily due to the rigid nature of commonly used programming models like MPI, which made writing malleable jobs a laborious undertaking. Enabling malleability with MPI required the user to implement functionalities to listen to expand/shrink messages and manually manage the complex MPI communicators for the changing environment in the application. Although some efforts to produce malleable MPI applications with minimal programming overhead have been proposed [5] [6], malleable jobs remain largely unrealized in present cluster systems.

However, to meet the needs of improved fault tolerance, load imbalance, and energy efficiency in emerging exascale systems, adaptive programming paradigms (such as Charm++ [7]

and OmpSS [8]) are foreseen to play a significant role [9] [10]. For example, the Charm++ runtime can autonomically manage resources where it can identify and ameliorate load imbalance, adapt the application to a changing resource set, as well as cope with the intermittent loss of resources due to component failures. A fault-tolerant version of MPI slated to be released in the near future to fit exascale systems will also bring adaptiveness to its runtime system. Applications using these paradigms are automatically malleable. Using such programming paradigms paves the way for power-aware adaptive scheduling which has the ability to handle the energy challenge for future systems [10]. Furthermore, these paradigms become the first choice for writing evolving simulations as programmers do not have to manage the dynamically allocated resources manually.

Thus, it is an urgent necessity for batch systems to support dynamic allocation facilities and manage a mix of job types in order to reduce resource wastage, increase throughput and address the ever increasing user demand for faster response times. To this end, this paper advances the state of the art in batch job management by proposing an extended Torque/Maui batch system with a novel and an efficient scheduling and resource management strategy for malleable jobs. By combining it with our previous work on supporting unpredictably evolving applications [11], we present the first production batch system for the combined scheduling of rigid, malleable, and evolving jobs unlike past works which either simulated malleable jobs [12], [13] or demonstrated prototype/demo schedulers [14], [15], [16]. Towards establishing a next generation batch system and a tight coupling with parallel runtime, our main contributions are:

- An extended Torque/Maui batch system that is capable of shrinking/expanding malleable jobs
- A novel malleable job scheduling strategy
- A communication protocol between the batch system and the Charm++ runtime which enables malleability of applications [15]
- The integration with a scheduling algorithm for evolving jobs to support combined scheduling of rigid, malleable, and evolving jobs.

Evaluation of the batch system shows that our malleable job scheduling strategy consistently delivers superior performance in comparison to all other state-of-the-art strategies under varying dynamics of the workload.

The remainder of this paper is organized as follows. Section II discusses notable past work on scheduling malleable jobs. In Section III, we discuss the main objective of malleable job scheduling and outline the goal of our approach. In Section IV, we present the extended Torque/Maui batch system with shrink/expand facilities, the communication mechanism with Charm++, and the efficient scheduling strategy. We evaluate the batch system in Section V. Finally, we conclude the paper and discuss future work in Section VI.

II. Related Work

The advantages of malleable jobs were theoretically identified several years back. For this reason, frameworks for writing malleable applications were already introduced [17] [18] years ago. Kale et. al. [19] developed an adaptive runtime system for Charm++ and showed the benefits of malleable jobs compared to rigid ones with an experimental scheduler and equipartitioning policy.

Efficient resource management and scheduling for malleable jobs has been studied actively since then. Most of the work in this field pertains to theoretical aspects of scheduling and evaluation with simulations. For example, Carrol et. al [12] proposed a method for online scheduling of malleable jobs where the main goal was to assign resources to jobs such that the total running time is reduced. Users submit a job along with an indication of the amount of time that will be required by the job to run on a single processor. To ensure that users do not manipulate the scheduler by misreporting the job's parameters, incentives were given to users if their job was completed on the specified deadline. Sun et. al. [13] proposed a scheduling strategy whereby resources are distributed to malleable jobs using the equipartitioning technique but periodically adjusted based on application feedback of its scaling pattern. Similar approaches were taken by Mounie et al. [20] and Blazewicz et al. [21].

Below, we discuss notable prototypical/demo schedulers for malleable jobs. Their scheduling methodology follows a standard approach according to which, when the queued job with the next highest priority cannot be started anymore due to the lack of resources, the scheduler attempts to find nodes for the job by shrinking already expanded malleable jobs. When the next queued job cannot make use of any resources even by shrinking other jobs, then an expand phase is started where available idle resources are distributed across the running malleable jobs to improve system utilization and throughput. The order of jobs selected for shrinking and expansion varies according to the policy. Hungershöfer [16] showed that moldable and malleable jobs can significantly improve the response times through the equipartitioning strategy for shrink and expand. Utrera et. al. [14] proposed an FCFS-malleable strategy which distributes available nodes to malleable jobs in earliest-started-job-first order. They also investigated other strategies such as earliest deadline first, latest deadline first and the one with the least CPU utilization first. They showed that for a cluster composed only of malleable jobs, the earliest started first strategy generally performed better and improved the average response time by 31% in comparison to well-known EASY backfilling. The OAR resource manager [22] was also extended to support malleable MPI jobs. However, the problem of scheduling multiple malleable jobs was not discussed.

In the context of grids, Buisson et. al. [23] introduced malleability in the KOALA multicluster grid scheduler with an equigrow and equishrink policy, a different flavor of equipartitioning. While the equipartition policy tries to equalize the amount of malleable nodes held by each running malleable job, the equigrow policy simply distributes the current set of idle resources equally among the malleable jobs. Thus, irrespective of the number of nodes held malleably by a job, it will be expanded by an equal proportion of idle nodes when the scheduler starts an expansion phase.

In general, naive equipartitioning was often employed and benefits were shown through prototypical schedulers. In this paper, we present a production batch system for malleable jobs and show its benefits with Charm++ applications that become automatically malleable when run under the proposed batch system. We also evaluate a new malleable job scheduling strategy and compare it with naive equipartitioning, earliest started first, earliest deadline first and latest deadline first.

III. Basic Approach

In this section, we discuss the various aspects of scheduling malleable jobs and define the goal behind the approach taken in this paper.

A. Resource Utilization and Throughput

Malleable jobs help to considerably reduce the resource wastage by using idle resources when expanded. However, increased resource utilization does not always imply higher throughput. With a cluster running several malleable jobs, an inefficient selection of jobs for expansion and shrinkage can lead to higher resource utilization without any increase in throughput. In certain cases it may even be counterproductive to a gain in throughput. Such scenarios are shown in Section V with the evaluation of some of the scheduling strategies. Therefore, a malleable job scheduling scheme must analyze job and resource dependencies to deliver a better overall performance.

B. Fairness

Enabling some amount of fairness in expand/shrink is essential as it can motivate users to write more malleable applications as opposed to rigid ones. Equipartitioning is a reasonably good strategy towards enabling fair dynamic (de)allocations. However, equipartitioning alone cannot improve the global system throughput and response times for the same reason that it can contradict the best malleable job selection for expand/shrink. This is also exemplified with experiments in Section V. Therefore, a good malleable scheduling strategy must target system efficiency along with as much fairness as can be delivered.

C. Communication with the Parallel Runtime System

Apart from powerful scheduling schemes, enabling malleability also requires a scalable shrink/expand mechanism. Typically, expansion can happen almost instantaneously as the parallel runtime may be able to spawn new parallel tasks as soon as it obtains the fresh nodes. However, shrinking may require more time since the task running on the nodes to be removed needs to be completed (usually until the end of an iteration in iterative applications), data required by the rest of the application needs to be retrieved and the process must be killed. To facilitate immediate release, it is also possible to use the internal checkpointing mechanism of Charm++ to abort the processes immediately and restart the application from the latest checkpoint. For simplicity, we follow the former policy in this work. Another aspect of communicating with the runtime system is the option of making scheduling decisions based on application feedback. Typically, when running iterative applications, communicating the iteration times regularly to the batch system can help it select more responsive and well-scaling applications for shrink/expand. However, feedback mechanisms introduce other overheads such as too frequent communication, inconsistency (as iteration times are not always constant), and increased complexity for non-iterative malleable applications. Thus, efficient feedback mechanisms for malleable applications have been exclusively studied by many [24], [25]. Scheduling based on feedback from application on its scaling pattern is our interest for investigation in the future and is out of scope of this work.

IV. The Adaptive Batch System

In this section, we describe the dynamic allocation facilities in the Torque/Maui batch system and scheduling strategy for malleable jobs. We start by providing an overview and proceed to discuss the extensions.

A. Overview of Torque/Maui

The Torque/Maui batch system is one of the most commonly used middleware for batch job control. The Torque resource manager [26] is based on the PBS project [27], extended to improve scalability and fault tolerance, and is currently maintained by Adaptive Computing. Torque is usually integrated with sophisticated schedulers such as Maui [28], which provides advanced scheduling features such as job prioritization, fairshare, and backfill scheduling.

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 1 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 Maui allocates resources for this job, the list of nodes is sent to the server, which forwards it to one of the nodes called for the job that assumes the role of *mother superior*. The mother-superior node and the other allocated nodes perform a *join* operation, after which the user application starts execution. The Maui scheduler communicates with the server and schedules jobs iteratively. A scheduling iteration is followed by a period of sleeping or

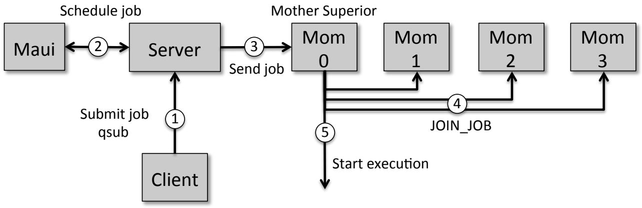

Fig. 1. Workflow of the Torque/Maui batch system. Circled numbers indicate the sequence of steps.

Algorithm 1 Maui Iteration

```
while TRUE do
    Obtain resource information from Torque
    Obtain workload information from Torque
    Update statistics
    Refresh reservations
    Select jobs eligible for priority scheduling
    Prioritize eligible jobs
    Schedule the jobs in priority order and create reservations
    Backfill jobs
end while
```

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. During this process, two types of jobs are created according to the reservations: *StartNow* and *StartLater*. *StartNow* jobs are jobs with reservations that start immediately. *StartLater* jobs are jobs with reservations made to start at a later point in time due to the lack of resources preventing immediate job start. These reservations are created for the earliest time at which the resources will become available for the jobs. The number of *StartLater* jobs that need to be created can be configured by an administrator-configurable parameter `ReservationDepth`. 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 fairness, resource requirements, and waiting time of a job. A detailed description of these features is available in [28].

B. Shrink/Expand for Malleable Charm++ Jobs

Given the structure of the Torque/Maui batch system, the following features were implemented in Torque to enable shrink/expand facilities:

- An extended `qsub` command to submit a malleable job
- Functionality to shrink/expand a resource set at the server based on Maui's instruction
- Functionality to associate/disassociate nodes at the mom based on the server's instruction
- A communication mechanism between the mom and the Charm++ runtime system for malleability interaction

A malleable job can be submitted with the extended `qsub` command as shown in the example below:

```
$ qsub -l nodes=2:ppn=8,walltime=3600 \
> -L max=6,type=charm++ jobscript.sh
```

A user indicates the minimum number of nodes required for a job, the fixed number of processors required per node and the duration of the job with the minimum number of nodes through the `-l` option. To indicate the malleability of the job, the user must specify the `-L` option indicating the maximum number of nodes that can be used by the job and a job *type*. In general the shrink/expand facilities can be used for any job. However, as there is no standard way of interacting between the batch system and the parallel runtime, it requires development and integration of appropriate communication for every programming paradigm. The job *type* indicates to the batch system the type of programming paradigm used by the job so that the right mechanism can be chosen for communication. In the current version, only Charm++ jobs are fully supported.

For malleability interactions, the Converse Client-Server interface (CCS) [29] in the Charm++ runtime system was leveraged and a shrink/expand specific handler was developed. A separate management thread of a Charm++ job acts as a CCS-server that listens to shrink/expand via TCP/IP as soon as the application begins executing. The corresponding CCS-client has been integrated into the mom. Before starting the application through the `charmrun` command from the job script, the mother-superior assigns a unique port at which the CCS-server must listen by appending the highlighted code shown below:

```
> charmrun +p8 ./exec \
        ++server ++server-port=1234
```

Users are not permitted to manually activate the CCS-server. This allows the mother-superior to assign unique port numbers to all Charm++ applications that may run on the same space-shared node. Figures 2 and 3 illustrate the steps of an expand and shrink process in the Torque RMS, respectively. When the scheduler initiates an expand operation, it sends the new list of hosts to be added for the job to the server. The server updates the internal information and forwards the list to the mother-superior executing the job. The mother-superior modifies the nodes list (*hostfile*) and performs a *dyn_join* operation to dynamically associate the new nodes with the job. It then sends the CCSExpand message through the CCS-client API to inform the application. The reply to this message from the CCS-server is immediate and the Charm++ runtime starts

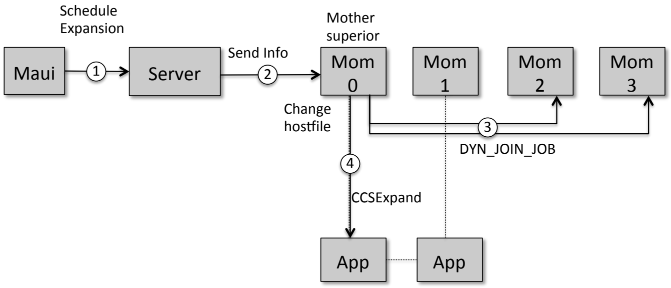

Fig. 2. Expanding a job by adding nodes 2 and 3. Circled numbers indicate the sequence of steps.

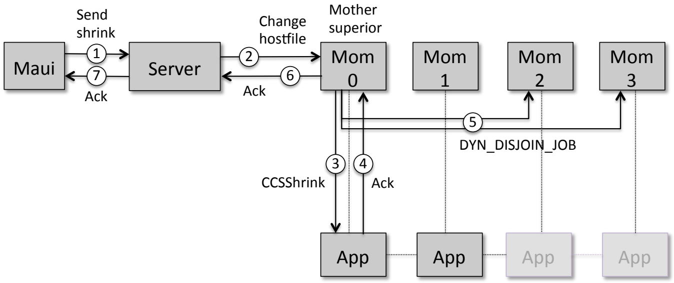

Fig. 3. Shrinking a job by removing nodes 2 and 3. Circled numbers indicate the sequence of steps.

using these resources after the next synchronization point in the application (typically between iterations). A similar process is carried out during a shrink operation, except that after the mother-superior sends the CCSShrink message to the application, the reply is not immediate. The CCS-server replies only after the data from the shrinking nodes are retrieved and the processes are cleaned during the next synchronization point.

C. Malleable Scheduling with Maui

By design, the Maui scheduler supports only rigid jobs. In our previous work [11], we extended the Maui scheduler to support evolving applications. To maintain fairness between evolving requests and static requests (jobs queued through `qsub`), new dynamic fairness policies were introduced. In this work, we further extended the Maui scheduler in the following ways:

- We enhanced the resource allocation mechanism to expand and shrink a resource allocation set
- We devised a *dependency-based expand/shrink* (DBES) algorithm for efficient scheduling of malleable jobs
- We enriched Maui's iteration with combined scheduling of rigid, malleable, and evolving jobs

All malleable jobs are always allocated according to their minimum requirements and later expanded. The DBES algorithm consists of two expansion steps. The first expansion step, contrary to other strategies, is based on analyzing job and resource dependencies, and targets increasing throughput. The earliest start time of a *StartLater* job is the deadline of that running job after whose completion all the resources requested by the *StartLater* job become available. For example, consider a four-node system with two running jobs A and B using one node each for a scheduled period of 1 hour and 1/2 hour, respectively. If a queued job C requires 3 nodes for execution, it can start as soon as B is completed because two nodes are already available. On the other hand, if C requires 4 nodes for execution, it has to wait longer until the completion of job A. In our approach, we determine the dependencies of all *StartLater* jobs in the order of their priority. If the job on which a *StartLater* job depends is malleable, it is expanded using the available resources up to its user-specified maximum. Jobs expanded in this step maintain information about the dependent job in the queue for which the expansion was

Algorithm 2 Maui Iteration

```
while TRUE do
    Obtain resource information from Torque
    Obtain workload information from Torque
    Update statistics
    Refresh reservations
    Prioritize eligible static requests
    Prioritize eligible evolving requests
    Schedule static requests in priority order and create reservations (without job start)
    for each evolving request do
        Allocate idle resources
        if Enough idle nodes not available then
            Shrink expanded malleable jobs to find resources
        end if
        if Enough idle nodes found then
            Apply fairness policies and determine if job expansion is allowed
            if Expansion is allowed then
                Allocate resources for evolving job
            else
                Reject the dynamic request
            end if
        else
            Reject the dynamic request
        end if
    end for
    Reschedule static requests and create reservations (with job start)
    Update job dependencies according to the new system state
    for each reserved job do
        Prioritize malleable jobs in the order: (i) malleable job expanded for this reserved job, (ii) malleable job expanded for no specific reserved job, (iii) malleable job expanded for other reserved jobs
        Analyze if expanded malleable jobs can be shrunk in the above order to make enough nodes available to start the reserved job
        if enough nodes were found then
            Shrink the selected malleable jobs
            Start the reserved job
        end if
    end for
    Reschedule static requests and create reservations
    Update job dependencies according to the new system state
    for each reserved job do
        if job depends on one malleable job then
            Expand the malleable job with the available nodes
        else if job depends on more than one malleable job then
            Equipartition available resources among these malleable jobs
        end if
    end for
    Update job dependencies
    Backfill non-reserved static requests from the job queue
    Equipartition available idle nodes among other running malleable jobs
end while
```

made. The second expansion step targets improving resource utilization and fairness, thereby equipartitioning the available resources across malleable jobs.

The complete Maui iteration for the combined scheduling of rigid, evolving, and malleable jobs is shown in Algorithm 2. In the first step, all the static and evolving requests are prioritized separately (line 6-7). Static requests are scheduled, which creates the *StartNow* and *StartLater* jobs (line 8). At this point (lines 9-24), *StartNow* jobs are not yet started. Evolving requests are now scheduled, which may steal resources from the *StartNow* jobs, thereby causing delay to the *StartNow* as well as to the *StartLater* jobs. In our previous work, we designed dynamic fairness policies for the Maui scheduler to ensure fairness between such unpredictably evolving requests and static requests, which were based on administrator-configurable parameters [11]. With these policies, it was shown that evolving jobs can be served (as long as idle nodes are available) with fairness by controlled and admissible delay that can also take historic delays for users into consideration. This strategy improved the system utilization and throughput as well as job response times. These fairness policies are applied at this point to decide if the evolving request should be satisfied or rejected. When no idle nodes are available, the system determines whether shrinking expanded malleable jobs can serve the evolving requests. At this point, the malleable jobs expanded in the second expansion step are considered first. If not enough nodes can be extracted from these jobs, the other malleable jobs expanded as part of the first expansion step are considered. If sufficient nodes are found, the malleable jobs are instructed to shrink to release the nodes. The dynamic fairness policies are then applied again to determine whether the evolving request can be satisfied with the newly available nodes. If yes, the evolving job is granted these nodes. Otherwise, the nodes obtained from the shrink operation are used later for expansion or backfilling. In the future, we plan to improve the system to enable it to apply dynamic fairness policies without having to shrink the jobs so as to reduce the overhead.

After all the evolving requests have either been satisfied or rejected, a new schedule of static requests is performed as the state of the system and job dependencies may have changed (line 25). *StartNow* jobs produced at this step are started immediately. Any expanded malleable jobs maintaining invalid job dependencies are cleared (line 26). Now a shrink phase is initiated to attempt to locate nodes for *StartLater* jobs (lines 27-34). Starting from the highest priority *StartLater* job, the scheduler analyzes whether shrinking expanded malleable jobs can produce enough nodes to start the *StartLater* job. Malleable jobs are considered for shrinking in the following order: (i) expanded malleable jobs that are dependency jobs of this *StartLater* job, (ii) malleable jobs expanded during the second expansion step (i.e., expanded for no specific *StartLater* job), and (iii) malleable jobs expanded for other *StartLater* jobs which have lower priority than this *StartLater* job. If enough nodes are found, the malleable jobs are instructed to shrink to release the required resources and are allocated to the *StartLater* job to start immediately. The same procedure is then applied to the next *StartLater* job until enough resources can be located for a *StartLater* job. At this point, the iteration proceeds to the next phase.

Since there might have been changes in the system state again (due to starting more jobs), a new schedule of the queued jobs is initiated to create `ReservationDepth` number of *StartLater* jobs, and job dependencies are recomputed (lines 35-36). The first expansion step is initiated where the computed job dependencies are used to expand the malleable jobs each *StartLater* job depends on, as explained already (lines 37-43). During this step, nodes can also be stolen from other malleable jobs that were (i) either expanded for no specific *StartLater* job (i.e., in the second expansion step) or (ii) expanded for a *StartLater* job that has lower priority than the currently considered *StartLater* job. Such a transfer of nodes allows a malleable job to be expanded as much as possible to increase its speedup and allow the *StartLater* job to be started early. In some cases, a *StartLater* job may also depend on two malleable jobs having the same completion time. In this case, the available resources are equipartitioned among these malleable jobs. Running malleable jobs on which no *StartLater* job depends are not expanded in this step. After the first expansion step, a backfill step is initiated which ensures that only those jobs are started that will not delay any *StartLater* job (line 45). Finally, after the backfill step, a second expansion phase begins where the malleable jobs on which no other job depends are expanded with the available resources through equipartitioning (line 46).

One of the important differences between the proposed algorithm and other approaches is that it gives due importance to backfilling with a two-step expansion process. As mentioned in Section II, other methods only perform a shrink operation if the next job in the queue cannot be started. This is followed by an expand phase where running malleable jobs are expanded. As a final step, backfilling is performed with nodes available after expansion. Some approaches such as [14] ignore backfilling, which may be suitable for a workload with 100% malleable jobs but not for a workload that also consists of rigid jobs. In our approach, we perform a "needful" expansion, followed by backfilling and equipartitioned expansion. Also, in every iteration, dependencies are recomputed only if there is a change of state in the system, thereby avoiding unnecessary and frequent dependency computations. In the presence of evolving jobs, our approach attempts to its best to select those malleable jobs for shrinking that will least affect the throughput. Furthermore, since the number of *StartLater* jobs can be configured by the `ReservationDepth` parameter, administrators can modify it to control the behavior of the scheduler according to the site's workload characteristics. At a site with large number of malleable jobs, the `ReservationDepth` can be increased to gain more from dependency-based expansion, while at a site with a generally low number of malleable jobs, it can be reduced to favor more backfilling. The resources are charged only for the amount of time they are used. In the future, we also plan to provide administrator commands

TABLE I
PROPERTIES OF ALL JOB TYPES OF THE MODIFIED ESP BENCHMARK

Job Type	Size [% of System]	Count	Static Execution Time [secs]	Malleable	Evolving
				Number of Cells	Execution Time [secs]
A	0.03125	75	267	6x6x6	-
B	0.06250	9	322	8x8x8	-
C	0.50000	3	534	15x15x15	-
D	0.25000	3	616	10x10x10	-
E	0.50000	3	315	15x15x15	-
F	0.06250	9	1846	8x8x8	1230
G	0.12500	6	1334	9x9x9	1067
H	0.15820	6	1067	11x11x11	896
I	0.03125	24	1432	6x6x6	-
J	0.06250	24	725	8x8x8	-
K	0.09570	15	487	7x7x7	-
L	0.12500	36	366	8x8x8	-
M	0.25000	15	187	10x10x10	-
Z	1.00000	2	100	4x4x4	-

to manually shrink or expand jobs, which is useful for fault tolerance and easy proactive migration.

V. EXPERIMENTAL EVALUATION

In this section, we evaluate the proposed batch system and analyze its performance with respect to throughput, system utilization, and overhead.

A. *Experimentation Setup*

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 extended Torque version 4.1.0 and Maui version 3.3.1. For a fair comparison, all the experiments were performed with `ReservationDepth` in Maui set to 5.

Common benchmark workloads for the evaluation of schedulers contain only rigid jobs. We are not aware of any benchmark with malleable or evolving jobs. Therefore, we modified the well-known ESP benchmark [30] to contain various percentages of rigid, malleable, and evolving jobs. 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. To evaluate the DBES strategy, the synthetic application was replaced by a Charm++ mini-application *LeanMD*. LeanMD is a Molecular Dynamics (MD) mini-application which performs a simplified version of the force calculations of NAMD [31], a widely used MD code. LeanMD uses two Charm++ object arrays: (i) *cells* - a collection of atoms in 3D space, and (ii) *computes* - perform force calculation on atoms. To comply with the benchmark, each LeanMD mini-application was executed with varying numbers of cells and iterations to fit the job type's running time. As an evolving job, a synthetic MPI application with an evolving pattern similar to the real-world application Quadflow [3] was introduced. Quadflow is an MPI-based CFD flow solver that solves the compressible Navier-Stokes equations using a cell-centered fully adaptive finite volume method on a locally refined grid. Our synthetic application imitates a typical high-enthalpy stagnation point problem of supersonic flow around a 2D Cylinder at Mach 5.28. The application evolves after 16% percent of its static runtime and requests 4 additional cores. If the resources are not available at that point, the job continues and requests resources again after 25% of the total static running time as a second chance to obtain resources. If both attempts fail, the job continues with the current allocation until its completion. However, if the evolving request was satisfied, a linear reduction of the execution time is assumed for the evolving job. Table I shows the various job types of the modified ESP benchmark, the fraction of resources each job uses in a cluster, the total number of jobs for each type, the static execution time of each job type, the number of cells used when the job type was converted to malleable job and the execution time of the job as an evolving job if granted resources during the evolution.

B. *Scheduling Malleable Jobs*

Figure 4 shows the comparison of the total execution time of the ESP workload with varying amounts of malleable and rigid jobs with the earliest started first (ESF), earliest deadline first (EDF), latest deadline first (LDF), naive equipartitioning (EP) and DBES strategies. The *rigid* strategy executes the workload without any expand/shrink operations—irrespective of the number of malleable jobs present. It can be observed that the DBES strategy has a lower execution time in all cases compared to the other strategies. With 100% malleable jobs, the DBES strategy performs best with about 32% higher throughput than rigid scheduling and about 7% higher throughput than the best-performing state of the art strategy (in this case, EP). For large systems with longer workloads, this impact will be of higher magnitude. Furthermore, it can be observed that DBES is consistent in achieving the best total execution time unlike the other strategies. For example, while the equipartitioning strategy is the one that performs best among the state of the art strategies for a workload with 60% malleable jobs, it delivers the worst performance with 10% and 30% malleable jobs, and second worst with 90% malleable jobs. Similarly, the ESF strategy performs best amongst the state of the art strategies for a workload with 20% malleable jobs, but worst with 50% malleable jobs. In certain cases the other strategies can perform even worse than rigid scheduling. For instance, the EDF strategy with 20% malleable jobs and EP strategy with 30% malleable jobs took longer execution times than rigid scheduling. This is a direct effect of inefficient job selection for expansion and shrinking whereby the nodes are allocated to malleable jobs which becomes ineffective while the rigid scheduling gains by backfilling rather than by expanding/shrinking. The DBES strategy never shows such a pattern as the dependency-based analysis ensures that malleable jobs are expanded only if this may facilitate an earlier start time for queued jobs. Otherwise, backfilling is given precedence to improve throughput. Thus, it extracts

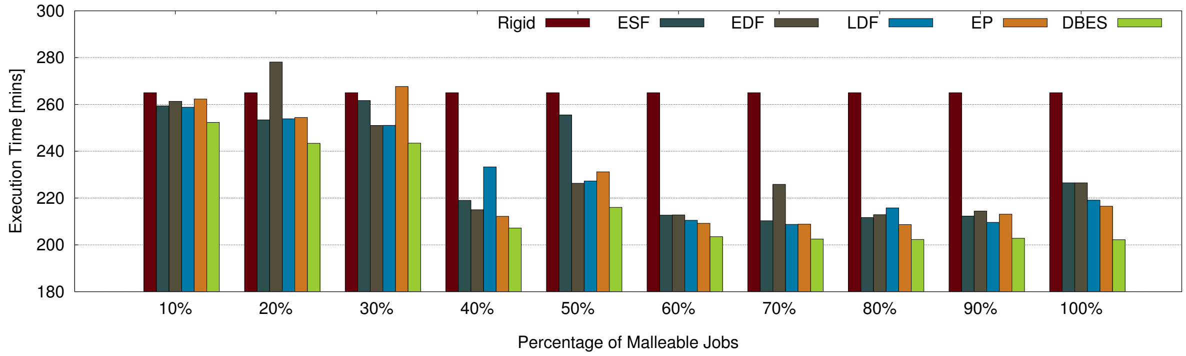

Fig. 4. Time for completion of the modified ESP workload with varying amounts of rigid and malleable jobs.

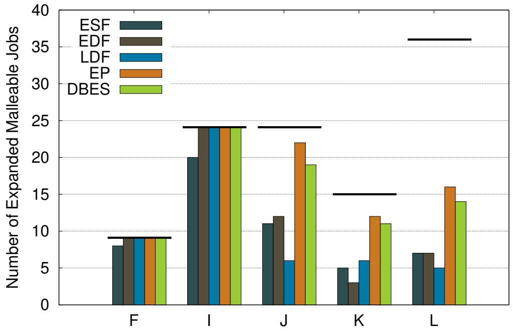

Fig. 5. Comparison of the number of expanded malleable jobs belonging each category under various strategies for 50% malleable jobs. The total number of actual malleable jobs in each category is indicated by a horizontal line.

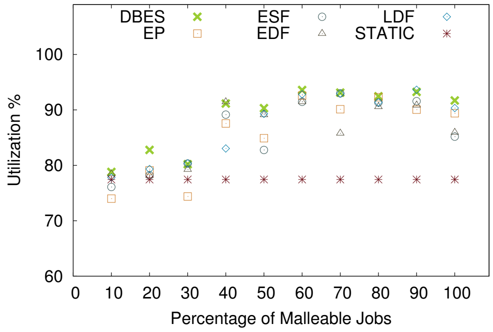

Fig. 6. Comparison of the average system utilization achieved by all the strategies for the ESP workload with various percentages of malleable and rigid jobs.

the best of malleability and backfilling, which other strategies fail to achieve. Overall, this implies that for a site with either a predictable number of malleable jobs or an unpredictable variation in the number of malleable jobs at a given point of time, the DBES strategy can be confidently applied to obtain the best throughput.

The reason behind the behavior of all these strategies and their resulting performance can be better understood by a deeper analysis of the pattern of expansion of all the strategies. As an example, Figure 5 shows the number of different types of malleable jobs expanded at some point in time during execution when run under each strategy. As ESF prefers to expand the job which started earliest, much of the expansion was made to type J jobs and it was not able to fully expand the long running F and I jobs which led to the least throughput. LDF preferred to expand jobs with long running times and therefore all F and I jobs were expanded at a very early stage. But this did not allow enough J, K and L jobs to be expanded to see a throughput gain. Similarly, along with F, I and J, EDF also expanded a few more L jobs but was not able to expand enough K jobs which actually have short running time. This was mainly due to the unavailability of idle resources when the majority of the K jobs were running. This was a result of backfilled A jobs using all the resources. As short jobs finished ahead of their walltime limit with more resources, they were used directly to start queued jobs instead of expanding running ones. Thus, due to a good use of resources, EDF performed best amongst ESF, LDF, and EP albeit by expanding only a smaller number of malleable jobs. EP performed worst after ESF although it expanded the greatest number of malleable jobs of all the strategies. This is due to the equal distribution of resources and frequent expansion without giving priority to backfilling. We can observe that DBES has a similar expansion pattern to EP but still has about 7% higher throughput than EP. This is not only because it expands a reasonably large number of malleable jobs, but also because it does so in the right order and at an effective point in time while giving priority to backfilling when a gain cannot be obtained from expansion.

Figure 6 compares the overall average system utilization maintained by the strategies for workloads with various percentages of malleable jobs. It can be seen that in general the DBES strategy maintains the highest system utilization. On the other hand, other strategies also achieve average utilization close to or even slightly better than DBES in some cases, but still have lower throughput. For example, with 40% malleable jobs, the EDF strategy maintained a slightly better average system utilization than DBES but still had about 5% less throughput than DBES. Thus, DBES not only increases system utilization but also assures increased throughput.

Note that the execution time of the workload with 50% malleable jobs was slower than that containing only 40%. This is because the workload for 40% malleable jobs was formed by making jobs F, G, H, I, K, and L malleable, while the 50% was made with F, I, J, K, and L. The non-malleability of long running G and H jobs caused the longer execution time of the workload with 50% malleable jobs. Thus, the presence of

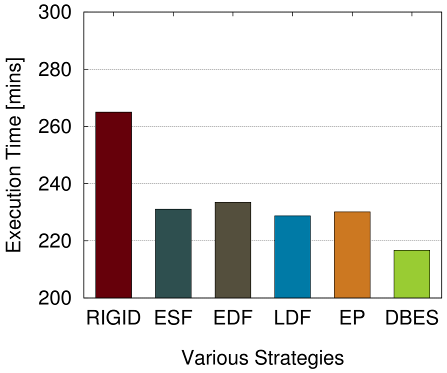

Fig. 7. Time for completion of the modified ESP workload under various strategies with 10% evolving jobs, 40% malleable jobs and 50% rigid jobs.

TABLE II
COMPARISON OF EACH TYPE OF EVOLVING JOBS SATISFIED WITH VARIOUS STRATEGIES.

	DBES	EP	ESF	EDF	LDF
Total no. of evolving jobs satisfied	11	11	13	17	12
Jobs shrunk to satisfy evolving jobs	9	9	9	10	8

larger numbers of malleable jobs does not always mean better performance than the presence of only a smaller number.

C. Combined Scheduling of Rigid, Malleable, and Evolving Jobs

We demonstrate and analyze the combined scheduling of rigid, malleable and evolving jobs with ESP workload containing 10% evolving (F, G and H), 40% malleable (I, J, K and L), and 50% rigid jobs. We are not aware of any other work that consists of a comprehensive scheduling method for the combined scheduling of the above job types. Therefore, we combined our evolving job scheduling strategy along with DBES and other strategies to compare the execution time. This is shown in Figure 7. We can see that DBES again has the fastest execution time with an increase in throughput of about 6% in comparison to the best performing state-of-the-art strategy (in this case, LDF). Table II presents the total number of evolving jobs that were satisfied in each case and the corresponding number of malleable jobs that were shrunk to obtain the resources for the evolving jobs. While EDF and ESF satisfied more evolving jobs, all strategies needed to shrink roughly an equal number of malleable jobs to obtain resources. This implies that a greater number of idle nodes were present in EDF and ESF during job evolution compared to other strategies. In other words, the inefficient system utilization was advantageously employed for the evolving jobs. Out of 9 malleable jobs that were shrunk in the DBES strategy, 5 malleable jobs had been expanded in the second expansion step (through equipartioning) and 4 malleable jobs had been expanded in the first expansion step (through dependency analysis). Thus, the improved performance is a combined result of the DBES strategy handling the malleable jobs and the choice of malleable jobs for shrinking to make resources available for the evolving jobs. The DBES strategy avoids as far as possible selecting malleable jobs expanded through the dependency analysis. Therefore, while almost the same number of evolving jobs were satisfied in all the strategies except EDF, the DBES still achieves better performance.

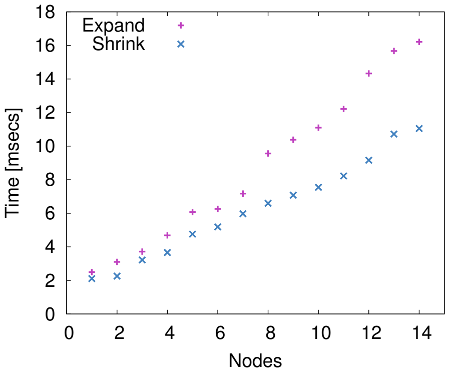

Fig. 8. The time taken for (i) adding 1 - 14 additional nodes to a job initially using 1 node (expansion), and (ii) removing 1 - 14 nodes from a job initially using 15 nodes (shrinking).

D. Overhead

Figure 8 shows the overhead of expand and shrink operations. For expansion, the total time required to expand a single node job with 1 to 14 additional nodes is plotted. Naturally, the time required for expansion increases with am increasing number of nodes due to communication with a larger number of nodes during the *dyn_join* operation. However, the time stays below 20 milliseconds for expansion with up to 14 additional nodes, which is fast and efficient. For shrinking, the plot shows the total time required for *immediately* removing 1 to 14 nodes from a job using 15 nodes. By “immediately”, we mean that nodes are released instantly after receiving a shrink message. The total time taken for such an operation increases with a larger number of nodes to shrink but remains in the millisecond range. This is generally faster than expansion since *dyn_disjoin* communicates much less data. However, as explained in Section III, the time taken for a shrink operation depends on the time required for the task running on the shrinking nodes to complete. Since the shrink message can be initiated at any time, the time required for the task to be completed cannot be predicted beforehand. Therefore, in reality shrinking usually takes longer than expansion. In the future, we plan to extend the communication mechanism to also include minimal application feedback in order to initiate shrink messages at a convenient point in time so as to reduce the waiting time until task completion.

VI. CONCLUSION AND OUTLOOK

As we move towards the next generation of supercomputers, the presence of malleable and evolving jobs is growing stronger. With adaptive programming paradigms taking the center stage, malleability of applications is a natural byproduct. Similarly, as the complexity of applications increases, unpredictably evolving jobs are also often seen to be on the rise. Thus adaptive resource management and scheduling is fundamental to support malleable and evolving jobs and thereby gain higher throughput and serve the ever increasing demand for faster response times. For many years, the technical challenge that had to be overcome before supporting adaptive jobs and the rigid nature of programming models prohibited the implementation of a fully fledged adaptive batch system in reality.

In this paper, we proposed the first production batch system that is capable of combined scheduling of rigid, malleable, and evolving jobs. We proposed a novel malleable scheduling strategy called DBES that expands and shrinks malleable jobs based on dependency analysis and combines it with backfilling to gain best performance for varying dynamics of the workload. Furthermore, the equipartitioning strategy was applied for fairness with resources that remained unused under both dependency-based expand/shrink and backfilling. Our results show that the DBES strategy demonstrates consistently superior performance in comparison to other state of the art scheduling strategies and that it is also the best strategy to be applied together with scheduling unpredictably evolving applications.

Adaptive resource management and scheduling is not only important for supporting malleable and evolving jobs, but also for fault tolerance. With small extensions, the facilities of the proposed batch system can be used for dynamic replacement of nodes during node failures and proactive migration. Also, to improve the gain that can be obtained from malleable jobs, a minimal application feedback can be established and a dependency-based malleable job scheduling with feedback considerations can be employed. In the future, we plan to enrich the batch system with the these features and also to support efficient combined scheduling with moldable jobs.

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.

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 Engg. Springer, 2003.

[3] 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.

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

[5] G. Utrera, J. Corbalan, and J. Labarta, "Implementing Malleability on MPI Jobs," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[6] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela, "Malleable Iterative MPI Applications," *Concurr. Comput. : Pract. Exper.*, vol. 21, 2009.

[7] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*. ACM Press, September 1993.

[8] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Springer, 2011.

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

[10] ETP4HPC. Strategic research agenda. http://www.etp4hpc.eu/strategy/strategic-research-agenda/.

[11] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf, "A batch system with fair scheduling for evolving applications," in *Proc. of the 43rd International Conference on Parallel Processing (ICPP), Minneapolis, MN, USA*, Sep. 2014.

[12] T. Carroll and D. Grosu, "Incentive compatible online scheduling of malleable parallel jobs with individual deadlines," in *39th International Conference on Parallel Processing (ICPP)*, 2010.

[13] H. Sun, Y. Cao, and W.-J. Hsu, "Fair and efficient online adaptive scheduling for multiple sets of parallel applications," in *IEEE 17th International Conference on Parallel and Distributed Systems*, 2011.

[14] G. Utrera, S. Tabik, J. Corbalan, and J. Labarta, "A job scheduling approach for multi-core clusters based on virtual malleability," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[15] A. Gupta, B. Acun, O. Sarood, and L. Kale, "Towards Realizing the Potential of Malleable Jobs," in *21st IEEE International Conference on High Performance Computing*, 2014.

[16] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *16th Symposium on Computer Architecture and High Performance Computing*, Oct 2004.

[17] S. S. Vadhiyar and J. J. Dongarra, "SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems," in *In: Parallel Processing Letters. Volume*, 2002.

[18] J. Buisson, F. André, and J. Pazat, "A framework for dynamic adaptation of parallel components," in *Proc. of the International Conference on Parallel Computing (ParCo) 2005.*

[19] L. V. Kalé, S. Kumar, and J. DeSouza, "A malleable-job system for time-shared parallel machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2002.

[20] G. Mounie, C. Rapine, and D. Trystram, "Efficient approximation algorithms for scheduling malleable tasks," in *Proc. of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1999.

[21] J. Blazewicz, M. Machowiak, G. Mouni, and D. Trystram, "Approximation algorithms for scheduling independent malleable tasks," in *Euro-Par 2001 Parallel Processing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.

[22] 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.

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

[24] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," in *Proc. of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[25] H. Sun, Y. Cao, and W.-J. Hsu, "Efficient adaptive scheduling of multiprocessors with stable parallelism feedback," *IEEE Transactions on Parallel and Distributed Systems*, 2011.

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

[27] 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.

[28] 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, 2001.

[29] F. Gioachin, C. W. Lee, and L. V. Kalé, "Scalable Interaction with Parallel Applications," in *Proceedings of TeraGrid'09*, 2009.

[30] A. T. Wong, L. Oliker, W. T. C. Kramer, T. L. Kaltz, and D. H. Bailey, "ESP: A System Utilization Benchmark," in *Proc. of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[31] A. Bhatele, S. Kumar, C. Mei, J. Phillips, G. Zheng, and L. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *IEEE International Symposium on Parallel and Distributed Processing, 2008.*