Towards Realizing the Potential of Malleable Jobs

Abhishek Gupta, Bilge Acun, Osman Sarood, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{gupta59, acun2, sarood1, kale}@illinois.edu

***Abstract*—Malleable jobs are those which can dynamically shrink or expand the number of processors on which they are executing at runtime in response to an external command. Malleable jobs can significantly improve system utilization and reduce average response time, compared to traditional jobs. To realize these benefits, three components are critical – an adaptive job scheduler, an adaptive resource manager, and an adaptive parallel runtime system. In this paper, we present a novel mechanism for enabling shrink/expand capability in the parallel runtime system using task migration and dynamic load balancing, checkpoint-restart, and Linux shared memory. Our technique performs true shrink/expand eliminating the need of any residual processes, requires little application programmer effort, and is fast. Further, we establish a bidirectional communication channel between the resource manager and the parallel runtime, and present an asynchronous split-phase mechanism for executing adaptive scheduling decisions. Performance results using Charm++ on Stampede supercomputer show the efficacy, scalability, and benefits of our approach. Shrinking from 2k to 1k cores takes 16s while expand from 1k to 2k takes 40s. Also, we demonstrate the utility of our runtime in traditional as well as emerging scenarios, e.g., proactive fault tolerance and clouds.**

I. Introduction

As we move towards exascale era in High Performance Computing, supercomputers will need to operate under power constraints and failing components [1]. In such environment, adaptivity will be crucial to achieve better utilization of system components. One direction to achieve such adaptivity is to enable malleable jobs – which can change the number of processors on which they are executing at runtime in response to an external command. Such jobs can expand when the cluster has low demand, and shrink when there is high demand. Figure 1 illustrates this with an example. Each box represents the current utilization of the compute capacity (say 100 nodes) of a cluster by jobs A, B, and C. Here, a long running job A can be made to shrink or expand to adapt to current demands. In the absence of malleability, job A which is using 60 nodes, can block job C which needs at least 50 nodes, resulting in wastage of 40 nodes. This wastage can be avoided if there are smaller jobs, but this may not always be the case. Malleable jobs are an excellent alternative solution to this problem, and have been shown to potentially improve system utilization by up to 25%, and also reduce mean job response time [2]–[4].

To enable malleable jobs, three components are critical (Figure 2) – (1) *a smart adaptive job scheduler*, which decides when and which jobs to expand or shrink, based on the job queue, current cluster state, and a job scheduling policy, (2) *an adaptive resource manager*, which allocates nodes to jobs (node scheduler) and executes the scheduling decisions by

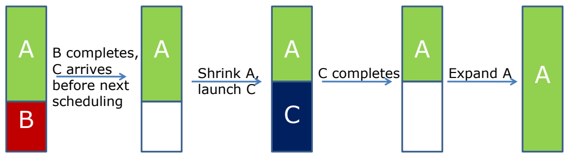


Fig. 1: Example use case

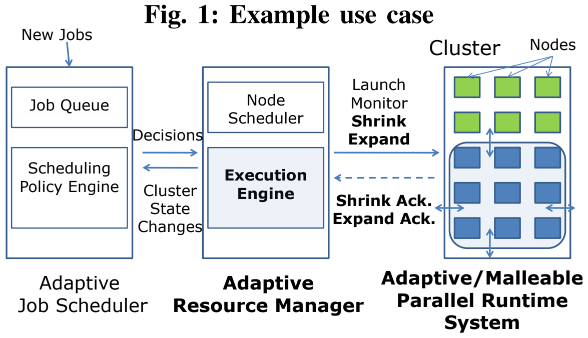


Fig. 2: System overview (focus and contributions in bold)

coordinating between the job scheduler and the cluster and running jobs (execution engine), and (3) *an adaptive parallel runtime system* which provides the dynamic shrink/expand capability. Although the job scheduling strategies for malleable jobs have been extensively researched [2], [3], [5]–[8], there are very few runtime systems which can actually perform shrink or expand on general purpose parallel programs. Existing techniques either perform pseudo shrink/expand by leaving residual process on nodes which are vacated as a result of shrink [2], [4] or require too much application-specific programmer effort for data re-decomposition after resize [5]. Further, the integration of above three components has been little researched. The scheduler needs to communicate to the application its shrink/expand decisions, and the application needs to acknowledge when done.

In this paper, we address the research challenges involved in designing an end-to-end system which can *provide* and *exploit* job shrink/expand capability. To this end, our contributions are:

- Study of the research challenges in designing end-to-end fast, scalable, and efficient shrink/expand capability in parallel runtime system. A key innovation of our approach is to combine task migration, checkpoint-restart, load balancing, and use of Linux shared memory (SHM) (§III).
- Split-phase execution of malleable job scheduling actions, incorporating scheduler-runtime communication (§IV).
- Implementation atop CHARM++ and analysis of malleability and associated benefits using mini-applications up to $2k$ cores on Stampede supercomputer (§ III-D, §VI).
- Exploration of novel use cases of shrink/expand capability, specifically proactive fault tolerance and price-sensitive scaling in cloud spot markets (§VII).

II. Related Work

Feitelson and Rudolph [9] classified parallel jobs into four categories based on – who decides the number of processors a job will be run on, and when it is decided (Table I). In both moldable and malleable jobs, users specify a range of processors a job can be run on, based on factors such as its strong scaling performance and memory limitations. In this paper, our focus is on malleable jobs where the scheduler can dynamically change the resources allocated to a job.

A. Runtimes with Shrink/Expand Capability

Kale et al. [4] demonstrated CHARM++ jobs with the ability to shrink or expand their node footprint by dynamic migration of work/data units (objects) to processors [4]. However, in case of shrink, individual processes (known as *residual processes*) are still left on processors that are removed from the available processor pool. These residual processes carry out low-level processor-based tasks, such as forwarding messages for migrated objects to their new homes and spanning tree-based reductions. The residual processes (one per job per processor) raise severe inter-job interference and security concerns, making this approach impractical for large clusters. Moreover, in this approach, for expand, the job size is limited to the number of nodes where it was initially launched. Hence, for efficient and true resize, one needs to eliminate these residual agents.

Cera et al. [2] demonstrated two techniques to provide malleable MPI applications: (1) dynamic CPUSETs mapping and (2) dynamic MPI, using OAR resource manager. Dynamic CPUSETs technique is specific to multi-core machines. It enables dynamic alteration in the number of cores per node allocated to an application. Their second technique is more general and allows shrinking or growing using MPI process spawning primitives (such as `MPI_Comm_spwan`). However, they do not vacate residual processes in case of shrinking. Moreover, significant application programmer effort is necessary to perform data re-decomposition after resize.

Perhaps the work most similar to ours is the research on dynamic malleability of iterative MPI applications using PCM (Process Checkpoint and Migration) library [5]. That work conceives malleability as split and merge operations supported using PCM calls added to application code. However, since MPI applications are processor-centric, their scheme needs significant application code modification for performing data re-decomposition after resize. We address this problem by leveraging the over-decomposition of data into migratable objects or medium-grained tasks. Our approach requires minimal application-level code changes to support malleability.

An approach which combines over-decomposition and checkpoint-restart uses Adaptive MPI (AMPI) [10]. The main idea is to perform shared file-system based checkpoint-restart and application re-launch. Its main drawback is slowness due to I/O and re-launch costs. Our advancements and contributions over [10] are: a) checkpoint to Linux SHM, which is fast and persistent, b) task/object evacuation, similar to [4], prior to checkpoint enabling fast restart from local SHM instead of shared file-system, and c) fast rebirth and modified re-launch protocol using `exec` avoiding complete application re-launch.

TABLE I: Job Type Taxonomy

Who decides	When it is decided	
	At submission	During execution
User	Rigid	Evolving
System/scheduler	Moldable	Malleable

B. Adaptive Schedulers and Resource Managers

Several studies have demonstrated the benefits of scheduling algorithms which consider malleable jobs, using theoretical analysis [3], [6] and/or simulation using job traces [3], [7]. For instance, Hungershofer demonstrated that simple strategies such as *equipartitioning* can result in significant improvement in response time and utilization using malleable jobs [7]. Utrera et al. [8] demonstrated benefits of a malleable processor allocation technique based on a combination of moldability and folding techniques – Folding by JobType (FJT). Adaptive scheduling policies have also been studied in context of grids, e.g., KOALA multicluster scheduler and DYNACO framework [11], AppLeS project [12], and others [13], [14].

In this paper, we do not intend to research adaptive job scheduling algorithms. Instead, we study the challenges in actual execution of malleable jobs by a resource manager. To this end, we present a runtime system which enables jobs to shrink/expand and a mechanism for interaction between the scheduler and running job. Also, we address the issues in enforcing scheduling decisions in presence of malleable jobs.

III. Shrink/Expand in Parallel Runtime System

To enable malleable jobs, the foremost requirement is a parallel runtime system (RTS) which can render applications malleable, preferably without much programmer effort. In this section, we discuss our approach towards malleability in an RTS. When we say runtime, we mean a parallel RTS. For enhanced understanding, we first define *shrink* and *expand* operations and present the challenges that we considered while designing such RTS.

A. Definitions and Design Goals

Shrink: A parallel application running on nodes of set A is resized to run on nodes of set B where $B \subset A$

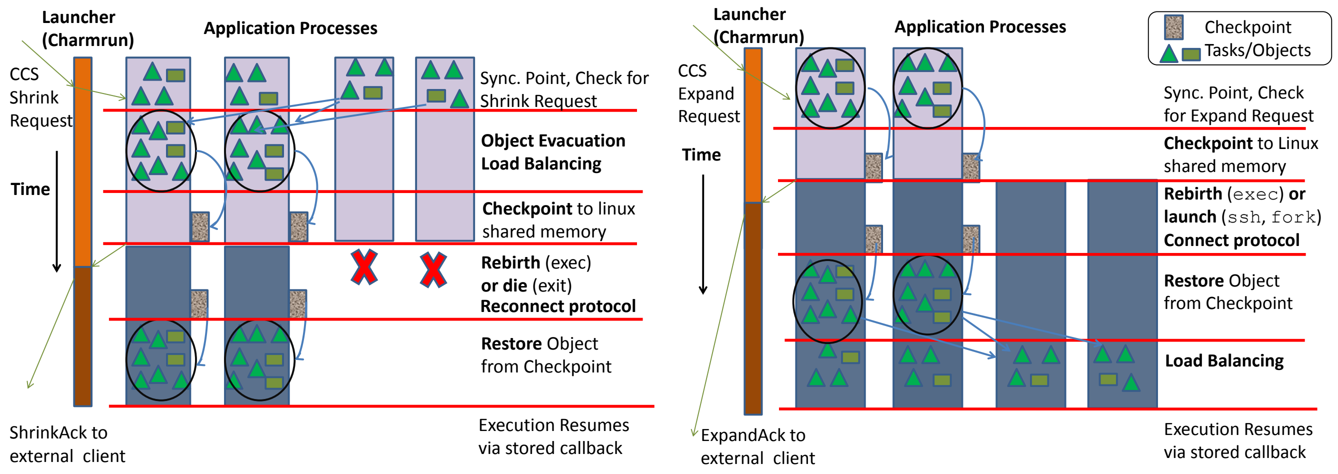
Expand: A parallel application running on nodes of set A is resized to run on nodes of set B, where $B \supset A$

Rescale: *Shrink* or *expand*

An alternative definition is possible, where the subset and super-set relationships for *shrink* and *expand* respectively are not necessary. For example, on *shrink*, a job may be allocated a new set of nodes to replace a subset of old nodes. One of the motivations for such re-allocations is to provide contiguous allocation on resize. In our definitions, such cases can be handled by performing *expand* followed by *shrink*.

While exploring mechanisms to provide *rescale* capability in an RTS, we focused on certain design challenges. Our approach towards a malleable runtime should be:

- *Efficient:* It should ensure that achieved performance after *rescale* is proportional to the compute power.
- *Fast:* The *rescale* time ($T_{rescale}$) should be small to satisfy the needs of its usage scenarios. We expect the granularity of *rescale* events to be few minutes or even more, so $T_{rescale}$ around 1 minute should be permissible.



Launcher (Charmrun)
Application Processes
CCS Shrink Request
Time
Sync. Point, Check for Shrink Request
Object Evacuation Load Balancing
Checkpoint to linux shared memory
Rebirth (exec) or die (exit) Reconnect protocol
Restore Object from Checkpoint
Execution Resumes via stored callback
ShrinkAck to external client
Launcher (Charmrun)
Application Processes
▨ Checkpoint
Tasks/Objects
CCS Expand Request
Time
Sync. Point, Check for Expand Request
Checkpoint to Linux shared memory
Rebirth (exec) or launch (ssh, fork) Connect protocol
Restore Object from Checkpoint
Load Balancing
Execution Resumes via stored callback
ExpandAck to external client

(a) Shrink (b) Expand

Fig. 3: Shrink expand runtime system design

- *Scalable:* The approach should scale well with increasing number of nodes and with increasing problem sizes.
- *Practical:* It should be applicable to most supercomputers, commodity clusters, and possibly even clouds. Also, it should be generic from runtime perspective.
- *Low-effort:* The runtime should ensure that there is little or no application-specific programmer effort required to render a parallel application malleable.

B. Assumptions

Our approach makes two assumption regarding an HPC job:

1. Over-decomposition and task (or object) migration: Over-decomposition refers to decomposing an application into medium-grained work/data units typically larger in number than the number of processors. These units can be mapped (and re-mapped when they are migratable) by the runtime to processors. These features are present in some established [15] and emerging [16] parallel runtimes, and are becoming more relevant for exascale [1]. MPI jobs can rescale without manual repartitioning using our approach with an over-decomposed MPI implementation (multiple ranks per-core) e.g., Adaptive MPI [17] or possibly Fine-Grained MPI (FG-MPI) [18]. These have negligible overhead, e.g., user-level threads [17].

2. Synchronization boundaries: Our approach requires application-specified synchronization points such as iteration boundaries. A large class of scientific applications are inherently iterative, making this a reasonable assumption.

C. Approach

Figure 3 describes our technique. The parallel application acts as a server which can listen to incoming *rescale* requests from external sources, such as a job scheduler. These requests can be received at any time and are recorded in the system. However, they are handled at the next synchronization point. We first discuss our approach for *shrink* (Figure 3a).

Shrink: If the request is of type *shrink*, the request needs to specify which processors out of the original set, does the application need to relinquish. One mechanism is to specify a bit-vector (with 1=available, 0=unavailable) of size P_{old}, where P_{old} is the number of processors before *shrink*.

Task Evacuation: At the synchronization point, a task evacuation module or a special load balancer is invoked to migrate the tasks away from the processors marked unavailable while balancing load over the remaining set of processors.

Removal of Residual Processes: After, the tasks/objects are evacuated, the next step is to eliminate the residual processes while ensuring that the application continues to seamlessly run after *rescale* is complete. This step is non-trivial since the application processes are closely tied. To ensure correctness, performance, and reliability after *rescale*, all the runtime system data structures which depend on P_{old} need to be modified. Examples of these include spanning trees, task location managers such as hash-tables, and any processor-level instrumentation and monitoring modules.

To avoid the need of such complex modification of runtime structures, we follow a three-step process: 1) checkpoint local application state before *rescale* on each available processor, nothing is checkpointed on processors which will be unavailable after *rescale*, 2) enable application rebirth, and 3) restore application state from checkpoint after rebirth. The naive approach would be to use disk-based checkpoint-restart and application re-launch, similar to [10]. However, interaction with disk can severely degrade *rescale* performance and prevent us from meeting the second design goal (Section III-A). Ideally, for fast *rescale*, in-memory checkpoint could be used. However, since we terminate and re-launch processes to achieve clean restart of application on the continuing set of processors, any state stored in process memory will be lost.

Checkpoint-restart using Linux Shared Memory: Our novel solution to this challenge of performing fast, stateful, and scalable process rebirth is to use *OS shared memory (SHM)*. SHM is a method of interprocess communication (IPC) where multiple processes share a single chunk of memory to communicate. Since SHM is not tied to a single process, it has the advantage of being persistent across process restart, and also being fast (since it resides in memory).

Hence, our approach is to 1) checkpoint the state before *rescale* to Linux SHM, 2) perform application process *rebirth* or *death* depending on whether the processor is marked available or not, 3) execute a reconnect protocol – a modified version of the application start-up protocol, and 4) restore the application state from the checkpoints. The essence of step 2 is to replace the current process image by a a clean application image using Linux `exec` system call for the available processors whereas *death* for the unavailable processors using Linux `exit` system call. A reconnect protocol, which is a modified version of application launch protocol, is necessary to establish appropriate communication channels among the reborn processes so that they can communicate after *rescale* is complete. Also note that SHM is allocated by the runtime only at *rescale* event, and freed after step 4. Scheduler-runtime coordination (§ IV) ensures proper management of SHM.

Summary: Our overall solution is to combine task migration, load balancing, SHM, and checkpoint-restart to achieve *shrink*. The task evacuation phase prior to checkpoint-restart ensures that the dying processes are stateless since all application state is already migrated to the continuing set of processors. Checkpoint-restart enables a clean state after *rescale* event, whereas the use of SHM allows the approach to be fast.

After the state restoration from checkpoints, RTS has completed *shrink* and an acknowledgment can be sent to the external source from which the request originated. Also, control needs to be transferred from the RTS to the application at a pre-registered application resumption point. An application can perform such registration at initialization.

***Expand*:** For *expand*, the basic idea is similar to the handling of *shrink* request. However, there are some important differences: 1) The *expand* request needs to specify the list of newly available processors. 2) There is no need of object evacuation before checkpoint, instead load balancing is required after restoring from checkpoint. This post-restore load balancing distributes tasks to newly available processors to effectively utilize the total compute power. 3) No processes need to be killed, instead new processes need to be launched. These and the reborn processes then participate in the reconnect protocol to enable inter-process communication after restart completes.

We discussed our approach to *rescale* in the context of malleable jobs, where the decision to *rescale* is external e.g., job scheduler driven. However, our techniques will be equally useful in other contexts, such as a) *evolving* jobs where the decision is application-intrinsic, and b) other non-traditional use cases (§VII). Also, our approach makes reasonable assumption about the parallel job (§III-B), and needs only SHM from the OS. According to the November 2013 top500 list, 96.4% of top supercomputers use Linux family OS [19]. Thus, our approach meets the design goal of being *practical*.

D. Implementation atop CHARM++

We implemented our techniques on the top of CHARM++ RTS [15]. In CHARM++, the objects *(chares)* form the basic unit of computation and can be redistributed dynamically among processors by the sophisticated load balancing framework [20]. These capabilities fulfill the needs of our approach. AMPI is a framework running atop CHARM++ and provides dynamic load balancing capabilities to MPI applications using migratable user-level threads [17]. Using AMPI, MPI applications can utilize our approach and implementation.

In CHARM++ programs, the application developer can specify synchronization points using `AtSync()` calls, which act as hints to the runtime to perform adaptive control such as dynamic load balancing. In our implementation, we service *rescale* requests by invoking a custom load balancer, which is aware of the bit-vector information about unavailable processors. We developed our load balancer on top of CHARM++ load balancing framework, which instruments the objects execution times and process wall clock time from previous `AtSync()` point. These instrumented times of previous iterations are used as estimates of loads of future iterations, which is a proven estimation technique for iterative scientific applications [20]. We incorporated two existing load balancing strategies – *RefineLB* and *GreedyLB* in our implementation. RefineLB performs periodic load refinement by moving objects from overloaded to under-loaded processors, whereas GreedyLB uses a greedy strategy which iteratively assigns heaviest compute object to most under-loaded processor.

For checkpoint-restart, we checkpoint the current state of chares, collection of chares (chare arrays), and CHARM++ groups. In CHARM++, groups are processor-level agents which can be used to perform system tasks such as load balancing. For checkpoint-restart, we use CHARM++'s pack-unpack (*pup*) serialization mechanism and Linux SHM (`shm`) calls. We perform the reconnect protocol using `Charmrun` – the start-up manager for CHARM++ applications. Our launch and reconnect protocol is a slightly modified version of the node-aware start-up discussed by Gupta et al. [21]. In our modified protocol, `Charmrun` perform `ssh` to launch the executable only on the newly added nodes rather than all the nodes. The processes on rest of the nodes use `exec` system call. When the processes start, they connect back to `Charmrun`. `Charmrun` also facilitates the exchange of communication information, such as data-port for Ethernet, necessary to enable inter-processes communication after restart is complete.

After *rescale* is complete, control is transferred to the application using CHARM++'s callback mechanism.

IV. ADAPTIVITY IN RESOURCE MANAGER

In the previous section, we provided a novel approach to enable an RTS to *shrink* or *expand* parallel programs. However, to realize the benefits of malleable jobs in a shared cluster environment, the job schedulers and resource managers also need to be made adaptive (see Figure 2). Researchers have shown significant benefits of adaptive job scheduling algorithms while simulating malleable jobs [3], [6], [7]. However, the research challenges which arise in the 'management' of malleable jobs and 'execution' of job scheduling decisions in presence of malleable jobs have mostly remained open. Prior solutions include stalling while the job reconfigures, or executing *shrink*, *expand*, and launch simultaneously, in the presence of residual processes. The primary research issues that we address here are *how* and *when* to (a) communicate the scheduling decisions to running application and (b) detect the success or failure of those actions. In next subsections, we present a general framework and protocol for resource management to address these questions.

Algorithm 1 *Shrink-Expand* Split-phase Execution

```
 1: while true do
 2:     jobDecisions = ScheduleJobs(jobQueue, clusterFreeNodes,
        runningJobs, $T_{rescale}$, optionalArgs)
 3:     nodeDecisions = ScheduleNodes(jobDecisions, clusterNodeState,
        optionalArgs)
 4:     UpdateSchedNodeMap(nodeDecisions)
 5:     dependentActions = ExecuteDecisions(jobDecisions)
 6:     repeat
 7:         ProcessBufferedShrinkAcks()
 8:         ExecuteDependentDecisions(dependentActions)
 9:     until (jobQueue != empty or a job finished)
10: end while

11: procedure UpdateSchedNodeMap(decision)
    Update scheduler's view of node to job mapping

12: procedure ExecuteDecisions(jobDecision)
13: for decision in jobDecisions do
14:     if decision.type == shrink then
15:         NotifyJobToShrink(decision)
16:     else if AreAllNodesFree(decision.jobid) then
17:         LaunchExpandJob(decision)
18:         UpdateActualNodeMap(decision)
19:     else
20:         dependentActions.Add(decision)
21:     end if
22: end for
23: return dependentActions

24: procedure AreAllNodesFree(jobid) Check if all the nodes of a job are
    marked free in actual node to job map

25: procedure UpdateActualNodeMap(decision)
    Update actual node to job mapping

26: procedure ProcessBufferedShrinkAck()
    Update actual node to job mapping on shrink completion

27: procedure LaunchExpandJob(decision)
28: if decision.type == launch then
29:     LaunchJob(decision)
30: else if decision.type == expand then
31:     NotifyJobToExpand(decision)
32: end if

33: procedure ExecuteDependentDecisions(dependentActions)
34: for decision in dependentActions do
35:     if AreAllNodesFree(decision.jobid) then
36:         LaunchExpandJob(decision)
37:         UpdateActualNodeMap(decision)
38:         dependentActions.Remove(decision)
39:     end if
40: end for
```

While performing the integration of the job scheduler, resource manager, and malleable parallel RTS, we made some important design decisions: (1) the mechanism used by resource manager for executing *rescale* decisions should be orthogonal to job scheduling algorithm and (2) the interaction between resource manager, application, and the scheduling algorithm should be orthogonal to parallel runtime's *rescale* mechanism. There is one exception, where information communication among the three components of our system can help scheduler make better decisions. This information is the expected time taken by an application to perform *rescale* ($T_{rescale}$). The scheduling algorithm can then decide the gap between any two *rescale* events for same job ($T_{gap\_rescale}$), such that $T_{gap\_rescale} >> T_{rescale}$.

A. Resource Manager – Parallel RTS Communication Channel

To answer the *how* question of communicating between running application and the resource manager, we establish a control and feedback channel between those two components. We leverage the Converse Client-Server interface (CCS), provided by CHARM++ RTS. CHARM++ application can act as a CCS server to which a CCS client can connect and send requests via a TCP/IP socket. Upon receiving a request, the CCS server runtime invokes appropriate pre-registered handler function, thus injecting a message into a running parallel computation. The main effort necessary on the runtime side was to implement handler functions to service incoming requests for the desired functionality – *shrink* or *expand*.

To demonstrate a working system, we implemented a simple job scheduler and resource manager in Python. To inform an application of a *rescale* decision, the resource manager starts a CCS client, connects to the CCS server using the host name and server port corresponding to that job, and send a message through that connection. Next, it listens for a response back from the application. The communication from application to resource manager happens through the same channel. To acknowledge that it has resized itself in response to the notification, the application sends back a completion notification after performing *rescale*.

We used CCS since it is inbuilt in CHARM++, and shown to be a *secure* and *scalable* method for interacting with the parallel application [22]. The communication mechanism in our resource manager is a pluggable module and it is easy to use another protocol, such as RPC, for this communication. However, the protocol needs to be secure and scalable.

B. Split-phase Execution of Scheduling Decisions

For traditional rigid jobs, the only scheduling decision that needs to be implemented by the resource manager is to start a new job. At a scheduling event, the job scheduler may decide to launch k jobs (Algorithm 1, line 2). After the node scheduler allocates them the corresponding number of nodes and updates its database (line 3–4), the resource manager can launch those k jobs simultaneously (`ExecuteDecisions` line 5).

In contrast, a resource manager for malleable jobs needs to handle three actions - launch, *shrink*, and *expand*. Having developed a mechanism for communicating between the running application and the resource manager, the next challenge is to decide *when* to execute the scheduling decisions. The challenge is that the k decisions provided by the job scheduler may have inter-dependencies. For example, considering the example of Figure 1, when job B completes, the scheduler decides to *shrink* job A from 60 to 50 nodes and launch job C on remaining 50 nodes of say 100 node cluster. These two decisions cannot be executed simultaneously since the nodes of C include those which are currently used by A and will be available only when A has finished shrinking. Similarly *expand* decisions on one job may also depend on *shrink* decisions of another. To tackle this problem, we perform split-phase execution of the scheduling decisions. A naive solution would be to first issue all *shrink* requests, wait till completion acknowledgements arrive from all of them, and then perform launch and *expand* actions. However, this results in unnecessary blocking and prevents other jobs from getting launched or scheduled. Hence, we optimize our split-phase approach to asynchronously send all the *shrink* requests (line 14–15 in procedure `ExecuteDecisions`). Next, we launch or *expand* jobs with no dependencies (line 16–17), and record jobs for which launch or *expand* needs to be delayed (line 20). Later, while the process is waiting for the next scheduling trigger, such as a new job arrival or completion of a running job, it periodically checks and processes any buffered *shrink* completion acknowledgements and updates appropriate data structure to reflect the actual node to job mapping (line 7 `ProcessBufferedShrinkAck`). After `ProcessBufferedShrinkAck` the decisions for which execution was postponed due to dependencies are checked to see if they can be executed now based on the current state (line 8, procedure `ExecuteDependentDecisions`).

In our implementation, to track the dependencies between jobs, we kept two data structures – $SchedNodeToApp$ which reflect the scheduler's view of nodes to jobs mapping, and $ActualNodeToApp$, which tracks the actual state. The scheduler's view gets updated at scheduling event (line 4) or when a job completes (line 9) whereas the actual state is updated on *shrink* completion acknowledgement(line 7, 26), new job launch (line 18, 37), and issue of *expand* request (line 18, 37).

V. Evaluation Methodology

To analyze the performance and scalability of our approach to malleability, and to evaluate it against the design goals, we used following benchmarks and applications.

- **Stencil2D** is a 5-point stencil kernel which iteratively averages values in a 2-D grid using Jacobi relaxation. This benchmark is a widely used kernel in HPC applications.
- **Wave2D** is a 2-D mesh based mini-application for simulating wave propagation. It is computation intensive and uses discretized finite differencing method.
- **LeanMD** is a Molecular Dynamics (MD) mini-application which performs simplified version of the force calculations of NAMD [23], a widely used MD code. LeanMD uses two CHARM++ object arrays – *cells*, which are collection of atoms in 3-D space, and *computes*, which perform force calculation on atoms.
- **Lulesh** is the CHARM++ implementation of LULESH hydrodynamics mini-application [24]. It simulates explicit shock hydrodynamics in 3-D space using Lagrangian formulation with leap frog time integration.

We conducted experiments on Stampede supercomputer. Stampede has Dell PowerEdge server nodes, which have 2 Intel Xeon E5 processors each, accounting for 16 cores per node. Each node has 32GB memory, and allows up to 16GB memory to be used for Linux SHM. We used interactive job allocation on Stampede, which allowed us to send CCS requests to running applications from external client and demonstrate an end-to-end malleable jobs system. However, the allocation in interactive mode is limited to 256 processors on Stampede. Hence, for larger scale runs (512–2k cores) we modified the application to initiate a *rescale* request itself.

For most experiment, we used CHARM++ `net-linux-x86_64-ibverbs` build that uses low-level Infiniband Verbs communication library, and one CHARM++ process (multiple objects) per core. We used `gcc` compiler, optimization level $-O3$, and show results with *RefineLB* load balancer unless specified otherwise.

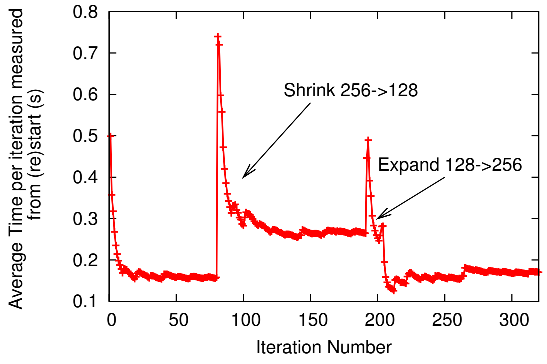


Fig. 4: Adapting load distribution on rescale (LeanMD)

VI. Results

We modified the applications presented in Section V to make them resizeable. Next, we analyze the effectiveness, performance, overhead, and benefits of our system.

A. Adapting Load Distribution on Rescale

Figure 4 illustrates LeanMD's response to *rescale* requests on 256 processors on Stampede. We plot the average time per iteration, measured from a *rescale* event completion, with respect to the iteration number. When the *shrink* request is handled at iteration 80, the number of processors are reduced to half, that is 128. The average iteration time doubles as expected since the average load on each processors doubles. The peaks in the graph reflect the time taken by load balancing, which is performed every 20 iterations. On *expand* (iteration 200), the number of processors doubles. For *expand*, the load redistribution happens at the next load balancing step after restart, which results in drastic reduction in the iteration time at step 220. Hence, our system is effective in adapting to the changes in compute power caused by *rescale* events, meeting the first design goal (efficient as stated in Section III-A).

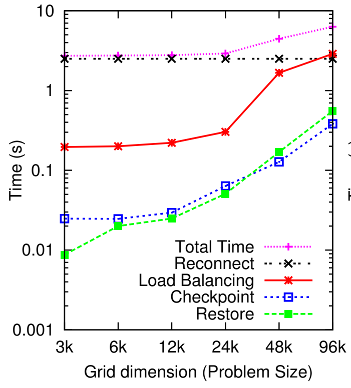
B. Shrink Expand Overhead

To quantify the overhead of application reconfiguration on a *rescale* event, we measured the breakup of time spent in different phases. We shrank various applications from 256 to 128 processors and expanded back to 256 processors. Here, we used the following configurations: Stencil2D: $12k \times 12k$ grid with block (or object) size of $2k \times 2k$, LeanMD: $4 \times 4 \times 4$ cell with 2432 computes, Lulesh: $512 \times 256 \times 320$ grid, and Wave2D: $64k \times 48k$ data on a 32×24 object grid.

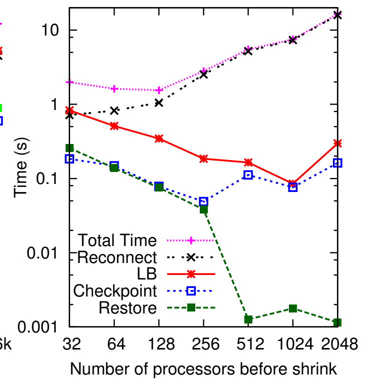
***Shrink* vs. *Expand*:** Table II shows the the time taken in different stages of our scheme. The total time required is 2.6–4s for *shrink* and 7.1–8.7s for *expand* for different applications (except last row). The reason for the difference in the time between *shrink* and *expand* is evident from the breakup, which shows that reconnect time is the dominating factor. Reconnect phase includes a) the time taken by the launcher to ssh and launch new processes, which is done only in case of *expand*, and b) the time taken by the connection establishment phase. For *expand*, launcher needs to start 128 new processes. Also, the connection establishment happens for 256 processes compared to 128 after *shrink*. Hence, the reconnect time is more for *expand* compared to *shrink*.

TABLE II: *Shrink expand* time breakup (in seconds) for different applications

Application	*Shrink*: 256 → 128					*Expand*: 128 → 256				
	LB	Checkpoint	Reconnect	Restore	Total	LB (Post)	Checkpoint	Reconnect	Restore	Total
LeanMD	0.515	0.039	2.102	0.003	2.658	0.056	0.016	7.079	0.003	7.154
Lulesh	0.560	0.531	2.533	0.432	4.056	0.458	0.520	7.083	0.436	8.496
Wave2D	1.219	0.243	2.542	0.336	4.340	1.046	0.244	7.067	0.337	8.695
Stencil2D	0.299	0.050	2.501	0.054	2.904	0.133	0.036	7.076	0.038	7.283
Stencil2D_Net	5.86	0.057	2.584	0.056	8.556	4.096	0.042	9.495	0.044	13.678



(a) Effect of problem size, shrinking from 256 to 128 cores



(b) Effect of strong scaling with $24k \times 24k$ problem size, shrinking to half

Fig. 5: Analysis of *rescale* performance using Stencil2D.

The overhead breakup enabled us to optimize the *rescale* time from around $9s$ to $2.5s$ for *shrink* and from $19s$ to $7s$ for *expand*. Knowing that reconnect phase is the bottleneck, we optimized it using startup techniques such as batching and node-awareness [21]. It is possible to further improve this using variations of advance startup mechanism such as multi-level startup [21]. Rest of the phases – load balancing (LB), checkpoint, and restore are very fast, for both *shrink* and *expand*. Considering that *rescale* events are expected to be infrequent – every tens or more minutes, our approach has very low overhead and meets our second design goal (fast).

Effect of network: We also evaluate the performance using Ethernet as the network option on Stampede. The motivation is to showcase the applicability of our scheme on commodity clusters and clouds since Ethernet is the network commonly available there. The last row in Table II shows that even with Ethernet, our scheme performs *rescale* in a reasonable time. Comparing the results of Infiniband and Ethernet (last two rows in Table II), it is clear that primarily LB, which involves data migrations, and reconnect phases are the ones which suffer due to worse network. Checkpointing and restore phases operate on local data and communicate only for synchronizations, hence their performance is not much affected.

C. Scalability Analysis using Stencil2D

Having shown that our approach is efficient and fast, we next analyze the scalability of our technique with respect to problem size and increasing node counts.

1) Effect of Problem Size: Figure 5a shows the effect of changing the problem size of Stencil2D, shrinking from 256 to 128 processors. As the problem size grows, load balancing, checkpoint, and restore times increase. This can be attributed to increased data per process. The checkpoint size is around 10MB per process for 12k size and 640MB per process for 96k grid dimension. As the grid dimension doubles, checkpoint size increases by 4X, resulting in slowdown of checkpoint and restore phases. It is evident from Figure 5a that our SHM based approach works well since even for 640MB data, only 0.5s is spent in checkpointing. The load balancing time also increases with problem size since more data needs to be migrated. The reconnect time remains constant since it is independent of the problem size. For very large memory applications, where the data cannot fit in SHM, our approach can be extended to seamlessly switch to disk-based checkpointing.

2) Effect of Strong Scaling : Next, we analyze the scalability with respect to increasing number of processors, but constant problem size (24k for Stencil2D). Here, we show results till 2k cores – external program initiated *rescale* (interactive mode) till 256 cores and application initiated *rescale* for 512–2k cores. The basic approach has no impediments to running on even larger scale but we were restricted by allocation limits.

Figure 5b shows that with increasing scale, the time for reconnect phase slowly increases and dominates the total *rescale* time at large scale. The checkpoint, restore, and load balancing phase scale very well till a particular point – 256 for checkpoint and 1k for load balancing. With increasing processor count, the per-process memory footprint proportionally decreases. This results in reduced checkpoint size and less communication per processor. However, as we scale further, the barrier synchronizations present in these phases become notable, resulting in increase in time. Overall though, the time is still small (16s for $2k \rightarrow 1k$ shrink). The corresponding time for expand ($1k \rightarrow 2k$) is 40s. Figures 5 shows that our approach *scales* reasonably well with respect to both, problem size and core counts, hence meeting our third design goal.

TABLE III: Application-specific development effort

Application	Original SLOC	Modified SLOC
Stencil2D	207	31
LeanMD	703	37
Lulesh	4066	15
Wave2D	363	37

D. Programmer Effort

To quantify the programmer effort needed to make applications malleable using our runtime, we measured the original and the modified source lines of code (SLOC) for our benchmarks and applications, using `sloccount`. Table III shows that we needed to modify very few SLOC, meeting our last design goal (low-effort). For Lulesh, which is the largest of the mini-applications, we needed to modify only 15 SLOC, which was very little effort (<0.4% of original SLOC). The primary modifications required were to register the resume callback with the runtime and make the `mainChare` (main or entry point object in CHARM++) as a migratable entity by providing its migration constructor and pack unpack routine.

E. Case Study with Adaptive Scheduler

To demonstrate that our approach towards integrating the resource manager and parallel runtime works in practice, we

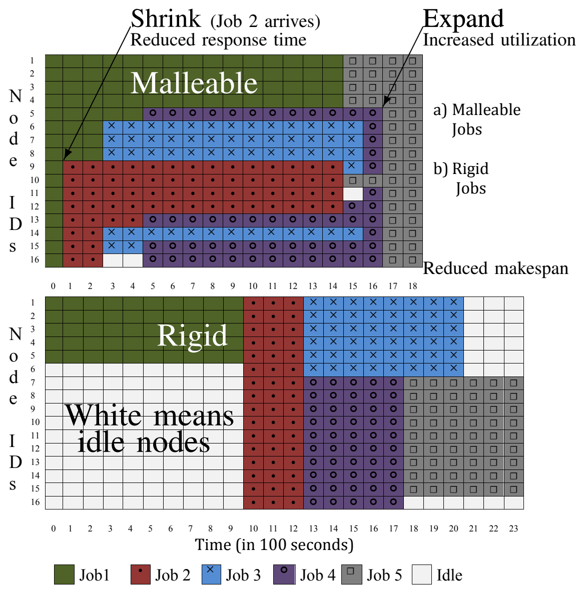


Fig. 6: Nodes allocated over time

conducted a case study with an adaptive job scheduling algorithm. We implemented a variant of dynamic equipartitioning strategy which has previously been shown to have significant performance gains [4], [25]. The strategy first assign each job its minimum required nodes on a first come first serve basis. After that, if each job received its minimum needs and more nodes remain available, they are equally distributed to the scheduled jobs. We modified this policy to consider $T_{gap\_rescale}$, that is the gap between two scheduling actions (launch, *shrink*, or *expand*) on same job. Only those jobs are considered for scheduling for which at least $T_{gap\_rescale}$ seconds have elapsed since they were launched or since they were last shrunk or expanded.

We used this job scheduler in conjunction with our resource manager and the malleable runtime. For the purpose of this case study, we consider the interactive mode allocation on Stampede as our small cluster (16 nodes, 256 cores). Five jobs were submitted to the scheduler with arrival times of 0, 1, 3, 7, and 7 minutes from the start time respectively. For simplicity all the five jobs run same application (Stencil2D with 10000 iterations each). The range for all the applications to *shrink* and *expand* is from 4 to 16 nodes, with 16 cores per node. $T_{gap\_rescale}$ was set to 40s for this experiment. Figure 6a (top) shows the nodes to job mapping over time, hence depicting overall cluster utilization and showing how these jobs are *rescaled*. For example, when job 2 arrives, job 1 is shrunk from 16 to 8 nodes to give the rest to job 2. This is reflected by the change in the nodes to job mapping at time=100s. Similarly, towards the end, when job 4 finishes, job 5 is expanded from 4 to 16 nodes at time=1700s.

Figure 6b shows the allocation of nodes when the same jobs are run but they are rigid. Here, we used a First Come First Serve policy. Also, we used a random number generator over the range [4-16] to choose the number of nodes needed by a job. Figure 6 illustrates the improvement in system utilization and total completion time using malleable vs. rigid jobs.

Table IV compares the achieved performance with different job types and different values (40s, 100s, and 500s) of $T_{gap\_rescale}$ for malleable jobs. To emulate moldable jobs,

TABLE IV: Comparison of different scheduling policies

Scheduling Type	Total Time (s)	Mean Response	Mean Completion	Util--zation
Malleable40	2751	201	1767	97%
Malleable100	2672	142	1699	98%
Malleable500	2844	287	1454	97%
Moldable	3792	289	1685	62%
Rigid	3816	928	1817	70%

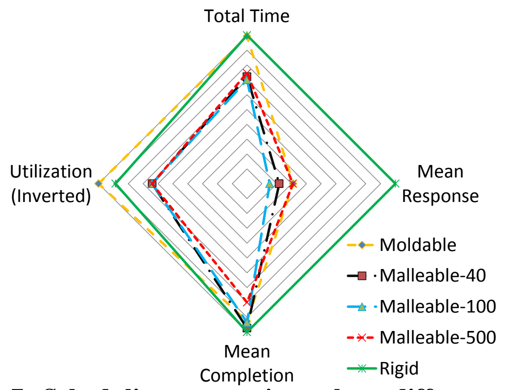


Fig. 7: Scheduling comparison along different metrics

we set a very large value of $T_{gap\_rescale}$, which eliminates any *rescale* events. The results for rigid case are the average of three runs with different random assignments for number of nodes to jobs. The data of Table IV is normalized and visualized using a spider chart (Figure 7). Here, the four dimensions represents our comparison metrics, with smaller being better. For utilization, we plotted the inverted values to get a consistent visualization. In Figure 7, the solid (green) quadrilateral which corresponds to rigid jobs is the worst since it perform poorly on all the dimensions. Figure 7 also shows that for this case study, most benefits of malleability are obtained in terms of mean response time, followed by utilization, total completion time, and mean completion time. Moreover, $T_{gap\_rescale}$ can have significant impact on achieved benefits. If $T_{gap\_rescale}$ is very small, there can be very frequent *rescale* events, leading to high performance overhead, which can increase mean completion time. If $T_{gap\_rescale}$ is very high, the system will not benefit much as there will be very few *rescale* events. For our case study, all the three values $T_{gap\_rescale}$ yielded benefits but the optimal value depends on the metric of interest. This can also be seen in Figure 7 by observing the malleable-40 and malleable-100 shapes along mean response and mean completion time dimensions.

In practice, we expect a good value of $T_{gap\_rescale}$ to be few tens of minutes. In this work, we ran short-duration jobs since our primary intention was to demonstrate the working system on a real supercomputer rather than quantifying the exact benefits of malleable job schedulers. Moreover, we were constrained by the the time limit on an interactive mode allocation on Stampede. The exact benefits of malleability depend on various factors, such as job arrival rate, runtimes, and the scheduling algorithm [2]–[5], [8].

VII. Non-traditional Use cases

In previous sections, we designed a malleable parallel runtime and demonstrated its integration and utility in conjunction with an adaptive resource manager and a job scheduler. The

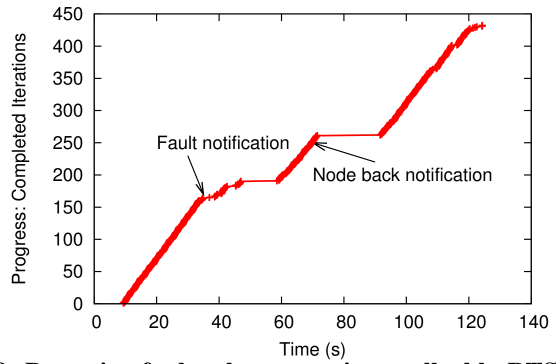


Fig. 8: Proactive fault tolerance using malleable RTS and resource manager to application communication channel

ability of a parallel runtime to *rescale* can be applied to other contexts. Here, we show two such emerging use cases.

A. Reliability: Proactive Fault Tolerance

High performance parallel systems with millions of cores are currently being used and even bigger systems are being planned as we move towards the exascale era. One of the biggest challenges for operating under such massive scale is to achieve fault-tolerant application execution since failures become more and more frequent as the number of system components increase [1]. The traditional solution to deal with failures is to *react* to failures by using mechanisms, such as checkpoint-restart. A less explored approach is to predict failures, and *proactively* vacate the processor where fault is imminent [26]. Failure prediction can be done using hardware devices supporting early indication of failures, sensors, monitoring core temperatures, and other techniques. Inspired by the work of Chakravorty et al. [26], we demonstrate how we can leverage the ability of our runtime system to *shrink* and *expand*, and the bi-directional communication channel between the resource manager and the parallel runtime to enable proactive fault-tolerance.

Once the cluster resource manager predicts that a failure is imminent on a node, it can inform the application by sending a CCS request containing information regarding the failing node. Figure 8 demonstrates how an application (here LeanMD), initially running on 16 nodes (256 processors) on Stampede, reacts to the information communicated by the resource manager. In Figure 8, first few seconds are taken by the job to start-up, hence no application progress is made during that time. At the next synchronization point after receiving the fault notification, the application re-configures itself using *shrink* and continues running on remaining 15 nodes. Once, the node is up again, the application can be informed. On this notification, the parallel runtime expands the job to use 16 nodes again.

Our *rescale* mechanism provides us with rich proactive fault tolerance capability. We can tolerate failures at the level of a node, which could translate into k application processes (e.g. $k = 16$ on Stampede) rather than a single application process. Most current fault-tolerance mechanism tolerate a single process failure. Furthermore, most current fault-tolerance mechanisms make an inherent assumption that either the failing node will be available instantaneously after failure or not at all, or a spare node will be available to replace the failing node. However, they do not allow the possibility of reusing a node which comes back into operation sometime after it failed, e.g., it may just need a restart. Our scheme allows to reuse that node at a later time using *expand*. Finally, proactive fault-tolerance gives an additional advantage by eliminating any execution rollback, such as restart from previous checkpoint in traditional fault tolerance schemes, when the failure actually occurs.

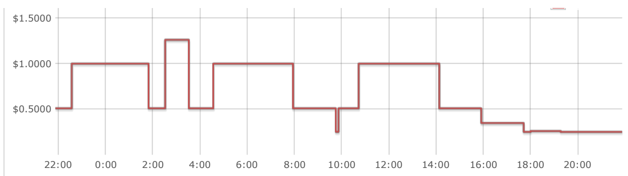


Fig. 9: Amazon EC2 spot price variation for cc2.8xlarge instance (zone us-west2c) on Jan 7, 2014

B. Price-sensitive Rescale in Cloud Spot Markets

Another emerging direction for HPC, especially for small and medium-scale users who have limited or no access to supercomputers, is HPC in cloud [27]. For such users, *renting* rather than *owning* a cluster provides the advantages of pay-as-you-go pricing and elasticity. Amazon EC2 [28] has emerged as the leader in providing such Infrastructure-as-a-Service (IaaS). For HPC, Amazon offers the Cluster Compute instance (CC) [29], of three kinds – reserved, on-demand, and spot. Reserved instances require long reservations at a lower price and are only suitable when a user has long-term static demands. On-demand instances allow the user more flexibility but at a higher price. Spot instances offer the unused cloud capacity at a dynamically varying price, typically very small compared to on-demand instance price. Spot instances work on a bidding based model. A users places her bid for compute power. If and when her bid exceeds the current spot price, instances are allocated to the user. When the spot price exceeds the bid, the instances are terminated without notice. The user is charged with the spot price at the start of each instance-hour. The spot price changes periodically based on supply and demand. Figure 9 shows the spot pricing variation within a day (Jan 7, 2014) of Amazon cc2.8.xlarge instance (zone us-west-2c). This data was obtained from the pricing history available from Amazon EC2 management console.

The malleability support in an HPC runtime can be used to exploit this dynamic spot pricing to achieve cost benefits. Our main idea is to 1) keep a certain minimum number of instances needed for running a job in the on-demand instance pool (static set) and 2) perform *price-sensitive rescale* over the spot instance pool to add more compute power (dynamic set). By price-sensitive *rescale* we mean performing *expand* when the spot price falls below a threshold and performing *shrink* when it exceeds the threshold (not the bid price, one would still place high bid to avoid abrupt termination). By specifying the same availability zone and placement group when requesting the static on-demand and the dynamic spot instances, it can be possible to get them in the same physical cluster [29]. If that does not happen, one can construct the static set also from the spot instance by placing a high bid for that set.

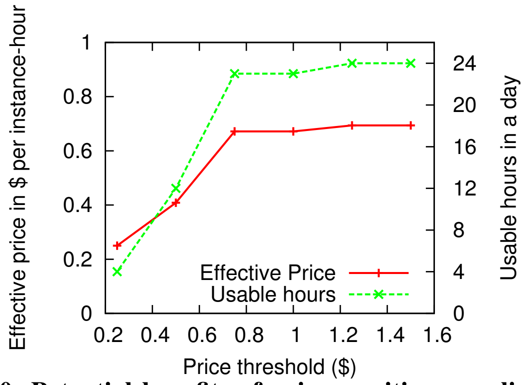


Fig. 10: Potential benefits of price-sensitive rescaling over Amazon EC2 spot instances, and the trade-off between effective price (left y-axis) and usable hours (right y-axis)

Running without *rescale* capability necessitates setting the bid at a very high price and paying whatever the spot price is at the start of the instance-hour. Using the data of Figure 9, that would entail setting the bid price greater than \$1.25, which results in a cost of \$16.65 over a period of 24 hours for a spot-instance. This results in an effective price of \$0.69 per instance-hour. In contrast, the combination of malleable parallel runtime and price-sensitive rescaling enables a user to choose the pricing point below which she wants to operate. As an example, setting a price threshold of \$0.5 results in operating costs of \$4.9 for 12 compute hours, resulting in average price of \$0.41 per instance-hour. Overall, that results in around 40% better effective price for the same instances compared to the default usage. However, the instances will be used only for the hours where the price at the start of the hour is less than \$0.5. Hence, there is a trade-off between the effective price and the usable hours as reflected in Figure 10, which shows that with a lower effective price attained using *rescale* , the usable compute hours are reduced. In most cases, this problem can be circumvented by either running more instances at this lower price or operating for longer periods.

VIII. Conclusions and Future Work

We presented a novel technique to enable malleability in a parallel runtime system using task migration, load-balancing, checkpoint-restart, and Linux shared memory. We implemented this approach using CHARM++ runtime and performed resize on one benchmark and three mini-applications. Through experimental evaluation and analysis on Stampede up to 2048 cores, we demonstrated that our approach is fast, scalable, practical, and effective. In addition, we integrated our malleable runtime system with a resource manager and demonstrated split-phase execution of job scheduling decisions through a bi-directional communication channel between application and the resource manager. Although our focus was on scheduler-triggered *shrink* or *expand*, the techniques developed here are also useful for evolving jobs and other emerging use cases such as proactive fault tolerance and HPC in cloud.

Future research directions include finding solutions for other practical issues in realizing malleable jobs, such as a charging model for HPC system usage by malleable jobs, user incentives for malleability, and scheduling issues including node topology-awareness and fairness.

References

[1] D. Brown et al., "Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale." U.S. DOE PNNL 20168, Washington, DC, Tech. Rep., 2011.

[2] 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," ser. ICDCN'10.

[3] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling," in *Job Sched. Strategies for Parallel Processing*, London, UK, 1997, pp. 1–34.

[4] L. V. Kalé, S. Kumar, and J. DeSouza, "A Malleable-Job System for Timeshared Parallel Machines," in *CCGrid 2002*.

[5] K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "Dynamic Malleability in Iterative MPI Applications," in *IEEE CCGrid 2007*.

[6] R. A. Dutton and W. Mao, "Online scheduling of malleable parallel jobs," in *19th IASTED Intl. Conference on Parallel and Distributed Computing and Systems*, ser. PDCS '07.

[7] J. Hungershofer, "On the Combined Scheduling of Malleable and Rigid Jobs," in *SBAC-PAD 2004*. IEEE.

[8] G. Utrera, J. Corbalan, and J. Labarta, "Implementing Malleability on MPI Jobs," in *13th IEEE Intl Conf. on Parallel Arch. and Compilation Techniques (PACT'04)*.

[9] D. G. Feitelson and L. Rudolph, "Toward Convergence in Job Schedulers for Parallel Supercomputers," in *JSSPP*, 1996.

[10] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[11] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling Malleable Applications in Multicluster Systems," in *IEEE Cluster, 2007*.

[12] B. Francine et al, "Adaptive computing on the grid using apples," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[13] G. Wrzesinska, J. Maassen, and H. E. Bal, "Self-adaptive applications on the grid," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 121–129.

[14] S. S. Vadhiyar and J. J. Dongarra, ""Self Adaptivity in Grid Computing"," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 235–257, 2005.

[15] L. Kale and S. Krishnan, "Charm++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA*, September 1993.

[16] G. R. Gao, T. L. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *IPDPS*, 2007, pp. 1–6.

[17] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger, "Object-Based Adaptive Load Balancing for MPI Programs," in *Proceedings of the International Conference on Computational Science*, 2001.

[18] H. Kamal and A. Wagner, "FG-MPI: Fine-Grain MPI for multicore and clusters," in *The 11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDESC)*. IEEE, Apr. 2010.

[19] "Top500 supercomputing sites," http://top500.org.

[20] G. Zheng, "Achieving high performance on extremely large parallel machines: Performance prediction and load balancing," Ph.D. dissertation, University of Illinois (UIUC), 2005.

[21] A. Gupta, G. Zheng, and L. V. Kale, "A multi-level scalable startup for parallel applications," in *Proc. of Intl. Workshop on Runtime and Operating Systems for Supercomputers*, Tucson, AZ, USA, 5 2011.

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

[23] A. Bhatele et al., "Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms," in *IPDPS 2008*.

[24] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[25] S.-H. Chiang and M. K. Vernon, "Dynamic vs. Static Quantum-based Parallel Processor Allocation," in *Job Scheduling Strategies for Parallel Processing*, 1996.

[26] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive Fault Tolerance in MPI Applications Via Task Migration," in *HiPC*, 2006.

[27] A. Gupta et al, "The Who, What, Why, and How of HPC Applications in the Cloud." in *5th IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom) '13*.

[28] "Amazon Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2.

[29] "High Performance Computing (HPC) on AWS," http://aws.amazon.com/hpc-applications.