Scheduling of Elastic Message Passing Applications on HPC Systems

Debolina Halder Lina(✉), Sheikh Ghafoor, and Thomas Hines

Tennessee Tech. University, Cookeville, TN, USA
dlina@tntech.edu

Abstract. Elastic parallel applications that can change the number of processors while being executed promise improved application and system performance, allow new classes of data and event-driven highly dynamic parallel applications, as well as provide the possibility of predictive proactive fault tolerance via shrinkage in increasingly larger and more complex HPC systems, where the mean time between component failures is decreasing. There are several challenges for elastic application to become mainstream: 1) a clear understanding of programming models for elastic applications, 2) adequate support from message passing libraries, middleware, and resource management systems (RMS), and 3) thorough investigation of scheduling algorithms. Scheduling elastic jobs requires communication between running jobs and the RMS, keeping track of pending jobs, and prioritizing jobs to expand or shrink at a certain point in time. These challenges make the task of finding an optimal schedule challenging. We have proposed three different scheduling algorithms to schedule elastic applications along with six different candidate selection policies to prioritize the shrinkable applications and investigated their impact on system and application performance. We have studied the impact of workload characteristics and algorithms on performance. Our simulations results indicate that workload characteristics as well as the range of elasticity (flexibility) of the elastics applications impact the system and application performance.

Keywords: Elastic applications · Malleable · Evolving · Scheduling

1 Introduction

High Performance Computing (HPC) systems are growing in capacity, complexity, and heterogeneity [5,23]. The upcoming and current large HPC systems have hundreds of thousands of computing cores in addition to networking and other components. On the other hand, parallel applications that generally run on such systems are also growing increasingly complex. They are increasingly data and event-driven and dynamic in nature. The current generation of message passing parallel applications can not change resources (grow or shrink in terms of resource usage) once they start executing. According to Feitelson [8], a malleable application can grow or shrink in response to commands by the resource management system and an evolving application can also grow or shrink, but the application itself decides when it needs to change size. We will refer to malleable and evolving applications as elastic applications. Currently, there is no or limited support from runtime libraries and/or resource management systems (RMS) for such applications.

One use case for elastic applications is for dynamic applications whose computational needs are not known when the application is launched. Consider an application that simulates both airflows around an airplane and the stresses on the frame from the airflow. If the stress simulation determines that a crack appears then the computational needs to simulate the airflow may increase by an order of magnitude. Using current inelastic (rigid) applications, the way to deal with this would be to give the application the worst-case number of nodes (which may be difficult to compute), wasting resources. An elastic application could automatically request more nodes from the system at the time they are needed. If no crack appears then there would be no need to waste resources for that potentiality. Another potential benefit of elastic applications is that it gives a way for applications to proactively respond to failures. If a node is giving signs of impending failures, such as temperatures running too hot, then the system could instruct the application to shrink down, off the failing node. Elastic applications provide a path to maximum possible utilization by expanding or shrinking applications. Elastic applications would open up the potential for new highly dynamic applications that are not developed as there is no support for running them. There is no support because there are no applications that need it. Breaking this cycle would allow opportunities for a new generation of applications.

There are many challenges to realizing elastic applications. Current applications use a distributed memory model. Data is transferred by message passing. The number of shared memory nodes remains fixed for the lifetime of the application. Elastic applications will need to reorganize their data as they shrink or grow. The nature of the reorganization would depend on the application. A parameter sweep application might simply need to migrate some runs to a new node. An iterative grid-based application might need to completely redistribute the data to a new rectangular layout. The resource manager will need to communicate with running and the elastic applications in order to allocate more or preempt (shrink) resources. Likewise, the application will need to be able to give up or acquire more resources. Evolving applications will need to request more resources. This may potentially involve multiple phases as the resource manager offers resources and the application counteroffers. Current resource managers only need to consider pending applications in the queue and keep track of used and free nodes. An elastic job scheduler will need to make decisions about when to grow or shrink malleable applications, as well as respond to evolving applications' requests to grow or shrink. This will need to be done in a way that is fair to inelastic and elastic applications.

We have proposed three algorithms to schedule workloads containing elastic as well as non-elastic (traditional rigid) jobs. Each algorithm has been evaluated with a different policy to select a running malleable job to preempt resources.

We have run a number of simulations to observe the impact of these different policies on different types of workloads. We have used modified workload traces from real systems [17,24] as well as synthetic workloads. Our simulation results indicate that as the workload increases the impact of candidate selection policy gradually becomes insignificant. We also found that the distribution of elastic jobs in the workload impacts the performance gain. Our simulation results are consistent with several previous works that the presence of elastic application in the workload improves both system and application performance compared to the same workload with rigid applications only.

The rest of the paper is organized as follows. Section 2 provides a brief discussion of related works found in the literature. Our elastic application model is presented in Sect. 3, followed by a description of scheduling algorithms and candidate selection policies in Sect. 4. The experimental results are presented in Sect. 5. Finally, Sect. 6 presents our conclusion and planned future works.

2 Related Works

Research in the field of elastic parallel systems is not as extensive compared in other HPC areas. The simplest way of expanding and shrinking is to checkpoint the application state at some point in time and then again start the application from that point with a different number of processors. This approach has been implemented by Vadhiyar and Dongarra [27] as the Stop Restart Services (SRS) framework. ReSHAPE, developed by Sudarsan et al. [25] combines a scheduler with a reconfiguration library for iterative MPI applications. The Parallel programming framework, AMPI [15], is built on top of Charm++ [13]. AMPI implements MPI as user level threads. Recently, Iserte et al. [11] have designed a library DMRlib which provides a series of predefined communication patterns for data-redistribution and communication with the RMS. They have designed a communication API using which Nanos++ OmpSs [1] can communicate with the Slurm resource manager [28]. CooRMv2 is an RMS to ensure efficient scheduling of non-predictable evolving applications developed by Klein et al. [16]. Process Management Interface-exascale (PMIx) is an abstract set of interfaces using which applications and tools can interact with the different components of the System Management Stack (SMS) as well as different SMS components can also interact with each other [3,19]. The PMIx standard provides APIs for applications to request allocation of additional resources, extend the reservation on currently allocated resources, and release currently allocated resources. These APIs are still being developed, yet to provide support for full flexibility, and have not been adopted by production RMSs.

Kale et al. [14] have designed a simple scheduling algorithm for elastic job schedulers where all jobs are initially allocated with their minimum number of processors and the rest of the processes are shared equally among the jobs considering the maximum allowable resources of a particular job. Utrera et al. [26] have proposed an algorithm that mainly focuses on reducing the average waiting time. Gupta et al. [10] have proposed a split-phase scheduling algorithm where shrinkage requests are performed asynchronously. D'Amico et al. [4] have proposed a dynamic slowdown-driven (SD) policy to schedule rigid and malleable jobs to reduce average response time. Iserte et al. [12] have modified Slurm and implemented a reconfiguration policy using the moldable submission mechanism of Slurm. Prabhakaran et al. [20] have proposed a scheduling algorithm to run evolving jobs with rigid jobs but they have only considered expansion. They have extended the algorithm to schedule rigid, malleable, and evolving jobs together [21].

Research on scheduling elastic applications is at an early stage. Recent research mainly focuses on the impact of different scheduling parameters on performance, but how workload characteristics impact performance along with scheduling algorithms has not been investigated adequately. Similarly, the communication and negotiation aspect between RMS and elastic applications has been under investigation.

3 Application Model

Before developing the scheduling algorithms, we developed a model for elastic application and their interaction with the RMS. We made the following assumptions for our proposed model and algorithms:

- All applications in the workload are parallel applications.
- Only processors are considered as resources.
- The HPC system is homogeneous and communication time between any pair of processors are identical.
- An elastic application can run on any number of processors between a predefined minimum and maximum allowable processors (this may not hold true for some applications).
- The overhead of interaction between a running application and the RMS is negligible.

An elastic message-passing application consists of ph phases and the number of resources allocated to the application does not change during a phase. Changing a phase involved a change in the number of resources either in response to the application request or the RMS request.

A phase ph_i can be defined by three tuples-

$$< R_i, W_i, T_i >$$

where R_i is the allocation of phase i, W_i is the amount of computation done at that phase and T_i is the execution time of the phase. The total runtime of the application T is a summation of all T_i.

A phase change may involve data re-distribution as the number of processors changes. So, the total time of a phase consists of five components: computation time tw, parallel overhead to, data reorganization cost td, synchronization cost ts, and other overhead such as process creation or terminations tp. So,

$T_i = tw_i + to_i + td_i + ts_i + tp_i$. The parallel overhead of an application in general increases and computation time decreases as the number of processors increases and vice versa when the workload remains constant. We have modeled the parallel overhead as a certain percentage ($x\%$) of total execution time before any phase change. The remaining$(1 - x\%)$ time is required for computation. x varies from application to application. Let us assume that, the application has p_i and p_{i+1} processors at phase i and phase $i + 1$. So,

$$to_{i+1} = to_i/p_i * p_{i+1}$$

$$tw_{i+1} = tw_i * p_i/p_{i+1}$$

Data-redistribution cost depends on two variables- the total number of processors involved in change and change in the number of processors. The data distribution cost decreases if the total number of processors involved is increased. Again, data redistribution cost increases with the increase in the difference in the number of processors. For example, the data distribution cost of 8 to 16 processors is lower than the data distribution cost of 4 to 16 processors. The total number of processors involved is 24 and 20, and the difference in processors is 8 and 12, respectively. Synchronization cost varies from application to application state and does not depend on the change in resources. Though synchronization cost depends on the current resources of the application, we are ignoring that for simplification.

$ts_{i+1} = \sigma$, where σ varies from application state to application state. Total processor involved in phase change $p_{total} = p_i + p_{i+1}$. Change in processor $p_{difference} = |p_i - p_{i+1}|$. Then, data redistribution cost-

$$td_{i+1} = \alpha * p_{difference} + \beta/p_{total}.$$

Here, α and β are constants. Other overhead like processor creation or deletion cost is directly proportional to the number of new processors. So,

$$tp_{i+1} = b * p_{difference}$$

where b is the cost of one processor. We have used [9] as the execution model of the elastic parallel application in this study.

4 Scheduling Algorithms

To simulate different scheduling algorithms and visualize their impact on different performance metrics, we have used a discrete event simulator. We have followed the pattern from [9]. The following data structure is used in the algorithms described in this section:

- system state:
    - Idle processors (p_c): Number of idle processors

    - total processors (p_t): Number of total processors
    - Running job list (J_r): List of all currently running jobs
    - Running malleable job list (J_rm): List of all currently running malleable jobs
    - Pending job list (J_p): List of all pending jobs yet to be scheduled
- Candidate schedule:
    - List of job to start list (J_s): Jobs that are scheduled to be started at this time
    - Agreement List (A): List of expansion and shrinkage that needs to be done at this time
- Shrinkable malleable job list (J_sm): List of malleable jobs that need to be shrunk
- Required number of processors p_r: Number of processors that are required by a job for execution

The main scheduling algorithm is described in Algorithm 1.

Algorithm 1. Main Scheduling Algorithm (FCFS & easy backfilling with evolving request priority over pending job with maximizing throughput)

```
input: The current system state
output: A candidate schedule & system state
 1: (schedule evolving request)#SatisfyEvolvingRequest()
 2: (schedule initial allocation)#InitialAllocation()
 3: if $length(J\_p) > 0$ and $length(J\_rm) > 0$ then
 4:     (schedule pending jobs by shrinking malleable jobs)#SchedulePendingJob()
 5: end if
 6: if $p\_c > 0$ and $length(J\_rm) > 0$ then
 7:     (expand running malleable jobs if possible)#ExpandRunningMalleableJobs()
 8: end if
 9: while $length(A)$ do
10:     Take the first agreement
11:     Calculate negotiation cost (Similar to [9])
12:     execute agreement
13:     return $J\_s$ and $A$
14: end while
```

Main Scheduling Algorithm

The algorithm SatisfyEvolvingRequest() is described in Algorithm 2. As evolving requests are given the highest priority, first the algorithm tries to schedule the evolving request with idle resources. If enough idle resources are not found, the algorithm tries to allocate necessary resources by shrinking malleable jobs. The candidate for shrinkage is chosen by select_shrinkable_job() algorithm. The select_shrinkable_job() algorithm is described later in this section.

The initial allocation is based on FCFS with an easy backfilling scheduling policy [18]. The algorithm is described in Algorithm 3. If there are jobs in

Algorithm 2. SatisfyEvolvingRequest(system state)

```
input: system state
output: $A$ & system state
 1: if Shrinkage Request then
 2:     Add shrink to $A$
 3:     Update system state
 4: else
 5:     Allocate idle resources
 6:     if Enough idle resources not found then
 7:         $J\_sm$ = select_shrinkable_job()
 8:         if length($J\_sm$) not zero then
 9:             Add the shrinkages to $A$
10:             Add the expansion to $A$
11:             Update system state
12:         end if
13:     end if
14: end if
15: return $A$ and system state
```

the pending job list and running malleable job list, the algorithm then tries to schedule pending jobs by shrinking running malleable jobs using the SchedulePendingJob() algorithm. The algorithm is detailed in Algorithm 4.

Algorithm 3. InitialAllocation(J_p, system state)

```
input: $J\_p$, system state
output: $J\_s$, $J\_p$, system state
 1: for each job in $J\_p$ do
 2:     if $p\_c == 0$ then
 3:         return $J\_s$, $J\_p$, system state
 4:     end if
 5:     if $job.p\_r$ ¡= $p\_c$ then
 6:         add the job to $J\_s$
 7:         update system state
 8:     end if
 9: end for
10: return $J\_s$, $J\_p$, system state
```

The ExpandRunningMalleableJob() algorithm expands running malleable jobs if idle resources are available after scheduling pending jobs. As expanding any job will result in higher system utilization, jobs with the highest runtime are chosen for expansion with the motivation to reduce average turnaround time.

We have proposed three different algorithms to select the shrinkable malleable jobs. Algorithm 5 does not look at any system or application state. It tries to shrink jobs if enough resources are not found. For the rest of the paper, we

Algorithm 4. SchedulePendingJob(J_p)

```
Input: $J\_p$
Output: $A$, $J\_s$
 1: for each job in J_p do
 2:     if Job is malleable then
 3:         $p\_r$ = minimum processor that is required to run the job
 4:     end if
 5:     $(J\_sm), J\_s$, system state = select_shrinkable_job()
 6:     if $length$(J_sm)! = 0 then
 7:         Add shrinkages to agreement list
 8:         Add the job to job_to_start_list and remove from pending queue
 9:         Update system state
10:     end if
11: end for
```

will refer to it as "Default". Algorithm 6 looks at the running applications first and sees if any application ends in the next t seconds. If it is the case, then the application waits for that application to finish before it shrinks any new application. We will refer to this algorithm as "Application" for the rest of the paper. Algorithm 7 looks at the system utilization before shrinking any job. If the utilization is greater than $u\%$, it does not shrink any job. For the rest of the paper, we will refer to this algorithm as "System". In each algorithm, malleable jobs are sorted according to a certain priority. These techniques are described later in this section. Algorithm 8 tries to shrink running malleable jobs and allocate necessary resources. The technique to set these priorities is called the candidate/ victim selection technique.

Algorithm 5. Algorithm 01 for selecting shrinkable jobs (SelectShrinkableJob (J_rm, p_r))

```
input: $J\_rm$ and $p\_r$
output: $(J\_sm)$ , $J\_s$ & system state
 1: Sort the running malleable jobs according to priority
 2: for each job in the sorted list do
 3:     $J\_sm$, $J\_s$, system state = AllocateResource($job, p\_r$, system state)
 4: end for
 5: return empty list, $J\_s$, system state
```

Setting Priority of Malleable Jobs (Candidate/Victim Selection Techniques)

We have used multiple policies to define the priority of malleable applications. These are called candidates of victim selection techniques. The priorities are described below:

- Random, r: Jobs are randomly selected without considering any parameter. Jobs selected first have the highest priority.

Algorithm 6. Algorithm 02 for selecting shrinkable jobs (SelectShrinkableJob (J_rm, p_r))

```
input: $J\_rm$ and $p\_r$
output: ($J\_sm$), $J\_s$ & system state
 1: if there is a job which is about to end in next t sec then
 2:     return empty list, $J\_s$, system state
 3: end if
 4: Sort the running malleable jobs according to priority
 5: for each job in the sorted list do
 6:     if the job is not $x$% done then
 7:         $J\_sm$, $J\_s$, system state = AllocateResource(job,$p\_r$, system state)
 8:     end if
 9: end for
10: return empty list, $J\_s$, system state
```

Algorithm 7. Algorithm 03 for selecting shrinkable jobs (SelectShrinkableJob (J_rm, p_r))

```
input: $J\_rm$ and $p\_r$
output: ($J\_sm$), $J\_s$ & system state
 1: if utilization is greater than $u$% then
 2:     if there is a job which is about to end in next t second then
 3:         return empty list, $J\_s$, system state
 4:     end if
 5: end if
 6: Sort the running malleable jobs according to priority
 7: for each job in the sorted list do
 8:     $J\_sm$, $J\_s$, system state = AllocateResource($job, p\_r$, system state)
 9: end for
10: return empty list, $J\_s$, system state
```

Algorithm 8. AllocateResource(job, p_r, system state)

```
input: $job, p\_r$, system state
output: ($J\_sm$), $J\_s$ & system state
 1: needed resource allocation = $p\_r$
 2: if available_shrinkable_resources of job $\geq p\_r$ then
 3:     shrinkable resources = needed resource allocation
 4:     needed resource allocation = 0
 5:     add pending job to $J\_s$
 6: else
 7:     needed resource allocation -= available shrinkable resources
 8:     shrinkable resources = available resource allocation
 9: end if
10: Add the malleable job to $J\_sm$
11: if needed resource allocation = 0 then
12:     Return $J\_sm$, $J\_s$, system state
13: end if
14: Return $J\_sm$, $J\_s$, system state
```

- Gain, g: Initially, every job has a gain (g) value set to 0. Every time a job expands or shrinks, the gain changes. For expansion, the number of expanded cores is added to the gain. For shrinkage, the number of shrunk cores is subtracted from gain. The job with the highest gain has the highest priority.
- Shrinkable Resources, sr: If the application is running on $P_{current}$ processors and the minimum processor for the application is P_{min}, then the application has $P_{current} - P_{min}$ shrinkable resources (sr). The application with the highest sr has the highest priority. If two applications have the same shrinkable resources, the application with the highest current resources has the highest priority.
- No. of expansion, e: The job with the highest number of expansions (e) has the highest priority.
- Adaptation Cost, a: The job with the lowest adaptation cost has the highest priority.
- Time, t: The job with the lowest remaining runtime (t) has the highest priority.

4.1 Evaluation Metrics

We choose average turnaround time to measure application performance and utilization to indicate system performance.

If the arrival time of a job i is Ta_i and completion time is Tc_i, and the workload has total n jobs then average turn around time (TAT) is -

$$average\ TAT \sum_{i=1}^{n} \frac{Tc_i - Ta_i}{n}$$

System utilization indicates the fraction of CPU cycles that has been used during the execution of the workload. If the scheduled span of a workload is SS and total processors is p, then total cpu cycle, $C_{total} = SS * p$. Let us assume that the CPU cycle used by an application i is C_i. If the application has total ph phases and the execution time and processor of phase p is T_p and P_p respectively then,

$$C_i = \sum_{p=1}^{ph} T_p * P_p$$

If the workload has total n jobs then,

$$utilization = \sum_{i=1}^{n} \frac{C_i}{C_{total}}$$

5 Experiment and Results

5.1 Workload

Selecting a workload to simulate a scheduling algorithm needs special attention. The workload should emulate the workload running on a cluster. Input data for simulating scheduling algorithms can be obtained in two ways. One is to derive it from workload traces of the existing HPC system and the other is to generate them using different workload models. For our study, we have used two real workloads and one synthetic workload. We have chosen the LLNL Atlas [17] and the KIT for HLR II [24] logs from the parallel workload archive [7]. We will refer to these two workloads as LLNL and KIT, respectively. For both workloads, we have considered the first 10,000 jobs for simulation. In order to increase the load and see the impact of that, we have further modified these two workloads. We have created two shrunk versions of LLNL and KIT by shrinking the inter-arrival time by 5% and 35% respectively. The modified workloads are referred to as LLNL-shrunk and KIT-shrunk respectively. In addition to the real workloads, one synthetic workload has been generated using Downey's model [6]. A workload containing 10,000 jobs was created with a cluster size of 10,000 processors. The model parameters to generate the workload are listed in Table 1.

Table 1. Parameters of Downey's model to generate synthetic

#Jobs	rho	seed	Job width ($ln(T_r)$)		Job size ($ln(P)$)	
$10,000$	0.75	17	Min	Max	Min	Max
			5.69	9.91	0.69	8.51

Table 2 summarizes the workloads. The max processor and min processors are the maximum and the minimum number of processors a job has in the workload, respectively. The relevant parameters of the workloads are:

Table 2. Workloads for simulation

Workload	# of jobs	Max processor	Min processor	Total processors	% of shrinkage in interarrival time
LLNL	10,000	9160	1	9,216	–
LLNL_shrunk	10,000	9160	1	9,216	5%
KIT	10,000	10,240	1	24,048	–
KIT_shrunk	10,000	10,240	1	24,048	35%
Synthetic	10,000	4994	2	10,000	–

- Id: A unique identifier for the jobs in the workload
- Type: A job can be of three types- rigid, malleable and evolving

- Arrival, T_a: Job arrival time
- Runtime, T_r: Execution time of a job
- Processors, P: Number of desirable processor allocation
- Minimum Allowable processor, p_{min}: Minimum required resource allocation for a job. p_{min} is equal to P for rigid jobs.
- Maximum Allowable Processor, p_{max}: Maximum allowable processor allocation of a job. p_{max} is equal to P for rigid jobs.

Creating Elastic Workload. All the workloads mentioned in Sect. 5.1 are rigid. We have generated an elastic workload by randomly selecting jobs to be elastic. If in a certain workload $x\%$ of jobs are elastic, then $x/2\%$ of them are malleable, and $x/2\%$ of jobs are evolving. For every workload, we have made in total 10 elastic versions with $10, 20, 30, 40, 50, 60, 70, 80, 90$, and 100% elastic jobs. Every elastic job has maximum and minimum processor requirements. We have selected 800% of P to be the maximum (p_{max}) and 50% of P to be the minimum (p_{min}) allowable resources.

Evolving jobs make expansion and shrinkage requests to the simulator. We refer to these requests as evolving requests. We choose the total number of evolving requests submitted by an evolving job chosen randomly from a predefined maximum and minimum. We have selected the event type (expansion or shrinkage) from a Bernoulli Distribution with a higher probability to be expansion. Then, we have chosen the number of processors involved in the evolving event from a predefined maximum and minimum. The time of occurrence of each evolving event is selected at a percentage of the remaining computation. The percentage is also chosen from a predefined minimum and maximum.

5.2 Experimental Setup

The discrete event simulator has been implemented using Python 3.4.1. Results shown in this section reflect the average of 10 runs. We have chosen the predefined parameters used in this simulation from an educated guess. Parameters were chosen to be the following:

- t in Algorithm 6 and 5 is set to be 5 s.
- u in Algorithm 5 is set to be 80.
- Maximum negotiation cost is set to be 0.05 s and minimum negotiation cost is set to be 0.005 s.
- Parallel overhead defined in the mathematical model (Sect. 3) is set to be between 0.5% to 1%.
- α and β defined in the mathematical model of Sect. 3 are randomly chosen from a uniform random distribution of 0.005 to 0.05.
- Synchronization cost defined in the mathematical model of Sect. 3 is randomly chosen from a uniform random distribution of 0.015 s to 0.1 s.
- Maximum and minimum evolving events requested by an evolving application are set to be 4 and 1 respectively.
- Probability of an evolving event to be an expansion event is set to be 0.8
- Expansion event can occur anytime when 30% to 60% of work is left.

5.3 Results

In all cases, including elastic application into the workload improves performance over only rigid workload. The performance of the only rigid workload is shown in the dotted line in all the plots of this section. Figures 1, and 2 show the comparison of algorithm System, Application, and Default in terms of average turnaround time, and system utilization of KIT workload, respectively. The algorithm System attains the best system utilization for all candidate selection techniques but performs worst in terms of average turnaround time. Algorithms Application and Default perform in a similar manner.

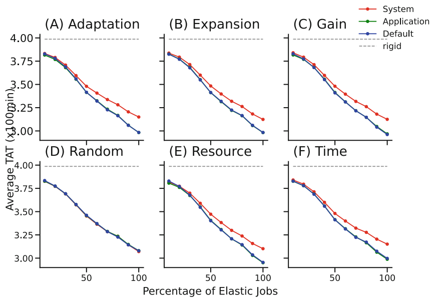

Fig. 1. Average turnaround time of KIT workload with different algorithms

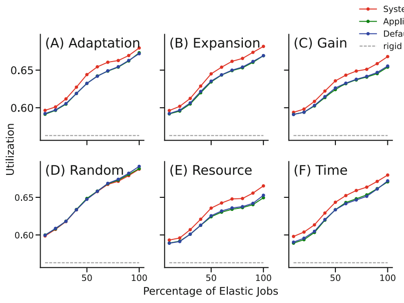

Fig. 2. System utilization with KIT workload and different algorithms

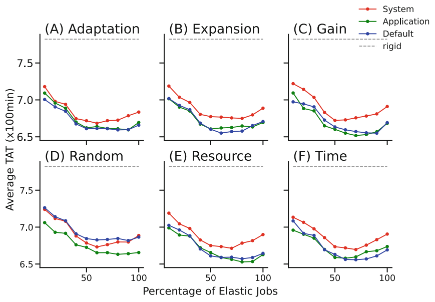

Fig. 3. Average turnaround time of KIT_shrunk workload with different algorithms

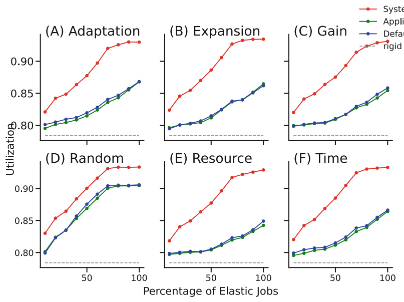

Fig. 4. System utilization with KIT_shrunk workload and different algorithms

Figures 3, and 4 show the comparison of Algorithm System, Application, and Default in terms of average TAT, and system utilization of KIT_shrunk workload, respectively. Random performs the best in terms of utilization for all three algorithms. The algorithm System attains the best system utilization

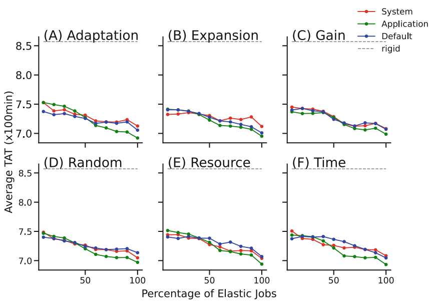

Fig. 5. Average turnaround time of LLNL workload with different algorithms

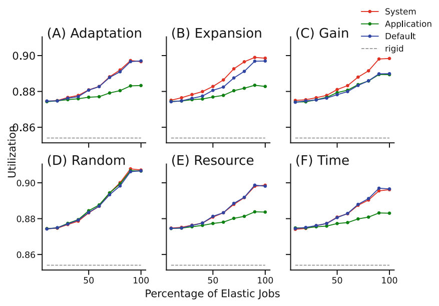

Fig. 6. System utilization with LLNL workload and different algorithms

for all candidate selection Algorithms but performs the worst in terms of average turnaround time. The algorithm Application performs the best in terms of average turnaround time (see Fig. 3).

Figures 5 and 6 show the comparison of algorithm System, Application, and Default in terms of average turnaround time, and system utilization of LLNL workload respectively. The algorithm System has the worst average TAT in the case of adaptation, expansion, and gain. The algorithm Default has the worst average TAT in the case of random, resource, and time. The algorithm Application provides the best TAT in all cases. In terms of utilization, the algorithm System gets the best turnaround time for adaptation, expansion, gain, and random. Algorithm Default generates the best utilization for resources and time. The worst TAT comes from the algorithm Application in case of adaptation, expansion, resource, and time, and from the algorithm Default in case of gain and random.

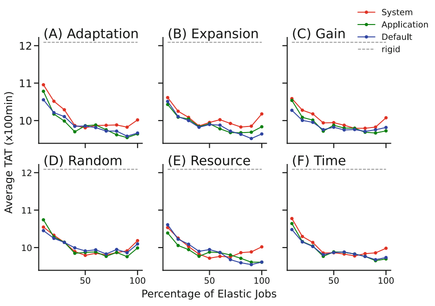

Fig. 7. Average turnaround time of LLNL_shrunk workload with different algorithms

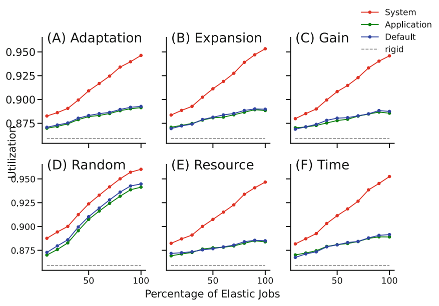

Fig. 8. System utilization with LLNL_shrunk workload and different algorithms

Figures 7 and 8 show the comparison of Algorithm System, Application, and Default in terms of average turnaround time, and system utilization of

LLNL_shrunk workload respectively. The algorithm System generates the worst TAT everywhere except random and resource. For random and resource, algorithm Default is the worst. The algorithm Application performs best in terms of average TAT for all candidate selection techniques except expansion and gain. In terms of utilization, the algorithm System performs the best and the algorithm Application performs the worst in all cases.

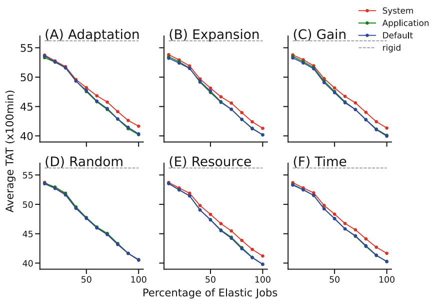

Fig. 9. Average turnaround time of synthetic workload with different algorithms

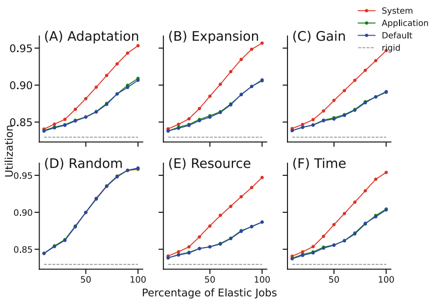

Fig. 10. System utilization with synthetic workload and different algorithms

Figures 9 and 10 show the comparison of algorithm System, Application, and Default respectively in terms of average TAT, and system utilization of Synthetic workload, respectively. In terms of average TAT and utilization, the results of the algorithm Application and Default are clustered together. The algorithm System is the best in terms of utilization and the worst in terms of average TAT.

5.4 Analysis

Table 3 shows comparisons between different candidate selection techniques for the scheduling algorithms for different workloads. For each metric, the best and the worst performing techniques along with the maximum difference between the best and the worst are presented. Table 3 shows that the maximum improvement gained over different candidate selection techniques of LLNL_shrunk workload is less than that of LLNL workload. It can be said that the results of different candidate selection techniques of LLNL_shrunk workload are more clustered than that of LLNL workload (see Figs. 5, 6, 7 and 8). A similar pattern is found in KIT and KIT_shrunk workloads. So, we can conclude that if the workload is high, the difference in performance between different candidate selection techniques becomes insignificant. Table 3 also shows that in terms of average TAT gain performs the best and random performs the worst in most of the cases. In terms of utilization, random performs the best, and resource performs the worst in most cases.

The performance of Synthetic workload increases as the percentage of elastic jobs increases. For other workloads, the increase saturates and stops at some

Table 3. Maximum improvement achieved by using different candidate selection techniques at any percentage of elastic jobs

Workload	Algorithm	Average TAT			Utilization		
		Best	Worst	Max difference	Best	Worst	Max difference
LLNL	Application	Gain	Random	2.62% at 40% elastic jobs	Random	Resource	8.075% at 70% elastic jobs
	System	Gain	Resource	2.09% at 100% elastic jobs	Random	Resource	3.65% at 70% elastic jobs
	Default	Random	Resource	3.08% at 30% elastic jobs	Random	Resource	9.0% at 90% elastic jobs
LLNL_shrunk	Application	Adaptation	Random	1.45% at 70% elastic jobs	Random	Time	3.23% at 90% elastic jobs
	System	Gain	Random	2.19% at 60% elastic jobs	Random	Time	1.65% at 90% elastic jobs
	Default	Resource	Random	2.36% at 100% elastic jobs	Random	Gain	2.49% at 90% elastic jobs
KIT	Application	Gain	Random	3.21% at 70% elastic jobs	Random	Resource	10.25% at 50% elastic jobs
	System	Adaptation	Random	3.04% at 20% elastic jobs	Random	Resource	3.11% at 40% elastic jobs
	Default	Gain	Random	5.7% at 70% elastic jobs	Random	Resource	10.9% at 50% elastic jobs
KIT_shrunk	Application	Gain	Random	1.23% at 80% elastic jobs	Random	Resource	0.85% at 90% elastic jobs
	System	Gain	Time	1.66% at 100% elastic jobs	Random	Resource	1.33% at 60% elastic jobs
	Default	Gain	Random	2.84% at 100% elastic jobs	Random	Resource	1.31% at 90% elastic jobs
Synthetic	Application	Gain	Random	1.51% at 60% elastic jobs	Random	Resource	8.58% at 80% elastic jobs
	System	Gain	Time	1.09% at 90% elastic jobs	Random	Resource	3.6% at 70% elastic jobs
	Default	Gain	Random	1.79% at 90% elastic jobs	Random	Resource	3.47% at 70% elastic jobs

point. Figures 11 and 12 shows the distribution of total jobs and the elastic jobs of KIT_shrunk workload and Synthetic workload when 30% and 60% of the workload are elastic respectively. The distribution for KIT, LLNL, and LLNL_shrunk workloads is similar to KIT_shrunk workloads. From these figures, we can see that in the case of Synthetic workload there are always elastic jobs present in the workload which is not the case for other workloads. For this reason, the Synthetic workload shows constant improvement with the increase in the percentage of elastic jobs.

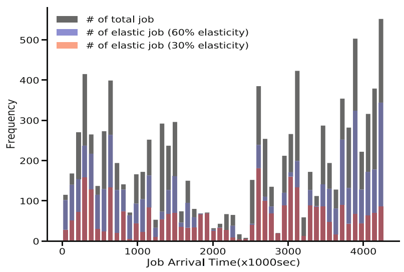

Fig. 11. Distribution of elastic job in KIT_shrunk workload

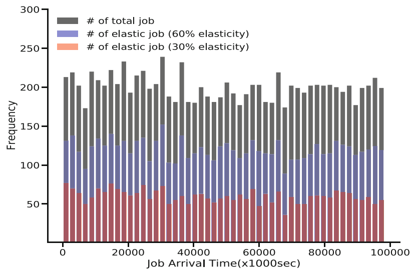

Fig. 12. Distribution of elastic job in synthetic workload

For KIT workload and KIT_shrunk workload, no algorithm can achieve utilization above 91% (see Figs. 2 and 4). The possible reason for this can be fragmentation and/or adaptation and negotiation overhead. Elastic jobs have a limit on how much they can expand. As a result, fragmentation can still exist even after the full expansion of all running elastic jobs. Again, the job distribution over time may be in such a way that at any point in time, there may not be any elastic job running at the cluster, and the cluster may remain underutilized. Also, adaptation cost and negotiation cost cause some utilization loss. To investigate this further, we have created another version of KIT and KIT_shrunk workload where every elastic job has a maximum resource of 24048 (equal to cluster size) and a minimum resource of 1. We refer to this phenomenon as 100% flexibility. We call these versions KIT_full workload and KIT_shrunk_full workload, respectively. Figures 13 and 14 show the utilization of KIT_full and KIT_shrunk_full workloads, respectively. Utilization of KIT_full workload saturates at 90%, but utilization of KIT_shrunk_full workload saturates at 98.5%. This proves that the KIT_full workload still has fragmentation as the inter-arrival time is high as well as adaptation and negotiation overhead. On the other hand, KIT_shrunk_full workload losses 1.5% utilization due to adaptation cost and negotiation cost. Also, from the figures, a knee is visible in the utilization curve. After a certain point, utilization saturates and does not improve with the increase in percentage elastic jobs. At 100% flexibility, improvement in utilization saturates at a certain percentage of elastic jobs.

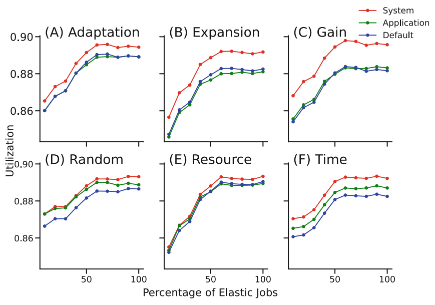

Fig. 13. Utilization of KIT_full workload

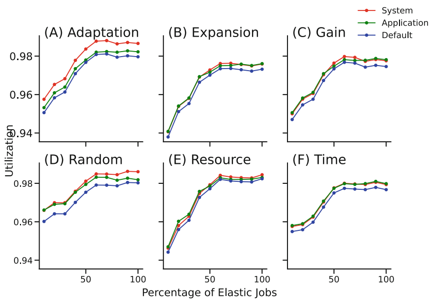

Fig. 14. Utilization of KIT_shrunk_full workload

The key findings of this research are as follows:

- When the load is high, the performance difference between many candidate selection techniques is insignificant.
- Impact of elastic jobs not only depends on the percentage of elastic jobs but also depends on the distribution of elastic jobs over time. The more uniform the distribution of elastic jobs over time is, the more evident the impact is.
- Even introducing 100% flexibility, utilization may not be 100% due to adaptation cost, negotiation cost, and fragmentation. Fragmentation may still remain due to the limit on the expansion capability of running elastic jobs.
- Algorithm System (Algorithm 7) always gets the highest system utilization. In most of the cases, the Algorithm Application (Algorithm 6) gets the lowest average TAT.

- In terms of average TAT, the case study shows that gain performs the best and random performs the worst in most cases. In terms of utilization, random performs the best and resource performs the worst in most cases.
- In all cases even a low percentage of elastic jobs (as low as 10% of the total job) improves the performance.

6 Conclusion and Future Works

The main objective of our research is to propose and evaluate different scheduling strategies for elastic applications under different workloads. We have proposed three different scheduling algorithms, and for every algorithm, we have proposed six candidate selection techniques to prioritize shrinkable jobs. We have evaluated the proposed algorithms using modified workload traces from real systems as well as synthetic workloads. The following are the main observations from our study: 1) With the increased workload, the difference in performance improvement between the proposed candidate selection techniques becomes insignificant. 2) The impact of elasticity not only depends on the number of elastic jobs but also depends on their distribution over time in the workload. The more uniform the distribution of elastic jobs over time is, the more constant the improvement will be with the increase in elasticity. 3) We have observed that even with 100% flexibility, 100% utilization can not be achieved. Adaptation and negotiation overhead limits the maximum achievable utilization. 4) In all cases, workload with elastic applications improves both system and application performance compared to the same workload with rigid workload only. 5) Even with a very small percentage of elasticity (as low as 10%), both system and application performance improved.

One of the limitations of the study is that the HPC systems we derived the workload traces for our simulation no longer represent current large HPC systems such ORNL Summit, Fugaku, etc., or upcoming systems like ORNL Frontier. In addition, simulation parameters such as adaptation cost used for simulation were derived from educated estimates based on sample runs of an iterative structured grid application running on a medium-size cluster. Experiments with the real application at scale should be used to estimate the value of such parameters. These limitations are mostly due to time, effort, allocation and access to large HPC systems, and availability of real workload traces from systems like Summit. Our planned future work includes: 1) investigating candidate selection policies for job expansion, 2) the impact of the ratio of malleable and evolving jobs in the workload on performance (all our workload has 50% malleable 50% evolving). It is difficult to estimate the execution time of an application if the number of processors is changed in the middle of execution. There exists some model to estimate the total execution time of an application on different sets of processors [2,22]. Further exploration/extension of such models for estimation of execution time on a different number of processors can be investigated.

References

1. Ayguade, E., Badía, R., Labarta, J.: OmpSs and the Nanos++ runtime
2. Bhimani, J., Mi, N., Leeser, M., Yang, Z.: New performance modeling methods for parallel data processing applications. ACM Trans. Model. Comput. Simul. (TOMACS) **29**(3), 1–24 (2019)
3. Castain, R.H., Hursey, J., Bouteiller, A., Solt, D.: PMIx: process management for exascale environments. Parallel Comput. **79**, 9–29 (2018)
4. D'Amico, M., Jokanovic, A., Corbalan, J.: Holistic slowdown driven scheduling and resource management for malleable jobs. In: Proceedings of the 48th International Conference on Parallel Processing, pp. 1–10 (2019)
5. Dongarra, J.: Report on the Fujitsu Fugaku system. University of Tennessee-Knoxville Innovative Computing Laboratory, Technical report ICLUT-20-06 (2020)
6. Downey, A.B.: A parallel workload model and its implications for processor allocation. Clust. Comput. **1**(1), 133–145 (1998)
7. Feitelson, D.G.: Parallel workload archive (2007). http://www.cs.huji.ac.il/labs/parallel/workload
8. Feitelson, D.G., Rudolph, L.: Toward convergence in job scheduling for parallel supercomputers (1996)
9. Ghafoor, S.K.: Modeling of an adaptive parallel system with malleable applications in a distributed computing environment. Mississippi State University (2007)
10. Gupta, A., Acun, B., Sarood, O., Kalé, L.V.: Towards realizing the potential of malleable jobs. In: 2014 21st International Conference on High Performance Computing (HiPC), pp. 1–10. IEEE (2014)
11. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Beltran, V., Peña, A.J.: DMR API: improving cluster productivity by turning applications into malleable. Parallel Comput. **78**, 54–66 (2018)
12. Iserte, S., Mayo, R., Quintana-Orti, E.S., Pena, A.J.: DMRlib: easy-coding and efficient resource management for job malleability. IEEE Trans. Comput. **70**(9), 1443–1457 (2020)
13. Kale, L.V., Krishnan, S.: Charm++ a portable concurrent object oriented system based on C++. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 91–108 (1993)
14. Kalé, L.V., Kumar, S., DeSouza, J.: A malleable-job system for timeshared parallel machines. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002), pp. 230–230. IEEE (2002)
15. Kale, L.V., Zheng, G.: Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: Advanced Computational Infrastructures for Parallel and Distributed Applications, pp. 265–282 (2009)
16. Klein, C., Pérez, C.: An RMS for non-predictably evolving applications. In: 2011 IEEE International Conference on Cluster Computing, pp. 326–334. IEEE (2011)
17. Minh, T.N., Wolters, L.: Modeling parallel system workloads with temporal locality. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2009. LNCS, vol. 5798, pp. 101–115. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04633-9_6
18. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6), 529–543 (2001)
19. Polyakov, A.Y., Karasev, B.I., Hursey, J., Ladd, J., Brinskii, M., Shipunova, E.: A performance analysis and optimization of PMIx-based HPC software stacks. In: Proceedings of the 26th European MPI Users' Group Meeting, pp. 1–10 (2019)

20. Prabhakaran, S., Iqbal, M., Rinke, S., Windisch, C., Wolf, F.: A batch system with fair scheduling for evolving applications. In: 2014 43rd International Conference on Parallel Processing, pp. 351–360. IEEE (2014)
21. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A batch system with efficient adaptive scheduling for malleable and evolving applications. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 429–438. IEEE (2015)
22. Raeder, M., Griebler, D., Baldo, L., Fernandes, L.G.: Performance prediction of parallel applications with parallel patterns using stochastic methods. In: Sistemas Computacionais (WSCAD-SSC), XII Simpósio em Sistemas Computacionais de Alto Desempenho, pp. 1–13 (2011)
23. Schneider, D.: The exascale era is upon us: the frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. IEEE Spectr. **59**(1), 34–35 (2022)
24. Soysal, M., Berghoff, M., Klusáček, D., Streit, A.: On the quality of wall time estimates for resource allocation prediction. In: Proceedings of the 48th International Conference on Parallel Processing: Workshops, pp. 1–8 (2019)
25. Sudarsan, R., Ribbens, C.J.: ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In: 2007 International Conference on Parallel Processing (ICPP 2007), p. 44. IEEE (2007)
26. Utrera, G., Tabik, S., Corbalán, J., Labarta, J.: A job scheduling approach to reduce waiting times. Technical report, Technical University of Catalonia, UPCDAC-RR-2012-1 (2011)
27. Vadhiyar, S.S., Dongarra, J.J.: SRS: a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process. Lett. **13**(02), 291–312 (2003)
28. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple Linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). https://doi.org/10.1007/10968987_3