Elastic Job Bundling: An Adaptive Resource Request Strategy for Large-Scale Parallel Applications

Feng Liu, Jon Weissman

April 16, 2015
Elastic Job Bundling: An Adaptive Resource Request Strategy for Large-Scale Parallel Applications  
(Extended Abstracts)

Feng Liu  
Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN  
liux2102@umn.edu

Jon B. Weissman  
Department of Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN  
jon@cs.umn.edu

Abstract—In today’s batch queue HPC cluster systems, the user submits a job requesting a fixed number of processors. The system will not start the job until all of the requested resources become available simultaneously. When cluster workload is high, large sized jobs will experience long waiting time due to this policy. In this paper, we propose a new approach that dynamically decomposes a large job into smaller ones to reduce waiting time, and lets the application expand across multiple subjobs while continuously achieving progress. This approach has three benefits: (i) application turnaround time is reduced, (ii) system fragmentation is diminished, and (iii) fairness is promoted. Our approach does not depend on job queue time prediction but exploits available backfill opportunities. Simulation results have shown that our approach can reduce application mean turnaround time by up to 48%.

I. INTRODUCTION

Scientific research today in areas such as fluid dynamics and climate modeling is largely dependent on simulations which have large computational needs [14]. Parallel computers are commonly used to address such problems of ever increasing scale [4]. With the rapid growth of scientific parallel programs designed to execute simultaneously on hundreds to thousands of processors, swiftly provisioning a large number of processors has become more challenging.

Massively parallel supercomputers have long been the most popular platform for executing large-scale scientific applications. Due to the high cost of these machines, users usually space-share them by submitting individual job requests to the batch queue system. Each job request contains the number of desired processors $P$ and run time estimation $R$. Once a job is scheduled, it gains exclusive use of the $P$ processors until it finishes before $R$, or is killed when $R$ expires.

Mapping each application’s resource request to a $P \times R$ shape is convenient for users to specify and simplifies batch scheduler design. However, this rigid scheme may also cause the following problems: (i) when system workload is high, it is difficult to find enough free processors for large jobs which leads to long waiting time; and (ii) when most jobs are large, a comparatively small number of free processors cannot be efficiently utilized, since these fragments are unusable for any waiting job. Giving higher priorities to large jobs will not solve these problems, particularly in the event that the workload is dominated by large jobs.

In this work, we propose a new technique addressing the queue waiting problem called Elastic job bundling. When a large job of size $P \times R$ is waiting in the queue, we decompose it into several smaller subjobs of size $P_x \times R_x$ ($P_x < P$) to reduce wait time. This technique then manages the time overlap of subjobs to allow the application to continuously execute and make progress.

In contrast to prior approaches such as [7], [8], [31], our technique: 1) does not require any changes to the batch scheduler, 2) does not depend on queue time prediction, and 3) does not require any changes to the application (e.g. moldability or malleability).

We evaluate our approach using real-world workloads. Preliminary results reveal that our approach can:

• on average reduce target job waiting & turnaround time by up to 69% & 48% respectively;
• on average reduce system-wide job waiting & turnaround time by up to 39% & 27% respectively;
• promote fairness in terms of waiting time between large and small jobs;
• lower system fragmentation by up to 59%.

II. ELASTIC JOB BUNDLING (EJB)

Elastic job bundling (EJB) is a software layer that operates between parallel application end-users and HPC batch systems. The goal of EJB is to reduce the turnaround time of parallel applications, especially those that demand a large number of processors. EJB accepts ordinary job requests and transforms them into multiple smaller subjobs which can start earlier than the original job. Applications initially start running on these smaller subjobs with downgraded performance due to over-subscription. During run time, the application will dynamically expand to processors subsequently acquired by EJB, through additional subjob requests when more resources become available.
A. The Formation of Elastic Jobs

Traditionally, one parallel application \( A \) is bound to a single job \( J \), with fixed processors \( P \) and run time estimation \( R \), which can be expressed as \( A \rightarrow J = (P, R) \). A batch scheduler will either allocate all of the \( P \times R \) resource or keep the job waiting. This all-or-nothing job scheduling strategy can lead to inefficiency. Consider the example in Figure 1a. Because all job requests are rigid, the three jobs experience long waiting time despite the presence of many idle processors. Intuitively, by changing the “shapes” of the waiting jobs in a way that they can adapt to the dynamic workload, we can not only reduce queue waiting time, but also improve resource utilization.

EJB implements this idea as follows: EJB treats a job request sent to it as a target job \( J_t \), and the application bound to \( J_t \) as the target application, \( A_t \rightarrow J_t = (P_t, R_t) \). EJB tries to improve \( J_t \)’s turnaround time by first decomposing \( J_t \) into several smaller subjobs \( J_x = (P_x, R_x), x = 1, \ldots, n \), \( P_x < P_t \). For example, if the jobs in Figure 1a were submitted to EJB, it would treat those jobs as target jobs and decompose them to smaller subjobs which can start much earlier and increase utilization (Figure 1b).

Second, EJB “bundles” the resource allocations from the independent subjobs to create an integrated malleable job \( J_e = (J_1, J_2, \ldots, J_n) \), called an elastic job, as Figure 1c shows. Third, EJB runs the target application continuously on the elastic job, \( A_t \rightarrow J_e \), which will be discussed in the next section. At any point in time, the number of total processors allocated to \( J_e \) will be \( \leq P_t \) since we maintain \( P_t \) processes of \( A_t \) at all times.

A subjob looks like an ordinary parallel job to the batch scheduler. The prefix “sub” is only meant to articulate a composition relationship between subjobs and the integrated elastic job. The notations introduced in this Section are summarized as follows:

- target job: \( J_t = (P_t, R_t) \);
- target application: \( A_t \);
- subjob: \( J_x = (P_x, R_x), x = 1, \ldots, n \);
- elastic job: \( J_e = (J_1, J_2, \ldots, J_n) \).

B. Running Applications on Elastic Jobs

When running a target application on an elastic job, the number of processors allocated to all concurrently running subjobs can change. The total duration of an elastic job can be divided into intervals. The number of processors in each interval stays the same. However, EJB does not change the number of parallel processes in the application. Instead, EJB adapts the target application to the elastic job through oversubscription and migration. Thus, the application structure or logic need not change.

Given \( A_t \rightarrow J_t = (P_t, R_t) \), we can know that \( A_t \) has \( P_t \) processes. By running \( A_t \) exclusively on \( J_t \), with one process per processor, the run time of \( A_t \) is \( R_t \),

\[ p_{A_t}(P_t) = R_t \quad (1) \]

Suppose \( A_t \) is compute-bound with balanced workload which is typical of many SPMD applications. Under oversubscription, \( A_t \) is run on \( q \) processors, \( q < P_t \). Under an even distribution, each processor is time-shared by up to \( \frac{P_t}{q} \) processes where each process on the same processor is given an equal share of the CPU. In this case, the expected performance degradation would be linearly proportional to \( \frac{P_t}{q} \), such that:

\[ p_{A_t}(q) = \left\lfloor \frac{P_t}{q} \right\rfloor \cdot R_t \quad (2) \]

Obviously, one processor can only support a limited number of processes for over-subscription, due to memory constraints or context switching cost. We denote by \( O_{max} \) an upper bound on the degree of over-subscription. For simplicity in this study, we assume \( O_{max} \) to be the same for different applications, such that \( q \in \left[ \frac{P_t}{O_{max}}, P_t \right] \). Our technique is applicable to more complex degradation models or to differing values of \( O_{max} \), but these are the subject of future work.

When a new subjob \( J_y \) is added to \( J_e \), EJB migrates a subset of \( A_t \)’s running processes to \( J_y \)’s processors, lowering the degree of over-subscription. Before a running subjob \( J_y \) terminates, EJB must migrate all the processes running in \( J_y \) to \( J_e \)’s other continuing subjobs. This type of cross-subjob migration can be performed in bulk, such that all
Figure 2: Mapping a parallel application’s processes to an elastic job, including progress measurement.

the migrated processes are moved concurrently. \( A_t \) stops making progress during migration intervals. We can assume each bulk migration interval has a fixed maximum duration of \( \lambda \) seconds. This bound can be established because in each migration interval, each destination processor can have at most \( O_{max} \) processes migrated to it. Migrations to different destination processors can be carried out in parallel. To evaluate the impact of migration cost, we can vary \( \lambda \).

\( J_e \) has two types of intervals: RUN and MIGRATE. Suppose there are \( l \) intervals in \( J_e \). In interval \( k \) (\( k = 1, \ldots, l \)), \( J_e \) has \( q_k \) concurrently usable processors and interval length = \( L_k \). Now we model \( A_t \)'s progress on \( J_e \) as:

\[
\sum_{k} \frac{L_k}{P_{A_t}(q_k)} = 100\% \\
\text{where } k = 1, \ldots, l \text{ and } k.\text{type} = \text{RUN}
\] (3)

Based on Equation 3, we can: (i) estimate \( A_t \)'s progress at any point during it’s run time, (ii) estimate the time it takes to achieve a certain amount of progress, and (iii) given \( A_t \)'s current progress and upcoming intervals, estimate \( A_t \)'s completion time.

We demonstrate this progress model in Figure 2. In this example, a 4 process \( A_t \) is submitted with \( J_t = (4, 800s) \). \( J_e \) contains 3 subjobs: \( J_1 = (1, 1460s) \), \( J_2 = (1, 1060s) \), and \( J_3 = (2, 440s) \). \( J_e \) has 7 intervals, the durations of which are marked below the interval number. In interval 1, since only \( J_1 \) is running, \( J_e \) has 1 processor. Each of the 4 processes of \( A_t \) over-subscribe the same processor in a time-shared manner, such that \( A_t \) makes progress at a rate of \( \frac{1}{4} \). By the end of interval 1, when \( J_2 \) starts running, \( A_t \)'s progress is 12.5%. With \( J_2 \)'s 1 processor added, \( J_e \) has 2 processors. Interval 2 is a MIGRATE interval. Suppose its length \( \lambda = 20s \), within which process 3 and 4 are migrated to \( J_2 \). Then in interval 3, \( A_t \) makes progress at a rate of \( \frac{1}{2} \). By the end of interval 3, when \( J_3 \) starts running, \( A_t \)'s progress is 37.5%. Since \( J_3 \) terminates before \( A_t \)'s completion time, 2 MIGRATE intervals 4 and 6 are added. Processes 2 and 4 which were migrated to \( J_3 \) in interval 4, must be migrated back to \( J_1 \) and \( J_2 \) in interval 6 before \( J_3 \) terminates. There is no over-subscription in interval 5, by the end of which \( A_t \)'s progress is 87.5%. Interval 7 is the last interval, by the end of which \( A_t \)'s progress is 100%, \( A_t \) then completes. \( J_e \)'s runtime is the summation of its intervals: 1460s.

Table I: Upper-bounds of processor (\( P_{max} \)) and runtime (\( R_{max} \)) of two types of immediately backfillable job slots.

<table>
<thead>
<tr>
<th>Type-I slot</th>
<th>( P_{max} )</th>
<th>( R_{max} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( extra ) processors</td>
<td>none</td>
<td>( T_{shadow} - T_{now} )</td>
</tr>
</tbody>
</table>

of \( \frac{1}{2} \). By the end of interval 3, when \( J_3 \) starts running, \( A_t \)'s progress is 37.5%. Since \( J_3 \) terminates before \( A_t \)'s completion time, 2 MIGRATE intervals 4 and 6 are added. Processes 2 and 4 which were migrated to \( J_3 \) in interval 4, must be migrated back to \( J_1 \) and \( J_2 \) in interval 6 before \( J_3 \) terminates. There is no over-subscription in interval 5, by the end of which \( A_t \)'s progress is 87.5%. Interval 7 is the last interval, by the end of which \( A_t \)'s progress is 100%, \( A_t \) then completes. \( J_e \)'s runtime is the summation of its intervals: 1460s.

Actually, our degradation model is conservative. Prior work [13] has shown that a modest level of over-subscription (2, 4, 8 tasks per core) can improve throughput. Specifically, they have shown that for MPI, the end-to-end performance decreases by 10% of 2 tasks per core, and 18% of 4 tasks per core. While in our model, these two numbers are 100% and 300%, which can be seen as theoretical worst-case upper-bounds.

C. Taming Unpredictability

EJB needs to control the sizes of subjobs to enable them to be scheduled early, and to ensure that they overlap in run time to allow for migration. However, accurate queue wait time prediction is known to be a difficult problem despite many efforts in this area [6, 9, 10, 16, 22, 26, 30, 34]. We address this challenge by controlling the shape of the subjobs such that they can be immediately scheduled on the fragmented idle resources. E.g., production schedulers such as TORQUE [27] or SLURM [35] are capable of providing immediately available resources information through the user interface such as showbf or slurmbf.

At a first glance, one may think that it would be difficult to find sufficient idle resources especially on HPC clusters that are often over-committed. However, we argue that a large factor contributing to job waiting time is due to the shape of the queued jobs, as in the example given in Figure 1a. Due to its wide deployment, we present how EJB can work with EASY backfilling [21] and later evaluate its performance.

We now briefly revisit the EASY backfilling algorithm. Each time the scheduling algorithm runs, EASY tries to maximize utilization at that point of time, while only guaranteeing the start time of the first job in the queue. The example in Figure 3a shows at time “now”, three jobs are waiting and the number of available free processors < processors required by the 1st job. EASY first loops over the running
2) Target applications are compute-bound, and not memory-bound. Otherwise, a large memory footprint will prohibit processor over-subscription.

3) In this work, we assume that the underlying batch scheduler runs the EASY backfilling algorithm, without additional priority control policies.

III. EJB SCHEDULING ALGORITHM

EJB runs a heuristic event-driven scheduling algorithm executed at four types of events:

1) TargetJobArrivalEvent: A target job is submitted to the EJB scheduler (EJB-sched).
2) IdleJobSlotsAvailableEvent: New idle job slots become available.
3) SubjobStartEvent: A subjob starts running.
4) TargetAppCompleteEvent: The target application has run to completion.

The time at which the event happens is called $T_{now}$.

A. TargetJobArrivalEvent Handler

When EJB-sched receives a job request $A_t \mapsto J_t = (P_t, R_t)$, EJB-sched first needs to check if the shape $P_t \times R_t$ can be scheduled immediately by the batch scheduler. For this purpose, EJB-sched submits a special subjob $J_0 = (P_t, R_t)$ to the batch scheduler. If $J_0$ starts running immediately, then we are done. Otherwise, EJB-sched creates a new $J_e$ for $A_t$. EJB initializes $J_e$ as follows:

- Status: $J_e.\text{Stat} = \text{WAIT}$;
- Maximum processors needed: $J_e.P_{\text{max}} = P_t$;
- Currently usable processors: $J_e.P_{\text{current}} = 0$;
- List of subjobs: $J_e.\text{SubjobList} = [J_0]$;
- List of available intervals: an interval is a data structure having the following information:
  - Type: $\{\text{RUN}, \text{MIGRATE}\}$
  - Processors - concurrently usable processors,
  - StartTime - when the interval starts,
  - Duration - how long is the interval,
  - SubjobList - subjobs running during the interval,
  - MigrationPlan - valid for MIGRATE interval, initially $J_e.\text{IntervalList} = [\text{empty}]$;
- Current progress of $A_t$: $A_t.\text{Progress} = 0\%$;
- $A_t$’s estimated completion time: $A_t.T_e = \infty$.

$J_0$ functions as a placeholder in the batch queue.

B. IdleJobSlotsAvailableEvent Handler

1) When $J_e.\text{Stat} = \text{WAIT}$: EJB-sched checks whether $S_1$ and/or $S_2$ are big enough to run the entire $A_t$. There are three cases to check when satisfying this condition, as Figure 4 shows: (i) submit one subjob which could fit $S_1$, (ii) submit one subjob which could fit $S_2$, and (iii) submit two subjobs to fit both $S_1$ and $S_2$ respectively. EJB-sched estimates $A_t.T_e$, if feasible, for each of the three cases. EJB-sched will then submit subjobs that produce the shortest
Figure 4: Three cases for subjob submission in a waiting elastic job.

estimated completion time. If none of the three cases are met, then EJB-sched will do nothing.

Case 1: if \( \exists S_1 \) and \( S_1.R_{max} \geq \frac{P}{\sigma_{max}} \), then by submitting subjob \( J_1 = (P_1, L_1) \), \( P_1 = \frac{P}{\sigma_{max}} \), \( L_1 = p_A(P_1) \), \( A_t.T_e = T_{now} + L_1 \).

Case 2: if \( \exists S_2 \) and \( S_2.R_{max} \geq \frac{P}{\sigma_{max}} \), then by running on \( P_1 = \frac{P}{\sigma_{max}} \) processors, \( L_1 = p_A(P_1) \). If \( L_1 < S_2.R_{max} \), then \( J_1 = (P_1, L_1) \), \( A_t.T_e = T_{now} + L_1 \).

Case 3: if (i) \( \exists S_1 \& S_2 \), and (ii) \( S_1.R_{max} \geq \frac{P}{\sigma_{max}} \), and (iii) \( \frac{P}{S_2.R_{max}} > \frac{P}{S_1.R_{max}} \) EJB-sched may simultaneously submit two subjobs such that \( A \) will (i) run on both subjobs in \( L_1 \); (ii) migrate processes from \( J_2 \) to \( J_1 \) in \( L_2 \); (iii) resume in \( L_3 \) while running only on \( J_1 \) such that:

- \( P_1 = \frac{P}{S_1.R_{max}} \),
- \( P_2 = \frac{P}{S_2.R_{max}} - P_1 \),
- \( L_1 = S_2.R_{max} - \lambda \),
- \( L_2 = \lambda \),
- \( L_3 = \frac{100% - \frac{L_1}{p_A(P_1)}}{P_2(P_1 + P_2)} \cdot p_A(P_1) \),
- \( J_1 = (P_1, L_1 + L_2 + L_3) \),
- \( J_2 = (P_2, L_1) \),
- \( J_c.T_e = T_{now} + L_1 + L_2 + L_3 \).

2) When \( J_c.Stat = RUNNING \): EJB-sched checks whether adding more resources to \( J_c \) could advance \( A_t.T_e \). This is not always true because in order to increase speedup after adding more processors, \( J_c \) needs to pay the price of migration. EJB-sched will only decide to allocate more subjobs to \( J_c \) when the benefit outweighs the cost. EJB-sched needs to evaluate at most three possible schedules based on resource availability and the current status of \( J_c \).

Basically, \( J_c.IntervalList \) will be updated with the newly available processors. EJB-sched can then re-estimate the new \( A_t.T_e \) according to the updated \( J_c.IntervalList \):

Case 1: submit a new subjob \( J_x = (P_x, R_x) \) with \( R_x = new A_t.T_e = T_{now} \). This case applies for a type-I available job slot, or a type-II slot when the slot’s \( R_{max} \) is sufficiently long. This case instantly triggers a migration in which processes in existing subjobs are partially migrated to \( J_x \). All the subsequent intervals will increase their \( q_x \) by \( J_x.P \).

Case 2: submit a new subjob \( J_x = (P_x, R_x) \) with \( R_x < new A_t.T_e = T_{now} \). This case applies for a type-II job slot with small \( R_{max} \). Besides triggering an instant expansion migration, this case will also schedule a shrinkage migration before \( J_x \) terminates. The \(_processors \) associated to every interval in \( J_c.IntervalList \) between \( T_{now} \) and \( T_{now} + R_x \) will be incremented by \( q_x \).

Case 3: submit two new subjobs \( J_x = (P_x, R_x) \) and \( J_{x+1} = (P_{x+1}, R_{x+1}) \). \( J_x \) will run until the recalculated completion time and \( J_{x+1} \) will terminate earlier. This case is a combination of cases 1 and 2.

Many fine-grained optimization such as combining/removing migration intervals are considered in our algorithm. For brevity, we omit the how the shapes of \( J_x \) and \( J_{x+1} \) are determined and how \( A_t.T_e \) is recalculated; please refer to [18] for details. Based on the evaluation results in the above cases, EJB-sched will choose the schedule that can produce the earliest completion time. Whenever new resources become available, EJB-sched will call this event handler unless \( J_c \) has reached full parallelism \( J_c.P_{max} \).

C. SubjobStartEvent Handler

When a subjob starts, EJB-sched performs process migration as scheduled. However, if the place holder job \( J_0 \) starts, based on \( A \)’s current progress, EJB-sched has the options of (i) migrating all running processes to \( J_0 \); or (ii) continuing execution on existing subjobs and cancel \( J_0 \); or (iii) restarting \( A_t \) on \( J_0 \) and discarding currently achieved progress. EJB-sched will choose the option which can produce earliest completion time.

D. TargetAppCompleteEvent Handler

When the application terminates earlier than the projected finish time \( A_t.T_e \), EJB-sched will cancel all running subjobs. EJB-sched will also cancel \( J_0 \) if it is still in queue.

IV. EJB System Design
EJB system can submit job requests to EJB-sched through an interface similar to the batch submission. EJB-sched runs the scheduling algorithm described in last section. EJB-sched interacts with the batch scheduler only through ordinary calls such as show job queue status, submit job, and cancel job. In order to control and manage the elastic job, scheduling operations for all elastic jobs are placed on a Operation Queue. There are two types of Operations: launch which submits the application with a computed degree of over-subscription, and migrate-dest which decides two things: the group of processes that will be migrated, and the destination subjob that will receive the processes. The EJB Controller is in charge of sending these operations to the EJB Workers running in each subjob. This mode of operation can be seen as similar to the pilot job [19], in which a resource is first acquired by a pilot job, and then tasks are scheduled into that resource. In our case, when a subjob starts, the EJB worker will direct the target application to perform the scheduled operations in that subjob. There will be only a small number of messages sent between the EJB Controller and EJB Worker throughout an elastic job’s lifecycle.

To enable both over-subscription and migration, we advocate the use of lightweight application-level virtualization as in [24], [25] depicted in Figure 6. The key point is the type-2 hypervisor which runs as a user process on top of the OS. The benefit of the type-2 hypervisor is that the cluster owner does not need to virtualize the cluster as is required by a type-1 hypervisor, which runs on the bare machine. The EJB agent can launch a type-2 hypervisor on each compute node allocated to the subjob the EJB worker resides on. On top of each hypervisor, each process will be launched in a separate VM, which has its own OS. The hypervisor will be in charge of over-subscription. The EJB worker will set up a virtual network layer on top of all hypervisors of an elastic job. The network layer is a thin layer which only maps from the logical id of a process to the physical location within a compute node.

When migration happens, the hypervisor will pause the VMs that will be moved. Since all compute nodes have a shared file system, the VM image will be automatically synced among all compute nodes. The hypervisor on the destination node will simply resume the VM to complete the migration procedure.

V. TRACE-DRIVEN SIMULATION

Before implementing our proposed EJB system in a live setting, we first wanted to establish the benefits of this approach. To do this, we simulated EJB using logs of real parallel workloads from production systems [2] to assess feasibility. Table II lists the 4 selected traces used by our simulation. These 4 traces have been widely used by previous studies of parallel job scheduling algorithms.

Our simulator is based on PYSS [3] – an event-based scheduler simulator developed by the Parallel Systems Lab at Hebrew University. In order to emulate how EJB works in practice, the simulator’s EasyBackfillScheduler which functions as cluster batch scheduler is kept unchanged. Job traces contain both job walltime and runtime. The former is user estimated run length. The latter is the application’s actual run length recorded after it terminates, walltime $\geq$ runtime. In simulation, the job’s actual runtime is unknown to EJB-sched. EJB-sched calculates the projected completion time based on the job’s walltime. However, the simulator keeps track of the actual progress based on the runtime, and triggers the TargetAppCompleteEvent once the actual progress is 100% (see Equation 3).

In theory, any job in the trace can be submitted to EJB-sched. Nevertheless, jobs requesting only a few processors...
cannot be further optimized through over-subscription. If they experience long waiting time, it can be an indication of truly high system workload and our approach cannot find free slots under this condition. We set the minimum $P$ of an eligible elastic target job to be 8. We set the following default values: the maximum degree of over-subscription, $O_{max} = 8$, and the migration duration, $\lambda = 120(s)$.

Furthermore, each trace’s first 1% jobs, as well as the jobs that terminate after last job arrival are excluded from the performance analysis. This is a commonly used technique to reduce the impact of warm up and cool down effects.

VI. EVALUATION
We evaluate EJB through a series of experiments based on simulation. Our baseline for comparison is a system scheduler that runs EASY Backfilling only. Overall, the results reveal the following performance benefits of EJB:

- elastic job performance is significantly improved;
- non-elastic job performance is either not impacted or slightly improved;
- system fragmentation is reduced;
- fairness between jobs of different sizes is promoted.

We start by carefully choosing the appropriate performance metrics (VI-A). We then measure how elastic jobs are improved (VI-B). We then evaluate how migration cost impacts elastic job performance (VI-C). Finally, we study the cluster-wide performance when co-scheduling many elastic jobs together (VI-D).

A. Performance Metrics

The elastic job’s turnaround time ($tt$) is the time when the target job is submitted to EJB-sched to the point when the target application completes, which is also when all subjobs terminate. When dividing the elastic job’s $tt$ by the baseline $tt$, we have the speedup of turnaround time:

$$S_{tt} = \frac{\text{baseline } tt}{\text{elastic job } tt}$$

(4)

The elastic job’s waiting time ($tw$) is measured from the target job’s submission to the start of the first subjob of the elastic job. The elastic job’s run time ($tr$) is measured from the time the first subjob belonging to the elastic job starts, to the time the elastic job’s last subjob terminates. Elastic job’s bounded slowdown ($Slo$) is defined as

$$Slo = \frac{\text{elastic job } tt}{\text{baseline } tr}$$

(5)

Notice that we don’t use elastic job $tr$ in calculating slowdown, for the reason that slowdown should be compared against the runtime on a dedicated system, without over-subscription and migration. Bounded-slowdown substitutes a job’s baseline $tr$ with 10s when $tr \leq 10s$. Bounded-slowdown avoids super-short jobs generating very large slowdown values.

We measure system fragmentation as the average number of idle processors in the cluster while the batch queue is not empty. This measurement excludes the period when all the jobs in the cluster have received resources, yet there are still unallocated processors. For example, if jobs never wait, then system fragmentation will always be 0, independent of idle processors. As another example, if the scheduler is able to perfectly fill all resources with jobs, then the system fragmentation is also 0.

B. Improving Elastic Job Turnaround Time

As a first step towards evaluating EJB’s performance, we isolate EJB’s impact on a single target job by simulating one elastic job in each run of our simulator. We then compare the elastic job performance against the baseline of the target job. Target jobs are all the jobs which have $P \geq 8$ and baseline $tw > 0$. Note: we do not need to know $tw$ accurately but simply whether it is non-zero.

We can know this if $J_0$ starts immediately. As Table III shows, we simulated $\geq 100,000$ such jobs in four traces combined. Figure 7a provides a clearer visual view of the how turnaround time has been improved.

Elastic jobs’ mean $tt$ is 39% faster than the baseline value, with variations between traces. As expected, the EJB results in significantly shorter $tw$ (61.4% lower) at the expense of longer $tr$ (58.6% higher) due to over-subscription and migration. Detailed distributions of $S_{tt}$ are depicted in Figure 7b which shows that most target jobs benefit from being elastic. Some exceptionally well-performing jobs have $tt$ 1,000 times faster than before. $1/4$ of the target jobs’ $tt$ are unchanged and < 3% of the elastic jobs experience worse results.

Next, we perform sensitivity analysis to $O_{max}$ and $\lambda$. Figure 8 shows the results of the CTC trace only, as other traces reveal similar trends. First, in Figure 8a, we vary $O_{max} \in [2, 4, 8, 16, 32, 64]$. The larger $O_{max}$, the greater the benefit of EJB. After $O_{max}$ has reached 16, further increasing $O_{max}$ won’t bring evident performance gain.

Second, we evaluate whether the performance improvements are sensitive to the migration cost. In Figure 8b, we vary $\lambda$ from 1 second, up to 10 minutes. To our surprise, the performance gain is not very sensitive to the migration.
Table IV: Summarize migration related statistics.

<table>
<thead>
<tr>
<th></th>
<th>subjobs</th>
<th>migrations</th>
<th>migration duration</th>
<th>resource overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTC</td>
<td>3.3</td>
<td>2.1</td>
<td>8.5%</td>
<td>19.7%</td>
</tr>
<tr>
<td>SDSC</td>
<td>2.8</td>
<td>1.7</td>
<td>5.6%</td>
<td>16.4%</td>
</tr>
<tr>
<td>BLUE</td>
<td>2.7</td>
<td>1.5</td>
<td>8.6%</td>
<td>18.2%</td>
</tr>
<tr>
<td>KTH</td>
<td>2.7</td>
<td>1.6</td>
<td>6.4%</td>
<td>24.1%</td>
</tr>
</tbody>
</table>

On average each elastic job consists of about 3 subjobs, and conducts bulk cross-subjob migrations approximately twice. Actually, more than 60% of the elastic jobs contain more than one subjob, and around 40% of the elastic jobs have experienced at least one bulk migration. In very rare cases, the number of subjobs and migrations can reach > 20. This shows that the performance gain of EJB is not only a result of moldability, but also the result of migrations.

The migration duration column shows that migration durations on average account for 5 – 8% of an elastic job’s run time. Furthermore, extra CPU resources may be spent due to migration and over-subscription. The resource overhead column shows that elastic jobs have a 16 – 24% resource overhead which is measured by processor × time. A main factor contributing to this is the inaccuracy in the tr estimations. Based on user provided trs, the EJB algorithm may decide that it is beneficial to perform additional migrations. However, the real tr of these elastic jobs are much shorter, such that the migrations may be unnecessary. To address this issue in our future work, we can use the similar approach of [30] to more accurately estimate tr according to historical job information and make migration decisions based on the adjusted tr.

D. Multiple Elastic Jobs

In [VI-B] we analyzed how EJB impacts single job performance. In this section, we try to understand the comprehensive performance impact when many elastic jobs coexist, in effect competing for resources with each other and with other non-elastic jobs. The following simulations are meant to emulate real-world conditions when users arbitrarily submit job requests to EJB-sched.

The impacts of EJB are measured on: (i) elastic jobs, (ii) non-elastic jobs, and (iii) all jobs. The impact determined by measuring how jobs perform differently after introducing EJB can be tricky. Since for each separate job, its performance in terms of tt or tr can be largely dependent on background workload during its tt. From a single job’s perspective, its background workload can be totally different if EJB were to be deployed.

We solve this dilemma by applying a statistical method called Before-and-After Comparisons [17]. The Before-and-After comparison is designed to evaluate whether by adding some new features to a system, the performance change is statistically significant. In our context, the method works in

C. Migration behavior

Table IV characterizes elastic job overhead with respect to the number of migrations. The subjobs column shows that on average each elastic job consists of about 3 subjobs, and conducts bulk cross-subjob migrations approximately twice. Actually, more than 60% of the elastic jobs contain more than one subjob, and around 40% of the elastic jobs have experienced at least one bulk migration. In very rare cases, the number of subjobs and migrations can reach > 20. This shows that the performance gain of EJB is not only a result of moldability, but also the result of migrations.

The migration duration column shows that migration durations on average account for 5 – 8% of an elastic job’s run time. Furthermore, extra CPU resources may be spent due to migration and over-subscription. The resource overhead column shows that elastic jobs have a 16 – 24% resource overhead which is measured by processor × time. A main factor contributing to this is the inaccuracy in the tr estimations. Based on user provided trs, the EJB algorithm may decide that it is beneficial to perform additional migrations. However, the real tr of these elastic jobs are much shorter, such that the migrations may be unnecessary. To address this issue in our future work, we can use the similar approach of [30] to more accurately estimate tr according to historical job information and make migration decisions based on the adjusted tr.

D. Multiple Elastic Jobs

In [VI-B] we analyzed how EJB impacts single job performance. In this section, we try to understand the comprehensive performance impact when many elastic jobs coexist, in effect competing for resources with each other and with other non-elastic jobs. The following simulations are meant to emulate real-world conditions when users arbitrarily submit job requests to EJB-sched.

The impacts of EJB are measured on: (i) elastic jobs, (ii) non-elastic jobs, and (iii) all jobs. The impact determined by measuring how jobs perform differently after introducing EJB can be tricky. Since for each separate job, its performance in terms of tt or tr can be largely dependent on background workload during its tt. From a single job’s perspective, its background workload can be totally different if EJB were to be deployed.

We solve this dilemma by applying a statistical method called Before-and-After Comparisons [17]. The Before-and-After comparison is designed to evaluate whether by adding some new features to a system, the performance change is statistically significant. In our context, the method works in

C. Migration behavior

Table IV characterizes elastic job overhead with respect to the number of migrations. The subjobs column shows that on average each elastic job consists of about 3 subjobs, and conducts bulk cross-subjob migrations approximately twice. Actually, more than 60% of the elastic jobs contain more than one subjob, and around 40% of the elastic jobs have experienced at least one bulk migration. In very rare cases, the number of subjobs and migrations can reach > 20. This shows that the performance gain of EJB is not only a result of moldability, but also the result of migrations.

The migration duration column shows that migration durations on average account for 5 – 8% of an elastic job’s run time. Furthermore, extra CPU resources may be spent due to migration and over-subscription. The resource overhead column shows that elastic jobs have a 16 – 24% resource overhead which is measured by processor × time. A main factor contributing to this is the inaccuracy in the tr estimations. Based on user provided trs, the EJB algorithm may decide that it is beneficial to perform additional migrations. However, the real tr of these elastic jobs are much shorter, such that the migrations may be unnecessary. To address this issue in our future work, we can use the similar approach of [30] to more accurately estimate tr according to historical job information and make migration decisions based on the adjusted tr.

D. Multiple Elastic Jobs

In [VI-B] we analyzed how EJB impacts single job performance. In this section, we try to understand the comprehensive performance impact when many elastic jobs coexist, in effect competing for resources with each other and with other non-elastic jobs. The following simulations are meant to emulate real-world conditions when users arbitrarily submit job requests to EJB-sched.

The impacts of EJB are measured on: (i) elastic jobs, (ii) non-elastic jobs, and (iii) all jobs. The impact determined by measuring how jobs perform differently after introducing EJB can be tricky. Since for each separate job, its performance in terms of tt or tr can be largely dependent on background workload during its tt. From a single job’s perspective, its background workload can be totally different if EJB were to be deployed.

We solve this dilemma by applying a statistical method called Before-and-After Comparisons [17]. The Before-and-After comparison is designed to evaluate whether by adding some new features to a system, the performance change is statistically significant. In our context, the method works in

C. Migration behavior

Table IV characterizes elastic job overhead with respect to the number of migrations. The subjobs column shows that on average each elastic job consists of about 3 subjobs, and conducts bulk cross-subjob migrations approximately twice. Actually, more than 60% of the elastic jobs contain more than one subjob, and around 40% of the elastic jobs have experienced at least one bulk migration. In very rare cases, the number of subjobs and migrations can reach > 20. This shows that the performance gain of EJB is not only a result of moldability, but also the result of migrations.

The migration duration column shows that migration durations on average account for 5 – 8% of an elastic job’s run time. Furthermore, extra CPU resources may be spent due to migration and over-subscription. The resource overhead column shows that elastic jobs have a 16 – 24% resource overhead which is measured by processor × time. A main factor contributing to this is the inaccuracy in the tr estimations. Based on user provided trs, the EJB algorithm may decide that it is beneficial to perform additional migrations. However, the real tr of these elastic jobs are much shorter, such that the migrations may be unnecessary. To address this issue in our future work, we can use the similar approach of [30] to more accurately estimate tr according to historical job information and make migration decisions based on the adjusted tr.

D. Multiple Elastic Jobs

In [VI-B] we analyzed how EJB impacts single job performance. In this section, we try to understand the comprehensive performance impact when many elastic jobs coexist, in effect competing for resources with each other and with other non-elastic jobs. The following simulations are meant to emulate real-world conditions when users arbitrarily submit job requests to EJB-sched.

The impacts of EJB are measured on: (i) elastic jobs, (ii) non-elastic jobs, and (iii) all jobs. The impact determined by measuring how jobs perform differently after introducing EJB can be tricky. Since for each separate job, its performance in terms of tt or tr can be largely dependent on background workload during its tt. From a single job’s perspective, its background workload can be totally different if EJB were to be deployed.

We solve this dilemma by applying a statistical method called Before-and-After Comparisons [17]. The Before-and-After comparison is designed to evaluate whether by adding some new features to a system, the performance change is statistically significant. In our context, the method works in

C. Migration behavior

Table IV characterizes elastic job overhead with respect to the number of migrations. The subjobs column shows that on average each elastic job consists of about 3 subjobs, and conducts bulk cross-subjob migrations approximately twice. Actually, more than 60% of the elastic jobs contain more than one subjob, and around 40% of the elastic jobs have experienced at least one bulk migration. In very rare cases, the number of subjobs and migrations can reach > 20. This shows that the performance gain of EJB is not only a result of moldability, but also the result of migrations.

The migration duration column shows that migration durations on average account for 5 – 8% of an elastic job’s run time. Furthermore, extra CPU resources may be spent due to migration and over-subscription. The resource overhead column shows that elastic jobs have a 16 – 24% resource overhead which is measured by processor × time. A main factor contributing to this is the inaccuracy in the tr estimations. Based on user provided trs, the EJB algorithm may decide that it is beneficial to perform additional migrations. However, the real tr of these elastic jobs are much shorter, such that the migrations may be unnecessary. To address this issue in our future work, we can use the similar approach of [30] to more accurately estimate tr according to historical job information and make migration decisions based on the adjusted tr.

D. Multiple Elastic Jobs

In [VI-B] we analyzed how EJB impacts single job performance. In this section, we try to understand the comprehensive performance impact when many elastic jobs coexist, in effect competing for resources with each other and with other non-elastic jobs. The following simulations are meant to emulate real-world conditions when users arbitrarily submit job requests to EJB-sched.

The impacts of EJB are measured on: (i) elastic jobs, (ii) non-elastic jobs, and (iii) all jobs. The impact determined by measuring how jobs perform differently after introducing EJB can be tricky. Since for each separate job, its performance in terms of tt or tr can be largely dependent on background workload during its tt. From a single job’s perspective, its background workload can be totally different if EJB were to be deployed.

We solve this dilemma by applying a statistical method called Before-and-After Comparisons [17]. The Before-and-After comparison is designed to evaluate whether by adding some new features to a system, the performance change is statistically significant. In our context, the method works in
Figure 9: Decision tree for elastic job selection

this way: for each workload, we run the simulation twice before and after involving EJB. Then for each performance metric, we have a pair of results corresponding to each job’s before and after case. Next, we calculate a confidence interval for the means of the differences of each paired value. If this confidence interval does not include zero, then we can conclude with a certain confidence that there is a statistically significant difference before and after introducing EJB.

First, we simulate an extreme condition by submitting all jobs that request at least 8 processors to EJB-sched. Table V shows the Before-and-After comparison results. Notice that the number of elastic jobs are different from that of Table III. We use the decision tree in Figure 9 to determine which jobs will become elastic.

All jobs submitted to EJB-sched are candidate jobs. EJB-sched only transforms a candidate elastic job to an elastic job when the job’s original shape cannot be started immediately. The increase in the number of elastic jobs (e.g. the number of elastic jobs in CTC has increased from 16,167 in Table III to 21,035 in Table V) indicates that when we saturate the cluster with elastic jobs, a greater number of jobs are identified as eligible for elasticity. The reason is that the mutual influence between elastic jobs causes more jobs that were inelastic originally because \( \text{tw} = 0 \), to now become elastic. However, the mean turnaround time of elastic jobs is significantly reduced.

Table V shows that for all the workloads except KTH, wide use of EJB not only results in shorter \( \text{tt} \) for elastic jobs, but surprisingly improves the response time of non-elastic jobs, and the improvement is statistically significant. For KTH, elastic jobs are also significantly faster than before. Non-elastic jobs in KTH are on average 0.4% slower after EJB is applied. However this performance degradation is not of statistical significance since its confidence interval (−56, 141) crosses 0.

The performance results measured by bounded slowdown shown in Figure 10 are consistent with the turnaround time results such as in Table V (column 5). The maximum slowdown (which is too large to be shown in the graph) experienced by the most unlucky job also decreases. Table VI shows the measurement of fragmentation as defined in VI-A. The result shows that with EJB, average system utilization is higher when there are jobs in the queue which indicates EJB uses the idle processors to help queueing jobs start sooner.
Table V: Before-and-after comparison: confidence intervals are calculated at the 95% confidence level.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Job type</th>
<th>Mean tt</th>
<th>Mean tw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean tt</td>
<td>Mean tw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>change</td>
</tr>
<tr>
<td></td>
<td>conf. interval</td>
<td>conf. interval</td>
<td></td>
</tr>
<tr>
<td>CTC</td>
<td>elastic</td>
<td>21,035</td>
<td>30,276</td>
</tr>
<tr>
<td></td>
<td>non-elastic</td>
<td>59,123</td>
<td>19,268</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>76,446</td>
<td>25,049</td>
</tr>
<tr>
<td>SDSC</td>
<td>elastic</td>
<td>18,790</td>
<td>58,235</td>
</tr>
<tr>
<td></td>
<td>non-elastic</td>
<td>40,333</td>
<td>10,589</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>59,123</td>
<td>25,731</td>
</tr>
<tr>
<td>BLUE</td>
<td>elastic</td>
<td>135,302</td>
<td>16,863</td>
</tr>
<tr>
<td></td>
<td>non-elastic</td>
<td>105,560</td>
<td>4,096</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>240,862</td>
<td>11,268</td>
</tr>
<tr>
<td>KTH</td>
<td>elastic</td>
<td>5,811</td>
<td>32,457</td>
</tr>
<tr>
<td></td>
<td>non-elastic</td>
<td>22,392</td>
<td>11,523</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>28,203</td>
<td>15,836</td>
</tr>
</tbody>
</table>

Table VI: Fragmentation: np is the average number of idle processors, % is the percentage of the idle processors in the cluster.

<table>
<thead>
<tr>
<th>Trace</th>
<th>np</th>
<th>%</th>
<th>np</th>
<th>%</th>
<th>changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTC</td>
<td>22</td>
<td>6.6%</td>
<td>22</td>
<td>6.6%</td>
<td>-34.0%</td>
</tr>
<tr>
<td>SDSC</td>
<td>5,9</td>
<td>4.6%</td>
<td>5,9</td>
<td>4.6%</td>
<td>-57.4%</td>
</tr>
<tr>
<td>BLUE</td>
<td>5,9</td>
<td>4.6%</td>
<td>5,9</td>
<td>4.6%</td>
<td>-58.0%</td>
</tr>
<tr>
<td>KTH</td>
<td>5,9</td>
<td>4.6%</td>
<td>5,9</td>
<td>4.6%</td>
<td>-58.8%</td>
</tr>
</tbody>
</table>

Table VII: Before-and-after comparison: confidence intervals are calculated at the 95% confidence level.

Figure 12: Changing utilization: EJB is more resistant under high utilization.

Finally, in Figure 12, we measure EJB performance by synthetically decreasing/increasing system utilization through changing job's arrival rate. From the results we can see when cluster utilization is low, EJB performs similar to batch scheduling. However, in clusters with high utilization, EJB performs significantly better.

VII. DISCUSSION

We have shown that EJB reduces large job turnaround time with minimal impact on small jobs. We attempt to explain this interesting phenomenon as EJB is, in effect, homogenizing system workload, by decomposing large jobs into smaller ones. Compared to larger jobs, smaller jobs can allow schedulers to allocate resources more quickly and improve load balance [23]. Ultimately the performance improvement comes from reduced system fragmentation. When workload is high, EJB lowers the average size of jobs. When workload is low, EJB generates additional jobs to exploit idle resources.

Another point worth discussion is: On an EJB-ready HPC cluster, when should EJB be activated? Our view is that EJB can be dynamically switched on/off according to system workload. Users can be given the option of specifying whether they would like to pay a little bit more resource quota in return for faster turnaround time. When the batch queue length has exceeded a certain threshold, the administrator could decide to enable EJB to reduce wait time.

VIII. RELATED WORK

Characterized by different patterns of resource usage, parallel jobs are categorized in three types. Rigid jobs require a fixed number of processors. Moldable jobs can be executed on several processor sizes. The actual number of processors is determined at the start, and never changes. Malleable jobs may change the number of processors during execution. Bringing flexibility to parallel jobs to adapt them to system workload has been extensively studied. The essentials of these studies are twofold. First, is the mechanism to allow parallel jobs to use different number of processors. Second, is the scheduling strategies used such as moldability or malleability. This section will briefly compare EJB with several representative approaches.

A. Moldable Jobs

Cirne’s works in [7], [8] rely on applications to be moldable and job waiting time to be predictable to improve moldable job turnaround time. It chooses the job size based on which size might produce the shortest $t_w + t_r$. The merit of this approach is that it requires no system changes. Nonetheless, estimating job waiting time can be very error-prone. Also, many applications are not moldable, e.g. some...
applications can only be decomposed into restricted degrees of parallelism such as powers of two. Moreover, by definition moldable jobs can not grow to a larger resource footprint to gain further speedup even when free resources become available after the moldable job starts running.

Commercial cluster schedulers like Moab support moldable job requests, in which the user provides several options for job size and walltime. The scheduler will choose an option based on whichever option can be met first. Basically, this is similar to our approach but the application must be moldable and migration to enable expansion of parallelism is not supported. We have evaluated this situation by setting migration cost to infinity, and the performance was shown to be inferior to EJB due to lack of adaptation to additional resources.

B. Malleable Jobs

Malleable (or adaptive) jobs have the attractive property that they dynamically adapt to system workload [32]. ReSHAPE [28], 29] is a framework supporting dynamically changing the number of processors of iterative, structured (2-D decomposition) applications, for the purpose of both selecting the best number of processors to yield the best efficiency by expanding/shrinking the processor size according to the system workload. The merit of their work is a implementation of a library which is capable of dynamically mapping data to different number of processors. The user of their approach needs to insert primitives into the code to indicate a resizing point. Our approach does not require application modification. Tightly-coupled malleable applications are difficult to implement, and require runtime support at the system level. Utrer et al. [31] proposed a job scheduling strategy based on virtual malleability: processes within the same node can be over-subscribed to use fewer processors, such that free processors could be allocated to queued jobs. However, their approach is based on the assumption that they can deploy their own scheduler to control the cluster, while our approach does not require any change to the system scheduler. Also, the migration within a node approach can not expand a running application to other available physical nodes.

C. Virtualization

Our work shares some of the goals of HPC virtualization projects [12], [15], [20], but at the same time complying with the existing batch scheduling framework. This hybrid use model of HPC shares similar goals as cloud-bursting [5]. However, instead of bursting to the cloud, EJB exploits additional resources internally.

IX. CONCLUSION

We have presented elastic job bundling (EJB), a new resource allocation strategy for large parallel applications. EJB decouples the one-to-one binding between parallel applications and jobs, such that one application can run simultaneously on multiple smaller jobs. By transforming one large job into multiple smaller ones, faster turnaround time is possible especially on HPC clusters with high workload. We simulated our algorithm using real-world job traces and show that EJB can (i) reduce target job’s mean turnaround time by up to 48%, (ii) reduce system-wide mean job turnaround time by up to 27%, and (iii) reduce system fragmentation by up to 59%. We have also presented an implementation architecture that can realize this approach. We have made the EJB code available on github [1], such that anyone interested can obtain and use the complete algorithm code and reproduce our experimental results.

X. ACKNOWLEDGMENTS

This work is sponsored by the Department of Energy under the AIMES (Abstractions and Integrated Middleware for Extreme-Scale Science) project (DE-FG02-12ER26115, DE-SC0008617, DE-SC0008651). Thanks to the participants of the AIMES project: Shantenu Jha, Matteo Turilli, Andre Merzky, Daniel S. Katz, Michael Wilde, Zhao Zhang, and Yadu Nand Babuji.

REFERENCES

[1] https://bitbucket.org/francis_liu/pyss