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

Feng Liu
University of Minnesota
Minneapolis, MN
fengl@cs.umn.edu

Jon B. Weissman
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%.

CCS Concepts

•**Software and its engineering** → **Middleware; Massively parallel systems;**

Keywords

parallel job scheduling, HPC, elasticity

1. 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 [6]. 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 [9,10,28], 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).

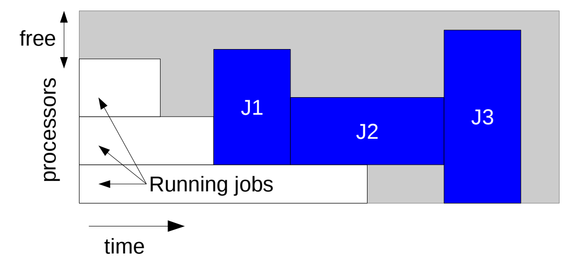
We evaluate our approach using real-world workloads. Preliminary results reveal that our approach can:

- 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%.

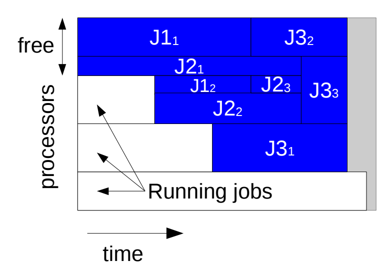
SC '15, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3723-6/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807610

2. ELASTIC JOB BUNDLING (EJB)

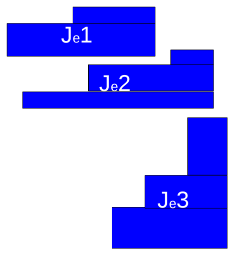
Elastic job bundling (EJB) is a software layer that operates between parallel application end-users and HPC batch systems. The goal of EJB is to reduce the turnaround time of parallel applications, especially those that demand a large number of processors. EJB accepts ordinary job requests and transforms them into multiple smaller subjobs which



(a) Rigid monolithic jobs experience long waiting time: Job priority $J1 > J2 > J3$.



(b) subjob decomposition: Jx_y are subjobs decomposed from monolithic job Jx.



(c) Elastic job composition: $J_e x$ is the elastic job corresponsing to monolithic job Jx.

Figure 1: Illustration of elastic jobs.

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.

2.1 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 $A \mapsto J = (P, R)$. 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*, $A_t \mapsto J_t = (P_t, R_t)$. 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, $A_t \mapsto J_e$, 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*: $J_e = (J_1, J_2, \ldots, J_n)$.

2.2 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 over-subscription and migration. Thus, the application structure or logic need not change.

Given $A_t \mapsto J_t = (P_t, R_t)$, 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,

$$p_{A_t}(P_t) = R_t \tag{1}$$

Suppose A_t is compute-bound with balanced workload which is typical of many SPMD applications. Under over-subscription, A_t is run on q processors, $q < P_t$. Under an even distribution, each processor is time-shared by up to $\left\lceil \frac{P_t}{q} \right\rceil$ 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 proportional to $\left\lceil \frac{P_t}{q} \right\rceil$, such that:

$$p_{A_t}(q) = \Delta \cdot \left\lceil \frac{P_t}{q} \right\rceil \cdot R_t \tag{2}$$

Where Δ is a penalty factor that models the severity of performance degradation due to over-subscription. In the ideal case, a $\Delta = 1$ indicates that the performance degradation is linearly proportional to the degree of over-subscription.

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 $q \in [\left\lceil \frac{P_t}{O_{max}} \right\rceil, P_t]$. 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_x is added to J_e, EJB migrates a subset of A_t's running processes to J_x'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 the migrated processes are moved concurrently. A_t stops making progress during migration intervals. We first assume that each bulk migration interval has a fixed maximum duration

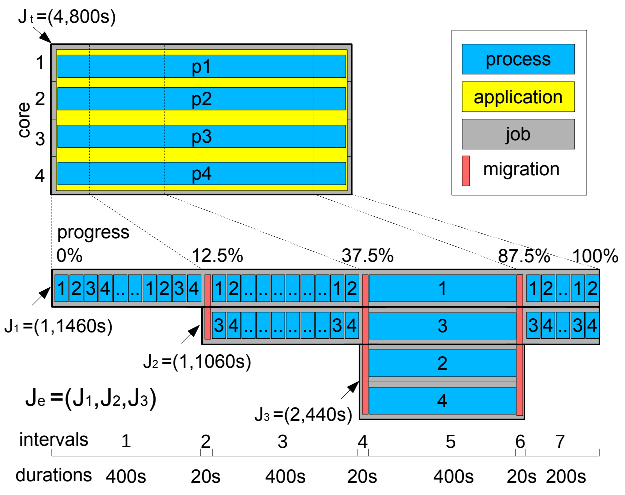


Figure 2: Mapping a parallel application's processes to an elastic job, including progress measurement.

of λ seconds. To evaluate the impact of migration cost, we can vary λ.

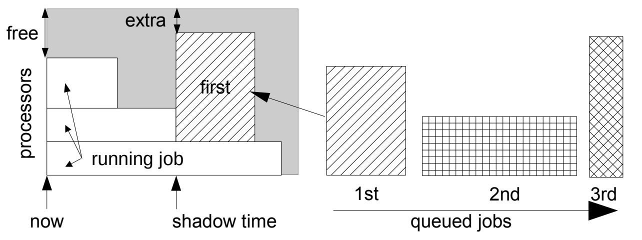
J_e has two types of intervals: RUN and MIGRATE. Suppose there are l intervals in J_e. In interval k ($k = 1, \ldots, l$), J_e has q_k concurrently usable processors and interval length$=L_k$. Now we model A_t's progress on J_e as:

$$\sum_k \frac{L_k}{p_{A_t}(q_k)} = 100\% \qquad (3)$$
$$where\ k = 1, \ldots, l\ and\ k.type = RUN$$

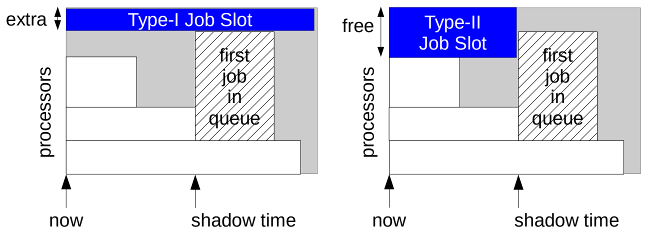
Equation 3 can be understood as follows: the completion of A_t requires that the accumulated progress made by every interval sums to 100%. Based on Equation 3, we can: (i) estimate A_t's progress at any point during it's run time, (ii) estimate the time it takes to achieve a certain amount of progress, 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 3 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.

2.3 Taming Unpredictability



(a) EASY backfilling algorithm.



(b) Immediately backfillable job slots: type-I and type-II

Figure 3: Finding immediately usable resources under EASY backfilling.

Table 1: Upper-bounds of processor (P_{max}) and runtime (R_{max}) of two types of immediately backfillable job slots.

	P_{max}	R_{max}
Type-I slot	*extra* processors	none
Type-II slot	*free* processors (*free* > *extra*)	$T_{shadow} - T_{now}$

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 [8, 11, 12, 16, 21, 23, 27, 31]. We address this challenge by controlling the shape $P \times R$ of the subjobs such that they can be immediately scheduled to run on the fragmented idle resources. E.g., production schedulers such as TORQUE [24] or SLURM [32] 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 [20] 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 jobs in the order of their expected termination time, until the available processors are sufficient for the 1st job, when the 1st job is guaranteed to start. EASY calls this time the shadow time T_{shadow}. If the available processors at $T_{shadow} >$ processors required by the 1st job, the surplus processors are *extra*. As a second step, EASY finds backfillable jobs according to the condition that they do not delay the 1st job. In our example, both the 2nd and 3rd job do not satisfy this condition, so they will keep waiting. If any lower-priority job satisfies the backfill condition, they will be selected as backfill jobs to start immediately, and they may add unbounded delay to the 2nd and 3rd job in our example.

Figure 3b shows the upper-bound in both processor and time dimensions of the shape of backfillable jobs. Table 1 lists the upper-bounds, which can be spatially imagined as slots with height=P_{max} and width=R_{max}, which we call *immediately backfillable job slots*. There are two types of immediately backfillable job slots. The P_{max} in a *type-I* slot = *extra* processors. *Type-I* slot has no upper-bound for R_{max}. The P_{max} in a *Type-II* slot = *free* processors. Simply speaking, jobs submitted to fill the *type-I* slot can run on a smaller number of processors with unlimited runtime. Jobs submitted to fill the *type-II* slot can run on a larger number of processors, but with limited runtime.

2.4 Assumptions and Limitations

In summary, we made the following assumptions for EJB:

1. EJB targets the optimization of large tightly-coupled (such as MPI) parallel applications. Embarrassingly parallel, or bags of tasks are comparatively easier to schedule, since they do not require co-scheduled subjobs, nor cross-subjob migrations.
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.

3. 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}.

3.1 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: $J_e.Stat = WAIT$;

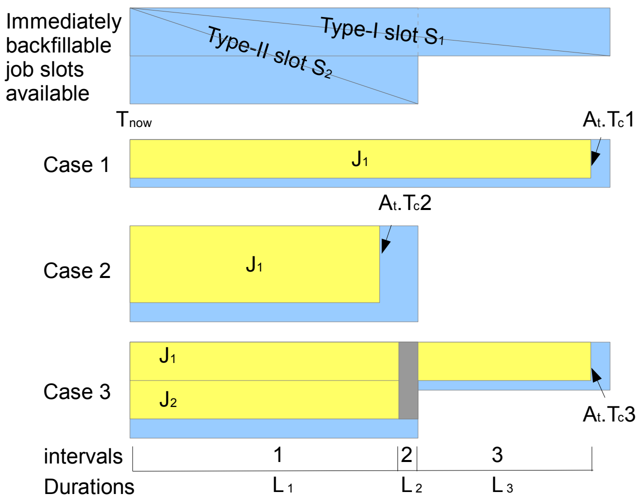


Figure 4: Three cases for subjob submission in a waiting elastic job.

- Maximum processors needed: $J_e.P_{max} = P_t$;
- Currently usable processors: $J_e.P_{current} = 0$;
- List of subjobs: $J_e.SubjobList = [J_0]$;
- List of available intervals: an *interval* is a data structure having the following information:
  - *Type* - $[RUN|MIGRATE]$,
  - *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 $J_e.IntervalList = [empty]$;
- Current progress of A_t: $A_t.Progress = 0\%$;
- A_t's estimated completion time: $A_t.T_c = \infty$.

J_0 functions as a place holder in the batch queue.

3.2 IdleJobSlotsAvailableEvent Handler

3.2.1 When $J_e.Stat = WAIT$

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 $A_t.T_c$, if feasible, for each of the three cases. EJB-sched will then submit subjobs that produce the shortest estimated completion time. If none of the three cases are met, then EJB-sched will do nothing.

Case 1: if $\exists S_1$ and $S_1.P_{max} \geq \frac{P_t}{O_{max}}$, then by submitting subjob $J_1 = (P_1, L_1)$, $P_1 = \frac{P_t}{\lceil \frac{P_t}{S_1.P_{max}} \rceil}$, $L_1 = p_A(P_1)$, $A_t.T_c = T_{now} + L_1$.

Case 2: if $\exists S_2$ and $S_2.P_{max} \geq \frac{P_t}{O_{max}}$, then by running on $P_1 = \frac{P_t}{\lceil \frac{P_t}{S_2.P_{max}} \rceil}$ processors, $L_1 = p_A(P_1)$. If $L_1 < S_2.R_{max}$, then $J_1 = (P_1, L_1)$, $A_t.T_c = T_{now} + L_1$.

Case 3: if (i) $\exists$ *both* $S_1 \& S_2$, and (ii) $S_1.P_{max} \geq \frac{P_t}{O_{max}}$, and (iii) $\frac{P_t}{\lceil \frac{P_t}{S_2.P_{max}} \rceil} > \frac{P_t}{\lceil \frac{P_t}{S_1.P_{max}} \rceil}$ 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:

- $P_1 = \frac{P_t}{\lceil \frac{P_t}{S_1.P_{max}} \rceil}$,
- $P_2 = \frac{P_t}{\lceil \frac{P_t}{S_2.P_{max}} \rceil} - P_1$,
- $L_1 = S_2.R_{max} - \lambda$,
- $L_2 = \lambda$,
- $L_3 = (100\% - \frac{L_1}{p_A(P_1+P_2)}) \cdot p_A(P_1)$,
- $J_1 = (P_1, L_1 + L_2 + L_3)$,
- $J_2 = (P_2, L_1)$,
- $A_t.T_c = T_{now} + L_1 + L_2 + L_3$.

3.2.2 When $J_e.Stat = RUNNING$

EJB-sched checks whether adding more resources to J_e could advance $A_t.T_c$. This is not always true because in order to increase speedup after adding more processors, J_e needs to pay the price of migration. EJB-sched will only decide to allocate more subjobs to J_e 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_e. Basically, $J_e.IntervalList$ will be updated with the newly available processors. EJB-sched can then re-estimate the new $A_t.T_c$ according to the updated $J_e.IntervalList$:

Case 1: submit a new subjob $J_x = (P_x, R_x)$ with $R_x =$ *new* $A_t.T_c - T_{now}$. 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_k by $J_x.P_x$.

Case 2: submit a new subjob $J_x = (P_x, R_x)$ with $R_x <$ *new* $A_t.T_c - T_{now}$. 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 $J_e.IntervalList$ between T_{now} and $T_{now} + R_x$ will be incremented by q_k.

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 how the shapes of J_x and J_{x+1} are determined and how $A_t.T_c$ 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_e has reached full parallelism $J_e.P_{max}$.

3.3 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.

3.4 TargetAppCompleteEvent Handler

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

4. IMPLEMENTATION

Figure 5 presents the architecture of the EJB system. At a high level, the EJB system consists of two parts: the EJB Manager and the EJB Worker. The EJB manager can be launched on any machine which has a connection to the HPC cluster's front node. Users of the 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.

Over-subscription is supported by most MPI implementations including OpenMPI, MPICH, and its derivatives. For example, the OpenMPI run-time environment detects over-subscription and sets MPI processes to degraded mode which means that they yield processors when idle. In order to validate our performance degradation model of Equation 2, we use the NAS Parallel Benchmarks (NPB [7]) and measure the over-subscription performance using the FutureGrid [2] testbed.

In Figure 6, NPB programs of fixed problem size and amount of parallelism were run on fewer processors to produce over-subscription. We measure the end-to-end execution time at different over-subscription levels. The measured times are compared against one-process-per-core. We can observe sub-linear (ft/is), linear (bt), and super-linear (lu/mg/sp) performance degradations from Figure 6. Our

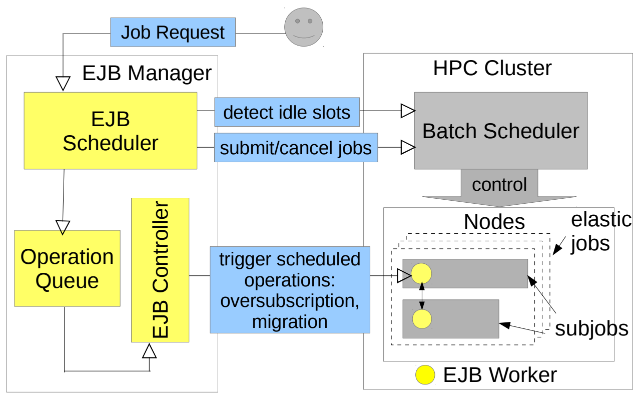


Figure 5: EJB system architecture

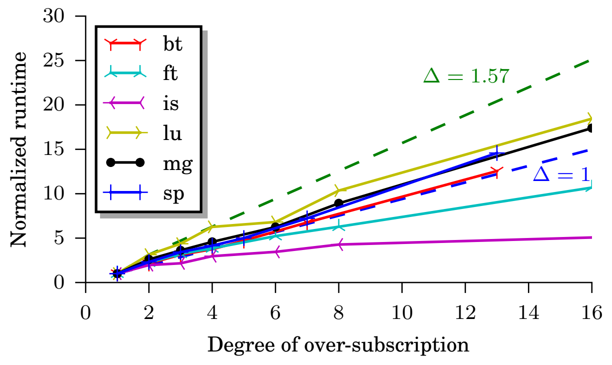


Figure 6: Performance measurement of six NPB programs under over-subscription. All programs are compiled of problem size CLASS=C, with NPROCS=100(bt,sp), 128(ft,is,lu,mg). Different problem sizes or NPROCS follow the same pattern.

key finding is: degradation correlates with scalability. E.g. two of the programs having super-linear degradations, lu and sp, also achieve super-linear speedup in our experimental cluster. ft and is, both show sub-linear degradation and also show sub-linear speedup. bt shows both linear speedup and degradation. The only exceptional case is mg, which shows slightly sub-linear speedup, but super-linear degradation. We are still looking for explanations for this exception.

Another key point is that the maximum number of processes on a single physical processor is limited by memory size. To evaluate whether EJB could be extended to cover the needs of memory-bound parallel applications, we are investigating how to apply out-of-core techniques [15] into our work scope.

To enable migration, we use DMTCP [5], a user-level checkpoint/restart tool for parallel applications including MPI. DMTCP does not require re-compilation of application nor system privileges. Migration includes three steps: global checkpointing, moving checkpoint images, and restart.

DMTCP supports checkpointing by adding a checkpoint management thread to every process at start-up time. During checkpointing, all application processes are simultaneously suspended and the checkpoint images are written to disk by every DMTCP checkpoint management threads. For

Table 2: Traces used in our simulation

Log Files	CPUs	Jobs	Duration	Uti%
CTC-SP2-1996-3.1-cln	338	77,222	7/96-5/97	85.2%
SDSC-SP2-1998-4.1-cln	128	59,725	4/98-4/00	83.5%
SDSC-BLUE-2000-4.1-cln	1,152	243,314	4/00-1/03	76.7%
KTH-SP2-1996-2	100	28,489	9/96-8/97	70.4%

compute nodes equipped with a shared file system (as on FutureGrid), explicitly moving the image is not required. For restart, each EJB Worker is directed to resume its own group of suspended processes, thus completing the migration.

Thus on FutureGrid, the migration time is predominantly spent on checkpointing, which is determined by the parallel application's total memory usage and disk-write speed. For example, in our experimental cluster, checkpointing a NPB bt program of size C and 100 processes takes about 30 seconds, generating in total 3GB checkpoint images at 100MB/s. We anticipate that on clusters supporting parallel I/O, the migration speed could be greatly accelerated. Another possible optimization is to only create checkpoint images for the processes that are actually moving during migration. This will be our future work.

5. TRACE-DRIVEN SIMULATION

Before deploying our 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 [3] to assess feasibility. Table 2 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 [4] – 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, $O_{max} = 8$, the migration duration, $\lambda = 120(s)$, and the performance degradation factor, $\Delta = 1$.

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.

6. 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 (6.1). We then measure how elastic jobs are improved (6.2). We then evaluate how migration cost impacts elastic job performance (6.3). Finally, we study the cluster-wide performance when co-scheduling many elastic jobs together (6.4).

6.1 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*:

$$S_{tt} = \frac{baseline\ tt}{elastic\ job\ tt} \tag{4}$$

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

$$Slo = \frac{elastic\ job\ tt}{baseline\ tr} \tag{5}$$

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 $tr \leq 10$s. 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.

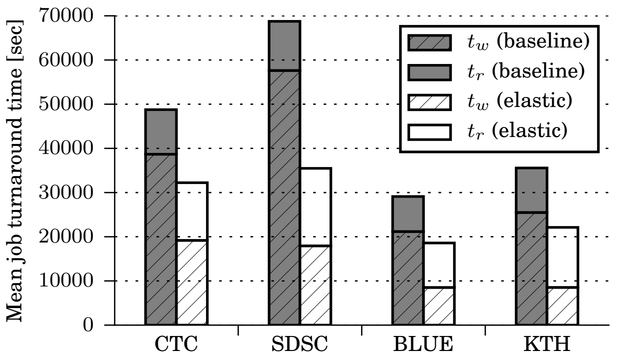
6.2 Improving Elastic Job Turnaround Time

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 3 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.

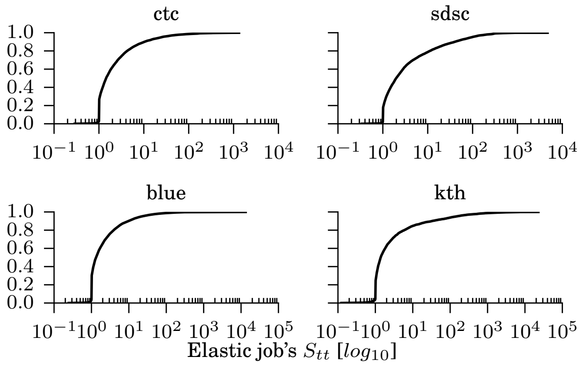
Elastic jobs' mean *tt* is 39.1% faster than the baseline value, with variations between traces. As expected, the EJB results in significantly shorter *tw* (61.4% lower) at the expense of longer *tr* (37.2% higher) due to over-subscription and migration. Detailed distributions of S_{tt} are depicted 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

Table 3: Increase '+' or decrease '-' percentages of mean wait, run, and turnaround time of elastic jobs compared to target jobs' baseline results.

trace	target jobs	percentage change of mean		
		t_w	t_r	t_t [95% conf. interval]
CTC	16,167	-50.4%	+29.3%	-33.9% [-34.7%,-33.1%]
SDSC	14,329	-68.9%	+57.6%	-48.4% [-49.5%,-47.3%]
BLUE	64,090	-59.8%	+26.9%	-36.1% [-36.7%,-35.6%]
KTH	4,399	-66.5%	+34.9%	-37.8% [-39.7%,-35.9%]
AVG		-61.4%	+37.2%	-39.1%

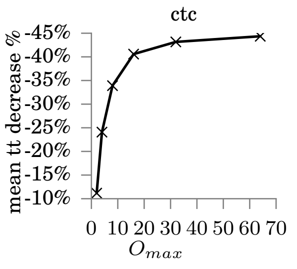


(a) Side-by-side view of how turnaround time improves by transforming target jobs into elastic jobs.

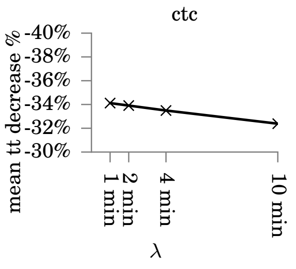


(b) Cumulative distribution function (CDFs) of the speedup of the turnaround time (S_{tt}) of all elastic jobs.

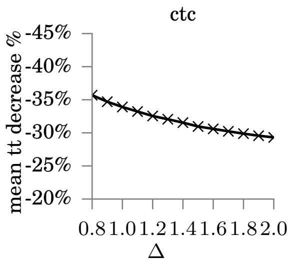
Figure 7: Elastic job's overall performance and variations.



(a) Elastic job mean *tt* percent reduction as a function of the max degree of over-subscription.



(b) Elastic job mean *tt* percent reduction as a function of the migration cost.



(c) Elastic job mean *tt* percent reduction as a function of the performance degradation factor.

Figure 8: Sensitivity analysis of O_{max}, λ, and Δ.

results.

Next, we perform sensitivity analysis to O_{max}, λ, and Δ. 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 λ from 1 to 10 minutes. The performance is not very sensitive to the migration cost. This can be explained as the number of migrations on average is small and the *tt* of the target job is much larger compared to migration time, e.g. on average 9 hours in the CTC workload.

In Figure 8c, we vary Δ from 0.8 to 2.0. The elastic job mean *tt* is not very sensitive to degradation. This is due to (i) the fact that many elastic jobs are capable of finding enough processors in the later stages of their life cycle, thus eliminating the over-subscription overhead afterwards, and that (ii) *tw* still accounts for a considerable proportion of *tt* even with EJB .

6.3 Migration behavior

Table 4 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

Table 4: Summarize migration related statistics.

trace	subjobs	migrations	migration duration	resource overhead
CTC	3.3	2.1	8.5%	19.7%
SDSC	2.8	1.7	5.6%	16.4%
BLUE	2.7	1.5	8.6%	18.2%
KTH	2.7	1.6	6.4%	24.1%

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 $\times$ time. A main factor contributing to this is the inaccuracy in the *tr* estimations. Based on user provided *tr*s, 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 [27] to more accurately estimate *tr* according to historical job information and make migration decisions based on the adjusted *tr*.

6.4 Multiple Elastic Jobs

In 6.2, 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 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 5 shows the Before-and-After comparison results. Notice that

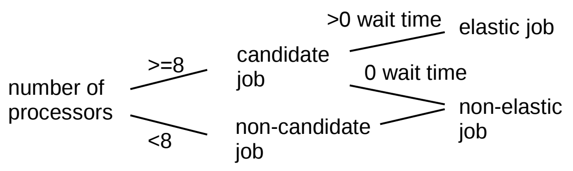


Figure 9: Decision tree for elastic job selection

the number of elastic jobs are different from that of Table 3. 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 3 to 21,035 in Table 5) 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 $tw = 0$, to now become elastic. However, the mean turnaround time of elastic jobs is significantly reduced.

Table 5 shows that for all the workloads except KTH, wide use of EJB not only results in shorter *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 5 (column 5). The maximum slowdown (which is too large to be shown in the graph) experienced by the most unlucky job also decreases. [13] indicates that the mean turnaround time and the mean slowdown are seperately dominated by long and short jobs, thus EJB is not biased toward any type of job. Actually, we observe that large jobs with short *tr* benefit greatly from EJB. These jobs previously suffered from long waiting time due to the height of their original shape. EJB enables these jobs to start earlier, hence they will complete in less time.

In order to evaluate how EJB promotes fairness, we did a linear regression analysis of all job's *tw* and over job size in Figure 11. We admit that job's *tw* does not have strict linear correlation with job's size. However, the trend is that larger jobs tend to wait longer. Actually, large jobs are known to suffer more than small jobs under scheduling policies that optimize mean *tt* or slowdown [30]. By comparing the slopes of regression lines generated from the results before and after EJB is added, we can see that the slope of the *tw* under EJB is flatter indicating less sensitivity to processor size (i.e. is more fair). We have also measured the total number of priority inversions, which drops about 20% when EJB is applied. This is further evidence of fairness.

Table 6 shows the measurement of fragmentation as defined in 6.1. 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

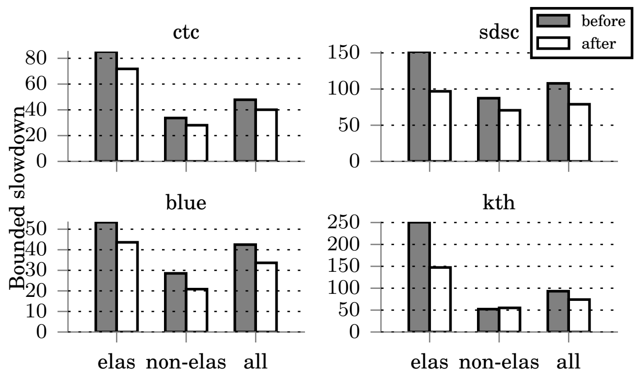


Figure 10: Bounded slowdown: side-by-side view before and after EJB is added, grouped into elastic, non-elastic, and all jobs.

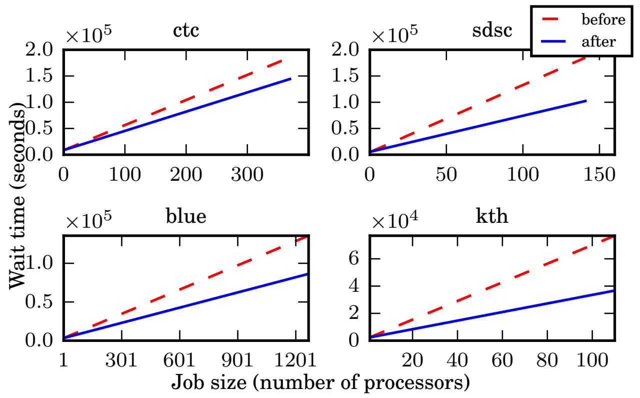


Figure 11: Linear regressions of *tw* over job size before and after EJB is added. Adjacent to x-axis indicates fairness.

Table 6: Fragmentation: np is the average number of idle processors, % is the percentage of the idle processors in the cluster.

Trace	before		after		changes
	np	%	np	%	
CTC	33.7	10.0%	22.4	6.6%	-34.0%
SDSC	13.8	10.8%	5.9	4.6%	-57.4%
BLUE	129.5	11.2%	53.8	4.7%	-58.0%
KTH	16.0	16.0%	6.6	6.6%	-58.8%

start sooner.

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.

7. 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, ho-

Table 5: Before-and-after comparison: confidence intervals are calculated at the 95% confidence level.

Workload	Job type	Elastic jobs	Mean tt			Mean tw			Mean tr	
			Before	After (change %)	conf. interval	Before	After (change %)	conf. interval	Before	After (change %)
CTC	elastic	21,035	40,276	35,512 (-11.8%)	(-5,091,-4,437)	31,695	25,863 (-18.4%)	(-6,165,-5,498)	8,581	9,649 (+12.4%)
	non-elastic	59,123	19,268	17,874 (-7.2%)	(-1,503,-1,285)	7,008	5,614 (-19.9%)	(-1,503,-1,285)	12,260	—
	all	76,446	25,049	22,728 (-9.3%)	(-2,442,-2,201)	13,801	11,186 (-18.9%)	(-2,737,-2,493)	11,248	11,542 (+2.6%)
SDSC	elastic	18,790	58,235	40,888 (-29.8%)	(-17880,-16813)	48,468	29,001 (-40.2%)	(-20,016,-18,917)	9,767	11,887 (+21.7%)
	non-elastic	40,333	10,589	8,676 (-18.1%)	(-2,052,-1,775)	5,412	3,499 (-35.3%)	(-2,052,-1,775)	5177	—
	all	59,123	25,731	18,913 (-26.5%)	(-7021,-6615)	19,096	11,604 (-39.2%)	(-7,701,-7,282)	6,636	7,309 (+10.2%)
BLUE	elastic	135,302	16,863	14,881 (-11.8%)	(-2,067,-1,899)	12,015	9,626 (-19.9%)	(-2,475,-2,302)	4,848	5,254 (+8.4%)
	non-elastic	105,560	4,096	3,186 (-22.2%)	(-941,-878)	1,109	199 (-82.0%)	(-941,-878)	2,987	—
	all	240,862	11,268	9,756 (-13.4%)	(-1,562,-1,463)	7,235	5,495 (-24.1%)	(-1,791,-1,690)	4,033	4,261 (+5.7%)
KTH	elastic	5,811	32,457	25,632 (-21.0%)	(-7,347,-6,302)	22,137	13,673 (-38.2%)	(-9,010,-7,918)	10,320	11,959 (+15.9%)
	non-elastic	22,392	11,523	11,565 (+0.4%)	(-56,141)	2,982	3,024 (+1.4%)	(-56,141)	8,541	—
	all	28,203	15,836	14,463 (-8.7%)	(-1509,-1236)	6,929	5,218 (-24.7%)	(-1,853,-1,568)	8,907	9,245 (+3.8%)

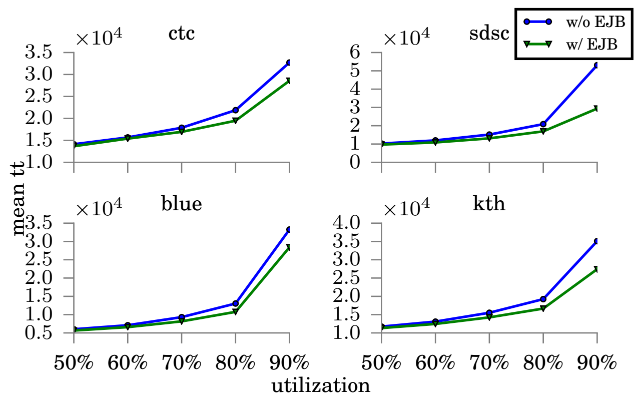


Figure 12: Changing utilization: EJB is more resistant under high utilization.

mogenizing system workload, by decomposing large jobs into smaller ones. Compared to larger jobs, smaller jobs can allow schedulers to allocate resource more quickly and improve load balance [22]. 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 a 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.

8. 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 a parallel job 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.

8.1 Moldable Jobs

Cirne's works in [9,10] 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.

8.2 Malleable Jobs

Malleable (or adaptive) jobs have the attractive property that they dynamically adapt to system workload [29]. *ReSHAPE* [25, 26] 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. [28] 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.

9. 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 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.

10. 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.

11. REFERENCES

[1] https://bitbucket.org/francis_liu/pyss.
[2] Futuregrid. futuregrid.org.
[3] Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.
[4] pyss - the python scheduler simulator. https://code.google.com/p/pyss/.
[5] J. Ansel, K. Aryay, and G. Coopermany. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.
[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
[8] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 8–pp. IEEE, 2001.
[9] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002.
[10] W. Cirne and F. Berman. When the herd is smart: aggregate behavior in the selection of job request. *Parallel and Distributed Systems, IEEE Transactions on*, 14(2):181–192, 2003.
[11] A. B. Downey. Predicting queue times on space-sharing parallel computers. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 209–218. IEEE, 1997.
[12] A. B. Downey. Using queue time predictions for processor allocation. In *Job Scheduling Strategies for Parallel Processing*, pages 35–57. Springer, 1997.
[13] D. G. Feitelson. Metric and workload effects on computer systems evaluation. *Computer*, 36(9):18–25, 2003.
[14] S. Jha, M. Cole, D. S. Katz, M. Parashar, O. Rana, and J. Weissman. Distributed computing practice for large-scale science and engineering applications. *Concurrency and Computation: Practice and Experience*, 25(11):1559–1585, 2013.
[15] D. LaSalle and G. Karypis. Mpi for big data: New tricks for an old dog. *Parallel Computing*, 40(10):754–767, 2014.
[16] H. Li, D. Groep, J. Templon, and L. Wolters. Predicting job start times on clusters. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 301–308. IEEE, 2004.
[17] D. J. Lilja. *Measuring computer performance: a practitioner's guide.* Cambridge University Press, 2000.
[18] F. Liu and J. Weissman. Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications. Technical report, TR15-006, Department of Computer Science and Engineering, University of Minnesota, 2015.
[19] A. Luckow, M. Santcroos, O. Weidner, A. Merzky, P. Mantha, and S. Jha. P*: A Model of Pilot-Abstractions. In *8th IEEE International Conference on e-Science 2012*, 2012.
[20] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.
[21] D. Nurmi, J. Brevik, and R. Wolski. Qbets: queue bounds estimation from time series. In *Job Scheduling Strategies for Parallel Processing*, pages 76–101. Springer, 2008.
[22] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters.
[23] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Job Scheduling Strategies*

for Parallel Processing, pages 202–219. Springer, 1999.

[24] G. Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[25] R. Sudarsan and C. J. Ribbens. Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 44. IEEE, 2007.

[26] R. Sudarsan and C. J. Ribbens. Scheduling resizable parallel applications. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[27] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):789–803, 2007.

[28] G. Utrera, S. Tabik, J. Corbalan, and J. Labarta. A job scheduling approach for multi-core clusters based on virtual malleability. In *Euro-Par 2012 Parallel Processing*, pages 191–203. Springer, 2012.

[29] J. B. Weissman, L. R. Abburi, and D. England. Integrated scheduling: the best of both worlds. *Journal of Parallel and Distributed Computing*, 63(6):649–668, 2003.

[30] A. Wierman. Revisiting the performance of large jobs in the M/GI/1 queue. In *Proceedings of the Forty-Fifth Annual Allerton Conference On Communication, Control, and Computing*, pages 607–614, 2007.

[31] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41–49, 2003.

[32] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.