Scheduling Many-Task Workloads on Supercomputers: Dealing with Trailing Tasks

Timothy G. Armstrong, Zhao Zhang
Department of Computer Science
University of Chicago
tga@uchicago.edu, zhaozhang@cs.uchicago.edu

Daniel S. Katz, Michael Wilde, Ian T. Foster
Computation Institute
University of Chicago & Argonne National Laboratory
d.katz@ieee.org, wilde@mcs.anl.gov, foster@anl.gov

***Abstract*—In order for many-task applications to be attractive candidates for running on high-end supercomputers, they must be able to benefit from the additional compute, I/O, and communication performance provided by high-end HPC hardware relative to clusters, grids, or clouds. Typically this means that the application should use the HPC resource in such a way that it can reduce time to solution beyond what is possible otherwise. Furthermore, it is necessary to make efficient use of the computational resources, achieving high levels of utilization.**

Satisfying these twin goals is not trivial, because while the parallelism in many task computations can vary over time, on many large machines the allocation policy requires that worker CPUs be provisioned and also relinquished in large blocks rather than individually.

This paper discusses the problem in detail, explaining and characterizing the trade-off between utilization and time to solution under the allocation policies of Blue Gene/P *Intrepid* at Argonne National Laboratory.

We propose and test two strategies to improve this trade-off: scheduling tasks in order of longest to shortest (applicable only if task runtimes are predictable) and downsizing allocations when utilization drops below some threshold. We show that both strategies are effective under different conditions.

***Keywords* - Many-task computing; scheduling; high-performance computing; supercomputer systems.**

I. Introduction

Many-task applications are characterized by two key features that, when combined, provide the motivation for this paper [1]. The first feature is that the application comprises many independent tasks coupled with explicit I/O dependencies. For high-throughput computing and many-task computing applications, each task typically will be single threaded or, if it is multithreaded, support parallelism only within a single node's address space. The number of tasks assigned to each node in the machine can vary. Multicore compute nodes are typical, so a machine node might run several tasks in parallel. For example, on the Blue Gene/P *Intrepid* at Argonne National Laboratory, each compute node has four single-threaded cores [2], and on Blue Waters, each compute node has eight cores, with four virtual threads per core [3]. We call the unit that a task is allocated to a "worker." The nature of a worker is both machine and application dependent: if tasks are multithreaded, it may be most efficient to treat each node as a worker and allocate one task per node. If tasks are single-threaded, each core or virtual thread can be treated as a worker.

The second feature of many-task applications is an emphasis on high performance. The many tasks that make up the application effectively collaborate to produce some result, and in many cases it is important to get the results quickly. This feature motivates the development of techniques to efficiently run many-task applications on HPC hardware. It allows people to design and develop performance-critical applications in a many-task style and enables the scaling up of existing many-task applications to run on much larger systems. The many-task style has definite advantages for applications that can be expressed as a directed acyclic graph of tasks. Existing applications can often be embedded in the task graph, eliminating the need to rewrite them. Additionally, the task-graph structure of many-task applications lends itself to easy reasoning about application behavior and provides a straightforward model for failure recovery.

In this paper we discuss the challenges of executing many-task applications on large supercomputers. To justify using supercomputing resources for an application, we must demonstratetwo results: that the application uses the machine's processing power efficient, thereby achieving high utilization, and that the use of supercomputing resources allows the time to solution to be reduced significantly below what is possible with less powerful systems.

If computing system worker nodes are individually allocated to applications, it is straightforward to achieve fairly efficient utilization of the allocated resources, at least if tasks are compute bound: if a worker falls idle, it can simply be released. However, parallel job schedulers typically assume that all resources requested will be held for the same duration, and allocating resources individually through the scheduler typically adds overhead and delays. A user of a supercomputer such as a Blue Gene or Cray XT series system must typically request compute nodes as a large block for a prespecified duration through the supercomputer's scheduler. Particular constraints on the sizes and numbers of blocks that can be requested are imposed because of intrinsic factors such as system interconnect topology and because of policies that, for various reasons, give priority to large jobs.

The degree of parallelism, and therefore the number of

runnable tasks, can fluctuate as the execution of a many-task application proceeds. When a large block of worker nodes or cores is allocated for a many-task application, and subsequently the number of runnable or running tasks dips to a much lower level, there may be many idle workers allocated for the application that cannot be released. One such case occurs in multistage workflows if some stages have a greater degree of parallelism than others.

This situation can be viewed as a shortage of available parallelism in the application: the number of running/runnable tasks is not enough to occupy all the workers that have been allocated. But another way of viewing the problem is to consider it a mismatch in the granularity of the application's tasks and the resource provisioning on parallel supercomputers.

A range of heuristics can, in some cases, increase the parallelism available at various points of execution [4], [5].

A specific case that can occur in many-task applications is what we term the *trailing task problem*, where an increasing number of workers have no further work to do and are sitting idle, but a tail of some number of *trailing tasks* continues to execute for some time. The defining feature of this situation is a gradual and monotonic decrease in the number of active tasks: as tasks complete, workers increasingly go from active to idle.

In this paper we focus on the trailing task problem and in particular on the simple case where the many-task application is a set of tasks with no interdependencies.

II. Related Work

Substantial work has been done on implementing load-balancing algorithms in parallel and distributed-computing systems. Load-balancing research has generally focused on producing practical, well-performing algorithms to even out disparities in load between workers, ensuring that lightly loaded workers receive work instead of heavily loaded workers and that idle workers are immediately given work. One of the major areas of research in load-balancing has been implementing efficient and scalable algorithms to detect and fix load imbalances. An example is ADLB, an MPI-based load-balancing library, which has scaled to 131,072 cores on *Intrepid* [6]. In this paper we investigate the minimization of the completion time of a set of tasks, rather than simply the balancing of load between workers. However, an efficient load-balancing and task dispatch system is essential to the implementation of any of the scheduling policies proposed in this paper.

The effects of skewed distributions of process lifetimes on load balancing have also been studied. It has been shown that, in the context of a network of workstations, process migration is a valuable tool for load balancing when process lifetimes are part of a heavy-tailed distribution [7].

Both allocation and deallocation policies have been explored in the context of many-task computing using the Falkon task dispatch system [8].

A special case of the trailing task problem is referred to as the "straggler" problem in the literature on data-intensive computing. In this problem tasks run for an excessively long time because of various unpredictable factors that affect individual "straggler" machines, such as hardware malfunctions or contention for CPU time or disks. There are various documented strategies to detect straggler nodes and mitigate the effect. The MapReduce framework, adopts a simply strategy to deal with stragglers. Upon detecting a task's slow progress, it simply replicates the task on other machines, under the presumption that the long running time of the task is because of some factor affecting the current machine, but not all other machines [9]. For obvious reasons, this strategy is not at all effective if the long-running task simple involves more work than others.

The paper does not consider the straggler problem; we assume that each task's runtime is determined solely by the task definition and input data. This is a fair approximation of the supercomputing environment we are considering, in contrast to the cloud or data center environment.

It is unusual in a supercomputing installation for compute nodes to be split between different allocations, so contention between different processes is mainly limited to shared file system and I/O resources. The problem of malfunctioning hardware is also something we can disregard, at least for the purposes of scheduling: we can reasonably assume that malfunctioning hardware will be proactively replaced, rather than demanding that the application or middleware be robust to misbehaving hardware.

Several bi-criteria scheduling problems with different objectives from the one in this paper have been studied, for independent jobs [10] and for computational workflows [11] on parallel machines.

III. Problem Description

If we ignore unpredictable variations in task runtime, then our problem, at a high level, is as follows. The parallel computer is made up of some number of identical workers $M_1, M_2, ...,$ (often called *machines* in the scheduling literature) that managed by the supercomputer's scheduler. A many-task application comprises n tasks (or *jobs*) $J_1, ..., J_n$ with runtimes $t_1, ..., t_n$. The scheduler, upon request, can allocate blocks of workers of differing sizes for the exclusive use of a user.

Two stages of decisions must be made to schedule a many-task application. First, workers must be acquired from the scheduler in a way fitting the allocation policies of the machine. The constraints imposed by allocation policies are discussed further in Section V. Second, the tasks must be scheduled on the available workers. It may or may not be possible to preempt running tasks. In this paper we assume tasks are not preemptable.

Our problem is a bi-criteria optimization problem: we want to both minimize time to solution (TTS, often called *makespan* in the scheduling literature) and maximize utilization. Utilization is defined as $u = \frac{\Sigma t_i}{\textit{total allocated worker time}}$, where numerator and denominator are both measured in CPU time.

Related problems are well studied in the scheduling theory literature, in particular the same problem in which m, the number of machines allocated, is fixed. The problem we encounter is atypical mainly because m must be chosen and may be set to different values over time as the application progresses. Hence, we must deal with the additional decisions of how many machines to allocate, and we must trade off the two competing goals of time to solution and worker utilization.

IV. Algorithms for Fixed Worker Counts

The problem is simpler to analyze with fixed m, since minimizing TTS and maximizing utilization are equivalent goals. This simpler problem is key to solving the more complex problem where we consider allocation: once we have made an allocation decision, the problem still remains to efficiently schedule tasks on that allocation of workers.

If we do not allow preemption and have no constraints on task runtimes, then this problem is closely related to the bin-packing problem and is NP-complete [12]. If there are no precedence constraints on tasks, then the problem allows various practical polynomial scheduling policies that produce schedules with makespans that are, in the worst case, within a fixed bound of the optimum [13].

Simple load-balancing approaches, where tasks are assigned to idle workers from a queue, can achieve results with makespans within fixed bounds of the optimum. Any arbitrary order of the queue will yield a schedule that requires, in the worst case, $2 - 1/m$ times the duration of the optimal schedule, where m is the number of processors. Sorting the task queue so that the longest tasks are assigned first gives a significant improvement, yielding a schedule with duration at most $4/3-1/m$ of the optimal duration [13].

Also present in the literature are more sophisticated approximation algorithms that still run in polynomial time and provide results provably closer to the optimal [13].

In practice, for many or most applications it is not possible or practical to estimate task runtimes with a great deal of accuracy, if at all. Therefore, any approach that assumes knowledge of runtimes may not be feasible.

To further understand how well the two load-balancing algorithms described above, random and sorted, will perform, we can identify two factors that frustrate attempts at achieving an even load balance between processors, and cause the trailing task problem to manifest itself.

The first factor is variance in task duration. Even if tasks are running the same code, differing input data or parameters typically will result in varying task run:times. Longer running tasks are likely to form part of a tail, particularly if they are scheduled late in the workload.

The second factor that occurs, even if tasks are all equal in runtime, arises if the number of tasks is not divisible by the number of workers. In this case the tasks cannot be split evenly between workers.

The severity of the first factor depends on the statistical distribution of task times in a workload: normally distributed runtimes are not too problematic, but unfortunately, left-skewed distributions of task times with a long right tail of tasks do occur in real-world many-task applications. Figure 1 illustrates one and shows the drop-off in utilization that occurs when it is run on a fixed number of processors. There are far fewer processors than tasks, but the load-balancing is insufficient to counter the skew of the distribution.

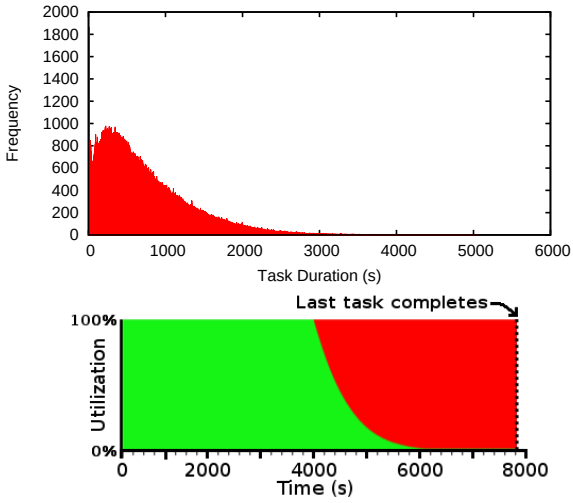

Figure 1. Top: histogram of task durations for a large DOCK5 run of 934,710 tasks, demonstrating a long tail of task durations. The longest task was 5,030 seconds. Bottom: worker utilization from a simulation of this run on 160,000 processor cores using the simple load-balancing strategy of assigning arbitrary tasks to idle workers. Red represents idle workers; green represents active workers. The last task completes after 7,828 seconds.

V. Constraints on Worker Provisioning

The constraints on what allocation sizes can be chosen may vary significantly between supercomputers, for technical and policy reasons. In some cases, the architecture or interconnect topology of the machine means that it is difficult or impossible to allocate blocks of otherwise desirable irregular sizes that would improve utilization. For example, on the 160K CPU Blue Gene/P *Intrepid*,

the partition size is bounded from below by the size of a *processor set* (pset), which comprises a single I/O node and 64 compute nodes. Spatial fragmentation of the machine is also a major issue on some machines when there are many jobs of different sizes. Interconnect topologies such as meshes or torus networks generally require that nodes for a job be allocated contiguously in rectangular-shaped

blocks; this requirement has a strong tendency to cause fragmentation of the machine [14]. Additional complications exist on some systems. For example, on *Intrepid*, certain partition sizes are particularly wasteful of interconnect resources when a torus rather than mesh network is used [15]. As a result of these and other problems, the allocation policy on *Intrepid* is such that its 4-CPU compute nodes can be requested through the scheduler only in block sizes of powers of 2, from 512 to 32,768. Partitions of 24,576 and 40,960 nodes (3 and 5 rows of 8,192 nodes respectively) are also made available by request [16].

In addition to these constraints on allocation size, in practice one often does not want to request too many separate allocations, as some schedulers impose a cap on the number of active allocations per user, and requesting many small allocations can add overhead and delays for many-task applications [8].

VI. Trade-Offs in Making Static and Dynamic Allocation Decisions

If we can dynamically or statically choose the allocation size, then in conjunction with either scheduling policy (random or sorted order) from Section IV, utilization can generally be improved by allocating far fewer workers than tasks. This strategy permits effective load balancing because assigning tasks from the queue to idle workers can keep utilization high for most of the allocation, until the queue of runnable tasks is exhausted. In practice this means that utilization levels can often be achieved with either policy far above the theoretical worst-case bounds, as demonstrated by the utilization chart in Figure 2.

However, sorting tasks is often not possible. Allocating far fewer workers than tasks is effective in applications comprising massive numbers of short tasks, but it is problematic otherwise. Reducing workers will naturally increase the time to solution for the application, resulting in the trade-off shown in Figure 2.

In this case the worker count must be low relative to the task count to achieve over 90% utilization (e.g., 16K workers for 935K tasks to achieve over 90% utilization). In this instance it might not be a problem, as tasks are minutes in length, and time to solution is approximately 12 hours. If tasks were longer in duration, however, this time would stretch out in proportion.

Furthermore, the utilization levels are likely to be sensitive to changes in the distribution of task runtimes: a different application with a longer or wider tailed distribution would likely require an even smaller number of workers relative to tasks. And even with this miserly proportion of workers to tasks there is no guarantee of acceptable levels of utilization on other workloads.

We seek here to develop methods that improve this trade-off, thereby allowing a shorter time to solution for a given utilization level. We also seek achieve to high utilization

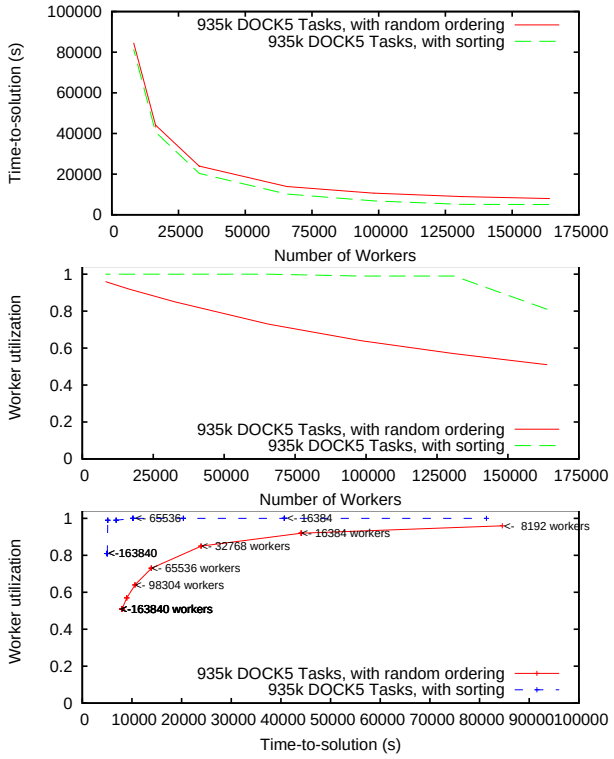

Figure 2. Results of a scheduling simulation of a many-task application with 935,000 single threaded tasks on partitions of different sizes. The random and sorted (long to short) load-balancing policies were both simulated. The task runtime data used in the simulation was collected from a large DOCK5 run on a Blue Gene/P system [17]. This relationships between worker count, time to solution, and utilization are shown for worker counts of 256 to 163,840 cores. The simulation is described in Section IX.

consistently and robustly without having to guess a "magic" number of processors to run the job on.

We could simply "move the goalposts" by noting that in some applications, a user might want a partial set of results quickly, with the remainder either ignored or completed at a later time. For many applications, even having 10%, 50%, or 90% of the final results is useful. For example, if a chemist is screening compounds for a particular property, it may be possible to use positive results as soon as they become available, and an exhaustive search of all compounds may not be necessary to the success of the endeavor.

Another method, which we explore in detail in the rest of the paper, changes the allocation dynamically to end a large block allocation of workers once there is only the "tail" of tasks left to run. After "chopping off the tail," a smaller block of the machine or another computational resource such as a smaller cluster, grid, or cloud can be allocated to run the remaining tasks. The tail-chopping process can, in principle, be performed several times, shifting to ever-smaller allocations. Intuitively, we can see that this approach could improve the trade-off between utilization and time to solution. The power of a supercomputer will be harnessed to complete the bulk of the work quickly on a large partition,

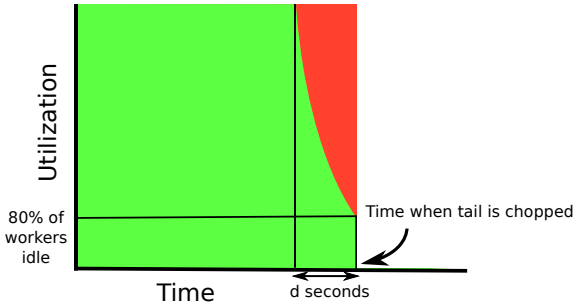

Figure 3. Illustration of several different ways of deciding when to chop the tail. The time to chop the tail can be selected in at least three ways: when 80% of workers are idle, when d seconds have elapsed after a worker falls idle, or when the amount of idle CPU time (the red area) exceeds a threshold.

but utilization will not be severely degraded by leaving a large partition underutilized.

VII. Tail-Chopping Approach: Assumptions and Heuristic

In the remainder of this paper, we investigate empirically the effects of choosing different allocation sizes and of tail-chopping: dynamically decreasing allocation sizes. For simplicity in this initial study, we made the following assumptions:

- Only one partition of processors will be used for the target workload at a given time.
- There is no time limit on processor allocations: allocations can be requested for any desired duration and can be terminated immediately at any time.
  (In practice, most supercomputers have a maximum allocation duration, and the duration of allocations must be specified ahead of time; but attempting to estimate the required partition duration ahead of time would complicate experiments without providing any particular insight into the value of tail-chopping.)
- A constant time is required to start or stop an allocation partition.
- Tasks cannot be migrated from one worker to another. To move a task, one must cancel and restart the task.

Given these assumptions, we investigate a range of approaches to "chopping off the tail" through the use of the following parametrized heuristic:

- We decide how many workers to allocate for a given number of tasks using a single parameter: a *minimum task/worker ratio* that is applied when selecting a new partition size. There will be a menu of available partition sizes for a given supercomputer, and the maximum size satisfying the task/worker ratio will be chosen. In the case where all are too large, the smallest will be chosen. This formula is applied each time a partition must be created: when the many-task application starts and every time a smaller partition must be allocated.
- A single parameter is used to decide when to shrink the number of workers: the *maximum fraction of idle workers* that is tolerated before a switch to a smaller allocation is triggered (provided a smaller allocation is available).
  We believe that this approach is adequate for our study, although many other heuristics are possible. As illustrated in Figure 3, this parameter is sufficient to specify any point on the downward curve of utilization. Different heuristics will behave differently as the workload characteristics vary; but for a given instance, this heuristic is sufficient to study the full range of tail-chopping approaches from very aggressive to very conservative.

These two parameters provide a simple way to investigate key trade-offs involved in making decisions about processor allocations.

The task/worker ratio trades off between utilization and time to solution, as discussed in Section VI. The maximum fraction of idle workers trades off between two sources of inefficiency: unused capacity, when workers sit idle due to lack of work, and lost progress, when tasks are canceled upon moving to a different partition. A strategy that aggressively moves to new partitions whenever workers start to become idle will reduce the first but increase the second source.

VIII. Tail-chopping: Hypotheses

Our expectations, which we have attempted to validate through simulation and experiment, are as follows:

- Tail-chopping will not completely solve utilization problems: utilization will still be lower with lower task/worker ratios, as more work is lost when a bigger partition is canceled, and more workers sit idle while waiting to drop below utilization threshold.
- It will be hard to achieve high utilizations with smaller workloads if the minimum allocation is comparatively high. For example, on Intrepid where the minimum allocation is is $2{,}048$ cores, the wastage of up to $2{,}047$ cores sitting idle while the tail finishes is high relative to the amount of remaining work.
- We expect tail-chopping to be more beneficial in cases where the task distribution is skewed and the tail has a large number of much-longer-than-average tasks. For these tasks the benefits from relocation in reducing the number idle workers exceed greatly the costs in lost progress.
- Tail-chopping is likely to provide a greater benefit when no task sorting occurs, because sorting mostly handles the long-running task problem. With sorting, if any of the leftover tasks are the longest running, canceling them is likely to be especially wasteful, since they will have been running since the start of the allocation, and much useful work will be lost.

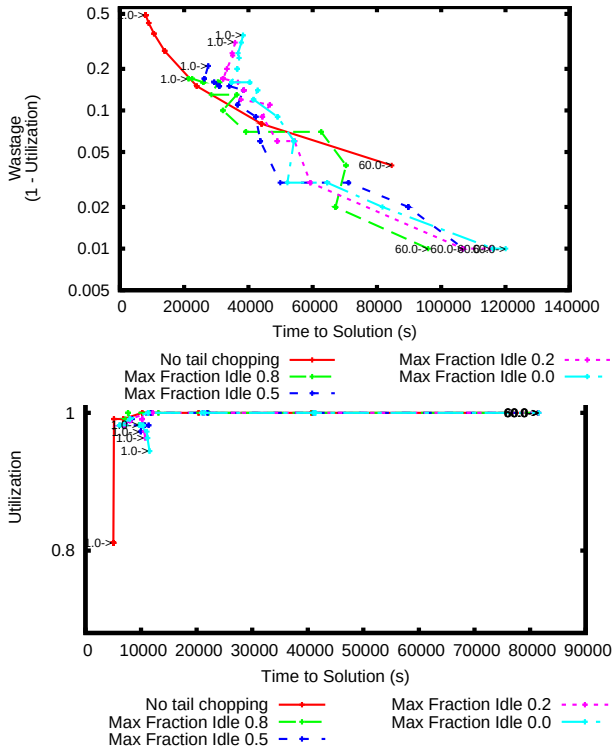

Figure 4. Effect of tail-chopping on trade-off between time to solution and utilization in simulation for the *real935k* workload. The points along each curve are derived from the measured (time to solution, utilization) observations for task/worker ratios of 1, 1.2, 1.4, 1.6, 1.8, 2, 2.2, 2.4, 2.6, 2.8, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 8.5, 9, 9.5, 10, 11, 12, 13, 14, 15, 20, 25, 30, 35, 40, 45, 50, 60 (following the curves left to right). The curves are not always convex: with tail-chopping, in some cases increasing the task/worker ratio could hurt time to solution or improve utilization because of the delays and lost work that occur as a result of tail-chopping. Top: with random task order (log scale). Bottom: with sorted task order (linear scale).

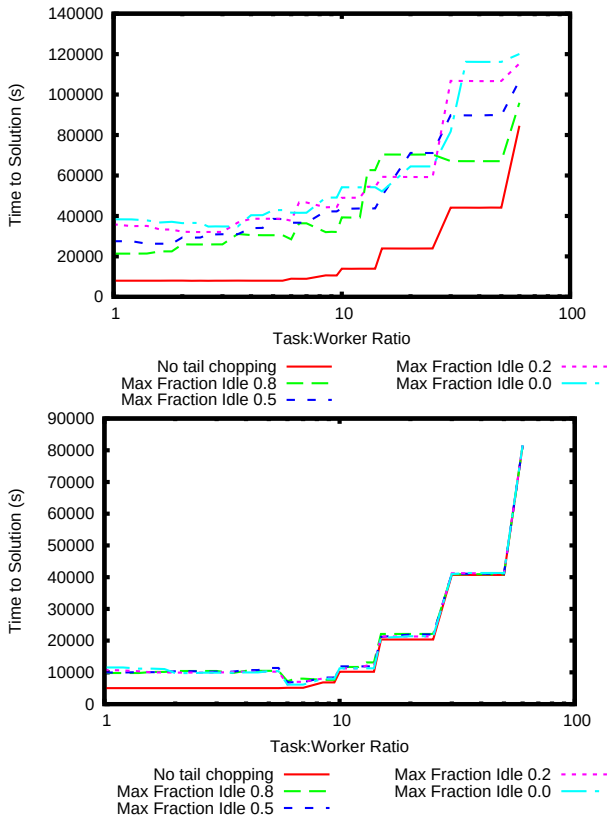

Figure 5. Effect of tail-chopping on time to solution in simulation for the *real935k* workload. Top: with random task order. Bottom: with sorted task order.

- Tail chopping is unlikely to provide any benefit combined with sorting if max_length/mean_length $>$ task/worker ratio. The reason is that case the longest-running tasks will be finished or almost finished when tail-chopping happens, and we would just be canceling and restarting shorter running tasks.

IX. Simulation

We conducted a number of simulations to investigate the effect of varying the parameters described in Section VII. From the simulation, we collected data for a wide range of scheduling algorithm parameters before implementing the idea in practice.

A. Method

We simulated the Intrepid Blue Gene/P configuration, modeling partitions with the following numbers of CPU cores: 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 98304, 131072 and 163840.

Three workloads with different characteristics were used to test the effectiveness of different policies. All tasks in the workloads were single threaded and ran on a single core.

Measurements of two workloads were obtained from DOCK [18] runs on Intrepid. Measurements for *real935k*, a large run of 934,710 DOCK5 tasks were collected during previous work [17]. Measurements for the *real15k* workload of 15,000 DOCK6 tasks were collected on *Intrepid*. Both workloads were executed with Falkon, and the total runtimes were collected from log files. A synthetic left-skewed workload of 15,000 tasks with a long tail (*synth15k*) was created by sampling from a log-normal distribution with the same mean runtime as *real15k* and four times the standard deviation of *real15k*.

We took further measurements to estimate the time taken to start up and close down a partition of the machine.

- The time from requesting an allocation through the resource manager (Cobalt) until Falkon reported that all the partitions requested were available to start executing tasks – 170.0 seconds on average for a partition of 64 nodes.
- The time taken between requesting an allocation to terminate (through `cqdel`) and the allocation finishing

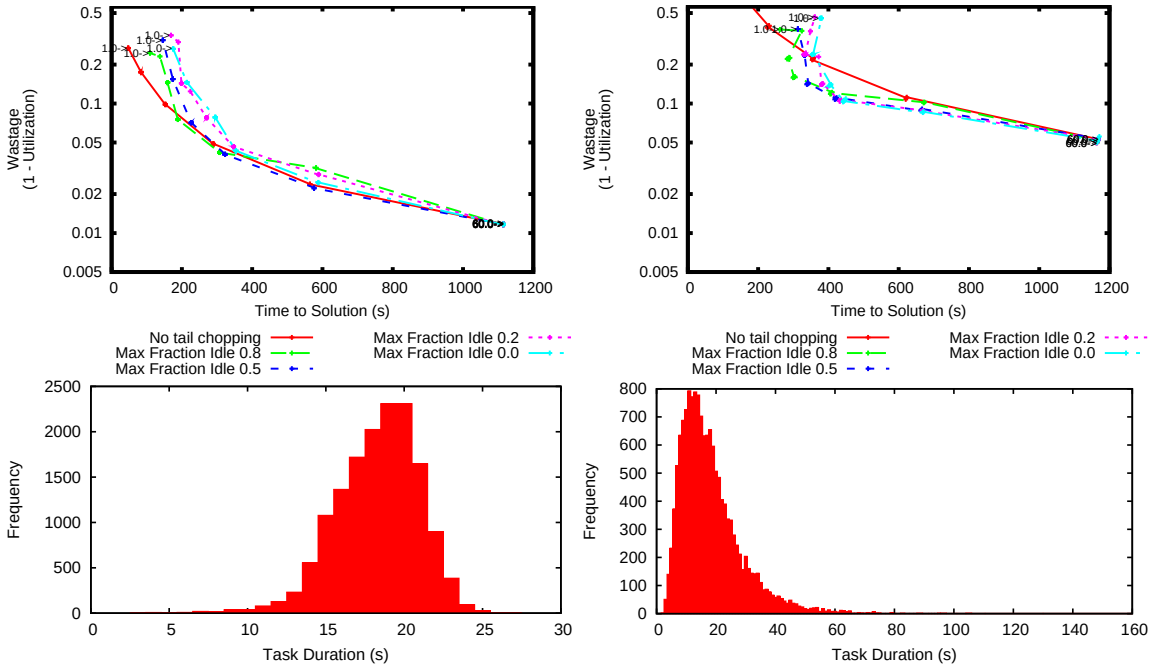

Figure 6. Top: Trade-off between utilization and time to solution observed in tail-chopping simulation for the *real15k* (left) and *synth15k* (right) workloads. The points along each curve are derived from the measured (time to solution, utilization) observations for task/worker ratios of 1, 1.2, 1.4, 1.6, 1.8, 2, 2.2, 2.4, 2.6, 2.8, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 8.5, 9, 9.5, 10, 11, 12, 13, 14, 15, 20, 25, 30, 35, 40, 45, 50, 60 (following the curves left-to-right). The curves are not always convex: with tail-chopping, in some cases increasing the task/worker ratio could hurt time to solution or improve utilization because of the delays and lost work that occur as a result of tail-chopping. Bottom: Histograms that demonstrate the different distributions for the two workloads.

– 2.4 seconds on average.

We then used these measurements to simulate the running of various scheduling policies. We assumed that tasks run for exactly the same duration as was measured for the real run and that the next task in the queue can always be scheduled with zero delay as soon as a worker becomes idle. Zero scheduling delay may not be a realistic assumption, since a scalable distributed algorithm will not likely feature a single, central task queue; but the results provide insight into how more scalable algorithms that approximate a central task queue will behave.

In the experiment, we have control over three parameters:

- Scheduling order – in random order or in descending order of runtime
- Minimum task/worker ratio
- Maximum fraction of idle workers

We simulated a large number of combinations of these variables. The simulation was straightforward: a list of task runtimes was loaded and either sorted or randomly shuffled as appropriate. At the start, m_0 workers are allocated, and the first m_0 tasks assigned to them, with m_0 chosen using the task/worker ratio. The simulation then simply uses a task queue to assign work to idle workers. This continues until all tasks are completed or the maximum fraction of idle workers is exceeded, at which point a partition change is triggered and the process repeats with the unfinished tasks.

With sorted task order, the simulation is deterministic, so we need to perform only one trial. With randomized task order, we perform 100 trials of the simulation, reshuffling the order of task execution each time and collecting the mean of all observed times.

We took several observations for each parameter combination:

- Time to solution (time from start-up until all tasks are completed) (milliseconds).
- Total active CPU time: total CPU time used by workers while running tasks (CPU-milliseconds).
- Total worker idle CPU time (CPU-milliseconds).
- Total allocation CPU time: the amount of CPU time provisioned. The time taken for a partition to start up and become available was not included (CPU-milliseconds).

We already knew from the input data the total amount of "useful" CPU time required to complete the workload, so from the available data we then calculated the following:

- Utilization, calculated as $\frac{Total\ work}{Allocation\ time}$
- Total wasted worker CPU time due to task cancellations: $Active\ CPU\ time - Useful\ CPU\ Time$.

B. Results

For a fixed task/worker ratio, enabling tail-chopping improved utilization for a wide range of parameter values

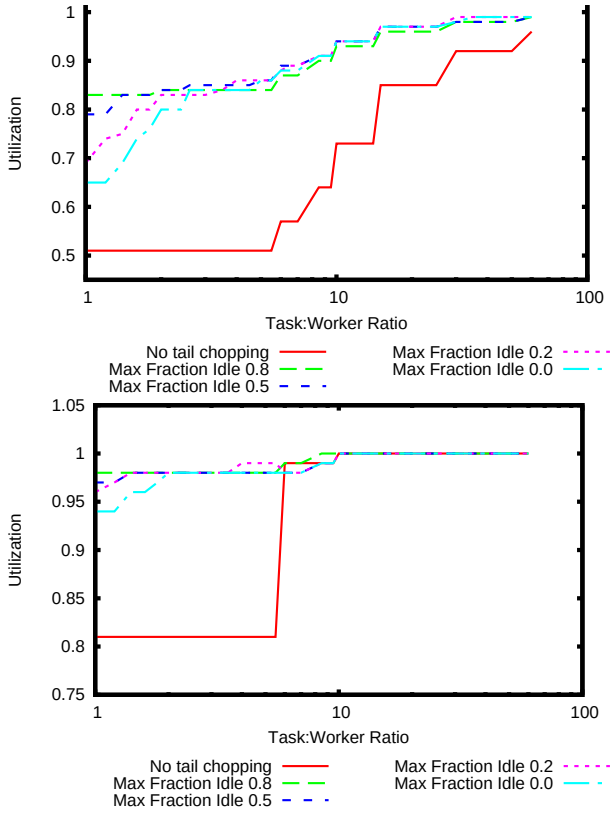

Figure 7. Effect of tail-chopping on utilization in simulation for the *real935k* workload. Top: with random task order. Bottom: with sorted task order.

for all three workloads (see Figure 7 for an example). This promising result suggests that reducing the inefficiency of idle processors typically outweighs the inefficiency of canceling tasks. Utilization levels of 90%+ and 95%+ were also more reliably achieved: utilization appears less sensitive to the task/worker ratio with tail-chopping enabled.

Enabling tail-chopping invariably increased time to solution, as was clearly expected. We must therefore look at the trade-off between TTS and utilization to determine if tail-chopping is a worthwhile exercise. If it cannot achieve a better trade-off than simply varying the task/worker ratio, then it provides little benefit.

Tail-chopping significantly improved the trade-off curves compared to the random load-balancing policy with no tail-chopping for two of three workloads: *synth15k* and *real935k*, which both had runtime distributions with elongated right tails. For example, in Figure 6, for several parameter values of maximum fraction idle, we were able to achieve 95% utilization while still getting a solution within 13 to 14 hours, which would be impossible without tail-chopping on this workload.

Comparing the trade-off curves for *real15k* and *synth15* in Figure 6, we get a strong indication that the skewedness of the distribution is a crucial variable in determining whether tail-chopping is effective. We can confidently attribute this difference to the higher variance and skewed distribution of the synthetic tasks: task count and mean runtime were identical.

The trade-off curves in Figure 6 and Figure 4 do not show clearly that one parameter for maximum fraction idle is better than another.

If we are aiming for a low time to solution, then 0.8 appears to be a good setting. The curves for 0.8 bulge out further to the left, presumably because this setting allows the vast majority of work to be done on the larger partitions, leaving only a small tail of tasks to complete on smaller partitions. For example, in Figure 1, if we chop off the tail when 80% of workers are idle, then the bulk of the work will be completed, leaving only long-running tasks.

A more sophisticated heuristic that uses available information about the workload and the runtimes of completed tasks will likely give a good trade-off more consistently than any single parameter value here.

Tail chopping provided no further benefit when applied in addition to sorting. All three workloads showed a trade-off similar to that observable in Figure 4 when tasks were sorted: enabling tail-chopping resulted in comparable or worse utilization for a given TTS. This is fairly predictable: by the time utilization starts to dip below 100% when tasks have been sorted, it is unlikely that there will be many long-running tasks left to complete.

X. Experiment

To validate our results and demonstrate the viability of the concept, we conducted an experiment, executing a real application on Intrepid with a set of parameters that had been found to be effective in the simulation.

A. Method

We augmented the Falkon task dispatcher with additional functionality to implement the approach described above to allocate and downsize partitions.

To validate the simulation results, we ran the workload of 15,000 DOCK6 tasks with and without tail-chopping. In both cases we used a task/worker ratio of 5.0. When the tail-chopping was enabled, we triggered it when 50% of workers became idle.

B. Results

The results of the experiment are illustrated in Figure 8. As expected, utilization is improved by the tail-chopping: with tail-chopping enabled we use 1,179 CPU-hours at 89.0% utilization, compared to 1,251 CPU-hours at 83.7% utilization without tail-chopping for the same workload.

Time to solution increases substantially with tail-chopping: from 39.6 minutes to 87.7 minutes. The large delay between the two allocations in the experiment with tail-chopping indicates that in our simulation we may have

underestimated the time it could take to get a new allocation of processors, but the time is likely to be highly variable on large-scale systems such as Intrepid where scheduling policies are not geared to minimizing waiting times for this particular situation. Job queues are likely be quite long and there are unlikely to be provisions in the scheduler to fast-track the requests that are continuations of a running job. Workarounds to this delay problem are be discussed in Section XI.

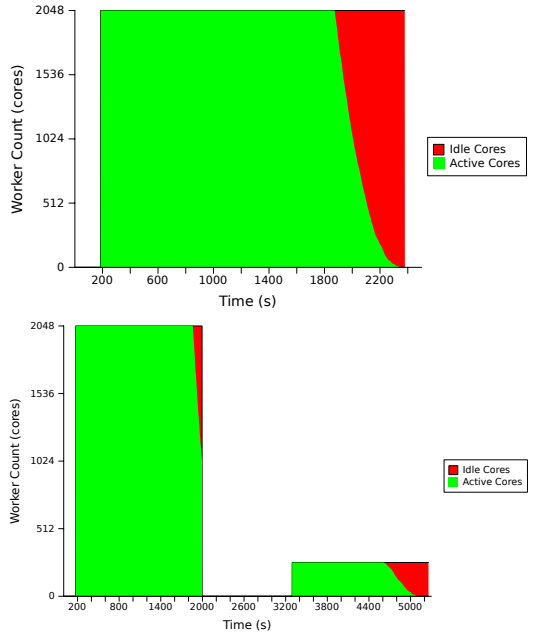

Figure 8. Worker utilization over time for the experiment with 15000 DOCK6 tasks. Top: without tail-chopping. Bottom: tail-chopping when 50% of workers are idle.

XI. Discussion and Proposals for Improvement

The simulation results in Section IX-B indicate that the basic tail-chopping approach could be valuable in enabling many-task applications to make efficient and effective use of supercomputing hardware.

We can identify three major problems that prevent tail-chopping from providing further improvements to the trade-off between utilization and time to solution: 1

1) The time spent waiting for a new allocation after tail-chopping can increase time to solution significantly, as was shown especially in Section X-B.
2) In our current system, tasks are canceled and restarted when the tail is chopped. This strategy can further increase the time to solution, as significant progress may be lost, particularly on long-running tasks, and can also decrease utilization because the CPU time used was wasted.
3) The currently used heuristics are not sophisticated, and any particular parameter choice will result in suboptimal decisions that could be avoided by using available information. For example, if the allocation is downsized when 50% of workers are idle, an allocation may be canceled even if the remaining tasks are all nearly finished. If there are a large number of long-running tasks, the allocation may be left to run for an overly-long period of time despite the fact that only 50% of workers are occupied. The fixed task/worker ratio for every partition (including after tail-chopping) is also likely suboptimal: it is probably too conservative in allocating a small proportion of workers compared with the number of trailing tasks. This is likely to increase the time to solution substantially, and lower utilization is likely affordable. Inefficient use of a small partition makes a relatively small impact on the overall utilization of a large MTC job.

Proactive provisioning of smaller processor allocations could address problem 1: one or more smaller processor allocations could be requested proactively and kept running alongside the large allocation given to the scheduler, reducing delays waiting for a new allocation after tail-chopping. If time to solution is paramount, the large partition could also be retained until a new partition came online.

The ability to downsize existing allocations (by releasing some fraction of the current processors without releasing all of them) would address problem 1, as no new allocation would need to be started. This would also partially address problem 2, as not all tasks would need to be canceled. This would require support from the supercomputer's scheduler to implement.

The ability to migrate tasks between workers within the same allocation and to workers in different allocations could address problem 2. This could be done either by checkpointing to a global file-system and restarting, which would impose significant overheads, or by peer-to-peer transfer of process state among workers. Either method requires additional application or middleware functionality.

If we have the option to downsize an allocation, then a migration feature would allow tasks to be consolidated into one region of the existing allocation, and the other region to be dropped.

Identifying tasks that have a long time remaining to run would be helpful in deciding which tasks to proactively migrate. If the distribution of tasks runtimes has a wide tail, then this strategy is possible, as tasks that have already run a long time are likely to continue to run a long time [7].

We can also imagine better heuristics that address problem 3. Using more available information allows more sophisticated estimates of the impact of tail-chopping on time to solution and utilization, given a particular workload. Runtimes of completed tasks and lower bounds on the runtime of currently running tasks are available. This information is useful for inferring some of the properties of the distribution of task runtimes, useful in estimating the

impact of scheduling decisions.

XII. Conclusion

We have described and provided examples and explanation of the "trailing task problem" that occurs in many-task computing and have characterized it relative to problems in scheduling theory.

We have shown through both simulation and experiment that the tail-chopping approach described in this paper is a promising way to address the trailing task problem. Not only can it improve the trade-off between utilization and time to solution, but it also provides a more robust way to achieve utilization levels of 90%+.

We see several directions for further research on this topic. The problem of provisioning blocks of workers could be formalized as a problem in scheduling theory, which might provide deeper insights.

Practically, however, the measures described in Section XI, if implemented in supercomputer schedulers and supported in many-task computing middleware, could greatly improve the trade-off between time to solution and utilization to the point where many-task applications become a significantly more attractive class of applications to run on supercomputers.

Acknowledgment

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research was also supported in part by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proc. IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, 2008.

[2] IBM Blue Gene team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, 2008.

[3] "Blue Waters computing system," [accessed 13-Sep-2010]. [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/system.html

[4] A. Khan, C. McCreary, and M. Jones, "A comparison of multiprocessor scheduling heuristics," in *Proc. 1994 International Conference on Parallel Processing*, vol. 2, pp. 243–250, 1994.

[5] G. Malewicz, I. Foster, A. Rosenberg, and M. Wilde, "A tool for prioritizing DAGMan jobs and its evaluation," in *Proc. 15th International Symposium on High-Performance Distributed Computing*, pp. 156–168, 2006.

[6] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, pp. 30–37, spring 2009.

[7] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[8] I. Raicu, "Many-task computing: Bridging the gap between high throughput computing and high performance computing," Ph.D. dissertation, University of Chicago, 2009.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 137–150, 2004.

[10] S. T. McCormick and M. L. Pinedo, "Scheduling n independent jobs on m uniform machines with both flowtime and makespan objectives: A parametric analysis," *ORSA Journal on Computing*, vol. 7, no. 1, pp. 63–77, 1995.

[11] M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer, "Bi-criteria scheduling of scientific workflows for the grid," in *Proc. 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pp. 9–16, 2008.

[12] P. Brucker, *Scheduling Algorithms*. Springer-Verlag, 2004.

[13] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *J. ACM*, vol. 34, no. 1, pp. 144–162, 1987.

[14] E. Krevat, J. Castaos, and J. Moreira, "Job scheduling for the bluegene/l system," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin / Heidelberg, 2002, vol. 2537, pp. 38–54.

[15] N. Desai, D. Buntinas, D. Buettner, P. Balaji, and A. Chan, "Improving resource availability by relaxing network allocation constraints on Blue Gene/P," *International Conference on Parallel Processing*, pp. 333–339, 2009.

[16] "Allocation management - Argonne Leadership Computing Facility," [accessed 13-Sep-2010]. [Online]. Available: https://wiki.alcf.anl.gov/index.php/Allocation_Management

[17] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proc. IEEE/ACM Supercomputing 2008*, November 2008.

[18] D. Moustakas, P. Lang, S. Pegg, E. Pettersen, I. Kuntz, N. Brooijmans, and R. Rizzo, "Development and validation of a modular, extensible docking program: DOCK 5," *Journal of Computer-Aided Molecular Design*, vol. 20, pp. 601–619, 2006.