JETS: Language and System Support for Many Parallel Task Computing

Justin M. Wozniak
Argonne National Laboratory
Argonne, IL, USA

Mike Wilde
Argonne National Laboratory
Argonne, IL, USA

Abstract

*Many-task computing is a now well-established paradigm for implementing loosely coupled applications on large-scale computing systems. However, few of the model's existing implementations provide efficient, low-latency support for the execution of tightly coupled applications as atomic tasks. Thus, a vast array of parallel applications can not readily be used effectively within many-task workloads. In this work, we present JETS, a middleware component that provides high performance support for many-*parallel-*task-computing (MPTC). JETS is based on a highly concurrent approach to parallel task dispatch and on new capabilities now available in the MPICH2 MPI implementation and the ZeptoOS Linux operating system. JETS represents an advancement over the few known examples of multi-level many-parallel-task scheduling systems by more efficiently scheduling many* short-duration *parallel application invocations; by overcoming the challenges of coupling the user processes of each application invocation via the messaging fabric; and by concurrently managing many application executions in various stages. We report here on the JETS architecture and its performance on both synthetic benchmarks and the NAMD molecular dynamics application.*

1 Introduction

Many-task computing (MTC) [12] has emerged as a powerful framework for the rapid development and execution of scalable scientific applications on large clusters and leadership-class supercomputers. MTC provides a simple framework in which individual sequentially-executing application jobs are scheduled en masse on individual cores (i.e., without intertask communication). These cores communicate only through the standard filesystem interfaces, although optimizations are possible [28]. The model thus is conducive to the use of scripting, workflow engines, and other familiar programming models that allow application developers to efficiently utilize large-scale parallel systems with little or no explicit parallel programming.

Since MTC makes no provisions for inter-task communication during a task's execution, it limits the flexibility available to developers who may wish to strike a balance between the MTC and HPC models. Application developers may wish to build a composite application which consists of an ensemble of parallel executions, linked together by a workflow or parameter sweep. The results of such a run may be integrated by statistical or optimization-based methods, such as a Monte Carlo algorithm, a parameter search, or other methods related to uncertainty quantification. When such an application faces challenges resulting from the large number of composite parallel executions, we consider it a many parallel-task computing (MPTC) problem. Systems that enable MPTC provide a powerful tool for scientific application developers.

A system that supports MPTC is also appealing from a systems perspective. The many-task computing model assumes that the task scheduling, startup, and shut down cycle is very fast. This rapid task rate is not supported by the native schedulers and application-launch mechanisms of today's supercomputers, but is possible through the development of a specialized, single-user scheduler. The interconnect fabrics of the largest of the TOP500 systems [24] constitutes a significant portion of the expense of the system. MPTC makes these resources available in a MTC-like model. Finally, powerful software implementations such as MPI-IO, which aggregates and optimizes accesses to distributed and parallel filesystems, are not available to MTC applications, resulting, by default, in uncoordinated filesystem accesses that are difficult to manage. The use of these algorithms and implementations could greatly increase data access rates to available cores.

In this work, we present JETS, which is designed from the ground up to support large batches of parallel tasks, in which each task execution consists of tightly coupled processes that use MPI for communication. The development of JETS involved modifications to the MPICH2 [14] package that are now publicly available. JETS runs on commodity clusters, optionally through SSH tunnels [16], and on the Blue Gene/P (BG/P) through the use of ZeptoOS func-

tionality. Thus it is applicable to clusters, grids, clouds, and high-performance systems. JETS is a highly usable system in the MTC tradition and is primarily concerned with dispatching application invocation commands to the available resources. Additionally, JETS has been integrated with the Swift workflow language [29] and Coasters [26] scheduler.

The remainder of this paper is organized as follows. Section 2 describes related work on the topic of MTC. Section 3 contains a motivating case study in MPTC and Section 4 presents an overview of the system architecture. In Section 5 we measure system performance and in Section 6 describe planned future work. We conclude in Section 7 with comments on possible new applications built using the JETS model.

2 Background

Many-task computing represents the intersection of sequential batch-oriented computing with extreme-scale computational resources. MTC is attractive to developers because of its broad portability and support from many toolkits. We emphasize that our target systems are single-site HPC resources; however, much of the foundational work in the area is based in grid and distributed computing.

Grid computing [9] provided an abundance of computational resources to scientific groups, necessitating the creation of a variety of toolkits to automate the use of these resources for common application patterns such as parameter sweeps and workflows. Parameter sweeps, supported by systems such as Nimrod [2] and APST [3], enable the user to specify a high-level definition of possible program inputs to be sampled, and the system generates the resulting job specifications and submits them to resources. In comparison, the JETS mechanism rapidly assembles independent available worker nodes into parallel jobs, without requiring support for such aggregation in the underlying resource manager. Further, JETS has been integrated with the Swift system for the management of jobs and data, and is not linked to a particular higher-level pattern.

An initial pilot job mechanism was the Condor Glide-In [10] mechanism that integrated with the full-featured Condor [22] scheduler. The Condor Glide-In mechanism essentially places a full Condor installation on the target site, including remote system calls and checkpoint/restart functionality. Panda Pilot factory [5] is a recent development that provides a pilot job wrapper mechanism that manages the distribution of worker agents as well as the initial data placement. Neither Glide-Ins nor Panda is capable, however, of aggregating multiple independent cluster compute nodes to assemble the resources needed for the execution of parallel MPI jobs.

The Falkon [19] system enabled MTC on Blue Gene/P resources, but only for single-job executions, and does not support the MPTC paradigm. On the Blue Gene/P, Falkon places workers on the system's compute nodes and communicates with them through an intermediate scheduler placed on the system's I/O nodes. Falkon primarily addresses task scheduling, although related project work produced DataDiffusion [20] to cache data for reuse among compute processes. In comparison, the JETS mechanism focuses on the deployment of MPI applications, which is not addressed by Falkon.

The SAGA BigJob [13] system enables the use of various underlying job submission mechanisms including Condor, Globus [8], and Amazon EC2 cloud allocation. SAGA places workers on the resources and coordinates with the BigJob system to place multiprocessor jobs on distributed resources. In comparison, SAGA is concerned primarily with questions regarding the distributed infrastructure and does not address the performance regime of many short-duration parallel tasks that JETS has achieved. SAGA has been used to perform replica-exchange simulations (see Section 3) with NAMD [23] in an investigation that focused on coupling the replica exchange trajectories.

Similarly, the Integrated Plasma Simulator (IPS [7]) is a dataflow-driven workflow specification system which wraps parallel MPI simulation applications into component-oriented Python objects. While originally designed for the specific needs of the plasma fusion simulation community, IPS is in fact quite general purpose in nature. Like JETS, it requests a large allocation of compute nodes as a single job from the system's underlying resource manager (such as PBS) and it manages the launching of individual application subtasks within this pool. Being more recent, JETS improves on two of IPS's limitations. First, in order to maintain its understanding of the number of free nodes in its compute-node pool, IPS must accurately predict how the underlying resource manager will assign nodes to IPS task creation requests. In complex systems such as the Cray with many NUMA and cpu-affinity issues being handled by the resource manager, this can be very tricky and error-prone logic to maintain. JETS overcomes this by doing its own node management with a JETS worker agent on each compute node. Second, IPS depends on the native systems underlying job placement and MPI launching service, such as `mpiexec` [14] on simple clusters and ALPS `aprun` on Cray systems [1]. This does not provide any straightforward way to run on systems with more complex job launching mechanisms, such as the Blue Gene/P. Again, JETS overcomes this limitations with its worker agents, which are started with simple scripts running under the native resource manager. While future Blue Gene systems may provide similar application launch capabilities to the Cray ALPS [4], the latency and performance of these capabilities are unknown, while the JETS architecture would be able to readily handle almost any imaginable architecture.

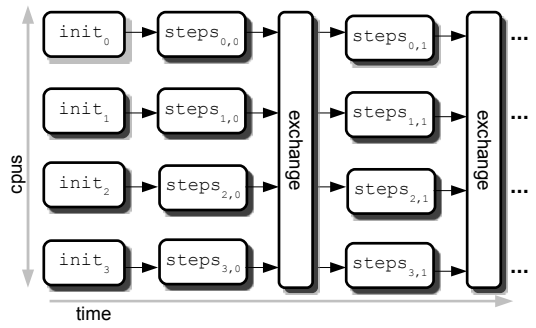

Figure 1. Workflow perspective of replica exchange method.

3 Use Case and Requirements

As a canonical example of the motivation for many-parallel-task computing, we consider a classical task and data-flow pattern from molecular dynamics. The replica exchange method (REM) [21] is a computational method to enhance statistics about a simulated molecular system by performing molecular dynamics simulation of the system at varying temperatures. These simulation trajectories, under varying conditions, are regularly stopped (typically at a rather high rate), sampled, and compared for exchange conditions. Data exchange may be required at each stopping point. The simulation is then restarted under the restart file of neighboring replica to accomplish the state exchange.

The computational workflow is diagrammed in Figure 1. The initial use case provided by our user group is as follows. Each set of CPUs is initialized with conditions including temperature. Each simulation runs as a NAMD [18] job of 256 compute cores. There are 64 concurrent simulations running on a total of 16,384 cores. Each simulation is expected to run for 10-100 simulated timesteps, for approximately 10-60 seconds of wall time depending on the configuration. Smaller individual runs produce finer granularity exchanges, which is desirable. The simulations are then stopped, and an external application process performs the replica exchange among the simulation snapshots. The simulations are then restarted from the snapshots, and the process repeats until a termination condition is satisfied approximately 12 hours later. Thus, to keep up with this workload, the scheduler would have to launch 6.4 MPI executions per second, requiring an individual process launch rate of approximately 1,638 processes per second, for a 12 hour period. The goal of this work is to present a system that provides an elegant scripting approach to the application task management while achieving this level of performance.

This process management and aggregation capability is not supported by previous systems and is notably difficult to achieve on our primary production target, the 160,000-core Blue Gene/P "Intrepid" at Argonne National Laboratory. Passing each job into a cluster scheduler is dramatically less efficient than our use of persistent worker agents, and cluster-specific policies often prevent such models of many-parallel-task computing by imposing a limit on the number of jobs in the queue per user, or other inhibiting constraints. For example, at Argonne, jobs must use a minimum of 1,024 nodes, whereas our initial application has an efficiency-based target of 64 nodes (256 cores). Thus, the scientists that motivated the REM use case above are currently running workloads using an inferior simulation approach due to the lack of MPTC support on the Blue Gene/P.

In order to support the MPTC programming model and its associated performance requirements, the work described here has achieved four primary contributions:

1. New MPICH2 features that enable individual MPI processes to be managed by an external scheduler.
2. An associated set of external routines used to control MPICH2, comprising the core JETS functionality.
3. Integration of this core JETS functionality with Swift parallel scripting language through its "Coasters" task execution provider [25]. This enables the application task flow to be specified as a high-level script that composes the individual application MPI jobs. Thus it may be used on any of the resources supported by Swift Coasters, including clusters, grids, clouds, and HPC systems.
4. A stand-alone tool (`jets`) which provides maximum performance for scripts that execute many small MPI task sets. This allows users to run very simple batches of MPI tasks without a Swift workflow using a simple task list.

4 Design

JETS is based on the basic MTC paradigm that enables users to rapidly submit large batches of ordinary command-line program executions to large resources. JETS assumes that the user can launch a worker script on the remote resources. The worker script requests work from a centralized dispatcher, which can assign work to given resources by using multiple scheduler components called *handlers*. Each handler has a specific input file format, which is basically a list of literal command lines.

Attaining high performance from the centralized JETS scheduler is critical. Additionally, deploying and using JETS could quickly become complex, since JETS involves multiple distributed resources as well as the management of user and system external processes. Thus, JETS observes the following principles:

1. Use simple, reusable threading abstractions. This task is accomplished through the use of existing concurrent data structures.

2. Separate service pipeline processes through simple interfaces. In JETS, socket management, handler processing, and external process management connect through obvious mechanisms and are each arbitrarily concurrent.

3. Support ready composition and decomposition. JETS components are easily composed into frameworks appropriate for different environments (e.g. for Swift, standalone usage, or use within other frameworks such as IPS). The components can also be decomposed for separate usage (e.g. the JETS worker agent can serve as a useful component of a benchmarking test framework).

4. Assume disconnection is likely. The JETS service and workers can operate independently and are individually diagnosable.

JETS integrates the technologies described in the following subsections. Although JETS may be used as a standalone system, its core features are available in Swift.

4.1 Swift/Coasters

An original design goal of JETS was to bring first-class support for MPI programs into the Swift system. Swift [26] is a highly concurrent programming model for deploying workflows to grids and clusters. Swift was originally developed for grid resources and is essentially a high-level language to build workflows for the Commodity Grid (CoG) Kit [25]. To support fast task scheduling, Swift uses as associated *provider*, called *Coasters*, that runs as a network of external services, including a CoasterService and worker scripts. Swift communicates with the CoasterService to schedule jobs and data movement to the distributed resources. The Swift/Coasters system can run directly on an HPC resource, launching sequential MTC tasks at high rates and employing filesystem access optimizations.

Swift/Coasters operations is diagrammed in Figure 2. First, the SwiftScript is compiled ① to the workflow language Karajan, which contains a complete library of execution and data movement operations. Tasks resulting from this workflow are processed by well-studied, configurable scheduling algorithms and processed by underlying service providers including local execution, PBS, SGE, Globus, Cobalt [6], or the Coasters provider. The Coasters provider consists of a connection to the CoasterService, which itself is deployed as a task ②. The CoasterService in turn uses

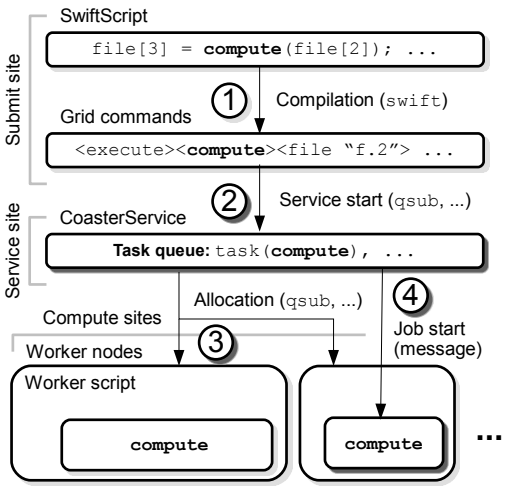

Figure 2. Swift/Coasters architecture.

task submission to deploy one or more allocations of workers, in blocks of varying sizes and durations ③. The CoasterService schedules user tasks inside these blocks of available computation time and rapidly launches them via RPC-like communication over a TCP/IP socket ④. Data transfer operations may also be performed over this connection, removing the need for a separate data transfer mechanism. On the BG/P, the CoasterService may be placed on a login node, communicating with its workers over the internal BG/P network.

4.2 MPI Process Management: Hydra

The ability to run MPI programs in JETS is built on facilities offered by the MPI implementation. The MPICH2 implementation of the MPI standard, used in this work, employs a *process manager* that is responsible for launching the individual user processes in coordination with user input and and existing scheduler such as the local operating system or a distributed scheduler such as PBS [11].

The default process manager in MPICH2 is currently Hydra [**?**]. The MPICH2 process manager is responsible for launching the user processes on the requested resources through a bootstrap control interface using an available bootstrapping mechanism such as `ssh`. Hydra was modified by this work by adding a bootstrap mechanism, called *bootstrap-none*, that employs no existing external scheduler: it simply reports proxy commands to its output and performs its ordinary network services. Thus, any other controlling process may use this specification to bring up the Hydra network and launch the MPI application. This works on any system that provides sockets, including the

ZeptoOS system described below.

4.3 ZeptoOS

JETS relies on the ability to dynamically bind MPI processes together using the sockets abstraction provided by POSIX-like operating systems. While TCP/IP sockets are a typical mechanism for MPI job coupling on commodity clusters, on the Blue Gene/P these APIs are not provided by default. The ability to perform sockets-based MPI messaging on the BG/P is made possible through the use of functionality provided by ZeptoOS. This Linux-based compute node operating system replaces the default IBM Compute Node Kernel (CNK) and enables the user processes to communicate over the BG/P torus interconnect using an ethernet network device. This virtual network is then used by the MPI programs launched by JETS.

4.4 JETS

JETS is designed to orchestrate the previously described systems into a simple framework for MPTC. JETS provides the following features:

1. **Speed:** JETS is designed to outperform process launchers like `ssh` while enabling security (e.g., an OpenSSH tunneling). JETS uses compute sites as they become available and quickly combines them into MPI-capable groups.

2. **Local storage:** JETS can cache libraries and tools (such as the MPICH2 proxy binary) and even user data on node-local storage, which boosts startup performance and thus utilization for ensembles of short jobs.

3. **Fault tolerance:** JETS automatically disregards workers that fail or hang, minimizing their impact on the overall system.

4. **Flexibility:** JETS enables fast submission of jobs to worker nodes unreachable by systems such as OpenSSH (e.g., the Blue Gene/P compute nodes), and enables the use of smaller MPI sizes than allowed by some site policies.

The essential concept in JETS is to transform an MPI job specification into a set of MPICH2 proxy job specifications by communicating with a background `mpiexec` process and rapidly submitting those proxy jobs to the worker scripts for execution. Performance benefits are obtained through the local, concurrent execution of the `mpiexec` processes and the use of the worker agents. Hundreds of `mpiexec` processes do not place a noticeable load on the submit site. Additional performance benefits are gained through the deployment of the proxy executable, the user executable, and related libraries in local storage on the compute sites. JETS contains features to automate these file transfers. JETS features are available through Coasters; in this work, we consider the design and performance characteristics of the new JETS functionality in stand-alone form, which can operate without Swift/Coasters.

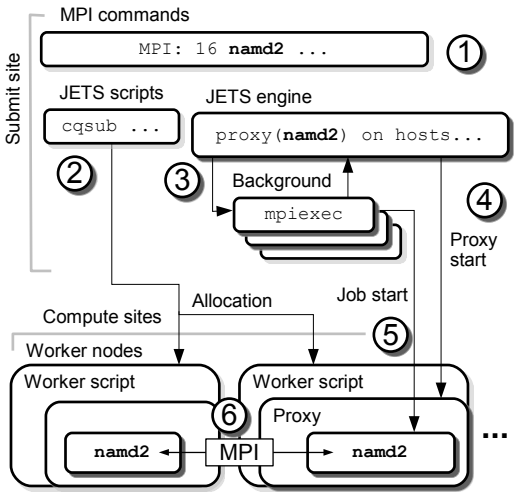

Figure 3. JETS architecture.

Stand-alone JETS operation is diagrammed in Figure 3. The input to JETS is a simple text file containing command lines to be executed and MPI-specific information such as the number of nodes on which to run ①. The user launches the worker scripts with provided allocation scripts, which use an external system such as `ssh` or Cobalt ②. Once running, each worker is persistent, capable of executing many tasks as a pilot job. Workers report readiness to the JETS engine. When the engine has obtained notification from the requisite number of workers for the next user job in the user list, it launches the `mpiexec` binary in the background, provides it with the host information from the ready workers, and obtains proxy startup information ③. The `mpiexec` process continues running in the background. The proxy jobs are issued to their respective workers ④, and the proxies connect to the `mpiexec` process to negotiate the MPI job start ⑤. The MPI application processes can locate each other to begin MPI communication ⑥. On job completion, the `mpiexec` process and its proxies terminate. The `mpiexec` output is checked for errors, and the workers request additional work, resuming the cycle.

Since the essential JETS functionality is to break MPI executions into composite single-process jobs, JETS functionality was readily made available in Coasters and thus to Swift workflow applications. Although not diagrammed for

lack of space, running MPI jobs in Swift through the JETS framework is performed by combining the task generation and worker management aspects of the Swift/Coasters architecture with the `mpiexec` process management of the JETS architecture. In the following section, we report on the performance of the stand-alone JETS system to avoid the measurement complexity in a full workflow.

5 Performance Results

In this section, we present performance results obtained by running JETS in a cluster setting, in a high-performance setting, in a faulty environment, and for use by the NAMD application.

5.1 Sequential Tasks

First, we demonstrate the basic task rate at which JETS can submit individual sequential tasks to a computing resource. In this series of tests JETS was configured to run on Surveyor, an IBM Blue Gene/P system at Argonne National Laboratory. Each task consisted of an external process that did no work; thus, only the cost of the process startup itself is considered. First, we measured the rate at which the BG/P compute node can launch processes without JETS (no communication), using all four available cores. This is shown as the "ideal" measurement. Then, JETS is used to submit jobs to allocations of increasing size.

As shown in Figure 4, JETS scales well, achieving over 7,000 job launches per second on the full rack of Surveyor, which consists of 1,024 compute nodes containing 4,096 cores. This result indicates that JETS will be very capable of submitting jobs generated by the more complex MPICH2-based mechanism described previously for MPI-based workloads.

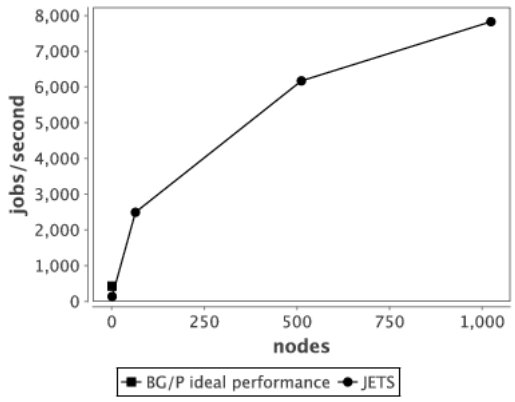

Figure 4. JETS results, sequential tasks.

5.2 MPI Communication Performance

Next, we measure the messaging performance penalty due to the use of the sockets-based MPICH2 communication mechanism used by the system. On the Blue Gene/P, the vendor-provided communication library is expected to be faster than the socket abstraction used by our MPICH2 library. As described above, the use of the ZeptoOS-based messaging abstraction is expected to increase message latency and reduce transmission bandwidth. In this test, a simple "ping-pong" MPI test was run on two nodes, each of which alternates between calls to MPI blocking send and receive functions. The buffer was filled once with random data of the given size and sent back and forth the given number of times. The runtime was measured using `MPI_Wtime`. The program was compiled and run in each of two modes: "native" mode, which was compiled with `bgxlc` and uses the default system kernel and settings; and "MPICH/sockets" mode, which uses the MPICH2 library running on the Zepto sockets layer.

As shown in Figure 5, using MPICH2 as we do results in much higher latency for small messages and slightly slower bandwidth for large messages. This is primarily due to the use of TCP by the ZeptoOS mechanism. While this performance penalty may be problematic for some applications, it must be weighed against the flexibility and functionality offered by ZeptoOS features, and the fault recoverability offered by TCP-based APIs. Possible network enhancements are considered below in Section 6, and the reliability characteristics are demonstrated below in Section 5.5.

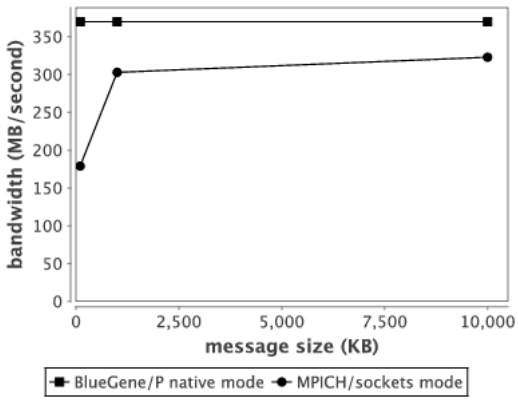

Figure 5. MPI messaging performance.

5.3 MPI Task Launch Performance: Cluster Setting

In our next series of tests JETS was configured to run on Breadboard, a network of x86-based compute servers at

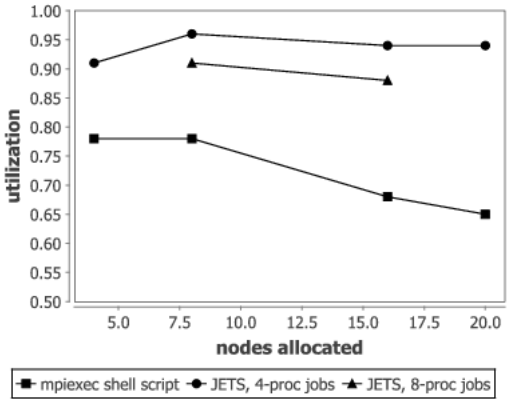

Figure 6. MPI/JETS results, cluster setting.

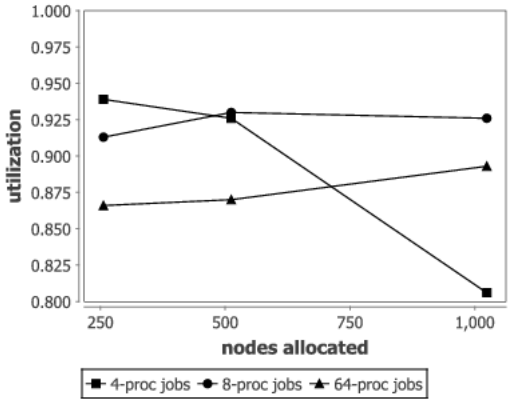

Figure 7. MPI/JETS results, BG/P setting.

Argonne National Laboratory. In this test, a simple MPI application was constructed for benchmarking purposes that starts up, performs an MPI barrier on all processes, waits for a given time, performs a second MPI barrier, and exits. The number of MPI processes in each invocation of this application is independent of the size of the whole allocation. In this test, each data point represents the utilization obtained by running a large batch of application invocations of varying sizes (shown as n-proc) inside an allocation of the size given on the x-axis. Each job wait duration was 1 second. System utilization is reported as

$$\text{utilization} = \frac{\text{duration} \times \text{jobs} \times n}{\text{allocation} \times \text{time}}. \quad (1)$$

The workload was run in each of two modes: a "shell script" mode, which simply calls `mpiexec` repeatedly, and a mode in which JETS was used. The shell script mode is capable of only using the entire allocation, whereas the JETS mode may be run at smaller, varying sizes. As shown in Figure 6, JETS can achieve approximately 90% system utilization for the extremely short (single-second) tasks submitted. This greatly exceeds the utilization available in an `mpiexec`-based shell script and indicates that the performance is capable of scaling to larger resources.

5.4 MPI Task Launch Performance: Blue Gene/P Setting

JETS was again configured to run on Surveyor. The user application in this case is the same application used in the cluster setting but was run for a 10-second duration. Each node here contains 4 cores. We only place one MPI process per core as additional processes per core are managed locally by the Hydra proxy and are not performance challenges visible to JETS.

JETS scripts are used to enable compatibility with ZeptoOS and high performance at this system scale. We used the ZeptoOS local RAM-based file system to store the application binary, the Hydra proxy, and requisite libraries in this local store. The script sets `LD_LIBRARY_PATH` to suppress any lookups to GPFS, which are much more time-consuming than local lookups. The scripts also add an entry to `/etc/hosts` to enable the Hydra proxy to find the JETS service on the login node. The ZeptoOS IP-over-torus feature was enabled to provide each node with an IP address obtainable through `ifconfig`. This address is connectable by all peer nodes in the allocation, and was used by the JETS components to connect the Hydra processes.

We ran the same application used in Section 5.3 with a 10 second duration. MPI executions were constructed from nodes in the allocation without regard for their relative network positions; the default JETS behavior is to group nodes in first come, first serve order.

Results are shown in Figure 7. Each line shown represents one component task size, 4, 8, or 64-processor tasks. These task sizes were chosen to highlight JETS performance characteristics. Each size task was run on allocation sizes of 256, 512, and 1,024 nodes, and only one core per node was used. The number of tasks in the batch was selected such that each node processed 20 10-second tasks. As shown in the figure, 4-processor tasks at this duration are sustainable up to about 512 nodes after which there is a significant degradation from the utilization achieved by the 8-processor tasks, this is due to the load on the central JETS scheduler becoming excessive. 64-process tasks are individually slower to start, which results in lower utilization in small allocations. However, this penalty becomes smaller as the task size becomes a smaller fraction of the available nodes.

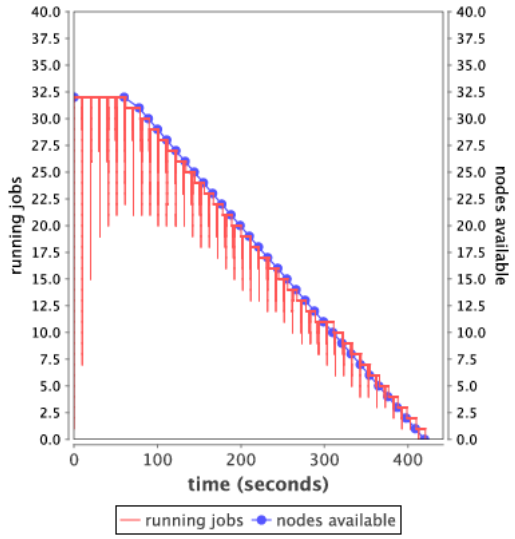

Figure 8. MPI/JETS results, faulty setting.

5.5 Task Management: Faulty Setting

In this series of tests, we demonstrate that JETS is capable of maintaining high utilization on the remaining useful compute nodes of a faulty allocation in which worker script processes terminate early due to system hardware or software failure. In this case, JETS was run again on Surveyor. The sequential application from Section 5.1 was used again. A fault injection script was run on the submit site that terminated worker scripts, one at a time, at random, at regular 10 second intervals. Due to skew among the application tasks, this could result in a worker being terminated during or between application task executions. The worker and user task start and stop times were recorded, which allowed the total system load and worker count to be obtained and plotted over time.

Results are shown in Figure 8. The number of worker nodes in operation are shown as "nodes available"; the number steadily decreases from the original level of 32 workers down to zero over a period of about 320 seconds. The number of running application jobs is plotted as "running jobs". Initially, the jobs execute in lockstep, resulting in large utilization dips which become smaller over time. These large dips are due to congestion on the JETS scheduler when multiple nodes become available for work simultaneously. The dips become less dramatic as skew reduces the number of simultaneous work requests. After the 100 second mark, the number of running jobs is bounded by the number of nodes available. The number of running jobs stays very close to the number of nodes available, indicating that JETS maintains a very high utilization rate on the available nodes.

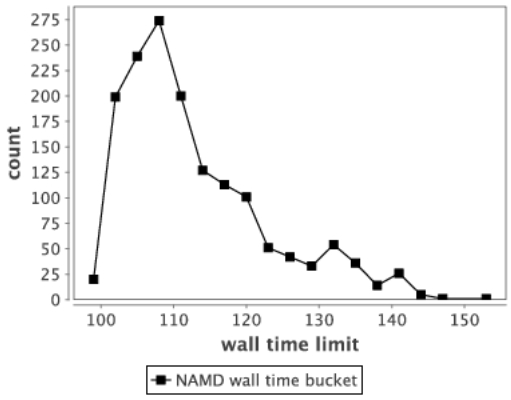

Figure 9. NAMD wall time distribution.

5.6 Application: NAMD

In this series of tests, we report utilization results observed when running a bag-of-tasks batch of NAMD executions, with settings similar to that of the replica exchange method. JETS was configured to run on Surveyor. The NAMD application was configured to run one process per node. A batch of 32 NAMD runs comparable to those used in an REM run was provided to us by a NAMD user. We duplicated those cases and ordered them in a round-robin fashion. For each allocation size from 256 to 1,024, we created a batch that would require 6 executions per node on average. Each run simulated an NMA [17] system of 44,992 atoms for 10 timesteps, which runs in NAMD for approximately 100 seconds on 4 BG/P processors.

Application I/O is as follows. The application reads 5 files totaling 14.8 MB of input and writes 3 files totaling 2.2 MB of output, in addition to about 11 KB on standard output. The I/O time is contained in the application wall time. The NAMD application performed I/O directly to the PVFS filesystem available on Surveyor. Standard output was directed back to the `mpiexec` process. In the JETS framework, standard output is directed from the application to the Hydra proxy, over the network to the `mpiexec` process, into the JETS process, and then into a file. For the largest run, this produced 16 MB of output over 11 minutes, which was not enough to cause congestion.

The full rack (1,024-node) batch consisted of 1,536 4-processor jobs. The run time distribution for these jobs is shown in Figure 9. While the majority of the tasks fall between 100 and 120 seconds, many tasks exceed this, running up to 160 seconds. The utilization results, shown in Figure 10, shows that utilization is near 90%. Load level for the full rack batch, computed as the number of busy cores at each point in time, is shown in Figure 11. For a longer run, utilization could be higher as the effect of the ramp-up

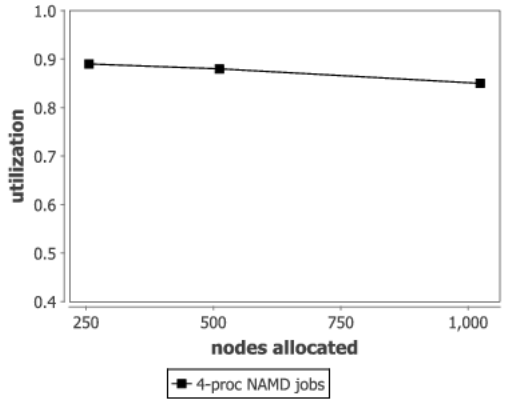

Figure 10. NAMD/JETS utilization results.

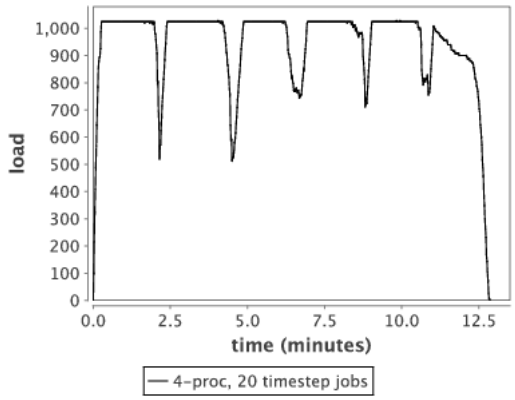

Figure 11. NAMD/JETS load level results.

and long-tail effects are amortized.

6 Future Work

At the time of this writing, JETS has only been recently developed, and while the performance measures presented here are preliminary, they are already very encouraging. However, many improvements and extensions to JETS are planned, including the following.

In order to simplify its implementation and focus on algorithms, the initial JETS version uses MPI over standard TCP sockets. In order to take better advantage of the native high performance interconnect fabric on petascale systems such as Blue Gene/P and Cray XE, we plan to enhance JETS with support for vendor-provided MPI over the native communication fabric libraries (such as Blue Gene DCMF and Cray GNI).

While JETS currently operates at high speed in part because it uses a simple FIFO queuing approach, we plan to explore the addition of priority-based scheduling and backfill, and to measure scheduler performance on workloads of varying size tasks. (At the same time, such workloads seldom occur in typical MPTC applications, and are thus of low priority to current user applications).

We plan to add the "multiple-job-size spectrum" allocator of the Coasters mechanism to JETS to enable it to request resources from the underlying system scheduler in a "spectrum" of various node counts, to enable it to obtain resources quickly in the face of unknown queue compositions and system load conditions.

Fruitful experiments can be conducted using MPI-IO from JETS-initiated MPTC workloads, and optimizations can be envisioned for supporting the passing of MPI-IO-written and -read datasets within an MPTC data flow. Similarly, such high-performance data passing schemes can also be evaluated using Global Arrays [15] or distributed hash tables [27].

7 Conclusion

The ensemble study, consisting of the composition of large numbers of many-processor MPI executions, is an increasingly popular paradigm that is poorly supported by existing systems. In this work, we described a new lightweight mechanism to manage the scheduling of large numbers of MPI application executions. Our work is focused on gaining high utilization rates for applications on large-scale HPC resources. We addressed the coordination of large numbers of CPUs, the management of many MPICH2 startup processes, the rapid distribution of job specifications to workers, and the construction of application scripts through integration with the Swift language and runtime system.

From a performance perspective, we demonstrated that the JETS task scheduler can launch jobs at a rate exceeding that of previous many-task schedulers. Additionally, we provided new mechanisms for the deployment of MPI applications into many-tasks systems; in particular, the new MPICH2 functionality could be reused by other groups interested in novel strategies to launch MPI applications.

We expect that JETS and related systems will emerge as powerful tools in important areas, including rapid prototyping of batches of existing codes and large ensemble studies based on loosely coupled MPI runs. Our system promotes the rapid development of large runs of existing codes through its simple model and optional scripting language interface. The system provides a shell script-like model but offers much better performance and management capabilities. New applications could be designed around the JETS model. These applications would benefit from the ability of the JETS to manage multiple scheduler allocations in a high-performance, fault-tolerant way, and the software development benefits from using the high-level Swift model.

Acknowledgments

The authors would like to thank Wei Jiang of Argonne National Laboratory for help in constructing the REM use case, as well as Ray Loy, Kazumoto Yoshii and Kamil Iskra for support in using the Blue Gene system tools and ZeptoOS.

This research is supported in part by NSF grants OCI-721939 and OCI-0944332 and by the U.S. Department of Energy under contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility, TeraGrid, the Open Science Grid, the UChicago / Argonne Computation Institute Petascale Active Data Store and Beagle.

References

[1] *Workload Management and Application Placement for the Cray Linux Environment, Document number S–2496–3103*. Cray Inc., Chippewa Falls, WI, USA, 2011.

[2] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid. In *Proc. International Parallel and Distributed Processing Symposium*, 2000.

[3] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4), 2003.

[4] T. Budnik, B. Knudson, M. Megerian, S. Miller, M. Mundy, and W. Stockdell. Blue gene/q resource management architecture. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), co-located with IEEE/ACM Supercomputing 2010*, 2010.

[5] P.-H. Chiu and M. Potekhin. Pilot factory a Condor-based system for scalable pilot job generation in the Panda WMS framework. *Journal of Physics: Conference Series*, 219, 2011.

[6] Cobalt web site. http://trac.mcs.anl.gov/projects/cobalt.

[7] S. S. Foley, W. R. Elwasif, A. G. Shet, D. E. Bernholdt, and R. Bramley. Incorporating concurrent component execution in loosely coupled integrated fusion plasma simulation. In *Component-Based High-Performance Computing 2008*, 2008.

[8] I. Foster. What is the Grid? A three point checklist. *GRIDToday*, 2002.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1st edition, 1999.

[10] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3), 2002.

[11] R. L. Henderson and D. Tweten. Portable batch system: Requirement specification. Technical report, NAS Systems Division, NASA Ames Research Center, 1998.

[12] I. F. Ioan Raicu and Y. Zhao. Many-task computing for grids and supercomputers. In *Invited Paper, Proc. Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.

[13] A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Proc. CCGrid*, 2010.

[14] MPICH web site. http://www.mcs.anl.gov/research/projects/mpich2.

[15] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10(2), 1996.

[16] OpenSSH web site. http://www.openssh.com.

[17] NMA structure in the Protein Data Bank. http://www.rcsb.org/pdb/ligand/ligandsummary.do?hetId=NMA.

[18] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kal'e, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16), 2005.

[19] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Towards loosely-coupled programming on petascale systems. In *Proc. SC'08*, 2008.

[20] I. Raicu, Y. Zhao, I. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, 2008.

[21] Y. Sugita and Y. Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, 314(1-2):141 – 151, 1999.

[22] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.

[23] A. Thota, A. Luckow, and S. Jha. Efficient large-scale replica-exchange simulations on production infrastructure. *to appear in: Philosophical Transactions of the Royal Society of London A*, 2011.

[24] Top 500 web site. http://www.top500.org.

[25] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9), 2001.

[26] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *to appear in Parallel Computing*, 2011.

[27] J. M. Wozniak, B. Jacobs, R. Latham, S. Lang, S. W. Son, and R. Ross. Implementing reliable data structures for MPI services in high component count systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*. Springer, 2009.

[28] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, and M. Wilde. Design and evaluation of a collective I/O model for loosely-coupled petascale programming. In *Proc. MTAGS Workshop and SC'08*, 2008.

[29] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. Workshop on Scientific Workflows*, 2007.

Notice