Noname manuscript No.
(will be inserted by the editor)

JETS: Language and System Support for Many-Parallel-Task Workflows

Justin M. Wozniak, Michael Wilde, and Daniel S. Katz

the date of receipt and acceptance should be inserted later

Abstract Many-task computing is a well-established paradigm for implementing loosely coupled applications (tasks) on large-scale computing systems. However, few of the model's existing implementations provide efficient, low-latency support for executing tasks that are tightly coupled multiprocessing applications. Thus, a vast array of parallel applications cannot 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 advance over the few known examples of multilevel many-parallel-task scheduling systems: it more efficiently schedules and launches many *short-duration* parallel application invocations; it overcomes the challenges of coupling the user processes of each multiprocessing application invocation via the messaging fabric; and it concurrently manages many application executions in various stages. We report here on the JETS architecture and its performance on both synthetic benchmarks and an MPTC application in molecular dynamics.

1 Introduction

> *What matters is that all of the components* work together. [25]

The high-performance computing (HPC) systems of today almost exclusively run Unix and Linux operating systems, yet lack a fundamental strength of the Unix philosophy: *program composability and modularity*. HPC systems are controlled by schedulers that often greatly restrict the feature set available to the shell programmer. Schedulers and resource management systems often enforce multiple policies that limit software flexibility. As a result, new compositions of existing software features in scientific software are often developed by editing C and Fortran source code, at great effort.

Address(es) of author(s) should be given

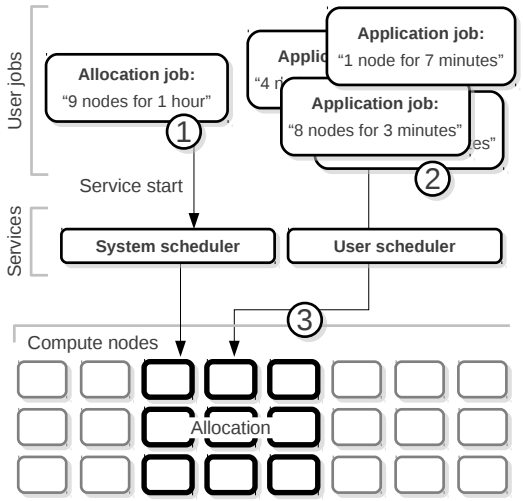


Fig. 1: Model for many-parallel-task computing.

Scheduled systems (PBS, Cobalt) run user programs inside *allocations*, which grant fixed computational resources to a user for a fixed amount of time (although programs may exit before the allocation expires). The user has little control over exactly when a program will start because the time is determined by other users in the queue. Scheduling multiple programs to work together is a complex co-scheduling problem that has not caught on with users. Allocation sizes may be restricted by site policy to minimum processor counts or times. Within an allocation, it is typically difficult or impossible to launch multiple processes in succession; if this capability is available, it is typically possible only with trivial control flow and simple resource usage. Among allocations, some schedulers offer rudimentary workflow functionality. Since separately allocated jobs are queued separately, however, a succeeding allocation is unlikely to start immediately when a preceding allocation exits. Allocations may take on the order of minutes to boot, making workflows constructed this way inefficient. Communication among allocations may be restricted for security reasons.

Solutions for interacting productively with these systems is related to problems addressed in grid computing. Among allocations, advanced workflow specification languages may be used, and scheduler abstraction interfaces may aid in workflow deployment. Additionally, within allocations, pilot job mechanisms may help in the reuse of allocation resources.

The model of interest to this work combines several of these grid-based solutions to HPC-specific problems. A model of our problem is shown in Figure 1. The user starts by creating one large allocation on the computing system ①. Next, the user provides or generates a (possibly dynamic) set of job definitions ②, which contain processor count and run-time requirements, and instantiates the user scheduler. These are launched efficiently inside the allocation ③.

Achieving this functionality requires multiple technical features. First, it requires a full-featured operating system on the compute nodes capable of communicating with the user scheduler and managing subordinate user properties. Second, it requires the development of a fast user scheduler. Third, it requires the ability to remotely invoke MPI programs without relying on previous process managers, the key contribution of this work. Fourth, it requires solutions to many related problems, such as workflow management, fault tolerance, and other systems challenges.

1.1 Many-Task Computing

Many-task computing (MTC) [36] has emerged as a powerful concept for the rapid development and execution of scalable scientific applications on large clusters and leadership-class supercomputers. The MTC model consists of many, usually sequential, individual applications (called tasks) that are executed on individually addressable system components, such as processor cores, without intertask communication. These tasks communicate only through the standard filesystem interfaces, although optimizations are possible [54]. MTC allows use of large-scale parallel systems with little or no explicit parallel programming.

Application developers may wish to build a composite application comprising 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. The model thus is conducive to the use of scripting, workflow engines, and other familiar programming models that allow application developers to efficiently utilize a variety of systems, from single systems where tasks are executed in sequence, to parallel and distributed systems, where the number of tasks that can be executed concurrently is limited by the scale of the system, the efficiency of scheduling and running the tasks, and the implicit parallelism of the application itself.

The MTC model has successfully been applied to problems in a variety of scientific domains, including computational biochemical investigations, such as protein structure prediction [10, 21, 50], molecular dynamics simulations of protein-ligand docking [37] and protein-RNA docking, and searching mass-spectrometry data for posttranslational protein modifications [26, 50]; modeling of the interactions of climate, energy, and economics [43, 50]; postprocessing and analysis of climate model results; explorations of the language functions of the human brain [17, 24, 42]; creation of general statistical frameworks for structural equation modeling [4]; and image processing for research in image-guided planning for neurosurgery [12] and astronomy [50].

Since MTC makes no provisions for intertask 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. It does not make the benefits of the high-performance interconnect available to the application. In this paper, we demonstrate the ability to make the interconnect available by allowing user tasks to use multiple processors and internally communicate through MPI. Thus, programs built on MPI and related technologies may be brought into the MTC model. In a

many-core setting, this also enables the use of shared-memory or other hierarchical or node-local communication mechanisms, although that is not developed here.

1.2 Many-Parallel-task computing

MTC, as a research field, addresses the problems that emerge from launching many individual sequential processes on a large-scale system. Analogously, an application that faces challenges resulting from a large number of parallel executions is an MPTC problem. Systems that enable MPTC provide a powerful tool for scientific application developers.

In addition to the the usability benefits of MTC, MPTC also provides benefits from a systems perspective, in that it allows many-task applications to make good use of HPC interconnects. First, the native schedulers and application-launch mechanisms of today's supercomputers do not support a sufficiently fast task scheduling, startup, and shutdown cycle to allow implementations of the many-task computing model to work efficiently, but the development of a specialized, single-user scheduler can allow many task applications to use a high fraction of the system compute resources. Second, MPTC makes the interconnect fabric resources available to the tasks in an MTC-like model. These constitute a significant portion of the expense of the largest of the TOP500 systems [47].

Additionally, MPTC allows tasks to use powerful software implementations such as MPI-IO, which aggregate and optimize accesses to distributed and parallel filesystems. The use of these algorithms and implementations can greatly increase data access rates to available cores, making better use of the storage system than is the case for today's MTC applications, which by default use uncoordinated filesystem accesses that are difficult to manage. For example, given N MTC processes, the filesystem would be accessed by N clients; however, for 16-process MPTC tasks using MPI-IO, the number of clients would be $N/16$.

In this work, we present JETS, which is designed from the ground up to make good use of supercomputer resources 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 [31] package that are now publicly available. JETS runs on commodity clusters, optionally through SSH tunnels [33], and on the Blue Gene/P (BG/P) through the use of ZeptoOS functionality. Thus it is applicable to clusters, grids, clouds, and high-performance systems. JETS is a highly usable system in the MTC tradition and is concerned primarily with dispatching application invocation commands to available resources. Additionally, JETS has been integrated with the Swift workflow language [55] and Coasters [18] scheduler.

This paper describes the MPTC problem in more detail and reports four main technical contributions:

1. We have designed, implemented, and demonstrated new MPICH2 features that enable individual MPI processes to be managed by an external scheduler.
2. We have designed, implemented, and demonstrated an associated set of external routines used to control the MPICH2 process manager (the Hydra component), constituting the core JETS functionality.
3. We have designed, implemented, and demonstrated a stand-alone tool (`jets`) that provides maximum performance for scripts that execute many small MPI

task sets. This allows users to run simple batches of MPI tasks using a simple task list produced by hand or by running a generator script, assuming all job specifications may be determined in advance.
4. We have integrated the core JETS functionality with the Swift parallel scripting language through the Swift execution layer. This enables a user to compose the application together as a loosely coupled collection of multiprocessor jobs. Such an application is driven by a high-level, dynamic script that is capable of making branch decisions at run time. Additionally, it may be used on any of the resources supported by Swift and Coasters, including clusters, grids, clouds, and HPC systems.

1.3 Applications, including Molecular Dynamics

Many scientific applications have the potential to benefit from MPTC techniques. The performance characteristics of the component jobs in an application workflow determine the applicability of MPTC techniques.

For a typical large-scale HPC system, such an application should necessarily be able to utilize $O(10,000)$ processors or more concurrently, across multiple running jobs. Individual jobs should run for seconds or minutes and require tens to hundreds of processors. Jobs that exceed these parameters in run time or processor count are candidates for using the traditional systems scheduler, jobs that run for subsecond run times on single processors are candidates for systems such as ADLB [30] or Scioto [11] (libraries that require code modification). Since MPTC logically encompasses MTC, it includes application components from that space as well.

In this work, we focus on replica exchange molecular dynamics (REM) via NAMD as an example application. The REM algorithm is described in more detail below. NAMD jobs for REM fall within the MPTC range. Existing techniques for REM in NAMD focus on modifying the NAMD codebase and recompiling. NAMD contains about 30,000 lines of Charm++ and C++ code in addition to Tcl features. Recompiling NAMD for the Blue Gene/P with optimizations takes multiple hours. Thus, a lightweight technique for recomposing application logic in workflows from component calls to NAMD jobs is highly desirable.

1.4 Contents

The remainder of this paper is organized as follows. Section 2 describes work related to the topic of MTC. Section 3 contains a motivating case study in MPTC, and Sections 4 and 5 describe the components used to build JETS and its system architecture. In Section 6 we measure system performance and in Section 7 describe planned future work. We conclude in Section 8 with comments on possible new applications built by using the JETS model.

2 Background and Related Work

Many-task computing represents the intersection of sequential batch-oriented computing with extreme-scale computational resources. MTC is attractive to develop-

ers 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 [15] 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 [1] and APST [3], enable the user to specify a high-level definition of possible program inputs to be sampled. The system then generates the resulting job specifications and submits them to resources. In comparison, the JETS mechanism rapidly assembles independent available compute 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 [16] mechanism that integrated with the full-featured Condor [45] scheduler. The Condor Glide-In mechanism essentially places a full Condor installation on the target site, including remote system calls and checkpoint/restart functionality. Workflows may be launched inside an Condor allocation with GlideinWMS [40]. Panda Pilot factory [7] is a recent development that provides a pilot job wrapper mechanism to manage the distribution of worker agents as well as the initial data placement. Neither Condor Glide-In 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 SAGA BigJob [29] system enables the use of various underlying job submission mechanisms including Condor, Globus [14], 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 [46] in an investigation that focused on coupling the replica exchange trajectories. Related work in a widely distributed context allowed MPI jobs to run across resource managers by using queue time estimates provided by a queuing time predictor [6].

Similarly, the Integrated Plasma Simulator (IPS [13]) is a dataflow-driven workflow specification system that 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 general purpose. 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, 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 task can be tricky and requires user error-prone logic. JETS overcomes this problem by doing its own node management with a JETS worker agent on each compute node. Second, IPS depends on

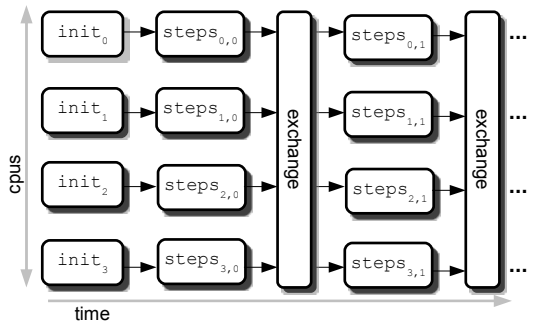


Fig. 2: Workflow perspective of replica exchange method.

the native systems underlying job placement and MPI launching service, such as `mpiexec` [31] on simple clusters and ALPS `aprun` on Cray systems [23]. 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 limitation with its worker agents, which are started with simple scripts running under the native resource manager. While future Blue Gene systems may provide application launch capabilities similar to the Cray ALPS [5], the latency and performance of these capabilities are unknown, whereas the JETS model would be able to readily handle almost any imaginable architecture.

The Falkon [37] system enables 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 [38] 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.

3 Use Case and Requirements

As a canonical example of the motivation for many-parallel-task computing, we consider a classical task and dataflow pattern from molecular dynamics. The replica exchange method (REM) [44] 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 2. 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 [35] task 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 are 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 our work is to present a system that provides an elegant scripting approach to 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 512 nodes, whereas our initial application has an efficiency-based target of 64 nodes (256 cores). Thus, the scientists who motivated the REM use case above are currently running workloads using an inferior simulation approach because of the lack of MPTC support on the Blue Gene/P.

Attaining high performance from the centralized JETS scheduler is critical. Additionally, deploying and using JETS could quickly become complex, because JETS involves multiple distributed resources as well as the management of user and system external processes. Thus, the JETS architecture 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, stand-alone 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.

4 Technologies

JETS integrates the three technologies described in the following subsections. Although JETS may be used as a stand-alone system, its features are also available in Swift.

4.1 Swift/Coasters

An original design goal of JETS was to support the MPTC model in the Swift system. Swift [51] 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 [48]. To support fast task scheduling, Swift uses an 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.

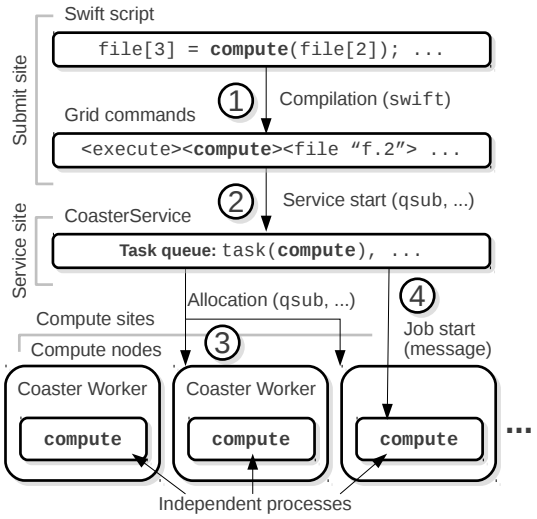


Fig. 3: Swift/Coasters architecture.

Swift/Coasters operations is diagrammed in Figure 3. First, the Swift script 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 scheduled by well-studied, configurable algorithms and distributed to underlying service providers (external schedulers) including local execution, SSH [33], PBS [20], SGE [41], Globus [9], Condor [28], Cobalt [8], or the Coasters provider [19]. The Coasters provider consists of a connection to the CoasterService, which itself is deployed as a task ②. The CoasterService in turn uses task submission to deploy one or more allocations of "pilot jobs" [29], called Coaster 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 an existing scheduler such as the local operating system or a distributed scheduler such as PBS [20].

The default process manager in MPICH2 is currently Hydra [22]. The MPICH2 process manager is responsible for launching the user processes ("bootstrapping", in Hydra terminology) on the requested resources through a bootstrap control interface using an available mechanism such as `ssh`. The user launches MPICH by invoking `mpiexec`, which is a Hydra component. Subordinate to `mpiexec`, `ssh` is the *launcher* that invokes processes on remote resources. In Hydra, the launcher invokes the Hydra *proxy*, which is given sufficient environment and arguments to connect back to `mpiexec` and receive control commands. The proxy then launches the user executable; thus commencing user processing and MPI communication.

Hydra was modified for this work through the addition of a bootstrap mechanism called *launcher=manual*, 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. Our complete solution uses this mechanism and is described in Section 5.

4.3 ZeptoOS

JETS relies on the ability to dynamically bind MPI processes together using the POSIX sockets facility provided by many operating systems. While TCP/IP sockets are a typical mechanism for MPI job coupling on commodity clusters, these APIs are not provided by default on the Blue Gene/P. 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 optionally replaces the default IBM Compute Node Kernel (CNK) and enables the user processes to communicate over the BG/P torus interconnect by using an ethernet network device. This virtual network is then used by the MPI programs launched by JETS.

5 Design

JETS is based on the basic MTC paradigm of users rapidly submitting large batches of ordinary command-line program executions to large resources. JETS assumes that the user can launch a pilot job on the compute resources (starter scripts are provided with the distribution). The pilot job 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.

JETS orchestrates the systems described in the previous section into a simple framework for MPTC. It provides the following features:

1. **Speed:** JETS is designed to outperform process launchers such as `ssh` while enabling security (e.g., 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. In practice, the files to be stored in this way are simply provided to the JETS start-up script as a simple list.
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.

JETS may be used to submit single-process jobs (as in Falkon) or to submit MPI jobs. The essential idea in JETS is to transform an MPI job specification into a set of MPICH proxy job specifications by communicating with a background `mpiexec` process and to rapidly submit those proxy jobs to the pilot jobs for execution.

Performance benefits are obtained through the local, concurrent execution of the `mpiexec` processes and the use of the pilot jobs. 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 when resources are allocated by simply copying them to local storage. JETS features are available in two software systems: stand-alone form, which submits jobs given by the user in a simple list, and MPICH/Coasters form, which enables MPI executions to operate on the Coasters infrastructure with job specifications delivered by a Swift workflow.

5.1 Stand-Alone Form

In stand-alone form, the user must know all of the job specifications in advance, including the command line and node count. This is formulated in a JETS input file formatted as:

```
MPI: 4 namd2.sh input-1.pdb output-1.log
MPI: 8 namd2.sh input-2.pdb output-2.log
MPI: 6 namd2.sh input-3.pdb output-3.log
```

Such an input file executes a wrapper script around NAMD, operating on the given files with 4, 8, or 6 nodes allocated to each respective job. Note that the hostnames to be used by each job are not specified; these are dynamically determined by JETS at run time based on availability. Files are accessed at the

given paths directly by the user application. As an optimization, these paths could contain local storage paths that refer to files copied in by JETS.

Stand-alone JETS operation is diagrammed in Figure 4. 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.

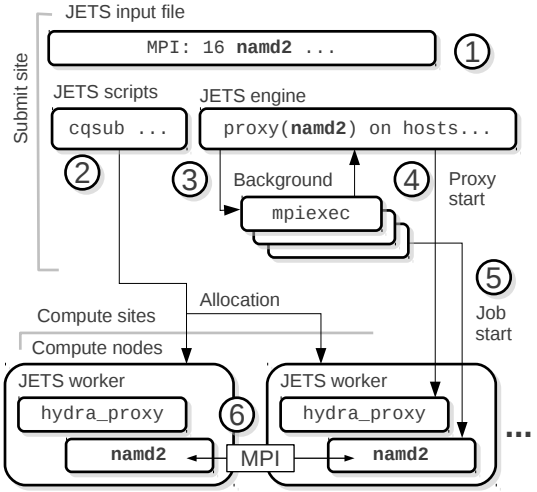


Fig. 4: JETS architecture.

5.2 MPICH/Coasters Form

Since the essential JETS functionality is to break MPI executions into composite single-process jobs, it was natural to make JETS functionality available in Coasters and thus to Swift workflow applications. Running MPI jobs in Swift through the JETS framework is performed by combining the advanced dataflow, task gen-

eration, data management, and worker management aspects of the Swift/Coasters architecture with the `mpiexec` process management of the JETS architecture.

MPICH/Coasters operation is diagrammed in Figure 5. The Swift script executes as normal, with the addition of configuration settings for MPI programs ①. These settings are packed with the job specification and sent to the CoasterService ②. The CoasterService processes the MPI settings, which affects the Coasters allocation strategy. For MPI jobs, the CoasterService waits for the appropriate number of available worker nodes before launching the `mpiexec` control mechanism, which is similar to that used in Section 5.1. When the job is ready to run, the `mpiexec` process is launched locally (not shown) concurrently with the Hydra proxies. Proxies launch the Swift wrapper script, which manages the user job in accordance with Swift features. The user executables are then launched, which are able to locate each other over sockets ③.

A common practice in workflow design is to write user wrapper scripts to manage files and set up program executions before calling the application binary program. This is commonly done in Swift applications as well. This practice is fully compatible with the MPICH/Coasters system, even though the process tree is deep (5 levels or more). A `PMI_RANK` (Process Management Interface Rank) variable is provided to all levels of user programs and may be used to coordinate such scripts. This value is equivalent to the MPI process rank in `MPI_COMM_WORLD`. For example, if a user desires to perform a simple shell command on the rank 0 process of a multiprocess job just before job start, this could be performed by the wrapper shell script that could branch to that shell command by referring to `PMI_RANK`.

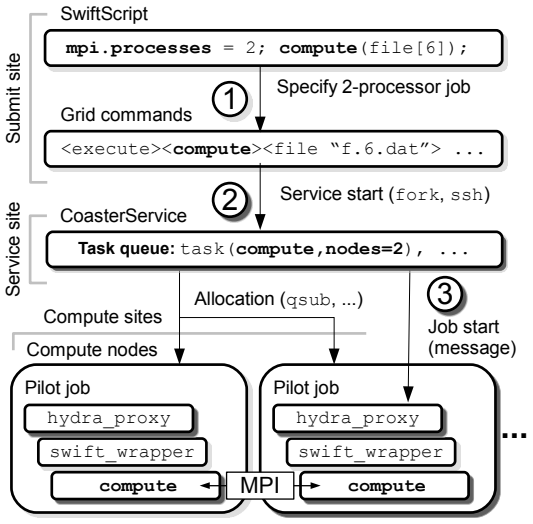


Fig. 5: MPICH/Coasters architecture.

6 Performance Results

In this section, we evaluate the many parallel-task computing model by measuring the performance of its implementations in multiple modes, for synthetic and realistic workloads. Specifically, we characterize performance parameters for use cases on three different computing systems and offer configuration details. In this section, we define a possibly parallel application invocation to be a single "job", and use "allocation" in the same sense as the Introduction.

6.1 Stand-alone JETS

Here, we present performance results obtained by running stand-alone JETS benchmarks in a cluster setting, in a high-performance setting, in a faulty environment, and in a NAMD-like application. Stand-alone benchmarks avoid the measurement complexity in a full Swift-based application workflow. Stand-alone JETS could be used in certain application patterns such as parameter sweep [49], and these results are relevant for that use case.

6.1.1 Sequential Tasks

Since JETS decomposes user MPI program invocations into a set of sequential user program invocations, we first measure the JETS performance for sequential tasks. This test demonstrates 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 single-point "ideal" measurement. Then, JETS was used to submit jobs to allocations of increasing size.

As shown in Figure 6, 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 capable of submitting the individual jobs generated by the more complex MPICH2-based mechanism described previously for MPI-based workloads.

6.1.2 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 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

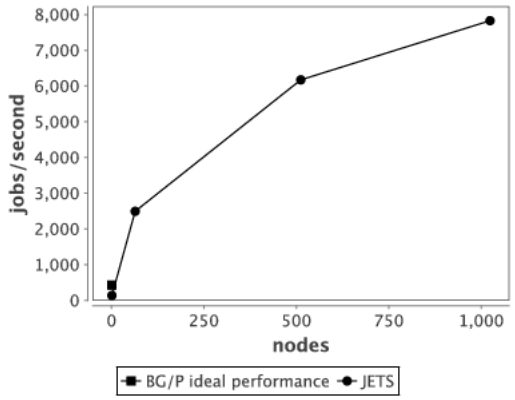


Fig. 6: JETS results for sequential tasks on the BG/P.

of the size given on the x-axis. Each job wait duration was 1 second. System utilization is reported as

$$\text{utilization} = \frac{\text{duration} \times \text{jobs} \times n}{\text{allocation size} \times \text{time}}, \tag{1}$$

where time is the total allocation time.

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 can use only the entire allocation, whereas the JETS mode may be run at smaller, varying sizes; the size of the MPI job is shown as either 4-proc or 8-proc, using 4 or 8 processes across 4 or 8 nodes, respectively. As shown in Figure 7, 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.

6.1.3 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 run time was measured with `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 ZeptoOS sockets layer.

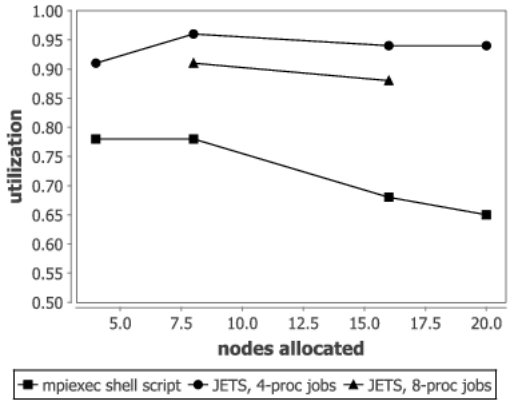


Fig. 7: MPI/JETS results, cluster setting.

As shown in Figure 8, 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 in Section 7, and the reliability characteristics are demonstrated in Section 6.1.5.

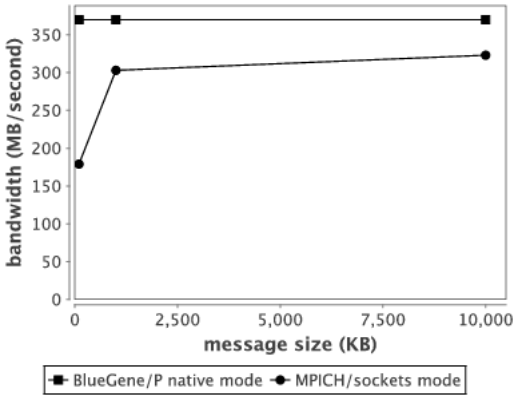


Fig. 8: MPI messaging performance on BG/P.

6.1.4 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

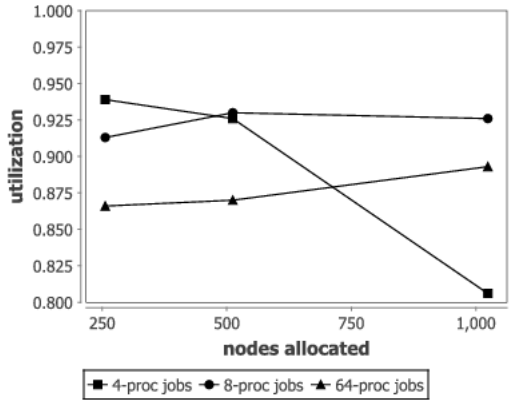


Fig. 9: MPI/JETS results, BG/P setting.

duration. Each node here contains 4 cores. We place only one MPI process per node in this test case. The n-proc number is as defined in the previous test.

JETS scripts were used to configure the system for compatibility with ZeptoOS and high performance at this system scale. We used the ZeptoOS node-local RAM-based filesystem to store the application binary, the Hydra proxy, and requisite libraries. The script sets `LD_LIBRARY_PATH` to locate the node-local system and user libraries and suppress any lookups to GPFS, which are much more time-consuming than are 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 and thus launch the MPI program.

We ran the same benchmark application used in Section 6.1.2 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 served order.

Results are shown in Figure 9. Each line shown represents one MPI 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. The 64-process tasks are individually slower to start, resulting in lower utilization in small allocations. However, this penalty becomes smaller as the task size becomes a smaller fraction of the available nodes.

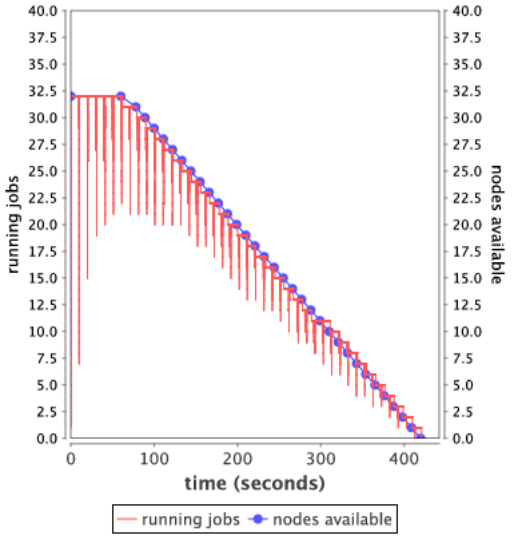


Fig. 10: MPI/JETS results, faulty setting.

6.1.5 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 because of system hardware or software failure. In this case, JETS was run again on Surveyor and the sequential application from Section 6.1.1 was used again. A fault injection script was run on the submit site that terminated randomly selected pilot jobs, one at a time, at regular 10 second intervals. Because of 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, allowing the total system load and worker count to be obtained and plotted over time.

Results are shown in Figure 10. The number of worker nodes in operation is shown as "nodes available"; the number steadily decreases from the original level of 32 workers 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 that 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 close to the number of nodes available, indicating that JETS maintains a high utilization rate on the available nodes.

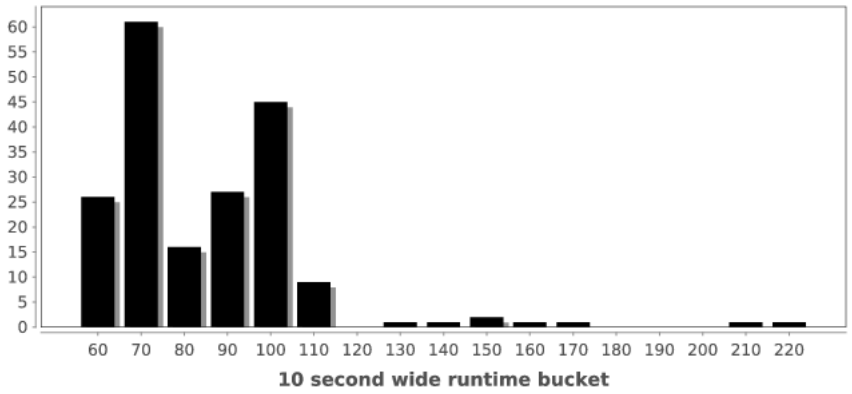


Fig. 11: NAMD wall time distribution.

6.1.6 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; the other cores were idle. 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 [34] 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 approach 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. A typical run time distribution for these jobs is shown in Figure 11. (This figure shows the full distribution for the batch that produced the 64-node result in Figure 18(b).) While the majority of the tasks fall between 100 and 120 seconds, many tasks exceed this, running up to 160 seconds. The utilization results (defined in Equation 1), shown in Figure 12, 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 13. For a longer run, utilization could be higher as the effect of the ramp-up and long-tail effects are amortized.

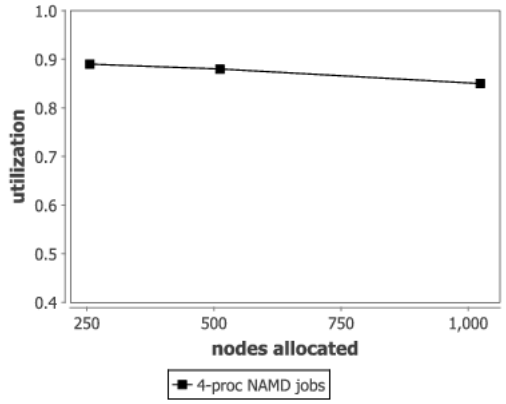


Fig. 12: NAMD/JETS utilization results.

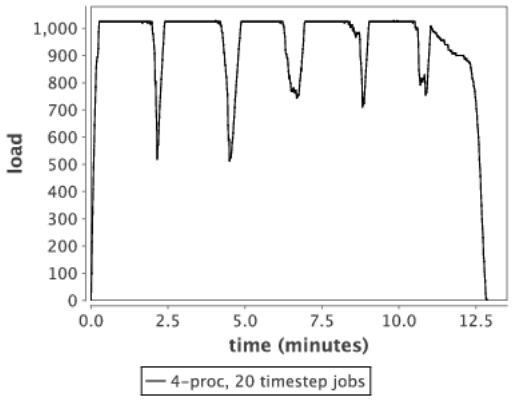


Fig. 13: NAMD/JETS load level results.

6.2 JETS/Swift Integration

Here, we report on the performance of the integrated JETS/Swift task distribution system. This allows the execution of complex Swift-based workflows. See Section 5.2 for a description of the software used in this section.

6.2.1 Synthetic Workloads

First, we report utilization results observed when running varying MPI configurations. The test suite constructed for this case allows us to measure utilization results for various allocation sizes, MPI job sizes, and MPI job core counts.

These tests were performed on Eureka, a 100-node x86-based cluster at Argonne National Laboratory. Each node contains two quad-core Intel Xeon E5405 processors running at 2 GHz for a total of 8 cores per node with 32 GB RAM. The system runs a GPFS [39] filesystem. Test allocations were started by Cobalt [8]

and maintained by a persistent Coasters service [19]. This allowed us to reuse the allocation for multiple Swift workflows, each of which issued hundreds of MPI jobs totaling thousands of individual processes.

We constructed an extremely simple MPI task that models useful work. The synthetic task used in this case runs a variable-sized MPI job. The task performs an `MPI_Barrier`, then each process sleeps for 10 seconds, then each process creates and/or writes its MPI rank to a single output file, then performs another `MPI_Barrier`, then exits. We wrote a trivial Swift script with a loop to generate tasks that execute this binary:

```
1  int total = 128;
2  int seconds = 10;
3  file input<"input.txt">;
4
5  foreach i in [0:total-1]
6  {
7    file output<single_file_mapper;
8         file=@strcat("data-",i,".txt")>;
9    output = mpi-sleep(input, seconds);
10 }
```

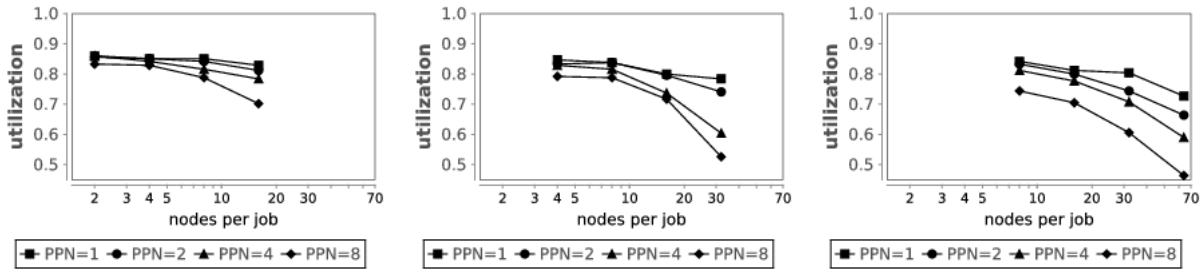
Fig. 14: Swift script for synthetic workload.

Three allocation sizes were chosen on Eureka: 16, 32, and 64 nodes. Each task was configured as an MPI job and thus processed by the MPICH/Coasters features as described previously. The number of processes per node is shown as PPN; this is the number of MPI processes allocated to a given node. Thus the total MPI size of a given task is the product of nodes per job and PPN. Results are shown in Figure 15. Note that in the following results, all plots use the same axis ranges for comparability, even if that resulted in interesting data appearing off-center.

The 10-second job duration was chosen to provide a useful spectrum of performance. For a given allocation size, at this duration, increasing task sizes decreases utilization. Increasing node counts or PPN reduce utilization. This is due to the increased relative delay in starting a job across a large fraction of an allocation. Additionally, increasing PPN exacerbates filesystem delays as the application program is read multiple times. In production, the primary approach that we would use to increase utilization for this challenging job duration is to move the application program to local storage and use data access optimizations available in Swift and Coasters.

6.2.2 REM Dataflow

Next, we describe the use of Swift to carry out a series of REM runs using NAMD. In this case, we perform a real replica exchange workflow in which segments and exchanges are data-dependent; each subsequent step in the workflow depends precisely on its inputs. Thus, a great deal of concurrency and *asynchronicity* is available. The individual NAMD segments are not just executed concurrently; they are



(a) Allocation size: 16 nodes (b) Allocation size: 32 nodes (c) Allocation size: 64 nodes

Fig. 15: Synthetic workload results.

also launched independently of the state of the workflow at large: they depend solely on their neighbor replicas for input data.

Describing this asynchronicity in a traditional scripting language such as Perl or Python would be quite difficult and would require user interaction with low-level operating system interfaces such as `waitpid()` as well as a nontrivial control loop, in addition to the complexity of quickly starting each available MPI job on free nodes. In our implementation, the dataflow diagrammed in Figure 16 is represented in under 200 lines of Swift script including comments and data definitions. Note that although the shapes are drawn to standard size to make the dependency structure readable, the times actually vary considerably (as illustrated in Figure 11). The core loop code is shown in Figure 17; some data definitions and data initializations are not shown.

The REM dataflow proceeds as follows. In Figure 16, each row from top to bottom represents a replica trajectory and is represented in the script as variable `i`. Each column from left to right represent progress made after exchange completion and is represented in the script as variable `j`. Each segment (`i`, `j`) is associated with a segment *index*; script variables `current`, `previous`, `neighbor`, and `total` are of this type. Script variables `c`, `v`, and `s`, represent conventional NAMD coordinates, velocities, and extended system files; `o` represents NAMD standard output (which contains application statistics) and `x` contains output from the exchange script (which is primarily used as a token for synchronization). These arrays are indexed by the segment index number and are mapped to real disk files through the Swift mapping abstraction (details not shown). Thus, segment k is associated with 5 dataflow files and represents 10 simulated time steps.

The statements in the Swift script are interpreted according to Swift semantics: they are all executed concurrently, limited by data dependencies. Each NAMD execution is handled by the `namd()` function which assigns the result of running NAMD on the files from the previous segment into the current segment. This creates a dataflow dependency from left to right in the diagram.

Similarly, the `exchange()` function executes the exchange required for REM. Notably, the exchange record in `x` is required for each `namd()` call in addition to NAMD application data. Certain conventional application data files are reused in each NAMD execution: `pdb_file`, `prm_file`, and `psf_file`. The `if` control statements ensure that the exchange is performed by alternating replicas. In Swift

scripts, the `%%` operator represents modulus, allowing for the determination of the parity of a number and for the exchange to wrap around in odd exchanges. The exchange function is implemented as a shell script that performs file operations to carry out the exchange. Swift was configured to perform the filesystem-bound exchange operations on the login node, freeing the compute nodes for the next ready NAMD segment.

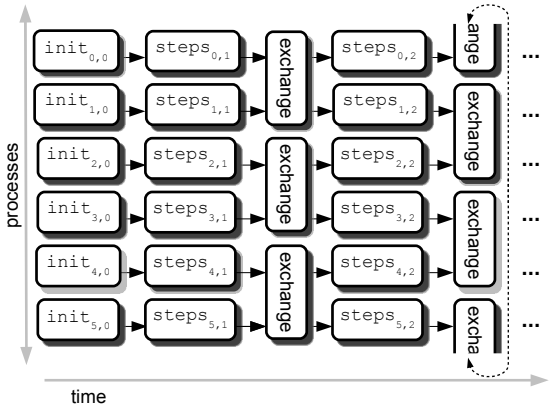


Fig. 16: Asynchronous REM dataflow as implemented in Swift.

The Swift script implementation is structured to allow maximal concurrency, mitigate the effect of the variation in NAMD execution time, and minimize the effects of the data-dependent synchronization. In both measurement series, the number of replicas in the ensemble is twice the hardware concurrency available, thus, when a NAMD invocation terminates, another is always ready to run. Additionally, since the performance of the single-process `exchange()` implementation is bound to filesystem latency, it is executed on the login node, preventing that process from delaying a ready multiprocess NAMD invocation.

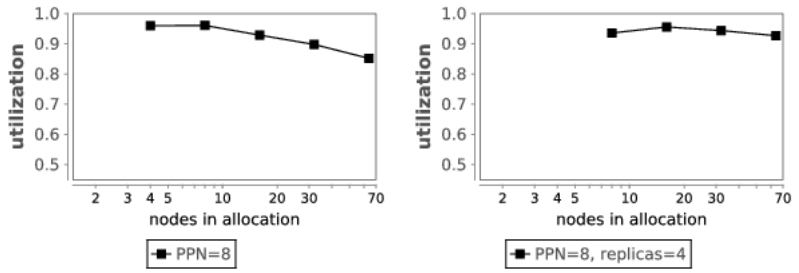
We executed this script on the same infrastructure as in the previous section (Eureka) under identical settings. In each case, utilization was measured by comparing the wall time reported by NAMD to the wall time in the allocation used by Swift, as calculated in Equation 1. Thus, any long tail effect [2] is charged against the utilization.

We first executed the script with NAMD configured to run in a single-process mode for each segment with up to 64 nodes, each segment running as a single process on a single node. In each script execution, the number of replicas in the dataflow was twice the number of nodes, and 4 exchanges were performed. The result of this series of measurements is shown in Figure 18a.

We then executed the script with NAMD configured to run as an MPI program for each segment with up to 64 nodes. For each allocation size from 8 to 64, the number of concurrently executing replicas was 4, and the total number of replicas in the dataflow was 8. All 8 cores on each core were used. Thus, for the 8-node allocation, each replica segment ran on 2 nodes with 16 MPI processes, and so

```
1  global coordinates c[]<array_mapper, ...>;
2  global velocities  v[]<array_mapper, ...>;
3  global xsc         s[]<array_mapper, ...>;
4  global output      o[]<array_mapper, ...>;
5  global exchange    x[]<array_mapper, ...>;
6
7  foreach j in [0:exchanges-1]
8  {
9    foreach i in [0:replicas-1]
10   {
11     // Index of previous segment
12     int previous = i*(exchanges+1) + j;
13     // Index of segment we are currently defining
14     int current = previous+1;
15
16     // Run NAMD
17     (c[current], v[current],
18      s[current], o[current]) =
19       namd(namd_template, fftw_file,
20              pdb_file, prm_file, psf_file,
21              c[previous], v[previous],
22              s[previous], x[previous],
23              i, current);
24
25     // Perform exchange
26     int neighbor;
27     if ((j %% 2) == 0)
28     {
29       // Perform exchanges from odd replicas
30       if ((i %% 2) == 1)
31       {
32         neighbor =
33           (current + (exchanges+1)) %% total;
34         (x[current], x[neighbor]) =
35           exchange(c[current], c[neighbor],
36                    v[current], v[neighbor], j);
37       }
38     }
39     else
40     {
41       // Perform exchanges from even replicas
42       if ((i %% 2) == 0)
43       {
44         neighbor = current + (exchanges+1);
45         (x[current], x[neighbor]) =
46           exchange(c[current], c[neighbor],
47                    v[current], v[neighbor], j);
48       }
49     }
50   }
51 }
```

Fig. 17: Swift script for REM core loop



(a) Single-process NAMD executions (b) NAMD executions as MPI processes

Fig. 18: REM/Swift results.

forth. Over the dataflow, 6 exchanges were performed. The result of this series of measurements is shown in Figure 18b.

Neither case used specialized data management functionality provided by Swift [53] or JETS as used in Section 6.1.4. Available optimizations that could be used in a production run include copying programs to local storage on compute nodes, changing operating system settings to prevent unnecessary filesystem accesses to GPFS, or prestaging reused data files (e.g. `pdb_file`) to local storage. Thus, all programs and data were read and written to GPFS, and our results show what a first-time user would experience in a straightforward use case.

For the single-process case, as the allocation size was increased from 4 to 64, utilization decreased down to 85.4%. In the MPI case, utilization did not change substantially over the measured range of allocation sizes, remaining between 92.7% and 95.6%.

Decreases in utilization comes from three sources: Swift/Coasters processing time, filesystem delays, and executable launch time. Swift/Coasters processing time is consumed by the Swift data dependency engine producing the task description and the Coasters system transmitting that task to a worker; however, the task rates in this use case do not approach known Swift/Coasters capabilities [19]. Filesystem accesses to GPFS are the likely cause of lost utilization for the single-process case, as the large number of independent replicas produce simultaneous small-file accesses. Most important for this work, the utilization for MPI use cases exceeds that of single-process use cases, showing that the use of the new JETS-based job launch features does not constrain utilization.

7 Future Work

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. 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.

We will experiment with MPI-IO from JETS-initiated MPTC workloads, and work on optimizations for supporting the passing of MPI-IO-written and -read datasets within an MPTC dataflow. Similarly, such high-performance data-passing schemes can also be evaluated using Global Arrays [32] or distributed hash tables [52].

JETS does not currently have a mechanism by which nodes may be grouped with respect to network location. This feature could be important if given workflow is running on multiple clusters simultaneously, and joining MPI processes on the same cluster should be preferred to running MPI jobs across clusters; in fact, some users would probably like to prohibit the latter.

8 Conclusion

As we're really not together at all, but parallel. [27]

The parallel ensemble application, 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 support MPTC. Our work is focused on gaining high utilization rates for applications on large-scale HPC resources. Our work includes 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 single-process jobs at a rate exceeding that of previous many-task schedulers, and showed that moving to multiple-process MPI jobs does not restrict performance. Additionally, we provided new mechanisms for the deployment of MPI applications into many-task systems; in particular, the new MPICH functionality could be reused by other groups developing other 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 fault-

tolerant way. Moreover, the software development would benefit from the high-level Swift model.

Acknowledgments

We 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 by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility.

References

1. David Abramson, Jon Giddy, and Lew Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid. In *Proc. International Parallel and Distributed Processing Symposium*, 2000.
2. Timothy G. Armstrong, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Ian T. Foster. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *Proc. MTAGS Workshop at SC'10*, 2010.
3. Francis Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.
4. Steven Boker, Michael Neale, Hermine Maes, Michael Wilde, Michael Spiegel, Timothy Brick, Jeffrey Spies, Ryne Estabrook, Sarah Kenny, Timothy Bates, Paras Mehta, and John Fox. OpenMx: An open source extended structural equation modeling framework. *Psychometrika*, 2011.
5. Tom Budnik, Brant Knudson, Mark Megerian, Sam Miller, Mike Mundy, and Will Stockdell. Blue Gene/Q resource management architecture. In *Proc. Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.
6. Promita Chakraborty, Shantenu Jha, and Daniel S. Katz. Novel submission modes for tightly coupled jobs across distributed resources for reduced time-to-solution. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1897):2545–2556, 2009.
7. Po-Hsiang Chiu and Maxim Potekhin. Pilot factory a Condor-based system for scalable pilot job generation in the Panda WMS framework. *Journal of Physics: Conference Series*, 219, 2011.
8. Cobalt web site. http://trac.mcs.anl.gov/projects/cobalt.
9. Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459, 1998.
10. Joe DeBartolo, Glen Hocky, Michael Wilde, Jinbo Xu, Karl F. Freed, and Tobin R. Sosnick. Protein structure prediction enhanced with evolutionary diversity: Speed. *Protein Science*, 19(3):520–534, 2010.
11. James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. *Intl. Conf. on Parallel Processing*, 0:586–593, 2008.
12. Andriy Fedorov, Benjamin Clifford, Simon K. Warfield, Ron Kikinis, and Nikos Chrisochoides. Non-rigid registration for image-guided neurosurgery on the TeraGrid: A case study. Technical Report WM-CS-2009-05, College of William and Mary, 2009.
13. Samantha S. Foley, Wael R. Elwasif, Aniruddha G. Shet, David E. Bernholdt, and Randall Bramley. Incorporating concurrent component execution in loosely coupled integrated fusion plasma simulation. In *Component-Based High-Performance Computing 2008*, 2008.

14. Ian Foster. What is the Grid? A three point checklist. *GRIDToday*, (6), 2002.
15. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1st edition, 1999.
16. James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.
17. Uri Hasson, Jeremy I. Skipper, Michael J. Wilde, Howard C. Nusbaum, and Steven L. Small. Improving the analysis, storage and sharing of neuroimaging data using relational databases and distributed computing. *NeuroImage*, 39(2):693–706, 2008.
18. Mihael Hategan, Justin M. Wozniak, and Ketan Maheshwari. Coasters: Uniform resource provisioning and access for scientific computing on clouds and grids. In *Proc. Utility and Cloud Computing*, 2011.
19. Mihael Hategan, Justin M. Wozniak, and Ketan Maheshwari. Coasters: Uniform resource provisioning and access for scientific computing on clouds and grids. In *Proc. Utility and Cloud Computing*, 2011.
20. Robert L. Henderson and David Tweten. Portable batch system: Requirement specification. Technical report, NAS Systems Division, NASA Ames Research Center, 1998.
21. Glen Hocky, Michael Wilde, Joe DeBartolo, Mihael Hategan, Ian Foster, Tobin R. Sosnick, and Karl F. Freed. Towards petascale ab initio protein folding through parallel scripting. Technical Report ANL/MCS-P1612-0409, Argonne National Laboratory, April 2009.
22. Using the Hydra process manager. http://wiki.mcs.anl.gov/mpich2/index.php/-Using_the_Hydra_Process_Manager.
23. Cray Inc. *Workload Management and Application Placement for the Cray Linux Environment, Document number S–2496–3103*. Cray Inc., Chippewa Falls, WI, USA, 2011.
24. Sarah Kenny, Michael Andric, Steven M. Boker, Michael C. Neale, Michael Wilde, and Steven L. Small. Parallel workflows for data-driven structural equation modeling in functional neuroimaging. *Frontiers in Neuroinformatics*, 3(34), 2009.
25. Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.
26. Sangkyu Lee, Yue Chen, Hao Luo, Andrew A. Wu, Michael Wilde, Paul T. Schumacker, and Yingming Zhao. The first global screening of protein substrates bearing protein-bound 3,4-dihydroxyphenylalanine in Escherichia coli and human mitochondria. *Journal of Proteome Research*, 9(11):5705–5714, 2010.
27. Ted Leo and the Pharmacists. Parallel or Together?, 2001.
28. Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A hunter of idle workstations. In *Proc. International Conference of Distributed Computing Systems*, 1988.
29. André Luckow, Lukasz Lacinski, and Shantenu Jha. SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Proc. CCGrid*, 2010.
30. Ewing L. Lusk, Steve C. Pieper, and Ralph M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17, 2010.
31. MPICH web site. http://www.mcs.anl.gov/research/projects/mpich2.
32. Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10(2):1–17, 1996.
33. OpenSSH web site. http://www.openssh.com.
34. NMA structure in the Protein Data Bank. http://www.rcsb.org/pdb/ligand/ligandsummary.do?hetId=NMA.
35. James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.
36. Ioan Raicu, Ian Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Proc. Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.
37. Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, and Ben Clifford. Towards loosely-coupled programming on petascale systems. In *Proc. SC'08*, 2008.
38. Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. Workshop on Data-aware distributed computing*, 2008.

39. Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. USENIX Conference on File and Storage Technologies*, 2002.
40. Igor Sfiligoi. glideinWMS a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119(6), 2008.
41. Sun Grid Engine web site. http://gridengine.sunsource.net.
42. Tiberiu Stef-Praun, Benjamin Clifford, Ian Foster, Uri Hasson, Mihael Hategan, Steven L. Small, Michael Wilde, and Yong Zhao. Accelerating medical research using the Swift workflow system. *Studies in Health Technology and Informatics*, 126:207–216, 2007.
43. Tiberiu Stef-Praun, Gabriel A. Madeira, Ian Foster, and Robert Townsend. Accelerating solution of a moral hazard problem with Swift. In *e-Social Science 2007*, Indianapolis, 2007.
44. Yuji Sugita and Yuko Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, 314(1-2):141 – 151, 1999.
45. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):325–356, 2005.
46. Abhinav Thota, André Luckow, and Shantenu Jha. Efficient large-scale replica-exchange simulations on production infrastructure. *Philosophical Transactions of the Royal Society of London A*, 369(1949):3318–35, 2011.
47. Top 500 web site. http://www.top500.org.
48. Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9), 2001.
49. Adianto Wibisono, Zhiming Zhao, Adam Belloum, and Marian Bubak. A framework for interactive parameter sweep applications. In Marian Bubak, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science ICCS 2008*, volume 5103 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008.
50. Michael Wilde, Ian Foster, Kamil Iskra, Peter Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11), 2009.
51. Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, September 2011.
52. Justin M. Wozniak, Bryan Jacobs, Robert Latham, Sam Lang, Seung Woo Son, and Robert 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.
53. Justin M. Wozniak and Michael Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proc. Petascale Data Storage Workshop at SC'09*, 2009.
54. Zhao Zhang, Allan Espinosa, Kamil Iskra, Ioan Raicu, Ian Foster, and Michael Wilde. Design and evaluation of a collective I/O model for loosely-coupled petascale programming. In *Proc. MTAGS Workshop at SC'08*, 2008.
55. Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Ioan Raicu, Tiberiu Stef-Praun, and Mike Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. Workshop on Scientific Workflows*, 2007.