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

Justin M Wozniak and Michael Wilde
Argonne National Laboratory

Presented at:

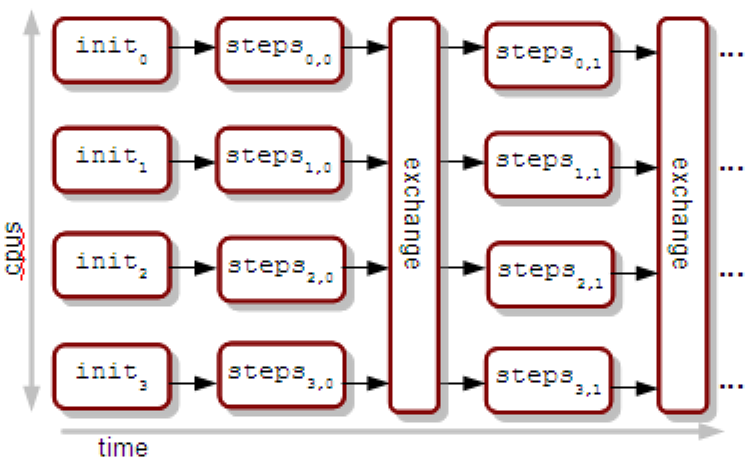
P2S2
Taipei – September 13, 2011

Outline

- Scientific applications
  - Batches, ensembles, parameter studies,
  - Scientific scripting tools to construct studies
  - Use case: Replica Exchange Method (REM) in NAMD
- Performance challenges
- Many Parallel-Task Computing - JETS
- Integration with Swift
- Ongoing work: ExM- Many-task computing at the exascale
- Summary

NAMD - Replica Exchange Method

- Original JETS use case
- Sizeable batch of short parallel jobs with data exchange



Application parameters (approx.):

- 64 concurrent jobs
  x 256 cores per job =
  16,384 cores
- 10-100 time steps per job =
  10-60 seconds wall time
- Requires 6.4 MPI executions/sec. ⟶
  1,638 processes/sec. over
  a 12-hour period =
  70 million process starts

Parameter studies

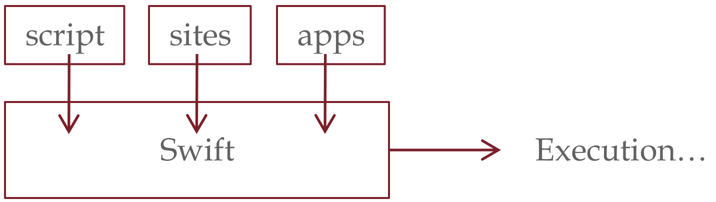
- Treat each application invocation as a function evaluation in a higher-level method
- Run the same application with varying input parameters
  - Parameter sweep: cover a known range of inputs to obtain outputs and produce statistical information or visualization
  - Parameter search/optimization: find inputs that produce interesting/extreme outputs
  - Application script: evaluate arbitrary user script
- REM is a form of parameter sweep with some relatively simple data exchange- easily expressed in a scripting language

Scientific scripting - SwiftScript

- Support file/task model directly in the language

```
app (file output) sim(file input) {
    namd2 @input @output
}
```

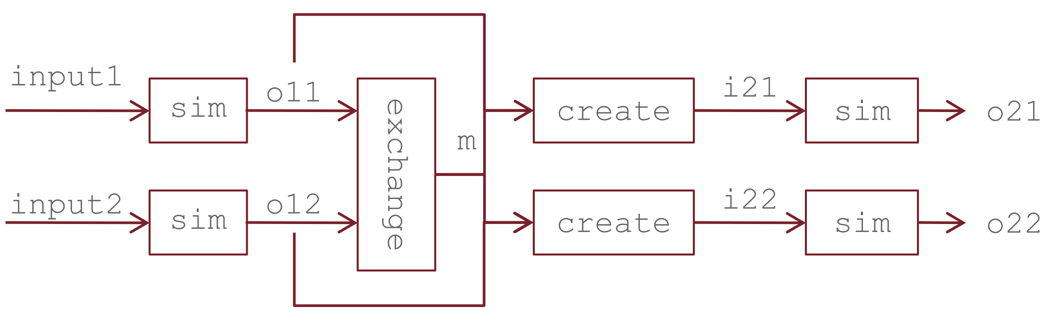
- Separate application script from site configuration details



- Provide natural concurrency through automatic data flow analysis and task scheduling

```
file o11 = sim(input1);
file o12 = sim(input2);
file m   = exchange(o11, o12);
file i21 = create(o11, m);
file o21 = sim(i21);
...
```

- Support scientific data sets in the language through language constructs such as structs, arrays, mappers, etc.



Task management

Task generation → Task distribution → Task execution

- Tasks may be generated by a simple list or by a running program or workflow
- Workflow execution produces "job specifications"- user tasks to be executed on the available infrastructure
- We are currently investigating the following infrastructures:
  - Coasters
  - Falkon
  - JETS
- Tradeoffs include performance, portability, and usability

Performance challenges for large batches

- For small application run times, the cost of application start-up, small I/O, library searches, etc. is expensive

- Existing HPC schedulers do not support this mode of operation
  - On the Blue Gene/P, job start takes 2-4 minutes
  - On the Cray, `aprun` job start takes a full second or so
  - Neither of these systems allow the user to make a fine-grained selection of cores from the allocation for small multicore/multinode jobs

- Solution pursued by JETS:
  - Allocate worker agents en masse
  - Use a specialized user scheduler to rapidly submit user work to agents
  - Support dynamic construction of multinode MPI applications

JETS: Features

- Portable worker agents that run on compute nodes
  - Provides scripts to launch agents on common systems
  - Features provide convenient access to local storage such as BG/P ZeptoOS RAM filesystem. *Storing application binary, libraries, etc. here results in significant application start time improvements*

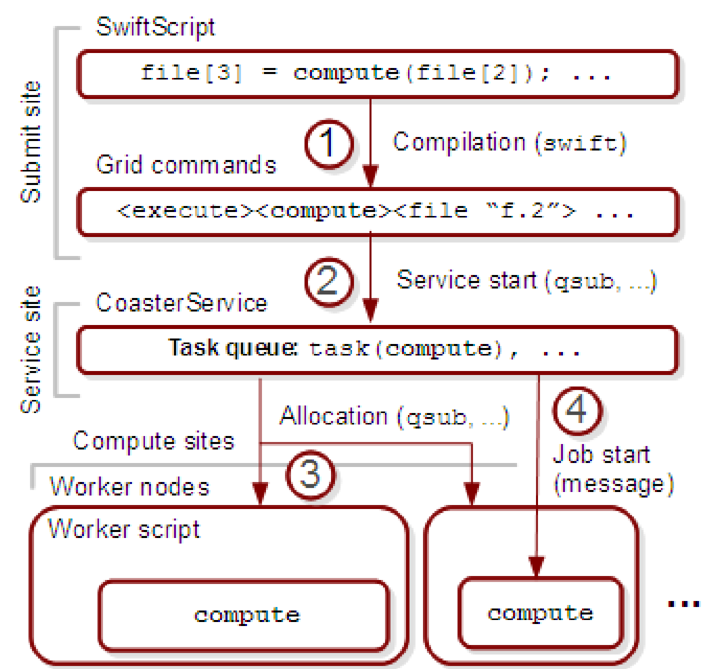
- Central user scheduler to manage workers: (Stand-alone JETS or Coasters discussed on following slides)

- MPICH /Hydra modification to allow "launcher=manual": tasks launched by the user (instead of SSH or other method)

- User scheduler plug-in to manage a local call to `mpiexec`
  - Processes output from `mpiexec` over local IPC, launches resultant single tasks on workers
  - Single tasks are able to find the `mpiexec` process and each other to start the user job (via Hydra proxy functionality)
  - Can efficiently manage many running `mpiexec` processes

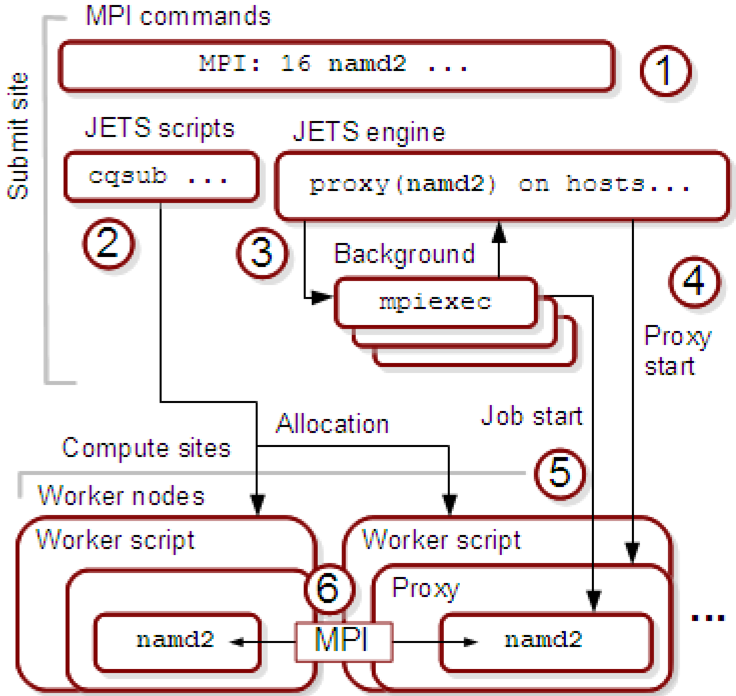
Execution infrastructure - Coasters

- Coasters: a high task rate execution provider (Previously developed for the Swift system)
  - Automatically deploys worker agents to resources with respect to user task queues and available resources
  - Implements the Java CoG provider interfaces for compatibility with Swift and other software
  - Currently runs on clusters, grids, and HPC systems
  - Can move data along with task submission
  - Contains a “block” abstraction to manage allocations containing large numbers of CPUs
  - **Originally only supported sequential tasks**



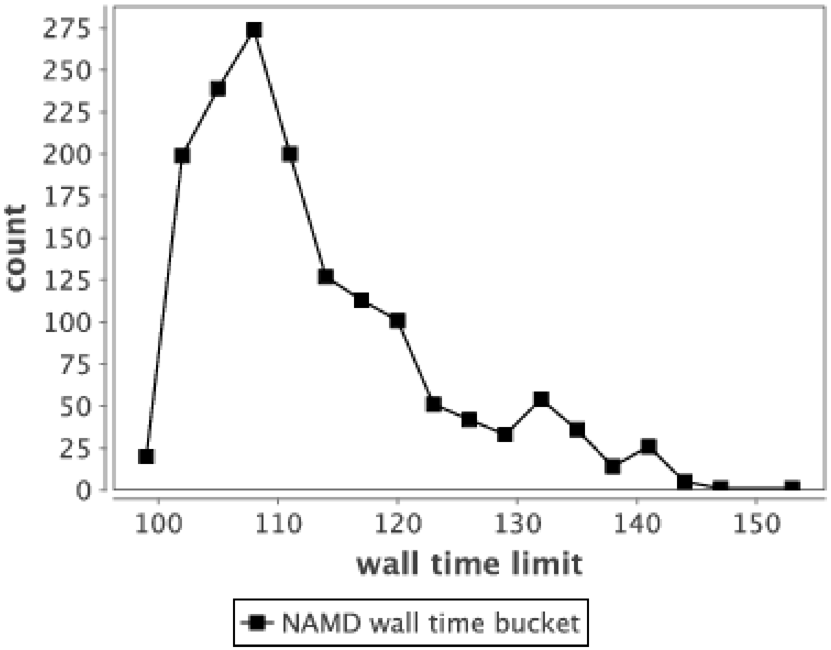
Execution infrastructure - JETS

- Stand-alone JETS: a high task rate parallel-task launcher
  - User deploys worker agents via customizable, provided submit scripts
  - Currently runs on clusters, grids, and HPC systems
    - Great over SSH
    - Runs on the BG/P through ZeptoOS sockets- great for debugging, performance studies, ensembles
  - Faster than Coasters but provides fewer features
    - Input must be a flat list of command lines
    - Limited data access features



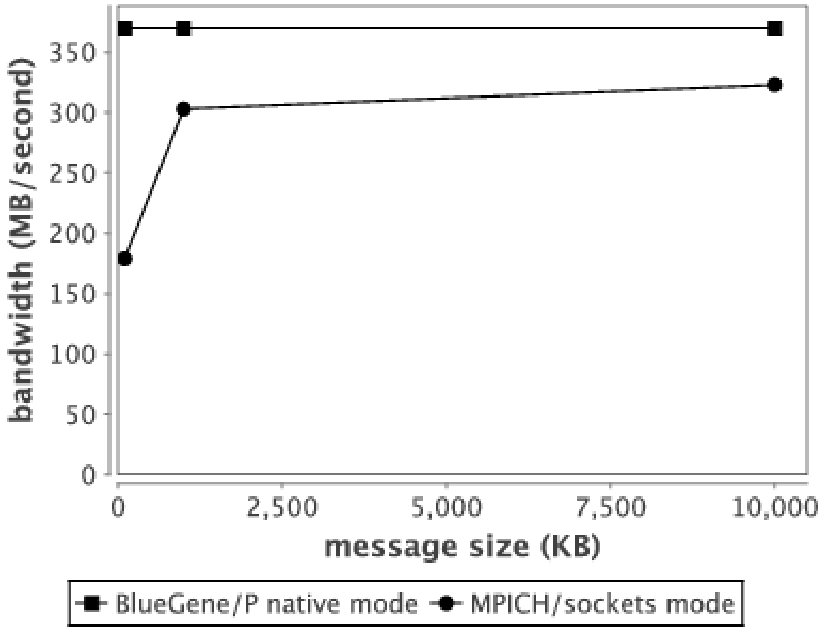
NAMD/JETS - Parameters

- NAMD REM-like case:
  Tasks average just over 100 seconds



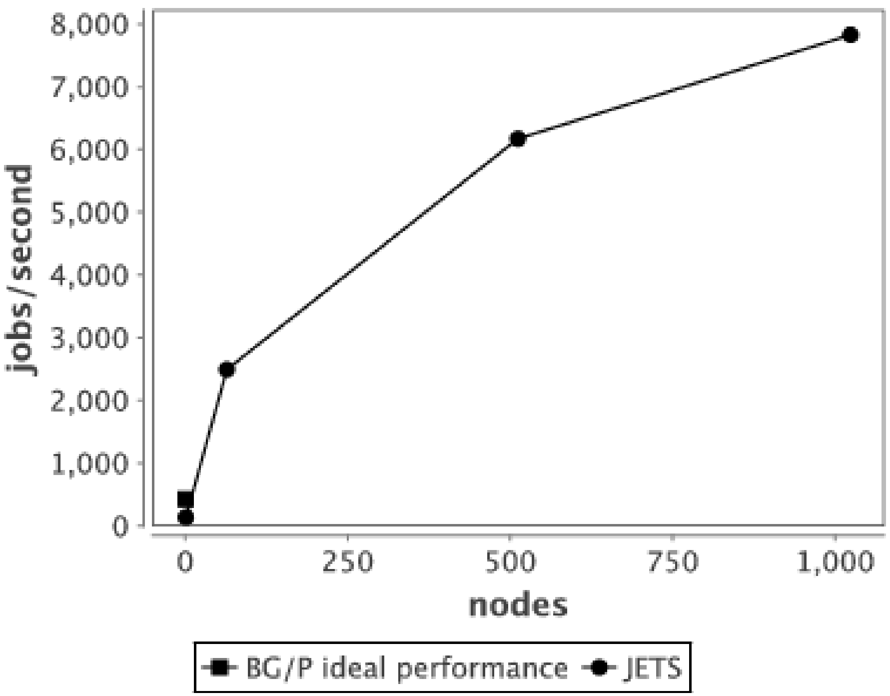
- Case provided by Wei Jiang

- ZeptoOS sockets on the BG/P
  90% efficiency for large messages
  50% efficiency for small messages

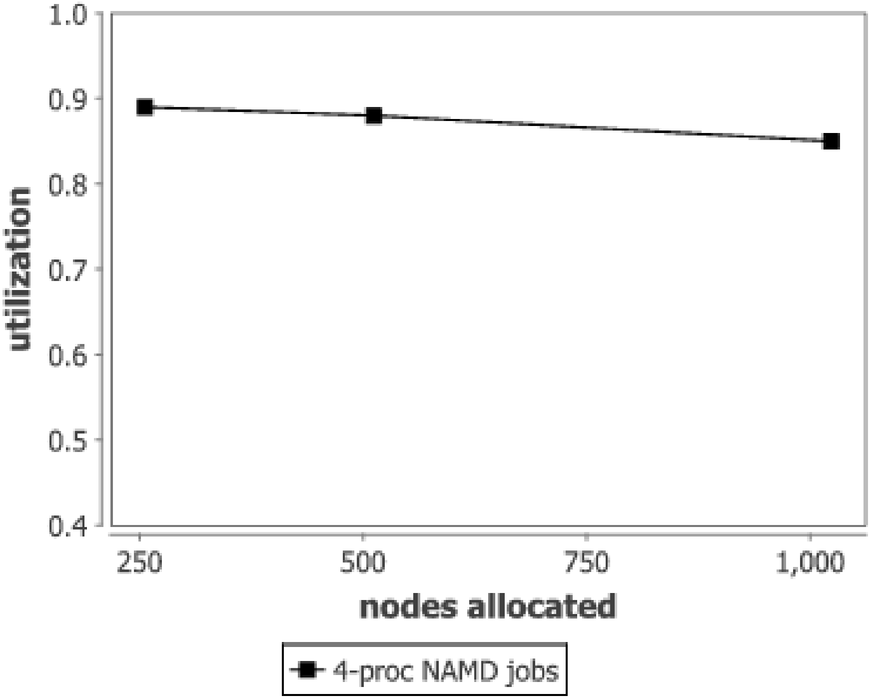


JETS - Task rates and utilization

- Calibration: Sequential performance on synthetic jobs:

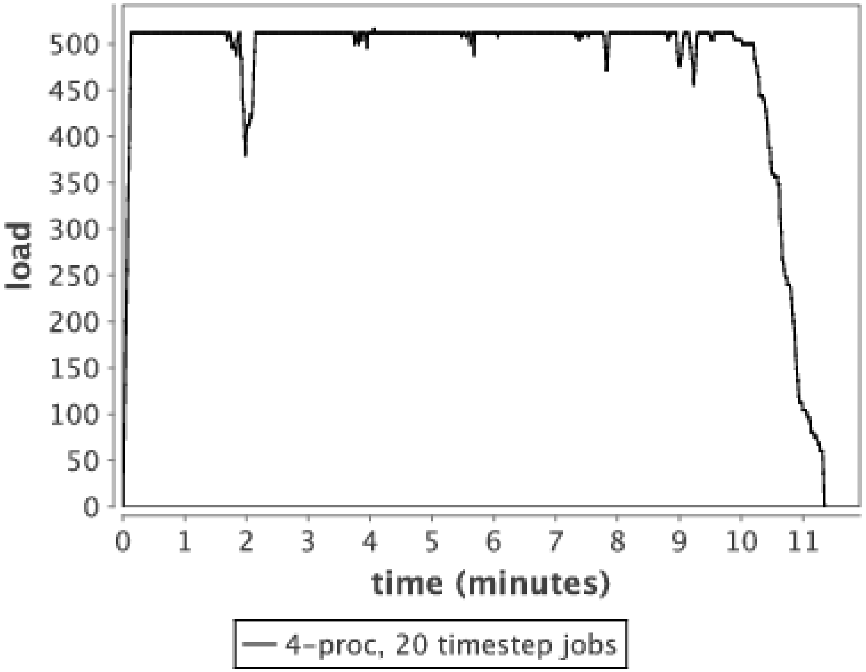


- Utilization for REM-like case: not quite 90%

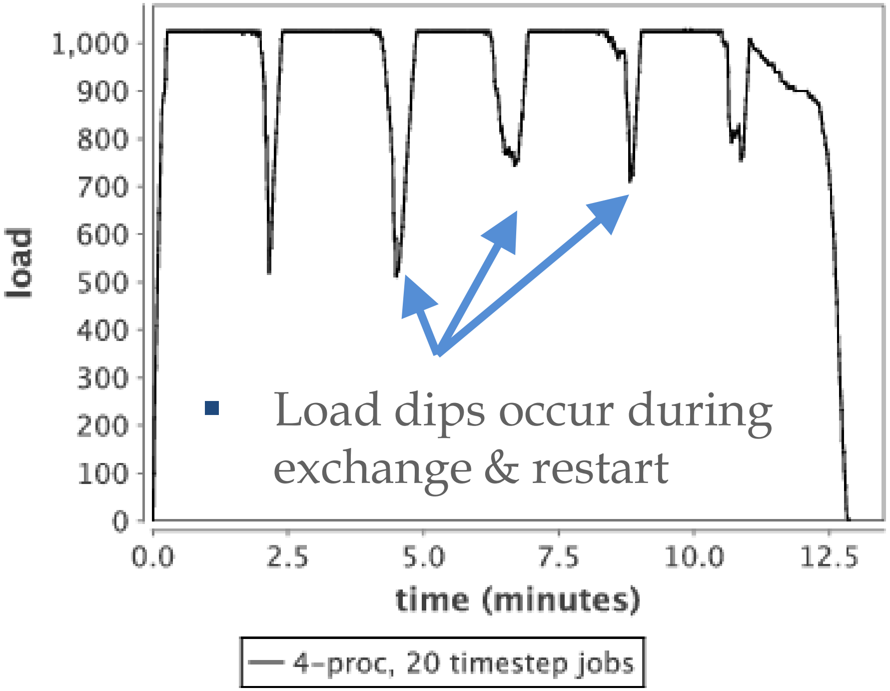


NAMD/JETS load levels

- Allocation size: 512 nodes



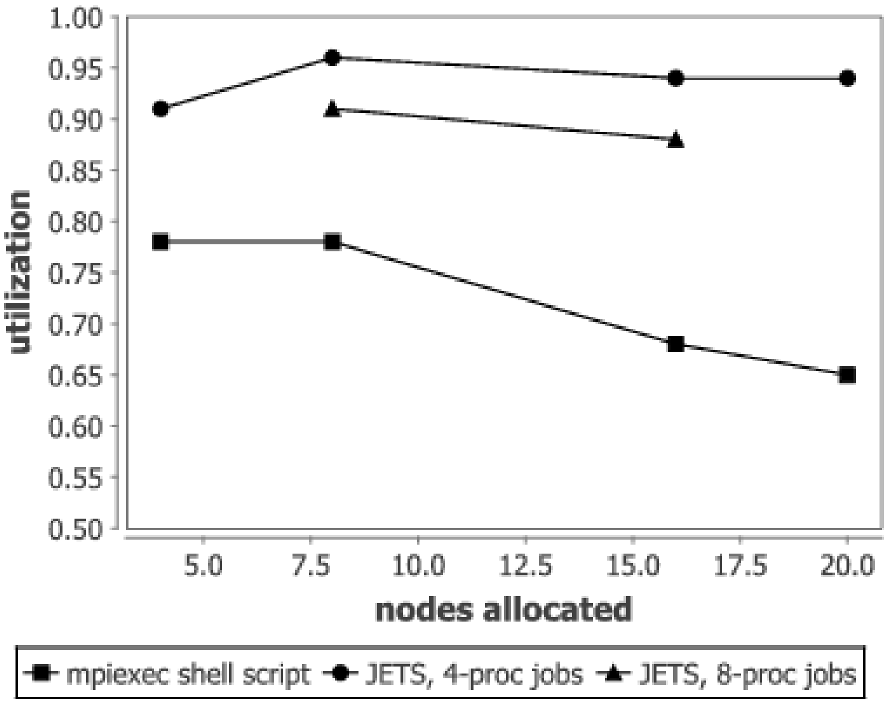
- Allocation size: 1024 nodes



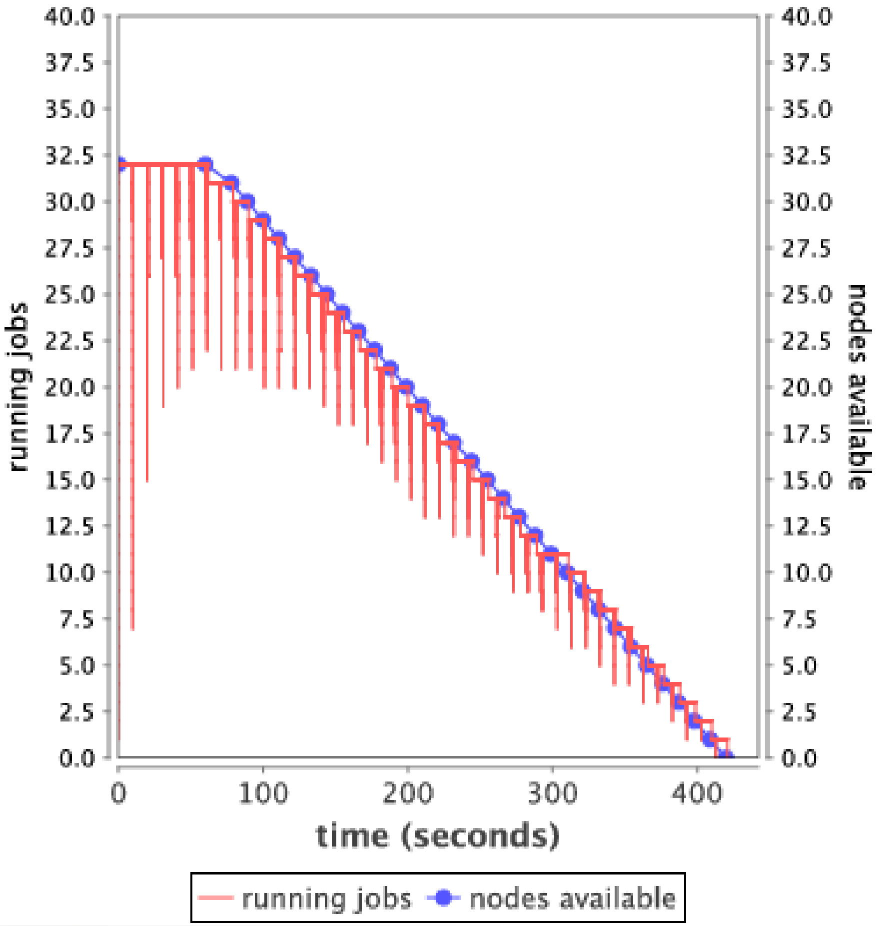
- Load dips occur during exchange & restart

JETS - Misc. results

- Effective for short MPI jobs on clusters
- Single-second duration jobs on Breadboard cluster



- JETS can survive the loss of worker agents (BG/P)



Future work:
ExM: Extreme-scale many-task computing

- Project goal- investigate many-task computing on exascale systems

  Possible benefits:
  - Ease of development – fast route to exaflop application
  - Investigate alternative programming models
  - Highly usable programming model: natural concurrency, fault-tolerance
  - Support scientific use cases: batches, scripts, experiment suites, etc.

- Build on and integrate previous successes

  - ADLB: Task distributor
  - MosaStore: Filesystem cache
  - SwiftScript language: Natural concurrency, data specification, etc.

Task generation and scalability

- In SwiftScript, all data items are *futures*
- Progress is enabled when data items are closed, enabling dependent statements to execute
- Not all variables, statements are known at program start
- SwiftScript allows for complex data definitions, conditionals, loops, etc.
- Current Swift implementation constrains the data dependency logic to a single node (as do other systems like CIEL) - thus task generation rates are limited
- ExM proposes a fully distributed, scalable task generator and dependency graph – built to express Swift semantics and more

Performance target

Performance requirements for distributing the work of Swift-like task generation for an ADLB-like task distributor on an example exascale system:

- Need to utilize $O(10^9)$ concurrency
- For batch of 1000 tasks per core
  - 10 seconds per task
  - 1 hour, 46 minute batch
- Tasks : $O(10^{12})$
- Tasks/s: $O(10^8)$
- Divide cores into *workers* and *control* cores
  - Allocate 0.01% as control cores, $O(10^5)$
  - Each control core must produce $O(10^3)$ = 1000 tasks/second

Recap and further reading…

Task generation → Task distribution → Task execution

Swift, ExM, Turbine | Coasters, JETS | Collective Data Management

- **Case studies in storage access by loosely coupled petascale applications**
  Petascale Data Storage Workshop at SC'09

- **Turbine: A distributed future store for extreme-scale scripted applications**
  Submitted to PPoPP: A preprint is available

Thanks

- Thanks to the organizers
- Swift team: Ketan Maheshwari, Mihael Hategan, Mike Wilde
- ExM team: Ian Foster, Dan Katz, Rusty Lusk, Matei Ripeanu, Emalayan Vairavanathan, Zhao Zhang
- Thanks to Wei Jiang (ANL) for providing the NAMD use case
- Grants:

This research is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contracts DE-AC02-06CH11357. Work is also supported by DOE with agreement number DE-FC02-06ER25777.

Questions