EPJ Web of Conferences **175**, 09007 (2018) https://doi.org/10.1051/epjconf/201817509007
Lattice 2017

Job Management and Task Bundling

Evan Berkowitz[1,⋆], *Gustav R.* Jansen[2,⋆⋆], *Kenneth* McElvain[3], and *André* Walker-Loud[3]

[1] *Institut für Kernphysik and Institute for Advanced Simulation, Forschungszentrum Jülich*
[2] *National Center for Computational Sciences and Physics Division, Oak Ridge National Laboratory*
[3] *Nuclear Science Division, Lawrence Berkeley National Laboratory*

Abstract. High Performance Computing is often performed on scarce and shared computing resources. To ensure computers are used to their full capacity, administrators often incentivize large workloads that are not possible on smaller systems. Measurements in Lattice QCD frequently do not scale to machine-size workloads. By bundling tasks together we can create large jobs suitable for gigantic partitions. We discuss `METAQ` and `mpi_jm`, software developed to dynamically group computational tasks together, that can intelligently backfill to consume idle time without substantial changes to users' current workflows or executables.

1 Introduction

Many large scientific calculations require enormous computational resources that are not available except at leadership-class computing facilities. To ensure that the computers at these facilities are used to their full potential, administrators often incentivize users to aim for large jobs that cannot be executed except on these large machines. For example, NERSC discounts the spent computer time by 40% once jobs are larger than 683 nodes[1] and the queue on Titan, at Oak Ridge National Laboratory, artificially ages jobs larger than 3750 nodes so that they make it through the queue more quickly[2].

Lattice QCD (LQCD) calculations are not often suited to such large jobs. For example, the calculation of the nucleon axial coupling g_A presented by Chang at LATTICE 2017[3, 4] required thousands of propagators, thousands of sequential (Feynman-Hellman) propagators[5], and thousands of contractions. In that work the authors typically solved for the propagator using 32 nodes, each with a GPU, while the contractions were performed on 8 nodes using only CPUs. To run any of these tasks on hundreds or even thousands of nodes would be a waste of resources as the data would be spread too thin. Consequently, the CPUs and GPUs would largely sit idle while communication would dominate the calculation time.

⋆ e-mail: e.berkowitz@fz-juelich.de.
Corresponding slides are available at https://makondo.ugr.es/event/0/session/102/contribution/335.
⋆⋆ This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. (http://energy.gov/downloads/doe-public-access-plan).

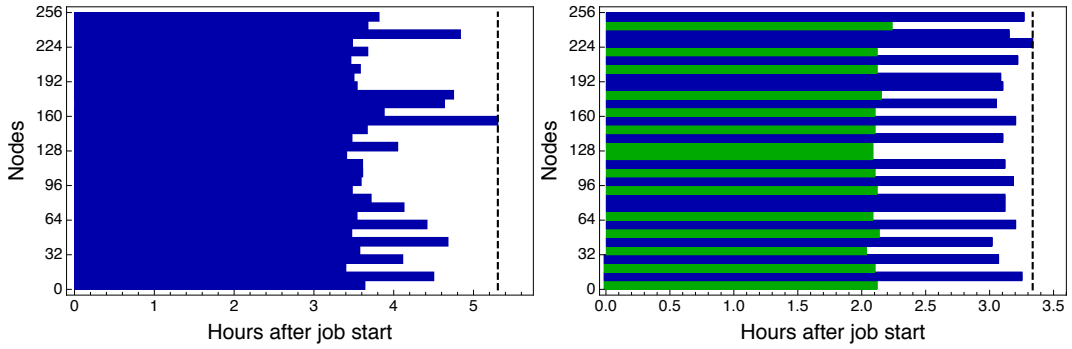

Figure 1. Two timelines of multiple tasks naïvely bundled into a single 256-node job run on Titan. The horizontal colored bars represent different computational tasks, and each color represents different types of tasks. White space is wasted, idle time. In the left panel we show how variation in run times for a single kind of task can vary, resulting in waste. In the right panel we show how a heterogeneous mix of tasks can result in waste, even when the run time for each kind of task is relatively consistent.

One way LQCD measurements can take advantage of the substantial incentives for large jobs is to group, or bundle, different computational tasks together. As LQCD measurements are often embarrassingly parallel, in that the measurements performed on different gauge configurations are independent of each other, these tasks can be bundled together quite simply.

One simple solution is naïve bundling—finding similar computational tasks and starting them all at once with, for example, `wraprun` [6]. This bundling is naïve, in the sense that it assumes different instances of similar tasks will finish at the same time. However, as illustrated in the left panel of Fig. 1, the wallclock time to complete different tasks of the same kind can vary dramatically. The most common causes include differences in instance difficulty, on-node performance, and inter-node communication, but there are many others and they all lead to a substantial amount of wasted, idle resources.

As illustrated in the right panel of Fig. 1, mixing different types of tasks can exacerbate the problem and lead to even more wasted resources, even if each task finishes roughly when expected. Waste has the potential to undo the benefits of the incentives of large jobs.

Backfilling computational tasks can markedly reduce the amount of wasted resources that can arise from naïve bundling. Backfilling is a widely adopted strategy in high-performance computing to minimize the amount of wasted resources by allowing light tasks to use idle resources, effectively at no additional cost. This enables large bundles that can qualify for administrative incentives. By providing information about what resources a task requires—how many nodes, how much wallclock time on those nodes, etc.—one can achieve substantially less waste in large bundled jobs. Figure 2 shows two example jobs where a simple backfilling strategy, even without extremely accurate timing estimates, reduced the amount of waste that would have resulted from naive bundling.

To perform effective backfilling, the scheduling application must have information about the computational workload of each task. The scheduler uses this information to solve an optimization problem to determine the optimal execution order at runtime. Determining which tasks to compute at runtime rather than when the job is submitted to the batch scheduler's queue leads to other nice features. For instance, multiple collaborators can create computational tasks that can be intermingled, and multiple collaborators can submit jobs that attack the same large set of computational tasks. Ad-

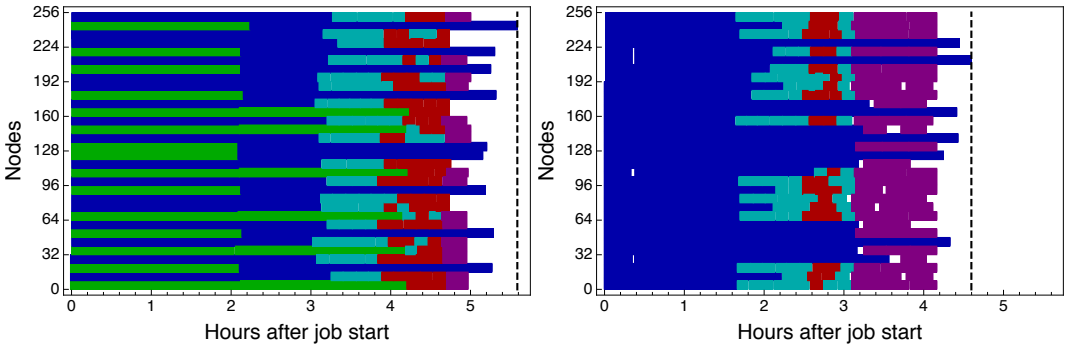

Figure 2. Two timelines of multiple tasks bundled and backfilled into a single 256-node job run on Titan. The horizontal colored bars represent different computational tasks, and each color represents different types of tasks. White space is wasted, idle time.

ditionally, a task can be changed after submission, and it is even possible to submit large jobs to the batch scheduler before deciding what tasks to schedule.

The dynamic selection of computational tasks at runtime also entails separating the job allocation request to the batch scheduler from the description of the computational work. Traditionally, these are intermingled in a single script that is submitted to the batch scheduler. Instead, to keep the job allocation independent from the tasks, independent descriptions are needed. The job script submitted to the batch scheduler must be aware of the job's properties, such as wallclock time, number of nodes, etc., in order to configure the backfilling software. The description of an individual task doesn't need these things—but it does need to know the resources required for the particular task, so that the backfiller can make a comparison between what resources are free and what are required to execute a task.

Here, we discuss `METAQ` [7] and `mpi_jm` [8], two pieces of software designed to make bundling supercomputing tasks into large jobs easy. Another pilot system[9], `RADICAL-Pilot`[10], focuses on smaller, shorter jobs, and therefore optimizes for launch rate and number of concurrent tasks. In contrast, LQCD calculations are often long-running and controlling communications efficiency and the ability to bundle tasks with different resource requirements are likely to yield the best performance.

2 METAQ

`METAQ`[1] is a simple implementation of software that can backfill computational tasks. A manual is available on the arXiv[7]. Implemented in `bash`, it forms a proof-of-principle that a backfilling strategy can be used to waste fewer cycles. However, as we will discuss, it has major drawbacks that has prompted the development of `mpi_jm`, which will be discussed in the next section. Nevertheless, `METAQ` has been used successfully in production, and allowed us to intermingle the computations for Refs. [3, 4] with those for Ref. [11] with little effort. We use it on smaller clusters too, where bundling may not be necessary but the feature that collaborators can perform one another's work is useful.

`METAQ` is designed to replicate the experience a user might already have grown used to when interacting with batch schedulers such as `SLURM` [12], `MOAB` [13], `TORQUE` [14], or `PBS` [15]. That is, the description of the computational tasks looks syntactically like a job script that might be submitted

[1] Available at https://www.github.com/evanberkowitz/metaq, and licensed under GPLv3.0

to one of those schedulers. The user provides, using `#METAQ` markup, information about the task and then is free to do any environmental setup and high-performance work (by invoking the batch scheduler's run command, such as `srun`, `aprun`, etc.). This design ensures that executables remain unchanged.

In contrast to a real batch scheduler, there is no 'submission'—the user simply puts the task description in file system directories. The job scripts that are submitted to the batch scheduler look through these (user-configurable) directories for task scripts, compares the needed resources and wall-clock time to what is currently available in the given allocation, and starts the task if possible. As long as the file system permissions are set correctly, any collaborator may contribute or execute tasks. Because the job scripts simply configure the backfilling, they are extremely uniform and simple to write.

Let us now discuss the drawbacks of `METAQ`. First, `METAQ` trusts the user, in that if a task claims to only need a certain set of resources, those needs are not enforced. Therefore, if a task script lies, `METAQ`'s accounting of busy and idle resources will be incorrect, and the discrepancy may cause problems.

Second, `METAQ` lacks the flexibility to execute the same task in different configurations. For example, some tasks might run most efficiently on 16 nodes but can be done on 8 nodes if that's all that is available. `METAQ` does not have the ability to make these kinds of task-configuring decisions at runtime.

Third, the ability to assign work to an accelerator (say, a GPU) independently of assigning work to the CPUs is dependent on administrative policy. For instance, this kind of overcommitting is not allowed at OLCF, but is possible on the Surface GPU cluster[16] at LLNL.

Fourth, the lack of a 'submission' utility means that the user must keep track of where the tasks belong. In principle, a simple script may do, but none has been written as yet.

Fifth, because the tasks are represented by files on disk, iterating over the tasks can be slow and require examining the disk frequently.

Sixth, and most serious, the backfilling logic happens on the shared resources where the batch scheduler runs. Because each run command (eg. `srun`, `aprun`, etc.) can be resource-intensive, and the backfiller itself launches monitoring processes for each task, it is possible to stress these resources. Indeed, when scaling to very large jobs on Titan, we once crashed these service nodes and brought the machine down. This incident resulted in a revision of the user guide, limiting the number of allowed simultaneous processes any job may use[17].

Many of these drawbacks are not necessarily problems but are consequences of `METAQ`'s implementation. In the next section we will discuss `mpi_jm`, software in preparation that is designed to address these drawbacks.

3 `mpi_jm`

The success we had using `METAQ` has inspired us to create `mpi_jm` [8], backfilling software that allows us to manage the computational resources allocated to a job much more finely. Unlike `METAQ`, which relied on the batch scheduler's run commands to move work to the compute nodes, `mpi_jm` puts a daemon on each node at a job's start. This daemon can monitor process IDs (PIDs), launch tasks as new processes, and can track resource consumption more reliably.

By design, `mpi_jm` requires very little modification of binaries. On compilation of an `mpi_jm`-aware executable, one must include one file and link against one library. Once compiled with `mpi_jm`, a binary works whether or not it is launched under `mpi_jm` management. This is crucial for software maintenance and easy debugging.

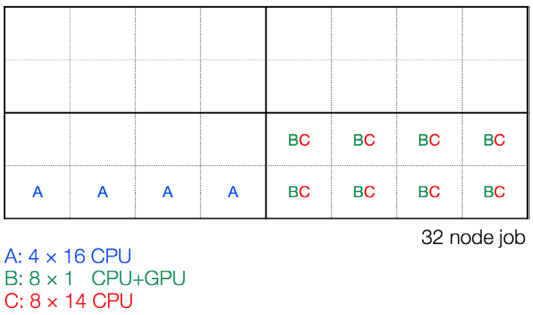

Figure 3. An example 32-node job on a hypothetical machine with 16 CPUs and 1 GPU per node, partitioned into 4 blocks of 8 nodes, being filled with three types of tasks (A, B, and C) which have different resource requirements. Each node is indicated by a dotted box, while a block is indicated by a box of thick lines. While A tasks require the whole node, B and C tasks can share nodes, B taking advantage of the GPUs, C only CPUs.

Additionally, two handshake calls are required, just after `MPI_Init` and `MPI_Finalize`. The daemon uses `MPI_comm_spawn` to provide your executable with its own `MPI_comm_world`. However, according to the `MPI` specification, if any of these spawned processes die, the highest-level communicator will die as well. Therefore, after the first handshake, the daemon disconnects from the executable so that a local error will only have local effects and will not cause the entire job and its other computational tasks to fail. The second handshake is needed to send exit information from the disconnected processes back to the monitor daemon.

These minor modifications means that it is easy to integrate `mpi_jm` into any software. Indeed, we have already successfully integrated it with `QMP`, the communications-wrapping layer of the USQCD software stack[2]. By simply adding `-with-mpi-jm=/path/to/mpi_jm` to the configuration of `QMP`, any additional USQCD software libraries or user applications built on top of that `QMP` installation will be `mpi_jm`-aware.

As was found with `METAQ`, inaccurate runtime estimates create waste. One important source of inaccuracy when using a run command to launch tasks in the allocation of compute nodes is that as tasks of different sizes start and complete, the available nodes become fragmented. Later launches may then be placed onto poorly-connected compute nodes. To resolve this issue `mpi_jm` partitions all the allocated nodes into user-configurable blocks. To maximize communication performance, `mpi_jm` uses information about the network topology to group neighboring nodes together[3]. Every task that can be considered must be able to use no more than the resources available in a block, but if it fits it will always get nodes with high performance connectivity, improving runtime and runtime predictability. If a task needs 64 nodes but the blocks are 32 nodes each, that task will not run.

Each block shares a communicator with the overall scheduler, which is responsible for assigning tasks to different blocks. The scheduler ensures that work isn't duplicated, and that work is only given to a block with the correct available resources. Within a block, smaller or resource-saturating tasks may be launched, as schematically depicted in Fig. 3, where A tasks consume entire nodes but B and C tasks can peacefully share a node.

[2] An `mpi_jm`-capable `QMP` is available at https://github.com/callat-qcd/qmp on the `mpi_jm` branch, and a pull request will be issued to the main `QMP` repository when `mpi_jm` is released.

[3] In future releases it might be possible determine configurations of fast blocks and storing that information, making it available machine-wide, or using more sophisticated methods for determining network bandwidth speeds.

In the case where there are accelerators, `mpi_jm` binds the appropriate CPUs to the GPUs based on configuration options and the hardware topology of the node. Suppose a GPU-capable task just needs 1 CPU. It may nevertheless make sense, to protect the CPU's caches, to think of it as requiring two CPUs, or at least to block other tasks' access. On a node with 16 CPUs and a GPU, for example, it may be best for performance to share a node between a (1 CPU + 1 GPU) task and a 14-CPU task, leaving one CPU idle. Moreover, it is conceivable that it's even better to give the CPU task only 12 CPUs, to limit competition for memory bandwidth, for example. Tuning this kind of balance can be done on small jobs. The resulting idle time is much less than otherwise achievable, and, as may increase overall performance and scientific output, should not be considered a loss, especially if one can execute different tasks on the same node simultaneously.

To create computational tasks we have implemented a `python` interface to `mpi_jm`. This allows the user to create a python class for every task type. That class can implement arbitrary pre- and post-execution steps and can implement a variety of configurations (for example, if a task can be run on different number of nodes). One has access to the full power of `python`, including its libraries, which makes it simple to handle logic surrounding task creation.

Currently, we describe tasks via `YAML` [18] dictionaries on disk, much as `METAQ`'s task scripts live on disk. This means that currently `mpi_jm` retains the fifth drawback discussed at the end of the previous section. However, the examination and parsing of task scripts happens less frequently and is done at a much lower level (compared to `METAQ`'s `bash` implementation). In addition, the `mpi_jm` scheduler doesn't run on the shared resources, but on the compute nodes. This removes the possibility of taking down the service nodes and rendering the machine temporarily useless, and is substantially more neighbor-friendly.

Traversing file directories is not the only possibility for storing task descriptions—`python` unlocks many possibilities. One can imagine interfacing with a graph database or other task-generation, task-management, or automated data analysis suites[19, 20]. Indeed, data reductions and analyses can themselves be run as tasks under `mpi_jm` if it is logistically or economically beneficial.

Finally, because `mpi_jm` relies only on the `MPI` interface for communication, it can be used to manage resources across grids—for example, if they are connected only by `TCP/IP` (so long as `MPI` works). This has some interesting potential big-data applications, including for sharing computational resources across computing facilities (see, for example, Ref. [21]) or eliminating idle cycles with tasks from users from afar. One might even imagining issuing overlapping allocations which take advantage of different resources.

4 Summary

As we move towards the exascale era, the resources needed per problem instance for some computational tasks of scientific interest will stop growing. To take full advantage of these machines, tasks can be bundled together. `METAQ` as a production-capable prototype, and `mpi_jm` going forward, provide a path towards intelligent backfilling of computational tasks.

A dynamic, backfilling task scheduler not only allows one to scale up to large jobs to take advantage of administrative incentives, but also provides logistical benefits, such as ease of collaboration and a major reduction in the amount of wasted computational cycles.

`mpi_jm` solves many of the issues with `METAQ`, at the cost of very mild modification of one's binaries. The needed modifications are already accessible to software built on top of the USQCD stack. We hope to have a release candidate of `mpi_jm` available soon, and anticipate distributing it widely with a liberal license.

Acknowledgements

METAQ was tested on aztec, cab, surface, and vulcan at LLNL through the Multiprogrammatic and Institutional Computing and Grand Challenge programs. METAQ was also tested and used in production on titan at Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and edison and cori at NERSC, the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was supported in part by the Office of Science, Department of Energy, Office of Advanced Scientific Computing Research through the CalLat SciDAC3 grant under Award Number KB0301052. This work was supported in part by the DFG and the NSFC Sino-German CRC110. This research used resources of the Oak Ridge Leadership Computing Facility located at ORNL, which is supported by the Office of Science of the Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] *How usage is charged*, http://www.nersc.gov/users/accounts/user-accounts/how-usage-is-charged/, accessed: 2017-01-23
[2] *Titan scheduling policy*, https://www.olcf.ornl.gov/kb_articles/titan-scheduling-policy/, accessed: 2017-01-23
[3] C.C. Chang, E. Berkowitz, D. Brantley, C. Bouchard, M. Clark, N. Garron, B. Jóo, T. Kurth, C. Monahan, H. Monge-Camacho et al. (CalLat), *The Nucleon Axial Coupling from Lattice QCD*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf.
[4] E. Berkowitz, D. Brantley, C. Bouchard, C.C. Chang, M. Clark, N. Garron, B. Jóo, T. Kurth, C. Monahan, H. Monge-Camacho et al. (2017), 1704.01114
[5] C. Bouchard, C.C. Chang, T. Kurth, K. Orginos, A. Walker-Loud, Phys. Rev. **D96**, 014504 (2017), 1612.06963
[6] OLCF, *wraprun*, https://github.com/olcf/wraprun (2015)
[7] E. Berkowitz (2017), https://github.com/evanberkowitz/metaq, 1702.06122
[8] K. McElvain, G. Jansen, E. Berkowitz, A. Walker-Loud, *mpi_jm*, in preparation (2017)
[9] M. Turilli, M. Santcroos, S. Jha (2015), 1508.04180
[10] A. Merzky, M. Santcroos, M. Turilli, S. Jha, CoRR (2015), 1512.08194
[11] A. Nicholson, E. Berkowitz, C.C. Chang, M.A. Clark, B. Joo, T. Kurth, E. Rinaldi, B. Tiburzi, P. Vranas, A. Walker-Loud, PoS **LATTICE2016**, 017 (2016), 1608.04793
[12] SchedMD, *SLURM: Simple Linux Utility for Resource Management*, https://slurm.schedmd.com/ (2006-2017)
[13] Adaptive Computing, *MOAB*, http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/ (2004)
[14] Adaptive Computing, *TORQUE: Terascale Open-source Resource and QUEue Manager*, http://www.adaptivecomputing.com/products/open-source/torque/ (2003-2017)
[15] Altair Engineering, *PBS: Portable Batch System*, https://github.com/pbspro/pbspro (1991-2017)
[16] *Surface*, https://hpc.llnl.gov/hardware/platforms/surface, accessed: 2017-02-13
[17] *Job Execution on Titan*, https://www.olcf.ornl.gov/kb_articles/job-execution-on-titan/, accessed: 2017-07-26
[18] O. Ben-Kiki, C. Evans, I. döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, http://www.yaml.org/spec/1.2/spec.html (2001-2009)
[19] D. Hackett, W. Jay, E. Neil, V. Ayyar, *Automated lattice data generation*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf.
[20] D. Hackett, W. Jay, E. Neil, V. Ayyar, *Modern tools for "automated" analysis of lattice data: A case study*, in *Proceedings, 35th International Symposium on Lattice Field Theory (Lattice2017): Granada, Spain*, to appear in EPJ Web Conf.
[21] PanDA team, *The BigPanDA Project*, http://news.pandawms.org/bigpanda.html (2015), accessed: 2017-07-26