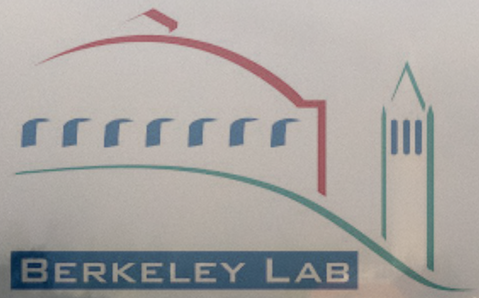
MPI Job Manager (MPI_JM)

Lattice 2022

André Walker-Loud



MPI Job Manager

- Ken McElvain (architect), André Walker-Loud, [Evan Berkowitz]

- What is the issue that this code is addressing?
- How does it work? (high level)
- What are future features?
- How and when can you get it?

What is the issue?

- The “measurement” stage of lattice QCD calculations requires the execution of millions of independent MPI tasks
- The tasks have mixed resource needs on heterogeneous systems
  - GPU-intense, CPU-only, heavy I/O
- The tasks often have chained/nested dependencies
  - A coherent sequential propagator for a 3-pt function can require 8 previous propagators to make a coherent sink
- A team may desire to run multiple projects with the same computing allocation with dynamic priorities (project A is higher priority generally, but you need results for project B for an upcoming seminar/conference…)
- Running this “swarm” of millions of MPI tasks as individual computations on “leadership computing facilities” is not tolerable by the supercomputing centers

Jargon

- job

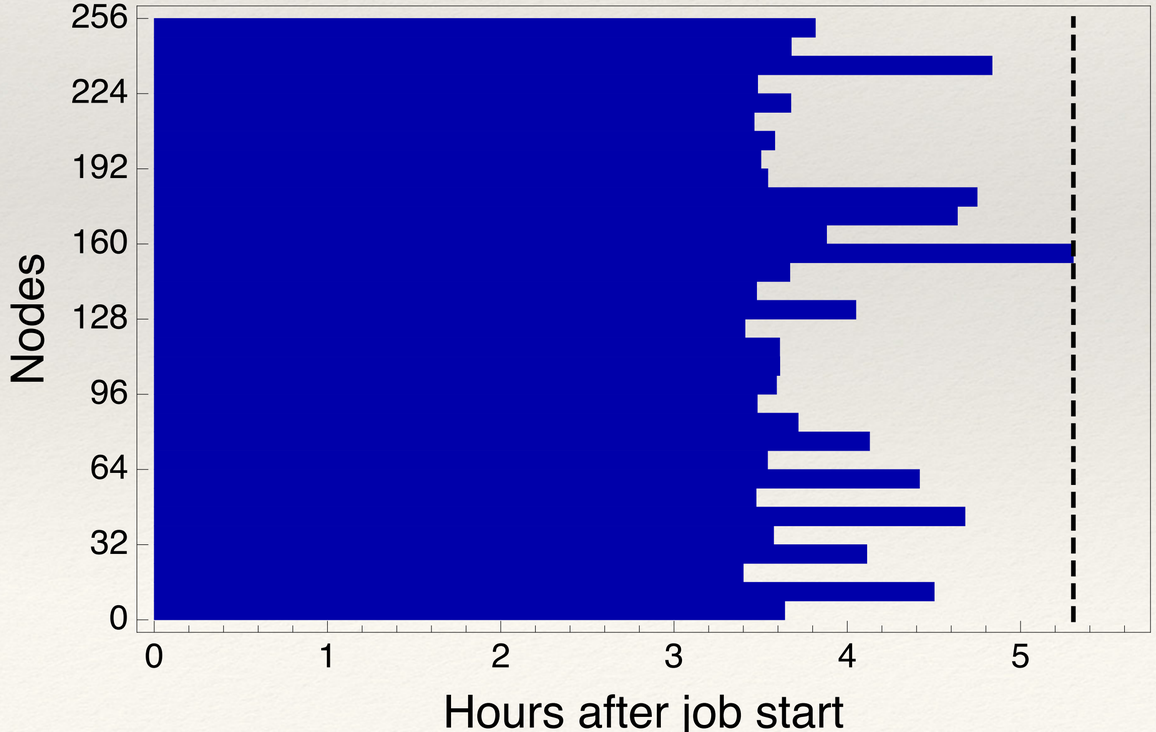
  a resource allocation on the compute cluster

- task

  an independent MPI task that could be run as a single job, or in a bundle on a larger job

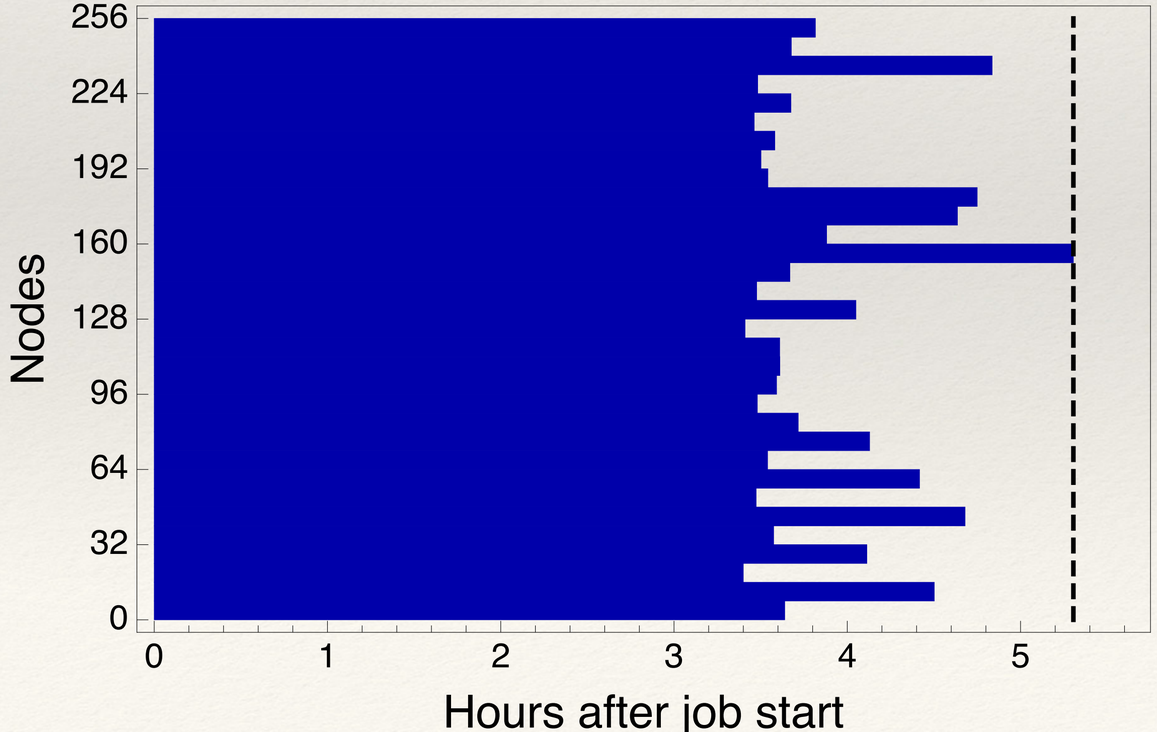
Managing millions of tasks on heterogeneous architectures

- Computing Centers often disfavor many small jobs
- We often bundle similar jobs into larger jobs
  - On heterogeneous architectures, different jobs may have very different resource needs
  - Even with similar jobs - significant waste can happen as fast/regular jobs wait for slow ones



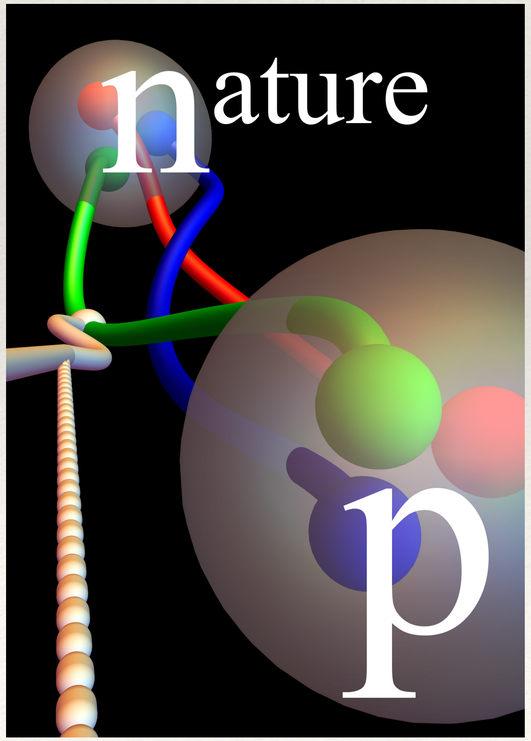
Managing millions of tasks on heterogeneous architectures

- METAQ - https://github.com/evanberkowitz/metaq - Evan Berkowitz
- Our first solution is a set of configurable `bash` scripts that can manage many different types of tasks and backfill idle nodes
  - Different colors represent jobs with different resource needs
    (on average, 30% wasted idle nodes reduced to ~5% wasted idle nodes)



Managing millions of tasks on heterogeneous architectures

- METAQ - https://github.com/evanberkowitz/metaq - Evan Berkowitz
- METAQ has served us very well - allowing us to very efficiently use Titan and its successor Summit (both at Oak Ridge Leadership Computing Facility (OLCF) @ ORNL)
  - METAQ was used to simultaneously run computations for these two projects
    without METAQ - we would not have been able to run either, let alone both, at OLCF



LETTER

https://doi.org/10.1038/s41586-018-0161-8

A per-cent-level determination of the nucleon axial coupling from quantum chromodynamics

C. Chang et al (CalLat)

Nature 558 (2018) no.7708, 91-94 [arXiv:1805.12130]

Heavy Physics Contributions to Neutrinoless Double Beta Decay from QCD

A. Nicholson et al (CalLat)

Phys. Rev. Lett. 121, 172501 (2018) [arXiv:1805.02634]

- additional requirements
  - Ludicrously fast QUDA code (MDWF)
  - Free configs from MILC
  - Access to Titan/OLCF through INCITE

Managing millions of tasks on heterogeneous architectures

- METAQ - https://github.com/evanberkowitz/metaq - Evan Berkowitz
- At the same time, METAQ has short comings that are not addressable with `bash`
- Managing tasks with `bash` means each MPI task requires an `mpirun` or equivalent call
  - This places significant strain on the service nodes that manage the compute nodes (we have crashed both Titan and Summit with too many tasks being simultaneously run)
- Desirable features:
  - Place tasks on neighboring nodes (don't communicate across racks)
  - Place ranks on specific cores/GPUs of node
  - Simultaneously run GPU-intense and CPU-only tasks on same nodes
  - Dynamically assign priority of tasks
  - Flexible run "configurations" for the same task (try this or that if this isn't possible)
  - Dynamically "shed" nodes in a job allocation
  - Collect node errors to identify problems (can't talk to disk, can't talk to GPU, slow fabric…)

MPI_JM

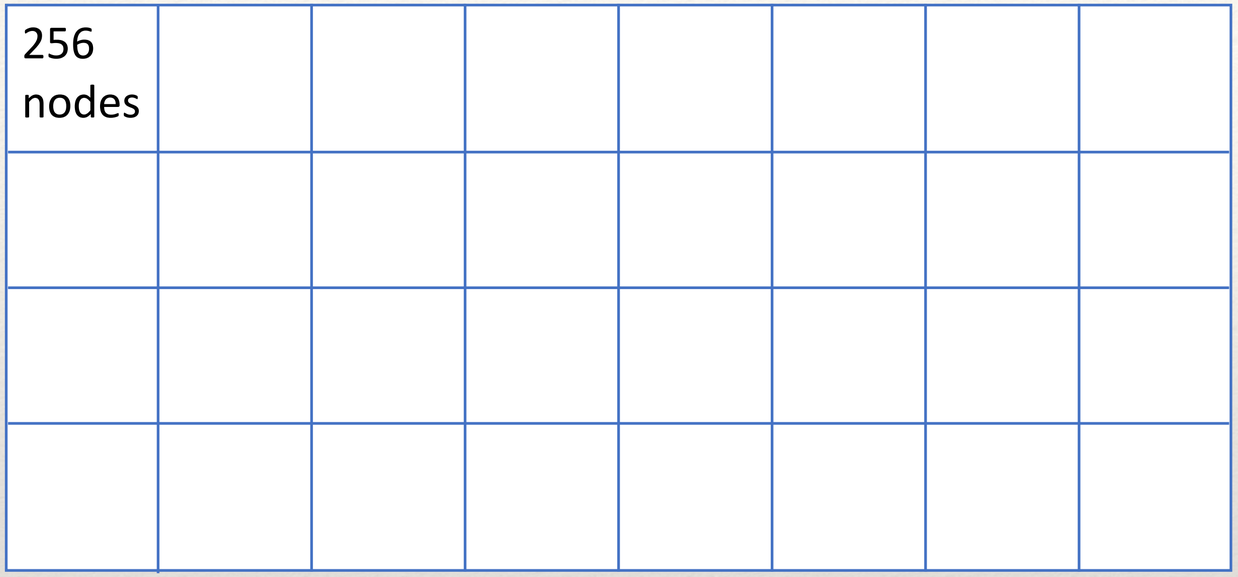
- MPI_JM - https://github.com/kenmcelvain/mpi_jm (temporarily private - license issue)
- To support these more desirable features, we have created MPI Job Manager (MPI_JM)
  - C++/Python code
  - MPI_JM uses a single `mpirun` call to start job-allocation and manage all tasks
  - MPI_JM requires minimal interface to your code to work (normal running still works)
  - MPI_JM uses a second Python Library (user written) that instructs MPI_JM where to collect `yaml` files that describe resource requirements. This library enables the user to build generic types of jobs (propagator, baryon-block constructor, …)
  - MPI_JM is configured to know about a machine
    - user only needs to specify resource requirements
    - MPI_JM handles locking ranks to various parts of the node to optimize performance

MPI_JM

- A job is launched with MPI_JM
  - MPI_JM carves up the nodes in “lumps” of size specified by the user
  - The lumps are all initialized concurrently (can bring up entire Summit in < 5 minutes)
  - MPI_JM begins to distribute tasks in each lump simultaneously
  - Tasks are located from user specified folders
  - User(s) can add more tasks during job (task list is refreshed regularly)
  - Multiple MPI_JM instances can work on the same list of tasks
  - Tasks are given user-specified priorities to help MPI_JM decide which jobs to run first
  - User can specify dependencies of tasks also
  - Tasks can have pre/post operations (check dependencies are ready - create dependent task)

MPI_JM

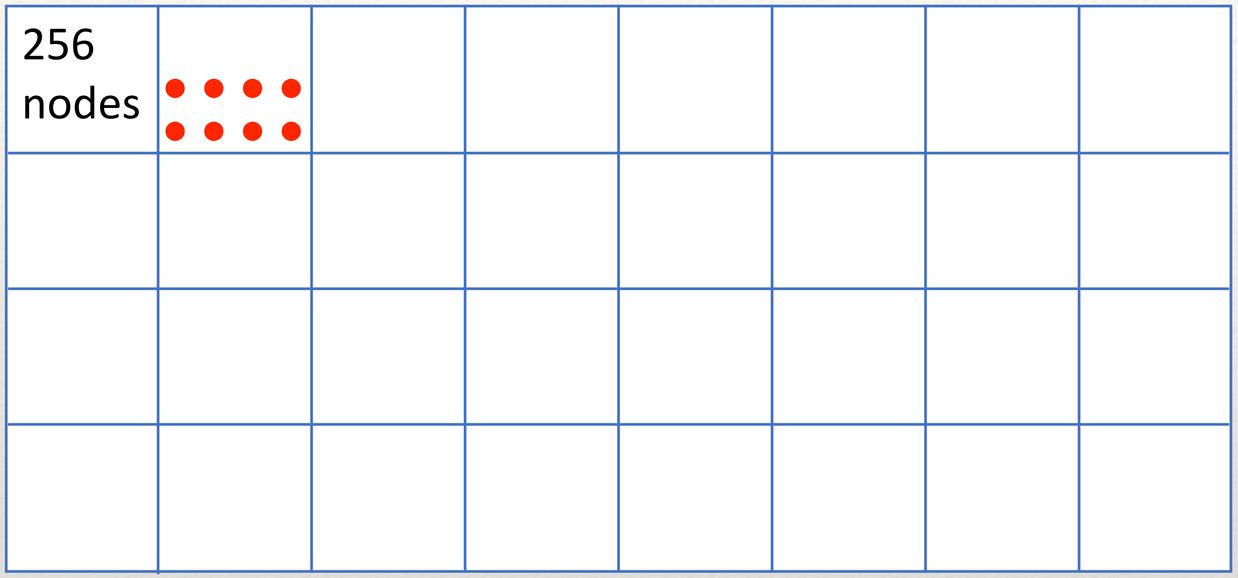
Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks



- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

MPI_JM

Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks

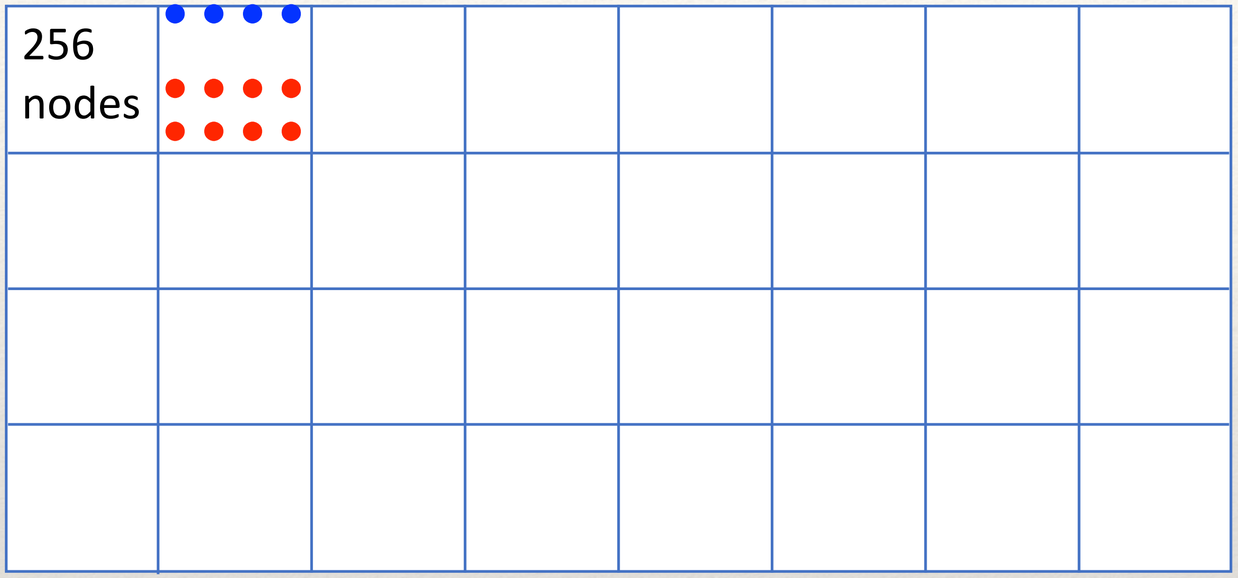


64 node CPU task ●
32 node GPU task ●
128 node CPU+GPU task ●

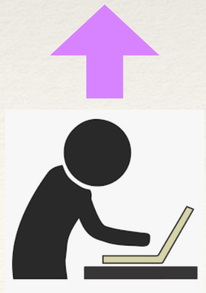
- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

MPI_JM

Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks



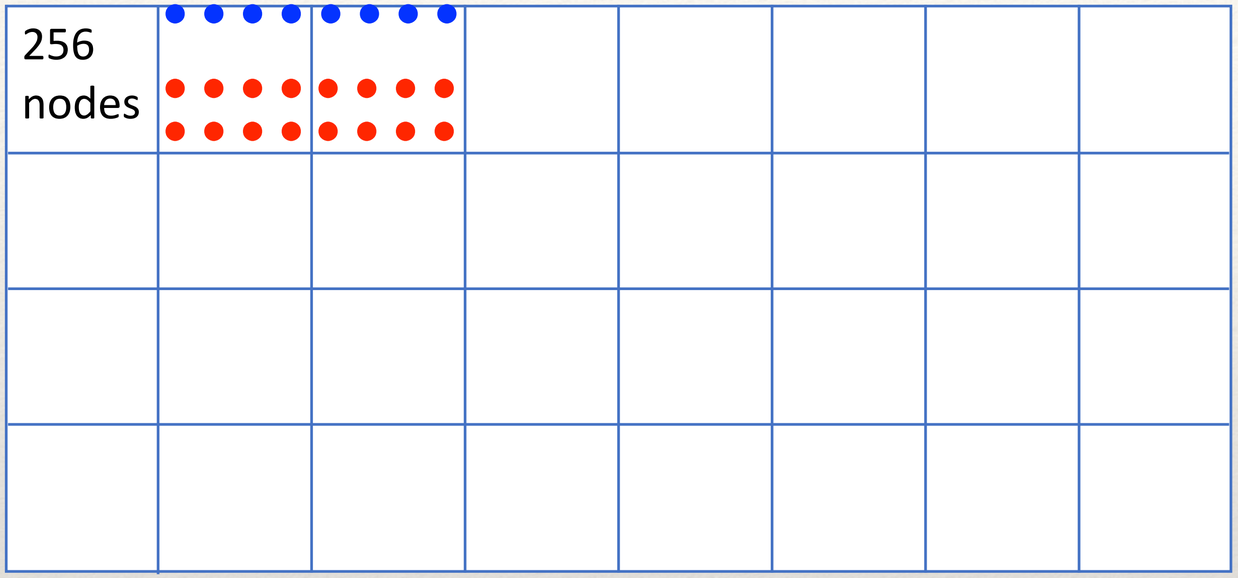
64 node CPU task
32 node GPU task
128 node CPU+GPU task



- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

MPI_JM

Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks

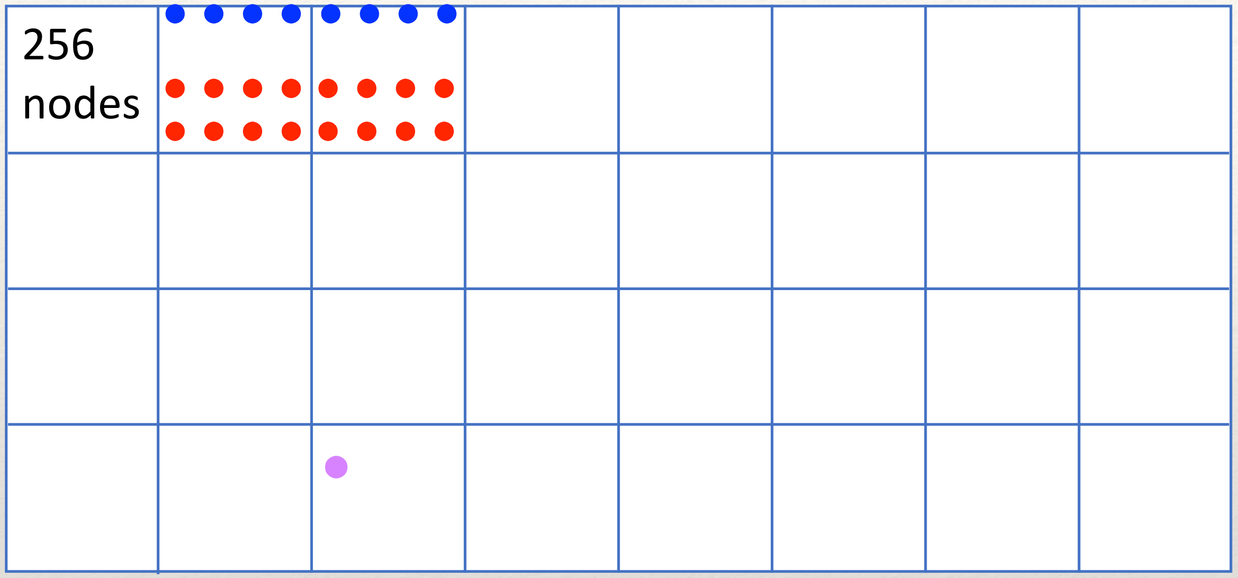


64 node CPU task
32 node GPU task
128 node CPU+GPU task

- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

MPI_JM

Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks

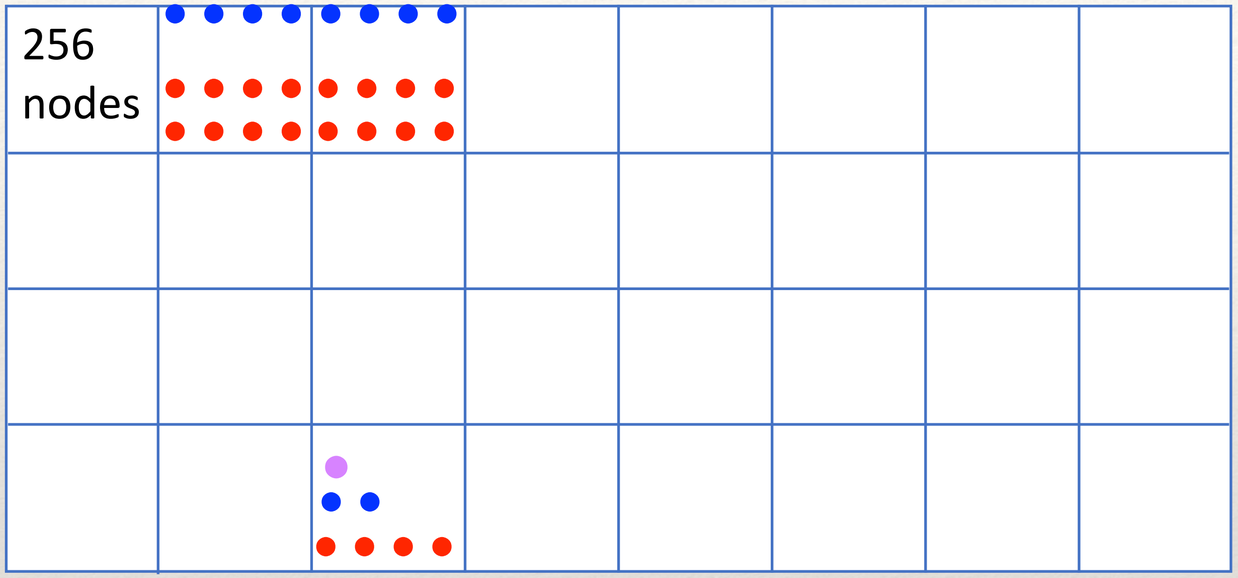


64 node CPU task ●
32 node GPU task ●
128 node CPU+GPU task ●

- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

MPI_JM

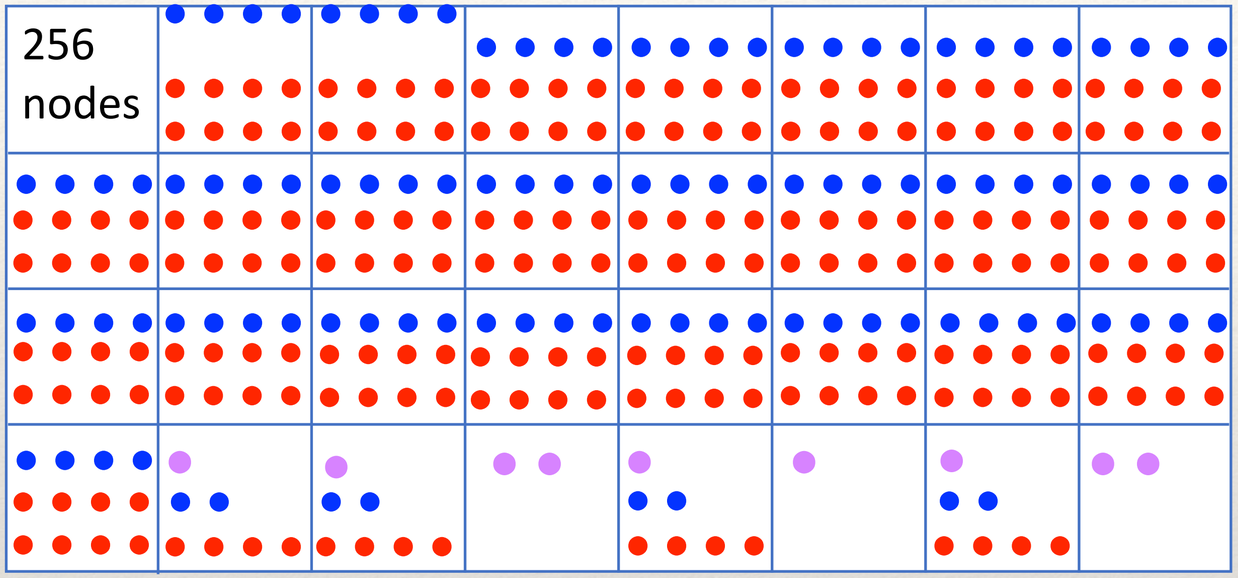
Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks



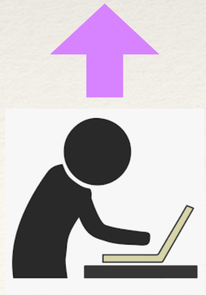
- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

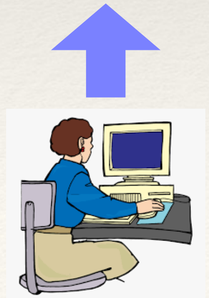
MPI_JM

Example: 4096 nodes, 1 mpirun, 32 “blocks” of 256 nodes, arbitrary number of “tasks” launched within blocks



64 node CPU task
32 node GPU task
128 node CPU+GPU task





- Main communicator organizes nodes in sorted list to keep nodes physically local
- User specifies “block size” (256) to distribute master communicator over all nodes
- Multiple users can continuously feed tasks to MPI_JM launch folder
- Tasks are launched via COMM_SPAWN
  (failed task does not crash entire allocation)

Minimal Interface

- Code that uses QMP can utilize MPI_JM
  (not yet in mainline QMP)

callat-qcd / qmp Public
forked from usqcd-software/qmp

Edit Pins | Unwatch 3 | Fork 14 | Star 0

Code | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

mpi_jm.no_call... | qmp / lib / mpi / QMP_init_mpi.c | Go to file

walkloud merging... Latest commit c2505cb on Jan 21, 2019 History

6 contributors

135 lines (117 sloc) | 3.18 KB | Raw | Blame

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdarg.h>

#include "qmp_config.h"
#include "QMP_P_COMMON.h"

#ifdef QMP_MPI_JM
#include "jm.h"
#endif
```

```c

#ifdef QMP_MPI_JM
  jm_parent_handshake(argc, argv);
#endif

```

```c
void
QMP_finalize_msg_passing_mpi (void)
{
    int flag;
    MPI_Finalized(&flag);

    if (!flag) {
        MPI_Finalize();
    }
#ifdef QMP_MPI_JM
    jm_finish(0, "QMP MPI finalized.");
#endif
}
```

```c
void
QMP_abort_mpi (int error_code)
{
#ifdef QMP_MPI_JM
    /* try to do clean shutdown instead of ABORT */
    /* BETTER: should really ask jm_master to kill job   */
    jm_finish(error_code, QMP_error_string(error_code));
    MPI_Finalize();//this seems out of step with the change made to QMP with the &flag and finalize above
    exit(error_code);
#else
    MPI_Abort(MPI_COMM_WORLD, error_code);
#endif
}
```

Known Future Features

- Performance monitoring
- Improve overlapped use of distinct resources
- Inter-job communication tasks
- Dynamic task configuration
- Enhance fault-tolerance and diagnosis
- …

Known Future Features

- Performance monitoring
  - User claims task will take 2 hours
  - MPI_JM will collect statistics on actual wall-clock time usage and use this to decide if a task can complete before end of allocation
  - MPI_JM can monitor performance of blocks, not just job type, in case some blocks of nodes are slower than others

Known Future Features

- Improve overlapped use of distinct resources
  - After a task is finished, but needs to write I/O, the GPUs and CPUs for that task are otherwise idle.
  - MPI_JM can be informed that the GPU-usage is complete, and a new GPU task can be placed on the nodes while I/O is being performed from the previous task

Known Future Features

- Inter-job communication tasks
  - Utilizing local NVME storage can be complicated for multi-node jobs with parallel I/O
  - Users can create “follow up” tasks/work with MPI_JM such that MPI_JM will launch a helper task to collect the NVME from the individual nodes and build the complete file
  - Task 37 may require the data files from Tasks 0-7, which were all written to NVME. MPI_JM can collect these and place them on the NVME that Task 37 will run on (MPI_JM can decide ahead of time which nodes Task 37 will run on)

Known Future Features

- Dynamic task configuration
  - The optimal number of nodes for task-A might be 4 (minimum required to fit job in memory)
  - Towards the end of a job-allocation, there may be insufficient time to complete task-A on 4 nodes
  - But, with a performance loss, the job could run on 8-nodes in the remaining wallclock time
  - The user can pre-specify possible task-configurations that MPI_JM can try to fit in the available resources, thus further reducing “wasted” time at the end of an allocation
  - One often observes 1000-3000 nodes available for only 45 minutes - with the optional run-configurations, especially in bottom feeder mode - users could opt to utilize these nodes that would otherwise be idle as the scheduler makes room for large jobs

Known Future Features

- Enhance fault-tolerance and diagnosis
  - Even mature systems exhibit random trouble. eg. the GPUs on a node may take too long to initialize, so the task aborts/times out, but the very next task will run successfully
  - MPI_JM can have a "re-try" option to catch such failures
  - MPI_JM can perform pretests of node health before launching the first task
    - Can the nodes see the file system?
    - Can the nodes see their NVME?
    - Can the nodes communicate via MPI? Is the MPI slower than it should be?
    - MPI_JM can detect if task is actually running (sometimes, failed I/O doesn't cause task to abort or time-out, MPI_JM could trigger a "kill" to this job)
  - All these errors/issues can be logged, ideally in some SQL database, that can be used to help users and the sys-admins identify problematic nodes

Conclusions

- MPI_JM has worked great for running our LQCD calculations on Summit
- While we haven't succeeded in getting $ to bring it to full beta-stage - we are determined to finally do it this Fall - and to share with anyone interested in trying it out
- MPI_JM is not specific to LQCD - it works with any compute problem that requires MANY individual tasks, and has dependencies etc.
  Ken has been running nuclear equation of state calculations with it
  Computation at one nuclear density can be used as initial guess for similar density
- There are many known features that will improve running as well as I am sure many not-yet-realized features
- It would be great to get users to get more feedback
- When it is ready - we'll email latticenews to get volunteers

Thank You