QCD on the Modular Supercomputer

Eric B. Gregory[*†]
Bergische Universität Wuppertal & Jülich Supercomputing Centre
Institute for Advanced Simulation, Forschungszentrum Juelich, Wilhelm-Johnen-Strasse, 52425 Juelich, GERMANY
E-mail: e.gregory@fz-juelich.de

Modular supercomputers contain heterogeneous computing resources on which the user can allocate a hardware mix meeting the demands of the calculation. I discuss motivations for generalizing QCD simulations for use on modular supercomputers, and identify possible models of LQCD simulations suitable for such an environment. I introduce QMOD, a system of libraries in development to enable task-parallelization of existing lattice QCD simulation codes for modular systems. The Jureca cluster at the Jülich Supercomputing Centre serves as a test bed for modular supercomputing strategies.

37th International Symposium on Lattice Field Theory - Lattice2019
16-22 June 2019
Wuhan, China

[*]Speaker.
[†]E. B. Gregory is supported by Deutsche Forschugsgemeinschaft grant SFB TRR 55.

1. Modular Supercomputing – an Introduction

In the past two decades increasing parallelism has driven vast growth in computing power. Fields like lattice QCD with large data-parallel codes have demonstrated particularly impressive performance gains. In other fields, data and kernels may be more varied and parallelization boosts more moderate.

Amdahl's Law limits the scalability of any calculation, with less-parallelizable elements of the calculation imposing a larger bottleneck as the number of parallel tasks increases. In some cases the less-scalable elements would perform better with a different parallelization scheme, or be better suited to different hardware.

A modular supercomputer has different types of computing resources networked together such that a user can allocate a heterogeneous mix of resources, tailored to the specific problem. Different kernels can be assigned to their optimal hardware within a single parallel job. Modular supercomputers are in development and use at the Jülich Supercomputing Centre (JSC). Building from ideas and prototypes developed in the DEEP Projects[1], the Jureca machine has "cluster" and "booster" modules, based on Intel Haswell and Knights Landing (KNL) nodes, respectively. They can be used in a common parallel job as described below in Section 4.

Juwels, the flagship machine at JSC has been designed from the start as an evolving modular supercomputer. It currently has a cluster module of 2271 Intel Skylake compute nodes. In 2020 this will be joined by a booster module of GPU accelerated nodes. The machine will continue to evolve over the coming years with additional modules, including a quantum computing device and a neuromorphic module, with architecture inspired by biological neurons. Figure 1 shows Juwels with some expected modules. Future modular supercomputers may include network-attached memory or GPU network-attached accelerators, both without a host CPU. This *disaggregation-of-resources* permits the user finer control over resource allocation.

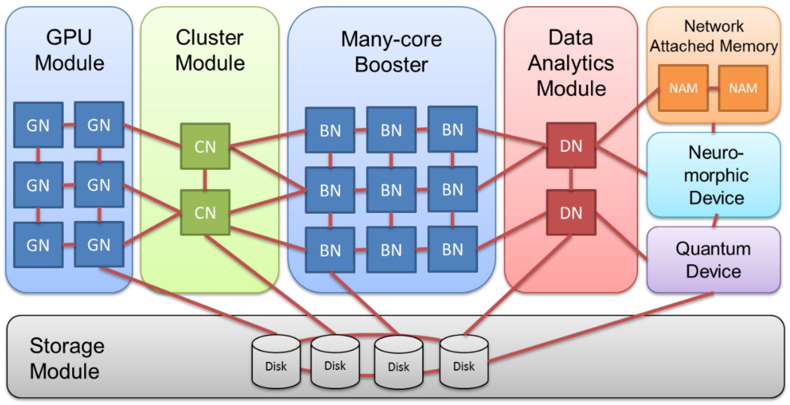

Figure 1: Projected Juwels supercomputing modules. Figure from [2].

2. QCD & Modular Supercomputing

The homogeneity of LQCD data allows near perfect weak scaling of a Dslash kernel up to more than a million threads distributed across MPI tasks. It may seem then that there is little point to pursue schemes beyond this simple data-parallelism and the embarrassingly parallel nature of measurement on a gauge ensemble or multiple Markov chains. However, the advent of large modular systems should lead us to consider whether the workflow of LQCD could present unexploited task-based concurrencies.

There is an additional bonus from considering more task-oriented parallelization. Even in a wholly homogeneous cluster, different functional tasks could be assigned to quasi-independent groups of nodes. While we may not be able to tailor the architecture for each task in this case, we could still tune the number of nodes assigned to each group according to workload and parallelizablity. With a sufficiently fast network, efficiency should improve.

Users of clusters with GPU accelerators already take advantage of the heterogeneous nature of the nodes with libraries like QUDA [7], where parts of the calculation are offloaded to the accelerator, while the CPU handles other work. We wish to consider more general and flexible task-parallelism, which could be mapped to a modular system.

We examine typical LQCD workflows for examples of elements that can be executed simultaneously or out of order. We assume a sufficiently fast network for this exercise, such that moving data between nodes is cheaper than writing it to disk and re-reading it.

One example is to have dedicated I/O nodes in an HMC job. We assign a group of booster nodes to evolve HMC trajectories. At appropriate check-pointing intervals they pass a copy of the gauge field to a group of I/O nodes, which would make simple essential measurements (e.g., Wilson flow) before writing the configuration to disk. The HMC evolution continues uninterrupted.

A second example is to assign generating propagators and performing contractions to separate hardware groups. Multi-nucleon correlators and correlators from stochastic sources require large numbers of propagators and even more contractions. The contraction group does not need to retain a copy of the gauge field in most cases. This example is depicted in Fig.4.

Separating tasks allows tailoring the size and characteristics of the hardware allocations to tune the load balancing.

3. The QMOD Project

The QMOD project at JSC has the goal of developing versatile libraries to investigate different concurrency models in the context of a modular supercomputing system. The software should allow node groups to execute different task components of an LQCD code concurrently. It should provide functionality to pass generic lattice data structures — gauge fields, propagators, eigenvectors — between hardware groups.

The software should also allow testing of a variety of concurrency schemes on a range of hardware architectures. It should interface with existing, well-developed open-source simulation codes. To this end, the QMOD libraries are designed to work as an add-on to the widely-used USQCD software stack [3].

The USQCD software stack is a hierarchy of layers containing software libraries which handle general utility aspects of a lattice simulation in a common manner. At the top are simulation applications, such as the MILC code [4] and Chroma[5] which can be linked to elements of the stack. The QDP++ library parallelizes lattice field data structures across processes and nodes, while QIO provides I/O functions for lattice field objects. QMP is a message-passing library providing a common communications interface. The libraries can be compiled to target a wide range of hardware architectures.

Of particular interest are the architecture-specific Dslash and inverter libraries QPHIX [6] and QUDA. The former is optimized for Intel many-core processors, and the latter for Nvidia GPUs. Since Intel and Nvidia hardware are each well-represented in the clusters at JSC, we have the opportunity to test simulation schemes with components executing concurrently on Intel or GPU hardware modules.

The QMOD project consists of two libraries interacting with different levels of the USQCD software stack, QMPadd and QMOD. QMPadd establishes the communication link between partitions and the QMOD library provides data transfer functionality.

3.1 The Intercommunicator

To avoid a massive re-design of existing libraries, code running on separate hardware groups must behave as if they were independent, so that QDP++ can provide data-parallelism (e.g., lattice layouts) on each, as normal. This means that the default QMP (or MPI) communicator must be local to each group. QMPadd establishes an additional communication channel, an *intercommunicator*, between groups for the transfer of lattice-wide data fields.

QMPadd is linked to the QMP library. When the simulation is initialized, all of the nodes in all hardware groups belong to the same default QMP communicator, normally mapping to MPI_COMM_WORLD. The QMP library provides a function, `QMP_comm_split()`, to split a communicator. This function is most often used for farming multiple simulations, but is exactly what we need here. QMPadd makes a copy of the default communicator as `global_comm`, then calls `QMP_comm_split()` to split the default communicator according to a key provided at run time. One might, for example, assign Haswell nodes to communicator 0 and KNL nodes to communicator 1. The new communicators then become the default QMP communicators of their respective groups.

The final step is to exchange the global node IDs of the local head nodes — the 0 nodes in each of the new default communicators — so that messages can be addressed appropriately later. Invoking the communicator split from the QDP++ *after* the initialization of QMP but *before* the initialization of the layout produces independent layouts on the different hardware groups. This requires the addition of a few lines in the appropriate `qdp_XXXX_init.cc` file.

3.2 Lattice field exchange

The QMOD library handles the exchange of lattice fields between hardware groups through the inter-communicator. This operation bears many similarities to I/O of lattice fields. Saving a lattice field to disk can be done in parallel, with every node writing its share of the layout, or one or more I/O nodes writing all of the data. In the latter case, the I/O nodes must collect the data from the other tasks, organize it and then write it to disk in a standard format.

So it is with sending a lattice field across the global intercommunicator. In the simplest “serial send” case, a single node collects the data from the local hardware group and sends it across the inter-communicator to the remote group. Alternately, the send could be handled in parallel by multiple comm nodes.

Likewise, reading a lattice field from disk is analogous to receiving of a lattice field through the intercommunicator. The recipient could be a single comm node "serial receive" which then distributes the data to the receiving group. Alternately the sending group could direct the data to multiple comm nodes in the receiving group.

Due to these similarities, the collection and communication functions in QMOD are closely modeled on corresponding functions in QDP++ and QIO. Currently, only the "serial comm" case of the a single sending node and a single receiving node is implemented. Developing the full range of functionality will allow optimization and tuning of the communication between hardware groups of different sizes or different architecture configurations. However, it presents a large programming task to consider all possible cases.

Inter-group communication differs from binary I/O in a significant manner. In a binary data file, there is normally a file header and the reading node knows the position of the file pointer within the full record. These provide context about the amount of data to digest and where the current bytes should go in the global data structure. The writing node can also set the file pointer to a known address beyond the start of the file.

In communication between hardware groups, the receiver expects to receive data in messages of a standard, pre-determined size (the full data structure may be too large for a single message). These messages may arrive out of order, so QMOD packages each with a header providing context. Crucially, the header contains the site indices corresponding to the field data in the payload.

3.3 Receive-from-any functionality

In its distributed form, QMP allows receipt of data only from a known sender. However, in the MPI standard, a message can be received from `MPI_ANY_SOURCE`. A single-line edit in the QMP library unlocks this functionality.[1]

A case where this is useful is when we have multiple sending groups. Consider a job with nodes divided into five groups. Four of the groups, perhaps on a partition of accelerator nodes, perform inversions to produce propagators. As each group completes a solution, it sends the propagator to group zero, which can receive from any group. Group zero continually takes propagators as they are available to contract appropriately into correlators, reducing waiting time.

In the receive-from-any case the receiving group listens for a message from any remote group. Upon receipt of the first packet it reads the header for the originating source, then locks the receive operation to only receive data that corresponds to the same sender and same lattice field until transfer is complete.

3.4 Chroma integration

New Chroma-style objects `QMODSendNamedObject` and `QMODReceiveNamedObject`, analogous to `QIOWriteNamedObject` and `QIOReadNamedObject`, provide access to the QMOD lattice field exchange functionality.

[1] In `lib/QMP_mem.c`, an existing assert insists that `(sourceNode >= 0)`. By relaxing that condition to `(sourceNode != -1)`, we can allow -2, the normal value of `MPI_ANY_SOURCE` to be passed as the souce node.

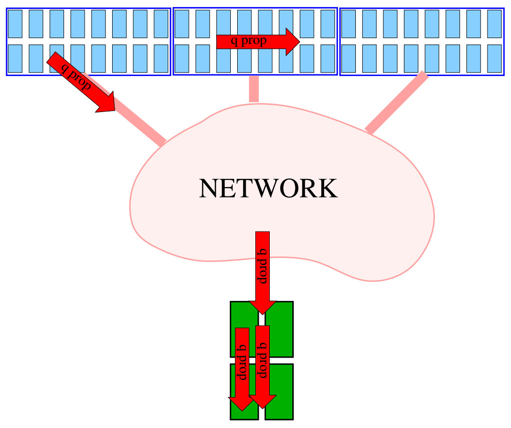

Figure 2: Booster nodes generating propagators and sending them to "cluster" nodes to contract into hadron correlators.

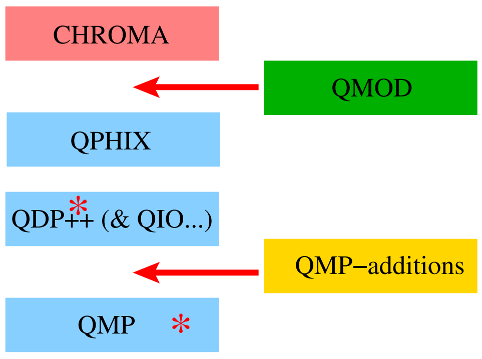

Figure 3: The QMOD library in the USQCD context with QPHIX. The $*$ indicates minor changes to the USQCD existing library.

4. QMOD modular computing tests

The Jureca cluster at Jülich Supercomputing Centre (JSC) is a first-generation modular cluster, with Intel Haswell and Knights Landing (KNL) nodes connected by a common network. It presents a unique test-bed for examining different concurrency schemes for modular supercomputing. The *packjob* feature of the Slurm scheduler allows support for multiple heterogeneous job components which can run as part of a common MPI communicator.

For this environment, we must first build separate QMOD-enabled Chroma executables for each architecture (i.e., Haswell and KNL). This means building the software stack for each with the insertion of QMPadd and QMOD libraries at the correct levels, illustrated in Fig. 4.

To run a Chroma modular job of node groups on the KNL and Haswell modules, we submit a *packjob*, a Slurm feature allowing heterogeneous job support. An excerpt from a sample QMOD two-group Slurm packjob script is in Fig. 4. Slurm options for different components are separated by "`#SBATCH packjob`" lines in the preamble (line 4). Partitions "batch" and "booster" correspond to Haswell and KNL nodes, respectively. In the `srun` launcher command, the two components of the job are separated by the ":" on line 11. At JSC the software environment is controlled by loading software modules. This must be done individually for packjob components on different architectures with the `xenv` command, as in lines 9 and 13. Note the Haswell and KNL components have different executables as well as input and output files. The `-modcolor` flags (lines 11 and 17) are read by the modified `QMP_initialize()`, and determine to which default communicator the component belong after the split.

5. Status and plans

The QMOD project is at a beginning, stage with much work remaining. An improved communication standard will allow the passing of XML metadata with lattice objects. Parallel communication modes will be added. To date, the code has been tested by exchanging propagators between

```
...
#SBATCH -N 2
#SBATCH -p  batch
#SBATCH packjob
#SBATCH -N 2
#SBATCH -p  booster
...
 srun  -n 2 xenv -L Intel -L ParaStationMPI -L imkl -L libxml2 -L GMP \\
env OMP_NUM_THREADS=24  ${EXEC_DIR}/sendtest_hsw  -i recv_test.xml \\
-o outhsw.xml -by 4 -bz 4 -c 24 -sy 1 -sz 1 -pxy 1 -pxyz 0 -minct 1 \\
-geom 1 1 1 2  -modcolor 0 :  \\
-n 2 xenv -L Intel -L ParaStationMPI/5.2.2-1-mt -L imkl -L libxml2 -L GMP  \\
env OMP_NUM_THREADS=68  ${EXEC_DIR}/sendtest_knl -i send_test.xml \\
-o outknl.xml \\
-by 4 -bz 4 -c 68 -sy 1 -sz 1 -pxy 1 -pxyz 0 -minct 1 \\
-geom 1 1 1 2 -modcolor 1
```

Figure 4: Slurm packjob batch script command for a modular QMOD test job.

groups using Chroma-style xml input file control. We will widen the tests to include the exchange of other lattice objects, such as gauge fields or eigenvectors. Development of full measurement routines will allow benchmarking and load balancing tests.

QMOD is tightly coupled to USQCD software and will be distributed open source. Interested parties can gain access to the pre-release code and necessary caveats by correspondence with the author. A more formal packaging and standardized build system is in the works.

6. Conclusion

We are entering an era of modular supercomputing, where a single parallel job may dynamically access different, specialized architecture to suit the needs of program components. For lattice simulations to reap the advantages, codes must accommodate task-parallelism. The goal of the QMOD project is to make it simple to test modular computing strategies and take advantage of upcoming modular supercomputers.

References

[1] https://www.deep-projects.eu/

[2] E. Suarez, N. Eicker, T. Lippert, John von Neumann Institute for Computing Series, **49**, 1 (2018).

[3] https://www.usqcd.org/usqcd-software/

[4] http://www.physics.utah.edu/∼detar/milc/milc_qcd.html

[5] R. G. Edwards *et al.* [SciDAC and LHPC and UKQCD Collaborations], Nucl. Phys. Proc. Suppl. **140** , 832 (2005) doi:10.1016/j.nuclphysbps.2004.11.254 [hep-lat/0409003].

[6] https://github.com/JeffersonLab/qphix/

[7] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, Comput. Phys. Commun. **181**, 1517 (2010) doi:10.1016/j.cpc.2010.05.002 [arXiv:0911.3191 [hep-lat]].