QMCPACK

Summary Version

QMPACK v3.4.0 released on 29 January 2018, included with this benchmark and also available via https://github.com/QMCPACK/qmcpack/releases

Purpose of Benchmark

Test performance of diffusion Quantum Monte Carlo methods. Obtain a fixed number of statistical samples within the shortest overall wall clock time.

Calculations are performed for a 256 atom nickel oxide (NiO) supercell containing 1536 up-spin and down-spin electrons/3072 electrons total.

Characteristics of Benchmark

QMCPACK is an open-source continuum quantum Monte Carlo simulation code. Electron positions are randomly sampled by a large number of Markov chains (or "walkers"). The benchmark uses a diffusion quantum Monte Carlo method that uses a diffusion-and-branch algorithm and a target population of walkers to evolve the walkers in time. During the simulation, the fluctuating population of walkers is load balanced between nodes at every time step of the simulation. In production simulations, measurements/statistics are collected at each time step of the simulation and for all the walkers.

Parallelization is performed over walkers: each MPI task has a number of walkers that are then distributed over OpenMP threads. QMCPACK can also use CUDA instead of OpenMP per MPI task. Strong scaling is high due to the high cost of computation relative to communication steps. However, each

walker has to be initially equilibrated, so the first 250 steps of each walker do not count towards useful statistics. The figure of merit measures the relative time to solution compared to reference run for a fixed number of total statistics. Thus, a hypothetical node that can run 50% faster improves the figure of merit more than a hypothetical 50% more nodes, because proportionately less time is spent in equilibration. Though the benchmark runs are somewhat shorter than what can be expected in production, the equilibration stage is essential to obtaining meaningful statistics.

The benchmarks are particularly sensitive to floating point, memory bandwidth and memory latency performance. To obtain high performance, the compiler's ability of optimize and vectorize the application is critical. Strategies to place more of the walker data higher in the memory hierarchy are likely to increase performance.

Mechanics of Building Benchmark

QMCPACK uses C++11, CMake, OpenMP, MPI, BLAS/LAPACK, Libxml2, HDF5, Boost, FFTW. CUDA is used with GPUs.

Full instructions to build and run QMCPACK are included in the README.md and described in more detail in https://docs.qmcpack.org/qmcpack_manual.pdf

This includes example build recipes for current leadership platforms, Linux workstations, and various compiler combinations. An initial build on RHEL or Ubuntu is recommended. Use of toolchains developed for older versions of QMCPACK is not recommended, supported, or tested.

To build the MPI-OpenMP version, typical steps are:

```
cd build
cmake -DCMAKE_C_COMPILER=mpicc \
      -DCMAKE_CXX_COMPILER=mpicxx \
```

```
      -DENABLE_SOA=1 \
      -DQMC_MIXED_PRECISION=1 ..
make -j 8
```

To build the MPI-CUDA version, typical steps are:

```
cd build
cmake -DCMAKE_C_COMPILER=mpicc \
      -DCMAKE_CXX_COMPILER=mpicxx \
      -DQMC_CUDA=1 ..
make -j 8
```

We require the OpenMP version to be built specifying QMC_MIXED_PRECISION=1, which enables mixed single/double precision numerics, consistent with the defaults for the CUDA GPU build.

ENABLE_SOA=1 enables "structure of array" data layout optimizations that significantly improves performance on modern CPU architectures. The CUDA implementation already includes these optimizations.

QMC_CUDA=1 enables CUDA builds. The CUDA version has a fixed ratio of 1 MPI task per GPU. QMCPACK can address multiple GPUs per node with multiple MPI tasks.

Known build problem with CUDA builds (only): on some systems, cmake must be run twice before make using same configure options.. Symptom: rapid make failure due to type definition problems. Solution: cmake -DQMC_CUDA=1 ..; cmake -DQMC_CUDA=1 ..; make

To assist characterizing single node performance, QMCPACK can be built without MPI, specifying QMC_MPI=0. The time per step is relatively constant but varies a little depending on the (random) moves taken. The benchmark includes multiple steps to obtain a more reliable average.

QMCPACK includes unit and integration tests accessible through the "ctest" system as described in

https://docs.qmcpack.org/qmcpack_manual.pdf . These use 16 CPU cores or a single GPU. e.g. "ctest –R short" runs the short integration tests. While the integration tests are statistical and may occasionally fail, the majority of the tests (>90%) should pass for a valid installation.

The CORAL2 benchmark is similar to the NiO performance tests integrated with ctest and included with the current QMCPACK distribution.

Running qmpack

For GPU runs:

```
#Single MPI task, assume 1 GPU per task
mpirun -n 1 ./qmcpack NiO-example.in.xml >out_1
```

If the example input is not changed, increasing the MPI task count will weak scale the run:

```
# Does 1024x work than single MPI
mpirun -n 1024 ./qmcpack NiO-example.in.xml >out_1024
```

For CPU runs, use a thread count that is a factor of the walker count:

```
grep walkers NiO-thread-example.in.xml
    <parameter name="walkers">32</parameter>
export OMP_NUM_THREADS=32
mpirun -n 1024 ./qmcpack NiO-thread-example.in.xml
# Gives 1024*32 total walkers.
# Walkers is specified per MPI task
```

Mechanics of Running Benchmark

The benchmark run consists of an initial short variational Monte Carlo run to prepare walkers for several “blocks” of diffusion Monte Carlo. Each block consists of a number of steps. The performance in the diffusion Monte Carlo sections is used to estimate a rate of work which is then used to construct the

figure of merit, as described below.

In the benchmark the number of blocks is fixed at 2. A minimum of 16 steps per block is required. The number of walkers per MPI task and the number of steps may be adjusted to optimize performance and to help extrapolate to the proposed system architecture and configuration. E.g. To align the walker count per MPI task with CPU or GPU compute cores or to balance NUMA domains etc.. Multicore CPU runs using OpenMP parallelization typically use one walker per thread or hyperthread. A negligible amount of I/O is performed at the end of each step and block.

To set the number of walkers ***per MPI task***, edit the "walkers" line in the input XML. The reference runs were computed with 14 walkers per GPU, equal to the number of SMs on K20X GPUs. Greater values may potentially be used to obtain higher performance, within memory constraints. For CPU runs, the number of walkers per MPI task is typically the number of OpenMP threads, or a multiple if efficient hyperthreading is available.

The figure of merit is based on the time taken to obtain 819200000=5*163840000 total samples (steps) after allowing 250 equilibration steps for each walker considered, computed relative to a 5x smaller reference run with 163840000 samples after 250 equilibration steps per walker, as measured on 18000 K20X GPUs on Titan at OLCF. The figure of merit therefore includes an element of weak scaling reflecting some of the improved accuracy desired in future.

For example, on a hypothetical system with 18000 nodes each with 80 walkers, each walker would have to perform 819200000/(18000*80)=569 production steps after equilibration or 819 total steps. Using the time per step

computed from the benchmark, the time to complete these 819 steps per node to achieve 819200000 total samples can be projected.

A commented awk script with the reference timings to compute the FOM is included. The projected FOM should factor the expected scalability of the application to the chosen node count. It can be computed by extrapolating timings from measured or estimated performance on lower node counts to the expected node count. e.g. a projected 90% parallel scalability would scale the FOM by a factor 0.9. Thus, a hypothetical machine with at least 6x more nodes each 10x more powerful than Titan would likely exceed 50x for the figure of merit.

Reference input and outputs are included for the provided 18000 node baseline, and for a range of node counts down to 1 node. These indicate currently achievable scalability and runtimes.

The 18000 node baseline takes an average 19.65 seconds per step, with 14 walkers per GPU and one GPU per node. Execution time was ~15 minutes. 163840000 samples therefore requires 650 steps per walker after equilibration or 900 steps total. The baseline used to compute the FOM is therefore 19.65*(250+(163840000/(18000*14)))=17685 seconds. **The baseline FOM is therefore 9262.1 samples/second.**

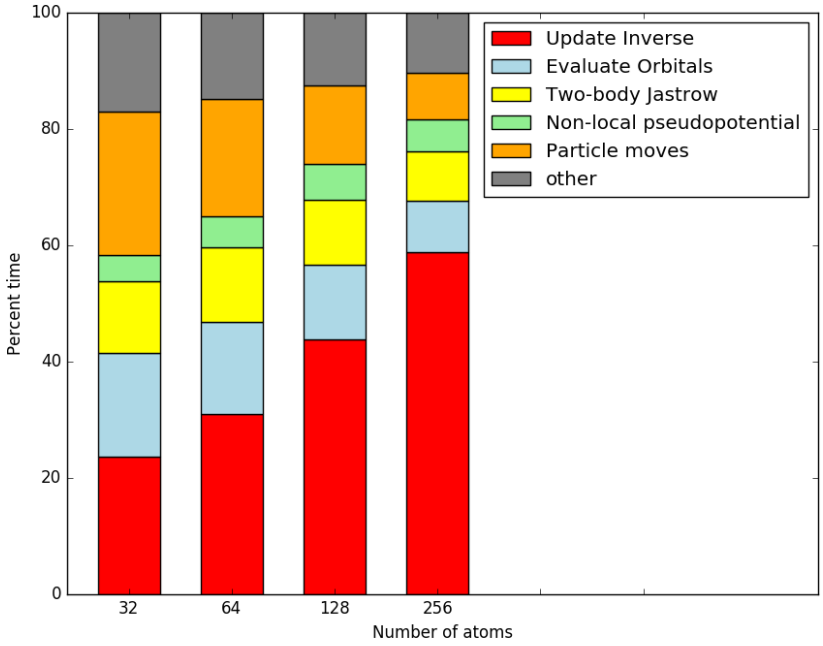
For the provided baseline note that the inputs downsample a large read only lookup array by specifying meshfactor =0.7 in the input XML to fit in the 6GB GPU memory. **For actual benchmarks the production meshfactor=0.9 should be used**, consistent with the provided NiO-example.in.xml and the desired increased accuracy on future hardware. This requires a

minimum ~12GB memory with 16 walkers per node.

Verification of Results

Due to the short nature of the benchmark runs they do not become fully equilibrated or reach the target error of production quality calculations. To verify the results, the average energies and statistical errors can be computed by postprocessing the *.scalar.dat using the qmca tool included with QMCPACK in nexus/executables. Reference energies are included with the titan reference files and README. Large deviations from the reference data may indicate e.g. numerical errors due to invalid compiler optimizations.

Example profile



Fewer than 10 kernels contribute to the majority of the runtime. The 256 atom entry in above profile was produced for a similar run to the CORAL2 benchmark on NVIDIA Kepler class GPUs. The inverse update usually takes a greater proportion of runtime on lower memory bandwidth architectures.

Change History

v20180112

Benchmark and figure of merit updated to include an element of weak scaling, with 5x more work specified. compute_fom script and documentation updated.

Updated to qmcpack v3.4.0. Fixed git-rev issue when building from tarball.

Shortened runs used to obtain baseline data.

Baseline inputs and outputs from 1-18000 nodes added.

Expanded cmake, toolchain, and build notes.

v20171206

Initial draft