LULESH

Summary Version
2.0

Purpose of Benchmark

LULESH is a shock hydro mini-app. While designed to test many machine and hardware features in particular it stresses compiler vectorization, OpenMP overheads and on node parallelism.

Characteristics of Benchmark

LULESH performs a hydrodynamics stencil calculation using both MPI and OpenMP to achieve parallelism. In general the compute performance properties of LULESH are more interesting than messaging as on a typical modern machine only about 10% of the runtime is spent in communication. Also, LULESH has been shown to scale with less than 10% performance degradation from 4 nodes to the entire Sequoia (BlueGene/Q) system.

Mechanics of Building Benchmark

LULESH can be built with any combination of OpenMP and MPI from a single source using the provided Makefile. To turn on MPI use an MPI compiler and the `-DUSE_MPI` flag. To turn on OpenMP use the appropriate flag for your compiler, for example `-fopenmp` for g++.

To build with your compiler and flags of choice edit the `MPICXX` or `SERCXX` values appropriately along with the `CXXFLAGS` and `LDFLAGS`. Then type `make` and you should get a built executable named `lulesh2.0`.

For a benchmarking, it is not necessary to include silo or hdf5 as these are only needed for data visualization and parallel I/O testing – neither of which are exercised in the default FOM.

Mechanics of Running Benchmark

There is one restriction in running LULESH. The number of domains, which is equal to the number of MPI tasks, must always be the cube of an integer.

We also suggest using about 25,000 to 30,000 elements per GB of memory. The number of MPI tasks and the `-s <size>` command line parameters control the

problem size for a single MPI task, or domain. The total aggregate number of elements can be computed by MPI tasks * <size>^{3}.

Other than the `-s` command line, there are other two options you may find useful during testing. These are the `-i` option to limit the number of iterations, which can be particularly helpful in reducing runtime when testing large problems and/or using simulators or emulators. And the `-p` option which prints out regular timestepping progress to stdout.

Example command line inputs for:

1. Small problem:
    a. For a single CPU run: `./lulesh2.0`
    b. For a single node with 16 cores and 16 GB of memory an example run would be: `mpirun <with 8 tasks> ./lulesh2.0 -s 38`
2. Medium problem: (<1K node) job
    a. For a 512 node system with 16 cores per node and 32 GB of memory per node an example run would be: `mpirun <4096 tasks> ./lulesh2.0 -s 48`
3. Large Sequoia problem:
    a. Throughput: The job to get the figure of merit was run as follows: `mpirun <32768 tasks> ./lulesh2.0 -s 38`
4. CORAL class problem:
    a. There are multiple ways to meet the CORAL throughput requirement for weak scaling by adjusting the benchmark job below are a few examples:
        i. `mpirun <32768 tasks> ./lulesh2.0 -s 48`
        ii. `mpirun <64000 tasks> ./lulesh2.0 -s 38`
        iii. `mpirun <125000 tasks> ./lulesh2.0 -s 31`

Verification of Results

The FOM is output as the last line of the simulation. This is the number that should be reported.

To verify the benchmark results are correct various values are output at the end. For a problem with the same configuration the final origin energies should be the same to the number of digits printed. Also, the iteration count should be the same. Finally the symmetry values for the final origin energy should all be less than 10^{-8} for small problem sizes <size> * (MPIRanks)$^{1/3}$ < 100.