EEL 6763 Parallel Computer Architecture
FINAL PROJECT REPORT

ANALYSIS, PROFILING AND IMPROVISATION OF THE LULESH PARALLEL CPU MODELS

Project Overview

LULESH is a mini application that simulates the behaviour of a shockwave propagating through a solid material. It is designed to be used as a benchmark for testing the performance of high-performance computing systems, particularly those with hybrid CPU-GPU architectures.

A brief overview of how LULESH works:

1. It represents the solid material as a collection of finite elements, each of which has a set of attributes such as position, velocity, stress, and strain.
2. The shockwave is modelled as a sudden increase in pressure on one end of the material, which propagates through the material at a constant speed.
3. LULESH solves a set of partial differential equations that describe the behavior of the material under the influence of the shockwave. These equations are solved using a finite element method, which involves discretizing the material into a grid of elements and solving the equations for each element.
4. It uses a time-stepping algorithm to advance the simulation from one time step to the next. At each time step, the positions, velocities, stresses, and strains of the elements are updated based on the results of the previous time step.
5. It also includes several performance optimizations that are designed to take advantage of the parallelism provided by modern high-performance computing systems. These optimizations include using OpenMP, MPI, and GPU acceleration to distribute the computation across multiple cores, nodes, and GPUs.

Approach

Source code downloaded from: https://github.com/LLNL/LULESH
What code is available? **Serial, MPI, OMP, MPI+OMP and CUDA**
What code are porting to HiPerGator for study/modification? **All models**
What are your proposed tasks?

- Perform analysis on all parallel CPU models.
- Profile the parallel models on HiperGator.
- Reproduce performance results collected from publications related to parallelism using LULESH.

Any value-added tasks?

- Modify programming scripts to improve performance (vary MPI calls/OpenMP constructs).
- Perform design-space exploration (vary parameters in SLURM script).

How is LULESH parallelized?

- The code is parallelized in a way that each processor works on a subset of the mesh, and the processors communicate with each other to exchange data for nodes that are shared between the subsets.
- The parallelization is based on the domain decomposition (data parallel) technique.
- It uses a dynamic load balancing technique based on the idea of threshold-based load balancing. The basic idea is to divide the mesh into smaller subdomains and assign each subdomain to a different processor.
- However, the size of each subdomain can be dynamically adjusted during the simulation to balance the workload across the processors.

Porting LULESH to HiPerGator

LULESH source code is written in a way that by modifying the Makefile, we can run all four models, i.e., Serial, MPI, OMP and MPI+OMP on any parallel computing system.

To run LULESH in serial mode, make following changes to Makefile:

1. CXX = SERCXX
2. Use default flags for a serial build.

In the same terminal, enter command **make all**
Then run binary file using command **./lulesh2.0**

To run LULESH in MPI only mode, make following changes to Makefile:

1. Change to MPICXX = mpic++ -DUSE_MPI=1 (use mpic++ instead of mpig++)
2. CXX = $(MPICXX)
3. Use default flags for a serial build.

In the same terminal, enter command: **module load intel openmpi**
Enter command: **make all**
Create a shell script to run MPI code and run using sbatch command.

To run LULESH in OpenMP only mode:

1. change WITH_MPI to default FALSE in CMakeLists.txt file in main directory
2. mkdir build -> cd build
3. **module load intel openmpi cmake**
4. Enter command: **cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=g++ -DWITH_MPI=Off ..**
5. **make all**
6. Create a shell script to run OpenMP only code using sbatch command.

To run LULESH in Hybrid mode, make following changes to Makefile:

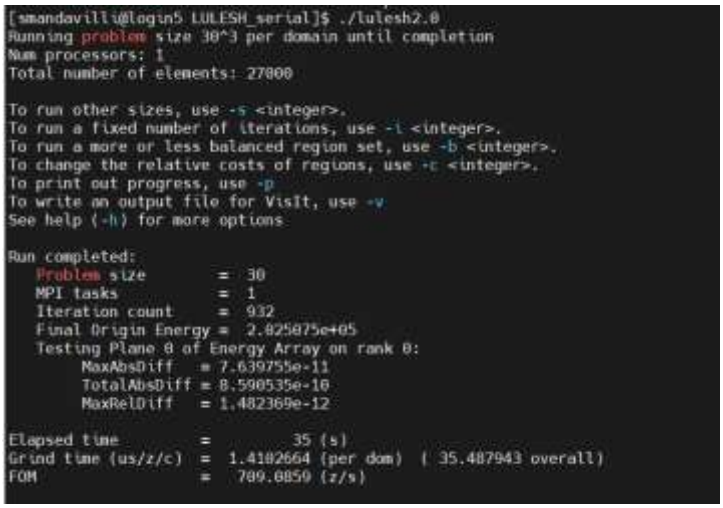
1. change MPICXX=mpic++
2. change -fopenmp flag to -qopenmp

In the same terminal, enter command: **module load intel openmpi**

Enter command: **make all**

Create a shell script to run MPI+OpenMP code using sbatch command.

Sample serial output –



In LULESH's output, there are several performance metrics that are reported. Here's what each of these metrics means:

1. **MaxAbsDiff**: This is the maximum absolute difference between the current and previous timestep's nodal positions. It measures the maximum displacement of any node in the simulation and is a measure of the accuracy of the simulation.
2. **TotalAbsDiff**: This is the total absolute difference between the current and previous timestep's nodal positions. It measures the overall displacement of all the nodes in the simulation and is another measure of the accuracy of the simulation.
3. **MaxRelDiff**: This is the maximum relative difference between the current and previous timestep's nodal positions. It measures the maximum percentage change in nodal position from one timestep to the next and is another measure of the accuracy of the simulation.
4. **Elapsed time**: This is the total time taken to run the simulation, including all initialization and cleanup steps.
5. **Grind time**: This is the time taken to perform the actual computation, i.e., the time taken to advance the simulation from one timestep to the next.
6. **FOM**: The figure of merit (FOM) is a performance metric that combines the accuracy and speed of the simulation. It is calculated as the reciprocal of the product of the simulation time and the maximum absolute difference, and it provides a measure of the simulation's accuracy per unit time.
7. **Final origin energy**: This is the total energy of the system at the end of the simulation. It is a measure of the stability of the simulation and can be used to check if the simulation has reached a steady state.

To build CUDA code on Hipergator:

- Run command: **srun --account=eel6763 --qos=eel6763 -p gpu --gpus=1 --time=03:00:00 --pty -u bash -i**
- Run command: **module load ufrc class/eel6763 cuda intel openmpi.** Check if driver is detected.

Many changes were made to makefile. It looks as follows:

```
MPI_FLAGS = -DUSE_MPI -I/opt/local/include/openmpi

NVCC            = nvcc
FLAGS           = -arch=sm_35 -std=c++11
DFLAGS  = $(MPI_FLAGS) -lineinfo
RFLAGS  = $(MPI_FLAGS) -O3 -DNDEBUG

LINKFLAGS = -lm -L/opt/local/include/openmpi/lib

all: release

debug: LINKFLAGS +=

release:        FLAGS += $(RFLAGS)
debug:          FLAGS += $(DFLAGS)

release: lulesh
debug: lulesh
```

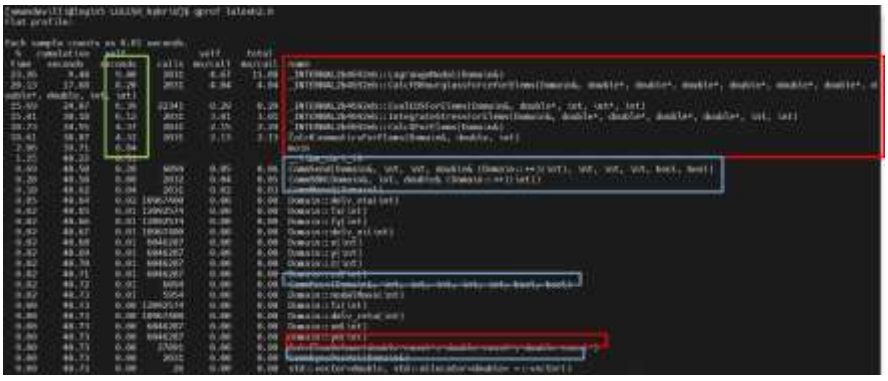
- Build object files individually:
  - **make allocator.o**
  - **make lulesh.o**
  - **make lulesh-comms.o**
  - **make lulesh-comms-gpu.o**
  - **make lulesh**

Run the binary file with the command: **./lulesh -s 45**

Profiling with gprof

Gprof gives us information about time consumed by each function, number of times it was called and the call tree of a program.

Given below is the gprof output for LULESH hybrid code. We will compare it with the serial implementation to make key observations.



- Red box indicates functions that compute physical metrics.
- Blue box indicates functions for communication (MPI).
- Notice that communication functions do not consume significant time but be aware that computation functions utilize OMP to divide work.

Serial code profile using gprof:

```
[smandavilli@login5 LULESH_serial]$ gprof lulesh2.0
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 61.71     11.47    11.47                             CalcHourglassControlForElems(Domain&, double*, double)
 17.75     14.77     3.30                             CalcKinematicsForElems(Domain&, double, int)
 15.87     17.72     2.95                             CalcQForElems(Domain&)
  4.68     18.59     0.87 5191000    34.54    34.54  CalcElemVolume(double const*, double const*, double const*)
  0.00     18.59     0.00       11     0.00     0.00  std::vector<int, std::allocator<int> >::_M_fill_insert(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::
allocator<int> > >, unsigned long, int const&)
  0.00     18.59     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14CalcElemVolumePKdS0_S0_
  0.00     18.59     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z23ParseCommandLineOptionsiPPcR11cmdLineOpts
  0.00     18.59     0.00        1     0.00     0.00  Domain::SetupCommBuffers(int)
  0.00     18.59     0.00        1     0.00     0.00  Domain::CreateRegionIndexSets(int, int)
  0.00     18.59     0.00        1     0.00     0.00  Domain::AllocateElemPersistent(int)
  0.00     18.59     0.00        1     0.00     0.00  Domain::SetupBoundaryConditions(int)
  0.00     18.59     0.00        1     0.00     0.00  Domain::SetupElementConnectivities(int)
  0.00     18.59     0.00        1     0.00     0.00  Domain::BuildMesh(int, int, int)
```

- Compare the time consumed by the computation functions between the current and previous slide. In serial code, these functions consume less time because there is no OMP in the for loop and no MPI between sub-domains of the mesh.
- The same behavior is observed for all the problem sizes we tested but as it increases, hybrid code becomes more efficient than serial because of its parallelism.

Profiling CUDA with nvprof

Nvidia provides a cmd profiler to summarize the performance details of a GPU execution.

```
[smandavilli@c0309a-s13 src]$ nvprof ./lulesh -s 45
==80219== NVPROF is profiling process 80219, command: ./lulesh -s 45
==80219== Profiling application: ./lulesh -s 45
==80219== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   38.59%  417.02ms      1477  417.75us  403.58us  576.67us  void CalcVolumeForceForElems_kernel<bool=1>(double const *, double const *, double const *, double const
                   28.02%  460.94ms      1477  312.08us  301.69us  423.55us  CalcKinematicsAndMonotonicQGradient_kernel(int, int, double, int const *, double const *, double const *,
                   16.76%  288.06ms      1477  181.49us  164.61us  230.78us  ApplyMaterialPropertiesAndUpdateVolume_kernel(int, double, double, double, double*, double*, double*, dou
                    5.34%  85.462ms      1477  57.862us  56.511us  61.119us  AddNodeForcesFromElems_kernel(int, int, int const *, int const *, int const *, double const *, double con
                    4.08%  65.276ms      1477  44.195us  42.528us  59.807us  CalcMonotonicQRegionForElems_kernel(double, double, double, double, double, int, int*, int*, int*, int*,
                    2.31%  36.899ms      1477  24.976us  22.843us  31.584us  void CalcTimeConstraintsForElems_kernel<int=128>(int, double, double, int*, double*, double*, double*, do
                    1.81%  29.013ms      1477  19.643us  18.464us  21.376us  CalcPositionAndVelocityForNodes_kernel(int, double, double, double, double*, double*, double*, double*, double*,
                    0.92%  14.723ms      1477  9.9680us  8.2240us  11.840us  CalcAccelerationForNodes_kernel(int, double*, double*, double*, double*, double*, double*, double*)
                    0.69%  11.022ms      4431  2.4870us  2.1110us  4.1590us  ApplyAccelerationBoundaryConditionsForNodes_kernel(int, double*, int*)
                    0.55%  8.8177ms      1477  5.9700us  5.7590us  8.0960us  void CalcMinDtOneBlock<int=1024>(double*, double*, double*, double*, int)
                    0.10%  1.6452ms        27  60.934us  1.1210us  428.19us  [CUDA memcpy HtoD]
                    0.01%  155.65us        40  3.8910us  2.6560us  13.920us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cud
                    0.01%  87.998us        17  2.0230us  2.3840us  5.4080us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cud
                    0.01%  36.992us        12  3.0820us  2.8800us  3.2960us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cud
```

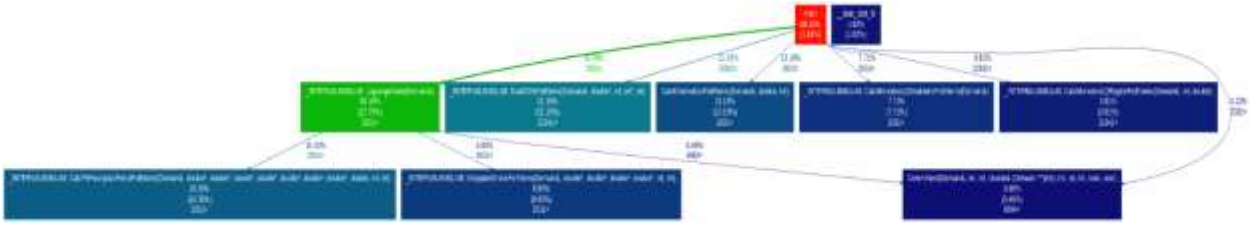
- Like gprof, it provides details on %time consumed by a function and number of times it is called.
- Observe that most of the functions under GPU activities are for computation.

- CUDA synchronization functions consume over 95% of the device's time. This signifies that in LULESH, the device was mainly responsible for setup, data transfer and kernel launch whereas the GPU performed computation.
- There are only 27 memcpy HtoD and 1 memcpy DtoH calls which specifies that all computation was processed and stored on the device memory.

Creating call graph with gprof2dot

Gprof2dot converts profiling output to a call graph, making it easier to analyze. It gives us information about the number of times a function is called, connection between functions and % of total time taken by that function.

MPI only call graph:



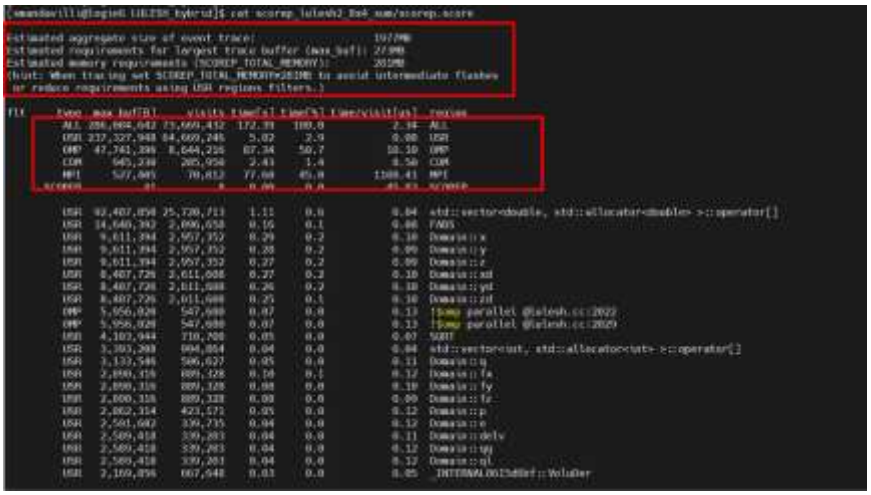
Generating call graph for mpi only code:

- Create a venv and install gprof2dot python module in it. Activate the environment.
- Add -pg flag in makefile. Compile and run the shell script.
- Run the command: **gprof lulesh2.0 | gprof2dot | dot -Tpng -o call_graph_mpi.png**

Profiling with ScoreP and Scalasca

ScoreP performs run-time profiling and provides data about program execution time and memory consumption. It is supported by various analysis tools, here we utilize Scalasca.

ScoreP profile of hybrid code:



- It provides information about total memory consumed by the application.
- It also provides a detailed split-up of the memory consumed by different components in the code (here, user, omp, mpi etc.).
- Compared to gprof, it provides a more detailed profile. For example, it also includes how much time and memory is consumed by the SQRT function.

Modifying code to improve/worsen performance

1. **Use vectorization**: Modern CPUs have vectorized instructions that can perform multiple calculations at once, which can significantly improve performance on some operations.

To enable vectorization in LULESH, add "-march=native" to the CFLAGS variable in the makefile. This will allow the compiler to generate code that is optimized for the specific CPU architecture on which the code is being run.

2. **Use fused multiply-add (FMA) instructions**: FMA instructions can perform multiplication and addition in a single operation, which can improve performance on some processors. To enable FMA instructions in LULESH, add "-mfma" to the CFLAGS variable in the makefile.

CPU Model	Time taken (s)		Memory consumption	
	Vectorization	FMA (compared to left column)	Vectorization	FMA
Hybrid	Reduced from 142.47 to 106.07	Increased back to 132.72	No change	No change
Serial	Reduced from 35.48 to 29.03	Slightly increased to 29.12	No change	No change
OMP only	Reduced from 54.91 to 31.03	Slightly increased to 31.85	No change	No change
MPI only	Reduced from 10.13 to 8.82	Slight reduced to 8.10	No change	No change

LULESH already makes use of SIMD (Single Instruction, Multiple Data) vector instructions to achieve better performance. For example, in the calculation of the rate of deformation tensor, the code is designed to process multiple elements of the tensor at once using vector instructions. Hence, FMA did not have any effect on the performance.

3. Modifying "#pragma omp parallel for" schedule from static to dynamic in lulesh.cc

Hybrid: Time consumption significantly increased, and the job did not complete in over 10 minutes.

OMP: Time consumption significantly increased, and the job did not complete in over 5 minutes.

4. Modifying "#pragma omp parallel for" by adding the ordered clause in lulesh.cc

Hybrid: Time consumption increased from 142.47 to 247.14561s.

OMP: Time consumption increased from 31.03 to 33.23s. Memory consumption also increased from 103MB to 120MB.

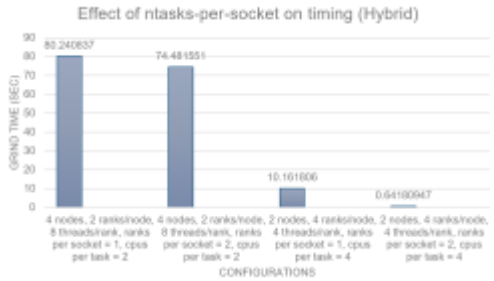
5. -Os: This flag instructs the compiler to optimize for code size rather than performance, which can reduce memory usage by producing more compact code.

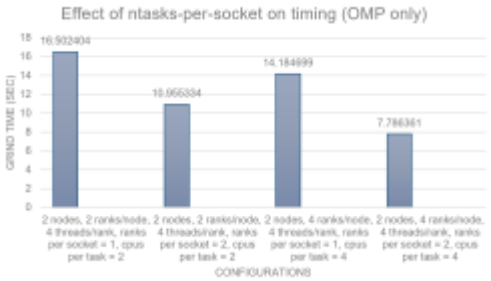
No change in memory consumption for all models.

Design Space exploration

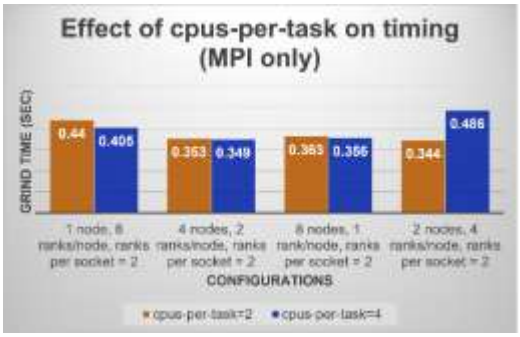
Objective of DSE is to study the effect on performance by varying configuration (SLURM) parameters and choosing the optimal solution for an application (here LULESH).

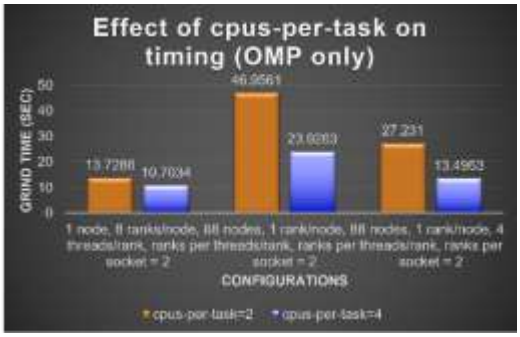
- From two different scenarios (Hybrid and OMP only) where we vary only the “ntasks-per-socket” parameter between 1 and 2, the time decreases for ntasks-per-socket=2.





- For different configurations where we vary *“cpus-per-task”* between 2 and 4, we observe that in 3/4 scenarios for MPI only and in all scenarios for OMP only, time taken decreases when we increase cpus-per-task.

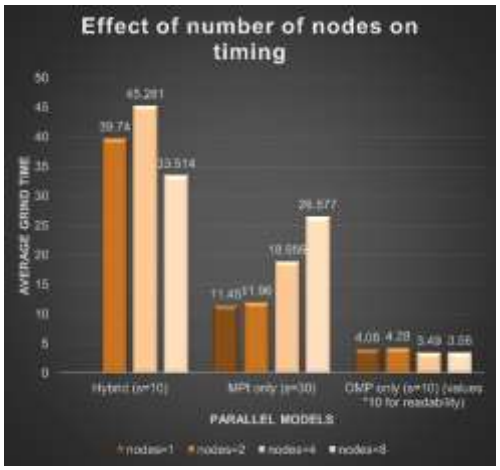




- Further, we observe that the *“number of nodes”* has an impact on timing as well.

We consider the weighted average for different values of –nodes = 1, 2, 4 and 8. Note that the hybrid model throws an error if we run it on 1 node.

Model	--nodes for best performance
Hybrid	8
MPI only	2
OMP only	4 or 8



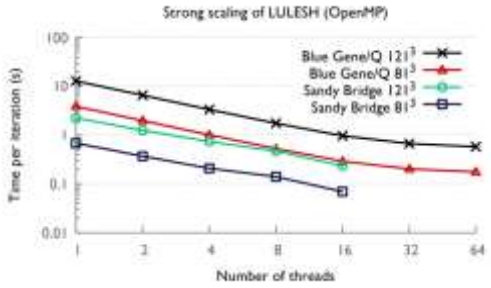
- Overall, LULESH offers better performance with larger node values (for huge problem size) because the application is designed to overlap communication with computation.

<u>Supporting research paper findings</u>

Here, we look at research papers to get data related to parallelism and performance using LULESH and run tests on HiPerGator to validate their findings.

Source: https://dl.acm.org/doi/abs/10.1145/3149412.3149416

1. Inference: Graph shows that the OpenMP variant exhibits good strong-scaling behavior, indicating that there is ample parallelism to be harvested in the code.

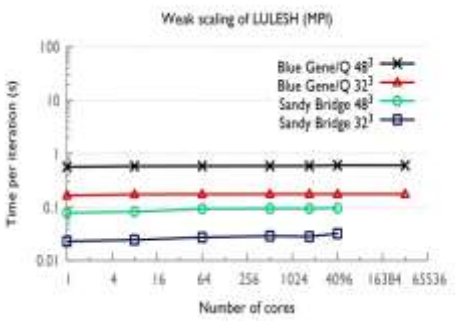


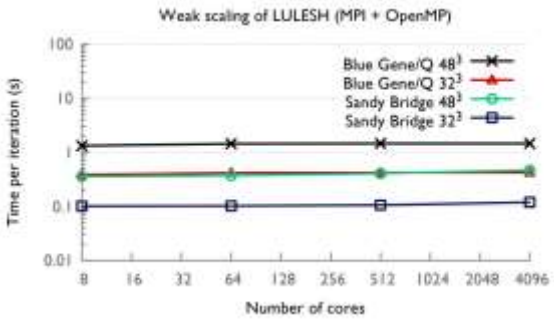
Our results: s=10, configuration = 2 nodes, 2 ranks/node, cpus per task = 2

Number of threads (OMP_NUM_THREADS)	Grind time (s)
1	16.375
2	14.902
4	12.639
8	35.121
16	57.052

As mentioned in the publication, OMP decreases time as the number of threads increase from 1 to 4, exhibiting strong scaling. Time increases again as we further increase threads.

2. Inference:
    a. The MPI version of the code exhibits near perfect weak scaling of the serial code, as there is not much communication and most of what is present is local.
    b. The hybrid version displays similar scaling characteristics, but significantly worse performance. Also, different experiments on both machines show that for low processor counts, the OpenMP overhead is higher than the MPI overhead and can cause hybrid codes to run slower.





Our results:

- MPI: --nodes=2, --ntasks-per-node=4, --cpus-per-task=2
- Hybrid: --nodes=2, --ntasks-per-node=4, --cpus-per-task=4, # threads=4

# of elements (problem size)	Time/# of elements (s)	
	MPI	Hybrid
8 (1)	0.0000241	0.00877
64 (2)	0.0000258	0.0101
216 (3)	0.0000253	0.00732
512 (4)	0.0000243	0.00484
1000 (5)	0.0000238	0.00299
1728 (6)	0.0000297	0.00252
4096 (8)	0.0000331	0.00174

As mentioned in the publication, MPI exhibits constant weak scaling. Also, hybrid code runs significantly slower than MPI. However, hybrid does not respond well to weak scaling as compared to MPI. For example, it increases time from 0.0877 (1) to 0.00252 (6).

Source:
https://www.researchgate.net/publication/301461578_Performance_analysis_of_OpenMP_on_a_GPU_using_a_CORAL_proxy_application

3. Statement: Table shows a performance comparison of LULESH OpenMP, CUDA and Liszt GPU variants. In this test the goal was to grow the problem size order to explore the performance characteristics of the OpenMP and CUDA versions, which share many similarities in terms of the style of coding. The CPU code was about a factor of two slower than the original CUDA code.

Version	45^3	55^3	65^3	75^3	85^3	96^3
CUDA	0.008	0.014	0.023	0.035	0.052	0.069
Liszt	0.016	0.029	0.047	0.071	0.103	0.147
OpenMP	0.017	0.032	0.053	0.086	0.128	0.182

Our results:

- OMP: 2 nodes, 2 ranks/node, 4 threads/rank, ranks per socket = 2, cpus per task = 2
- CUDA: 1 GPU (with index 0)

Version	45	55	65	75	85	96
CUDA	1.60	3.38	6.41	11.25	18.67	30.42
OMP	33.356	79.441	155.185	282.723	Timeout	Timeout

Unlike the publication which mentions the factor between CUDA and OMP is two, our results show that OMP is ~25 times slower than CUDA.

References

- https://asc.llnl.gov/codes/proxy-apps/lulesh
- https://www.researchgate.net/publication/320742167_Performance_and_Power_Characteristics_and_Optimizations_of_Hybrid_MPIOpenMP_LULESH_Miniapps_under_Various_Workloads
- https://openbenchmarking.org/test/pts/lulesh
- https://dl.acm.org/doi/abs/10.1145/3149412.3149416
- https://github.com/LLNL/LULESH