Analysis, Profiling and Improvisation of the LULESH Parallel CPU Models

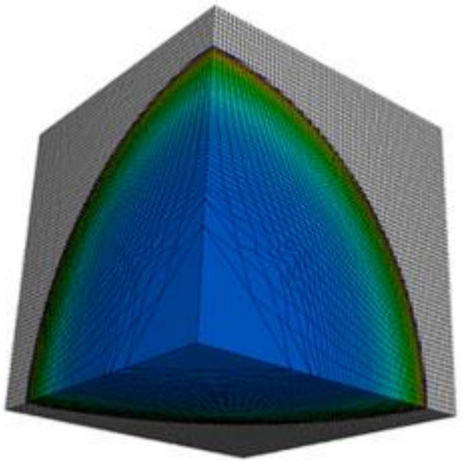
Project Overview

- LULESH is a mini-application that simulates the behavior 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 modeled as a sudden increase in pressure on one end of the material, which propagates through the material at a constant speed.



Shock wave propagating through a solid material

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).
- Describe any changes from your proposal: No changes

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.

- *CalcElemVolume* computes the "work fraction" used later in the load balancing algorithm to determine which processors are over- or under-loaded. It is part of the "Element-based computation" section of the code, which is responsible for calculating the volume, position, and velocity of each element in the mesh.

```
#pragma omp parallel for firstprivate(nodelist) private(dvdx,dvdy,dvdz) reduction(+:volo)
 for (Index_t ielem=0 ; ielem<domElemList.size() ; ++ielem) {
  //... compute volume of element ielem and add to volo ...
 }


 // ... compute total volume of mesh and broadcast to all processors ...


 // ... compute work fraction of each processor based on volumes of elements ...
```

- The function uses OpenMP directives to parallelize the computation of element volumes across multiple threads. The firstprivate and private clauses ensure that each thread has its own copy of the *nodelist, dvdx, dvdy*, and *dvdz* variables, while the reduction clause aggregates the *volo* variable across all threads.

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.
- Serial output:

Serial code uses only 1 processor

```
[smandavilli@login5 LULESH_serial]$ ./lulesh2.0
Running problem size 30^3 per domain until completion
Num processors: 1
Total number of elements: 27000

To run other sizes, use -s <integer>.
To run a fixed number of iterations, use -i <integer>.
To run a more or less balanced region set, use -b <integer>.
To change the relative costs of regions, use -c <integer>.
To print out progress, use -p
To write an output file for VisIt, use -v
See help (-h) for more options

Run completed:
   Problem size        =  30
   MPI tasks           =  1
   Iteration count     =  932
   Final Origin Energy =  2.025075e+05
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 7.639755e-11
        TotalAbsDiff = 8.590535e-10
        MaxRelDiff   = 1.482369e-12

Elapsed time         =         35 (s)
Grind time (us/z/c)  =  1.4102664 (per dom)  ( 35.487943 overall)
FOM                  =   709.0859 (z/s)
```

- MPI only output and its configuration (SLURM) script:

```
Running problem size 30^3 per domain until completion
Num processors: 8
Total number of elements: 216000

To run other sizes, use -s <integer>.
To run a fixed number of iterations, use -i <integer>.
To run a more or less balanced region set, use -b <integer>.
To change the relative costs of regions, use -c <integer>.
To print out progress, use -p
To write an output file for VisIt, use -v
See help (-h) for more options

Run completed:
   Problem size        =  30
   MPI tasks           =  8
   Iteration count     =  2031
   Final Origin Energy =  7.130703e+05
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 3.492460e-10
        TotalAbsDiff = 5.379491e-09
        MaxRelDiff   = 2.745918e-14

Elapsed time         =         55 (s)
Grind time (us/z/c)  =  1.0013376 (per dom)  ( 54.910348 overall)
FOM                  =  7989.3138 (z/s)
```

```
#!/bin/bash
#SBATCH --job-name=LULESH_mpi
#SBATCH --account=eel6763
#SBATCH --qos=eel6763
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=2
#SBATCH --mem-per-cpu=1000mb
#SBATCH -t 00:05:00
#SBATCH -o outfile
#SBATCH -e errfile
srun --mpi=pmix_v3 ./lulesh2.0
```

Output shows number of MPI tasks used.

- OMP only output and its configuration (SLURM) script:

```
Running problem size 30^3 per domain until completion
Num processors: 1
Num threads: 8
Total number of elements: 27000

To run other sizes, use -s <integer>.
To run a fixed number of iterations, use -i <integer>.
To run a more or less balanced region set, use -b <integer>.
To change the relative costs of regions, use -c <integer>.
To print out progress, use -p
To write an output file for VisIt, use -v
See help (-h) for more options

Run completed:
   Problem size        =  30
   MPI tasks           =  1
   Iteration count     =  932
   Final Origin Energy =  2.025075e+05
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 6.548362e-11
        TotalAbsDiff = 8.615093e-10
        MaxRelDiff   = 1.461140e-12

Elapsed time         =         10 (s)
Grid time (us/z/c)   = 0.40286159 (per dom)  ( 10.137609 overall)
FOM                  =  2482.2421 (z/s)
```

```
#!/bin/bash
#SBATCH --job-name=classdemo
#SBATCH --account=eel6763
#SBATCH --qos=eel6763
#SBATCH --nodes=1   #(single node)
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1   #(single process)
#SBATCH --cpus-per-task=2   #(32 class limit)
#SBATCH --mem-per-cpu=600mb
#SBATCH -t 00:05:00
#SBATCH -o outfile
#SBATCH -e errfile
export OMP_NUM_THREADS=8
./lulesh2.0
```

Output shows number of tasks and threads used.

- Hybrid output and its configuration (SLURM) script:

```
[smandavilli@login5 LULESH_hybrid]$ cat outfile
Running problem size 30^3 per domain until completion
Num processors: 8
Num threads: 4
Total number of elements: 216000

To run other sizes, use -s <integer>.
To run a fixed number of iterations, use -i <integer>.
To run a more or less balanced region set, use -b <integer>.
To change the relative costs of regions, use -c <integer>.
To print out progress, use -p
To write an output file for VisIt, use -v
See help (-h) for more options

Run completed:
   Problem size        =  30
   MPI tasks           =  8
   Iteration count     =  2031
   Final Origin Energy =  7.130703e+05
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 6.111804e-10
        TotalAbsDiff = 5.403820e-09
        MaxRelDiff   = 3.060749e-14

Elapsed time         =    1.4e+02 (s)
Grid time (us/z/c)   =  2.5944485 (per dom)  ( 142.27177 overall)
FOM                  =   3083.507 (z/s)
```

```
#!/bin/bash
#SBATCH --account=eel6763
#SBATCH --qos=eel6763
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks-per-socket=2
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=500mb
#SBATCH -t 00:10:00
#SBATCH -o outfile
#SBATCH -e errfile
export OMP_NUM_THREADS=4
srun --mpi=pmix_v3 ./lulesh2.0
```

Output shows number of tasks and threads used.

- **srun --account=eel6763 --qos=eel6763 -p gpu --gpus=1 --time=03:00:00 --pty -u bash –i** was used to allocate a GPU.
- **module load ufrc class/eel6763 cuda intel openmpi** : command to load modules and install GPU driver.
- CUDA output using 1 GPU:

```
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
```

Makefile configuration for the CUDA build.
Notice the “nvcc” compiler here.

```
[smandavilli@c0308a-s29 src]$ ./lulesh -s 45
Host c0308a-s29.ufhpc using GPU 0: NVIDIA GeForce RTX 2080 Ti
Running until t=0.010000, Problem size=45x45x45
   Used Memory         =  224.8750 Mb
Run completed:
   Problem size        =  45
   MPI tasks           =  1
   Iteration count     =  1477
   Final Origin Energy = 4.234875e+05
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 4.220055e-10
        TotalAbsDiff = 3.314958e-09
        MaxRelDiff   = 1.824377e-12


Elapsed time         =       1.62 (s)
Grind time (us/z/c)  = 0.012049175 (per dom)  (0.012049175 overall)
FOM                  =  82993.236 (z/s)
```

Output generated on GPU with index 0.

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.

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

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 23.26      9.48     9.48     2031     4.67    11.88  _INTERNAL2b4692eb::LagrangeNodal(Domain&)
 20.13     17.68     8.20     2031     4.04     4.04  _INTERNAL2b4692eb::CalcFBHourglassForceForElems(Domain&, double*, double*, double*, double*, double*, double*, d
ouble*, double, int, int)
 15.69     24.07     6.39    22341     0.29     0.29  _INTERNAL2b4692eb::EvalEOSForElems(Domain&, double*, int, int*, int)
 15.01     30.18     6.12     2031     3.01     3.01  _INTERNAL2b4692eb::IntegrateStressForElems(Domain&, double*, double*, double*, double*, int, int)
 10.73     34.55     4.37     2031     2.15     2.24  _INTERNAL2b4692eb::CalcQForElems(Domain&)
 10.61     38.87     4.32     2031     2.13     2.13  CalcKinematicsForElems(Domain&, double, int)
  2.06     39.71     0.84                             main
  1.25     40.22     0.51                               __libm_cbrt_l9
  0.69     40.50     0.28     6094     0.05     0.06  CommSend(Domain&, int, int, double& (Domain::**)(int), int, int, int, bool, bool)
  0.20     40.58     0.08     2032     0.04     0.05  CommSBN(Domain&, int, double& (Domain::**)(int))
  0.10     40.62     0.04     2031     0.02     0.03  CommMonoQ(Domain&)
  0.05     40.64     0.02 10967400     0.00     0.00  Domain::delv_eta(int)
  0.02     40.65     0.01 12092574     0.00     0.00  Domain::fx(int)
  0.02     40.66     0.01 12092574     0.00     0.00  Domain::fy(int)
  0.02     40.67     0.01 10967400     0.00     0.00  Domain::delv_xi(int)
  0.02     40.68     0.01  6046287     0.00     0.00  Domain::x(int)
  0.02     40.69     0.01  6046287     0.00     0.00  Domain::y(int)
  0.02     40.70     0.01  6046287     0.00     0.00  Domain::z(int)
  0.02     40.71     0.01  6046287     0.00     0.00  Domain::zd(int)
  0.02     40.72     0.01     6094     0.00     0.00  CommRecv(Domain&, int, int, int, int, int, bool, bool)
  0.02     40.73     0.01     5954     0.00     0.00  Domain::nodalMass(int)
  0.00     40.73     0.00 12092574     0.00     0.00  Domain::fz(int)
  0.00     40.73     0.00 10967400     0.00     0.00  Domain::delv_zeta(int)
  0.00     40.73     0.00  6046287     0.00     0.00  Domain::xd(int)
  0.00     40.73     0.00  6046287     0.00     0.00  Domain::yd(int)
  0.00     40.73     0.00    27001     0.00     0.00  CalcElemVolume(double const*, double const*, double const*)
  0.00     40.73     0.00     2031     0.00     0.00  CommSyncPosVel(Domain&)
  0.00     40.73     0.00       26     0.00     0.00  std::vector<double, std::allocator<double> >::vector()
```

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 25191000    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__Z23ParseCommandLineOptionsiPPciP11cmdLineOpts
  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%  617.02ms      1477  417.75us  403.58us  576.67us  void CalcVolumeForceForElems_kernel<bool=1>(double const *, double const *, double const *, double const
                   28.82%  460.94ms      1477  312.08us  301.69us  423.55us  CalcKinematicsAndMonotonicQGradient_kernel(int, int, double, int const *, double const *, double const *,
                   16.76%  268.06ms      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  59.007us  CalcMonotonicQRegionForElems_kernel(double, double, double, double, double, int, int*, int*, int*, int*,
                    2.31%  36.890ms      1477  24.976us  22.943us  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*,
                    0.92%  14.723ms      1477  9.9680us  9.2160us  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
                    0.00%  47.998us        17  2.8230us  2.3040us  5.4080us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::ParallelForAgent<thrust::cud
                    0.00%  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.

- It also provides data on the various CUDA APIs from the code.

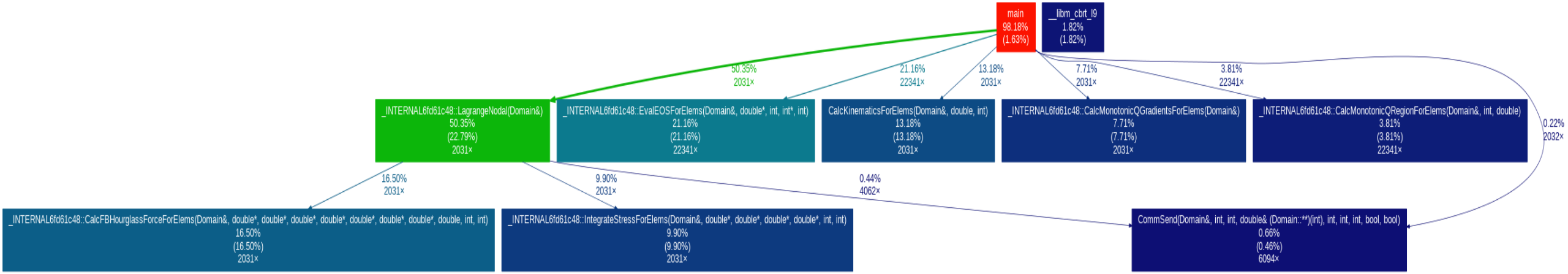
	0.55%	8.8177ms	1477	5.9700us	5.7590us	8.0960us	void CalcMinDtOneBlock<int=1024>(double*, double*, double*, double*, int)
	0.10%	1.6452ms	27	60.934us	1.1210us	428.19us	[CUDA memcpy HtoD]
	0.01%	155.65us	40	3.8910us	2.6560us	13.920us	void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::I
	0.00%	47.998us	17	2.8230us	2.3040us	5.4080us	void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::I
	0.00%	36.992us	12	3.0820us	2.8800us	3.2960us	void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__parallel_for::I
	0.00%	1.6320us	1	1.6320us	1.6320us	1.6320us	[CUDA memcpy DtoH]
API calls:	82.53%	1.62216s	1477	1.0983ms	2.1250us	34.639ms	cudaEventSynchronize
	13.28%	261.01ms	4	65.253ms	1.1210us	259.99ms	cudaDeviceSynchronize
	3.35%	65.876ms	17793	3.7020us	2.9770us	459.68us	cudaLaunchKernel
	0.21%	4.1916ms	60	69.859us	2.3870us	283.96us	cudaMalloc
	0.16%	3.0931ms	28	110.47us	3.2670us	578.34us	cudaMemcpyAsync
	0.13%	2.5277ms	97	26.058us	749ns	682.70us	cudaStreamSynchronize
	0.08%	1.6051ms	202	7.9460us	113ns	589.02us	cuDeviceGetAttribute
	0.06%	1.2610ms	2	630.48us	607.25us	653.71us	cudaGetDeviceProperties
	0.06%	1.0973ms	4	274.31us	3.1920us	1.0849ms	cudaHostAlloc
	0.04%	862.54us	1477	583ns	480ns	10.642us	cudaEventRecord
	0.03%	621.64us	1477	420ns	361ns	1.9640us	cudaFuncSetCacheConfig
	0.02%	441.20us	2	220.60us	220.39us	220.81us	cuDeviceTotalMem
	0.02%	413.04us	32	12.907us	1.0630us	189.88us	cudaStreamCreate
	0.01%	126.20us	2	63.102us	57.430us	68.774us	cuDeviceGetName
	0.00%	67.467us	586	115ns	93ns	552ns	cudaGetLastError
	0.00%	45.872us	139	330ns	225ns	1.2570us	cudaGetDevice
	0.00%	25.349us	69	367ns	253ns	2.2410us	cudaDeviceGetAttribute
	0.00%	16.761us	138	121ns	96ns	193ns	cudaPeekAtLastError
	0.00%	7.6490us	1	7.6490us	7.6490us	7.6490us	cudaProfilerStart
	0.00%	7.4110us	2	3.7050us	1.2660us	6.1450us	cuDeviceGetPCIBusId
	0.00%	5.4880us	1	5.4880us	5.4880us	5.4880us	cudaEventCreateWithFlags
	0.00%	5.2730us	1	5.2730us	5.2730us	5.2730us	cudaFuncGetAttributes
	0.00%	2.4590us	2	1.2290us	165ns	2.2940us	cudaGetDeviceCount
	0.00%	2.0490us	4	512ns	120ns	1.6530us	cuDeviceGet
	0.00%	1.7210us	1	1.7210us	1.7210us	1.7210us	cudaDeviceSetCacheConfig
	0.00%	1.3160us	1	1.3160us	1.3160us	1.3160us	cudaSetDevice
	0.00%	932ns	3	310ns	118ns	674ns	cuDeviceGetCount
	0.00%	392ns	2	196ns	174ns	218ns	cuDeviceGetUuid

CUDA synchronization functions consume over 95% of the device time. This signifies that in LULESH, device was mainly responsible for setup, data transfer and kernel launch whereas the GPU performed computation.

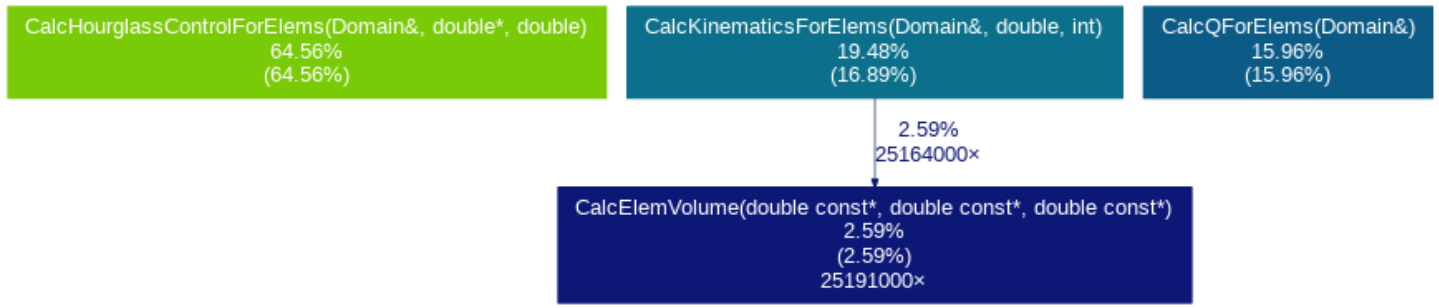
There are only 27 memcpy HtoD and 1 memcpy DtoH calls which specifies that all computation was processed and stored on the device memory.

Creating call graph using gprof2dot

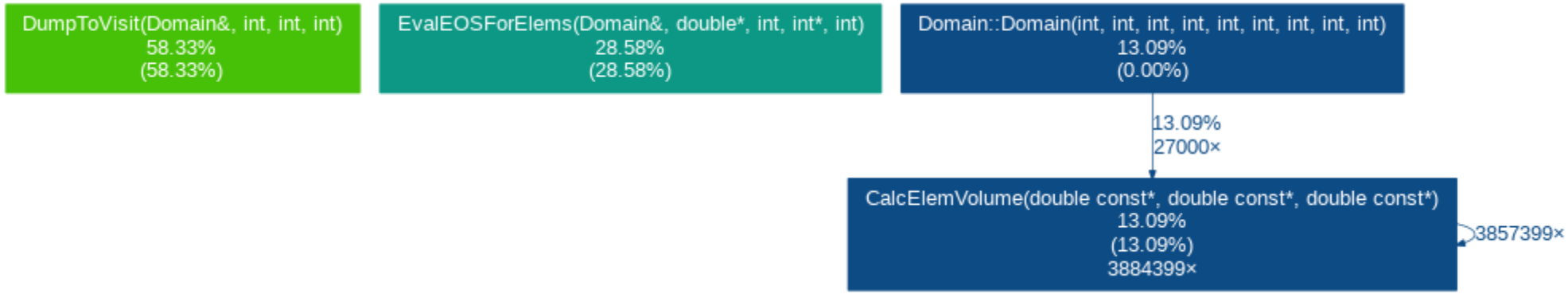
- Gprof2dot converts profiling output to a call graph, making it easier to analyze. It gives us information about number of times a function is called, connection between functions and % of total time taken by that function.
- MPI only call graph:



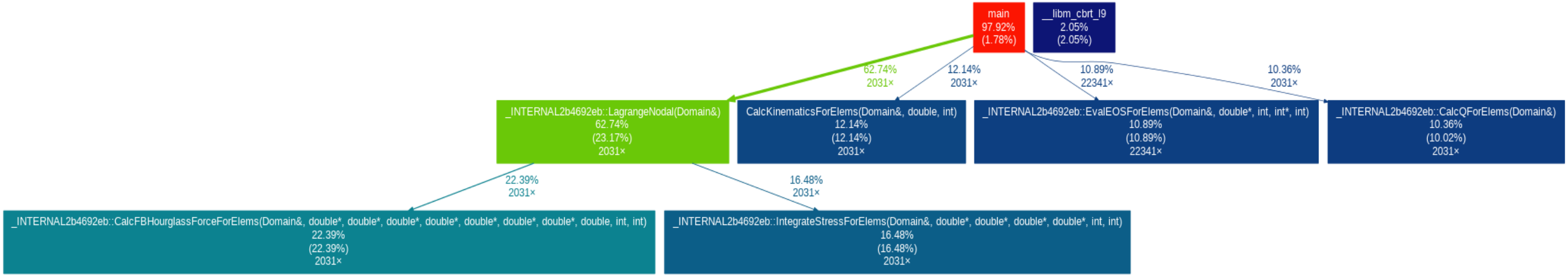
- Serial call graph:



- OMP only call graph:



- Hybrid call graph:



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:

```
[smandavilli@login6 LULESH_hybrid]$ cat scorep_lulesh2_8x4_sum/scorep.score

Estimated aggregate size of event trace:                   1977MB
Estimated requirements for largest trace buffer (max_buf): 273MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       281MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=281MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type  max_buf[B]     visits time[s] time[%] time/visit[us]  region
        ALL 286,084,642 73,669,432  172.39   100.0           2.34  ALL
        USR 237,327,948 64,669,246    5.02     2.9           0.08  USR
        OMP  47,741,396  8,644,216   87.34    50.7          10.10  OMP
        COM     945,230    285,950    2.43     1.4           8.50  COM
        MPI     527,405     70,012   77.60    45.0        1108.41  MPI
    SCOREP          41          8    0.00     0.0          45.82  SCOREP

        USR  92,487,850 25,720,713    1.11     0.6           0.04  std::vector<double, std::allocator<double> >::operator[]
        USR  14,640,392  2,096,658    0.16     0.1           0.08  FABS
        USR   9,611,394  2,957,352    0.29     0.2           0.10  Domain::x
        USR   9,611,394  2,957,352    0.28     0.2           0.09  Domain::y
        USR   9,611,394  2,957,352    0.27     0.2           0.09  Domain::z
        USR   8,487,726  2,611,608    0.27     0.2           0.10  Domain::xd
        USR   8,487,726  2,611,608    0.26     0.2           0.10  Domain::yd
        USR   8,487,726  2,611,608    0.25     0.1           0.10  Domain::zd
        OMP   5,956,020    547,680    0.07     0.0           0.13  !$omp parallel @lulesh.cc:2022
        OMP   5,956,020    547,680    0.07     0.0           0.13  !$omp parallel @lulesh.cc:2029
        USR   4,103,944    710,708    0.05     0.0           0.07  SQRT
        USR   3,393,208    994,864    0.04     0.0           0.04  std::vector<int, std::allocator<int> >::operator[]
        USR   3,133,546    506,627    0.05     0.0           0.11  Domain::q
        USR   2,890,316    889,328    0.10     0.1           0.12  Domain::fx
        USR   2,890,316    889,328    0.08     0.0           0.10  Domain::fy
        USR   2,890,316    889,328    0.08     0.0           0.09  Domain::fz
        USR   2,862,314    423,171    0.05     0.0           0.12  Domain::p
        USR   2,591,602    339,735    0.04     0.0           0.12  Domain::e
        USR   2,589,418    339,203    0.04     0.0           0.11  Domain::delv
        USR   2,589,418    339,203    0.04     0.0           0.12  Domain::qq
        USR   2,589,418    339,203    0.04     0.0           0.12  Domain::ql
        USR   2,169,856    667,648    0.03     0.0           0.05  _INTERNAL8615d6bf::VoluDer
```

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.

- Here we compare ScoreP data between MPI only (left) and OMP only (right) models.

```
[smandavilli@login6 scorep_lulesh2_8_sum]$ cat scorep.score

Estimated aggregate size of event trace:                   1747MB
Estimated requirements for largest trace buffer (max_buf): 245MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       247MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=247MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type  max_buf[B]     visits time[s] time[%] time/visit[us]  region
        ALL 256,147,356 70,364,302   25.27   100.0           0.36  ALL
        USR 255,556,184 70,274,674    5.18    20.5           0.07  USR
        MPI     527,405     70,012   19.96    79.0         285.09  MPI
        COM      63,726     19,608    0.13     0.5           6.48  COM
    SCOREP          41          8    0.00     0.0          19.91  SCOREP

        USR 102,328,486 28,748,601    0.97     3.8           0.03  std::vector<double, std::allocator<double> >::operator[]
        USR  14,640,392  2,096,658    0.08     0.3           0.04  FABS
        USR   9,611,394  2,957,352    0.24     0.9           0.08  Domain::x
        USR   9,611,394  2,957,352    0.23     0.9           0.08  Domain::y
        USR   9,611,394  2,957,352    0.22     0.9           0.07  Domain::z
        USR   8,487,726  2,611,608    0.22     0.9           0.08  Domain::xd
        USR   8,487,726  2,611,608    0.22     0.9           0.08  Domain::yd
        USR   8,487,726  2,611,608    0.21     0.8           0.08  Domain::zd
        USR   6,170,528  1,898,624    0.14     0.6           0.07  Domain::fx
        USR   6,170,528  1,898,624    0.14     0.5           0.07  Domain::fy
        USR   6,170,528  1,898,624    0.13     0.5           0.07  Domain::fz
        USR   4,103,944    710,708    0.03     0.1           0.04  SQRT
        USR   3,389,880    993,840    0.03     0.1           0.03  std::vector<int, std::allocator<int> >::operator[]
        USR   3,133,546    506,627    0.04     0.1           0.07  Domain::q
        USR   2,862,314    423,171    0.03     0.1           0.08  Domain::p
        USR   2,591,602    339,735    0.03     0.1           0.08  Domain::e
        USR   2,589,418    339,203    0.03     0.1           0.08  Domain::delv
        USR   2,589,418    339,203    0.03     0.1           0.07  Domain::qq
        USR   2,589,418    339,203    0.02     0.1           0.07  Domain::ql
        USR   2,169,856    667,648    0.03     0.1           0.04  _INTERNAL37529f0f::VoluDer
        USR   1,913,652    547,638    0.04     0.2           0.08  Domain::vdov
        USR   1,763,008    525,805    0.03     0.1           0.05  Domain::delv_zeta
```

```
[smandavilli@login6 scorep_lulesh2_0x8_sum]$ cat scorep.score

Estimated aggregate size of event trace:                   87MB
Estimated requirements for largest trace buffer (max_buf): 87MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       103MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=103MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B]    visits time[s] time[%] time/visit[us]  region
        ALL 90,651,689 3,077,809   31.72   100.0          10.30  ALL
        USR 64,288,822 2,472,647    0.25     0.8           0.10  USR
        OMP 26,119,440   595,800   28.10    88.6          47.16  OMP
        COM    243,386     9,361    3.36    10.6         359.39  COM
    SCOREP         41         1    0.01     0.0       10207.88  SCOREP

        USR 24,531,130   943,505    0.04     0.1           0.04  std::vector<double, std::allocator<double> >::operator[]
        USR  4,356,144   167,544    0.03     0.1           0.16  FABS
        OMP  3,288,600    37,800    0.01     0.0           0.16  !$omp parallel @lulesh.cc:2022
        OMP  3,288,600    37,800    0.01     0.0           0.16  !$omp parallel @lulesh.cc:2029
        USR  2,558,972    98,422    0.01     0.0           0.10  Domain::x
        USR  2,558,972    98,422    0.01     0.0           0.10  Domain::y
        USR  2,558,972    98,422    0.01     0.0           0.09  Domain::z
        USR  2,239,120    86,120    0.01     0.0           0.11  Domain::xd
        USR  2,239,120    86,120    0.01     0.0           0.11  Domain::yd
        USR  2,239,120    86,120    0.01     0.0           0.10  Domain::zd
        USR  1,842,178    70,853    0.01     0.0           0.16  SQRT
        OMP  1,096,200    12,600    0.00     0.0           0.14  !$omp parallel @lulesh.cc:2062
        OMP  1,096,200    12,600    0.00     0.0           0.17  !$omp parallel @lulesh.cc:2075
        OMP  1,096,200    12,600    0.00     0.0           0.13  !$omp parallel @lulesh.cc:2100
        OMP  1,096,200    12,600    0.00     0.0           0.17  !$omp parallel @lulesh.cc:2116
        OMP  1,096,200    12,600    0.00     0.0           0.14  !$omp parallel @lulesh.cc:2153
        OMP  1,096,200    12,600    0.01     0.0           0.74  !$omp parallel @lulesh.cc:2240
        OMP    982,800    37,800    0.02     0.1           0.48  !$omp for @lulesh.cc:2022
        OMP    982,800    37,800    4.78    15.1         126.38  !$omp implicit barrier @lulesh.cc:2027
        OMP    982,800    37,800    0.02     0.1           0.48  !$omp for @lulesh.cc:2029
        OMP    982,800    37,800    4.82    15.2         127.60  !$omp implicit barrier @lulesh.cc:2043
        USR    896,740    34,490    0.00     0.0           0.05  std::vector<int, std::allocator<int> >::operator[]
        USR    866,294    33,319    0.01     0.0           0.21  Domain::q
```

- MPI only code consumes 25.27s which is less than 31.72s taken by OMP only code. However, MPI only takes up 247MB of device memory which is significantly higher than 103MB taken up by OMP only code.
- Also, the number of function calls is very low in OMP (3.6M) as compared to MPI (70M).
- Overall, LULESH MPI is slightly faster, but OMP is cheaper and light on communication.

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.

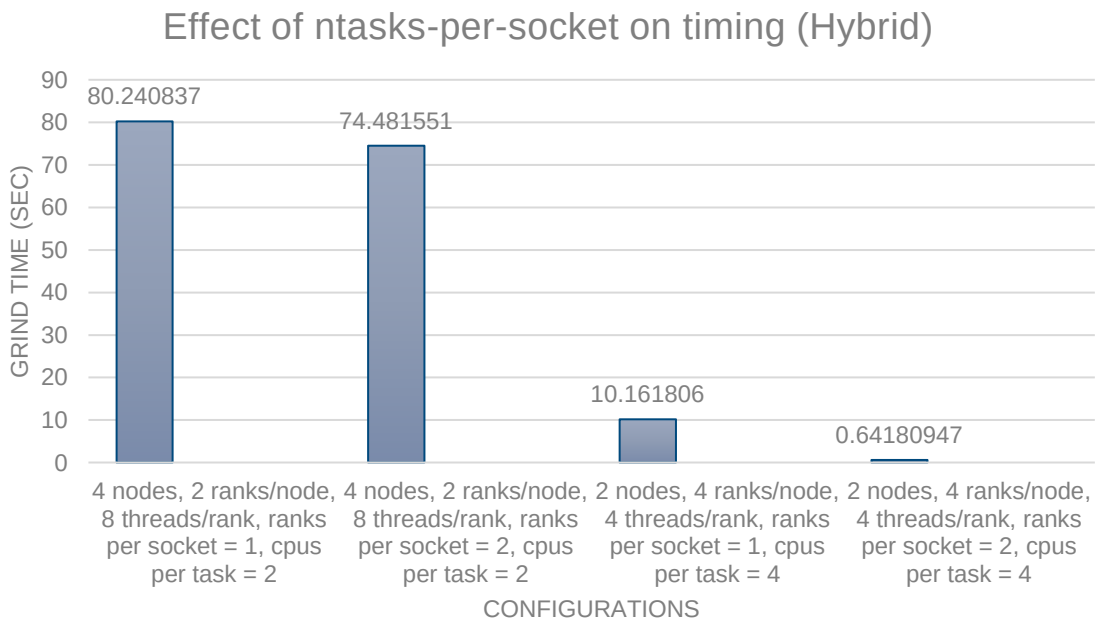
3. Modifying "#pragma omp parallel for" schedule from static to *dynamic* in lulesh.cc
   - Hybrid: Time consumption significantly increased, and the job did not complete in over 10 minutes.
   - OMP: Time consumption significantly increased, and the job did not complete in over 5 minutes.

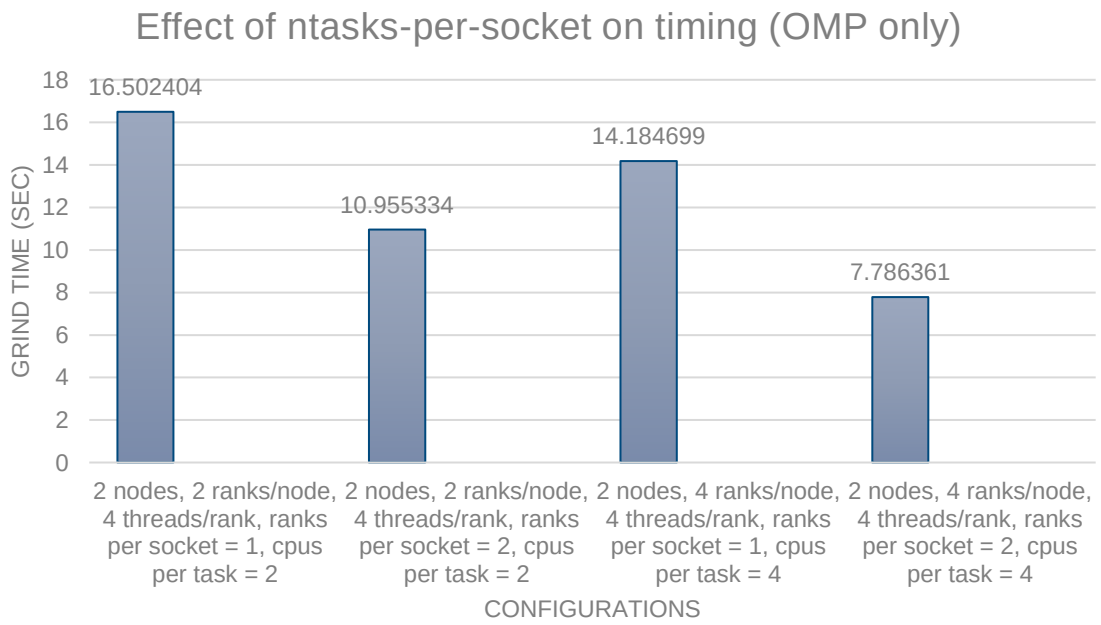
4. Modifying "#pragma omp parallel for" by adding the *ordered* clause in lulesh.cc
   - Hybrid: Time consumption increased from 142.47 to 247.14561s.
   - OMP: Time consumption increased from 31.03 to 33.23s. Memory consumption also increased from 103MB to 120MB.

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.

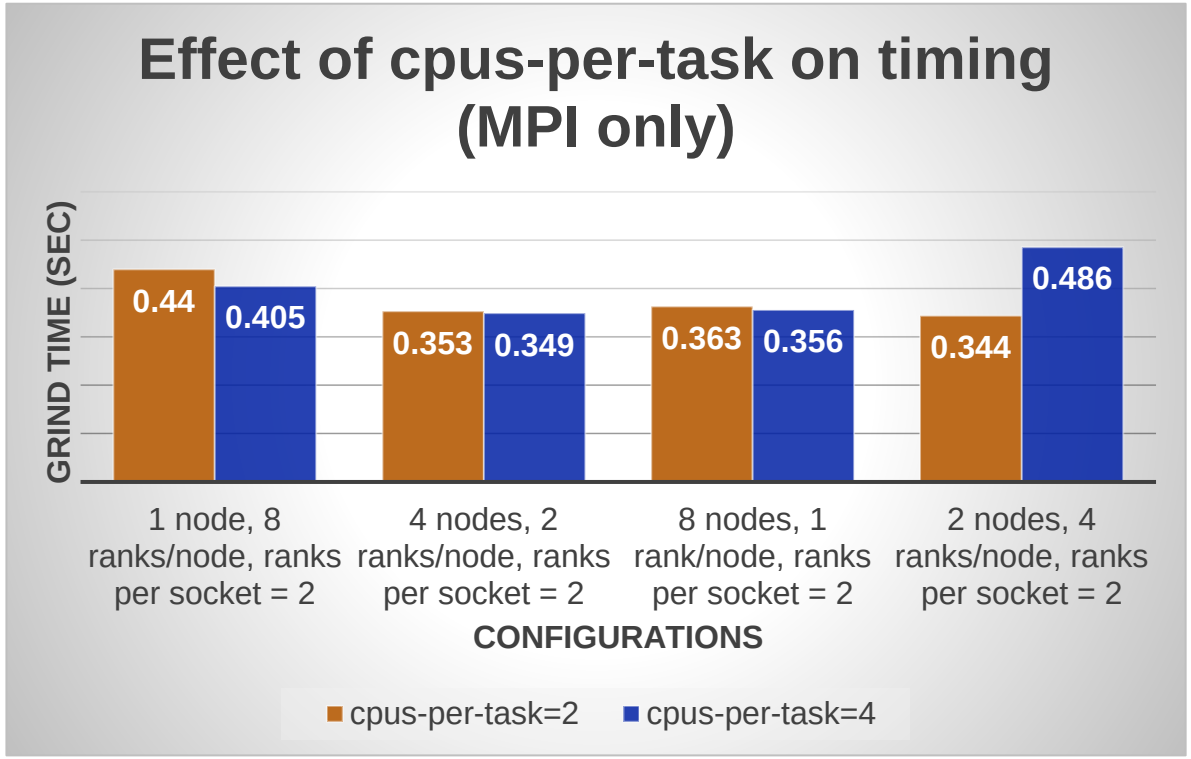
Design Space exploration

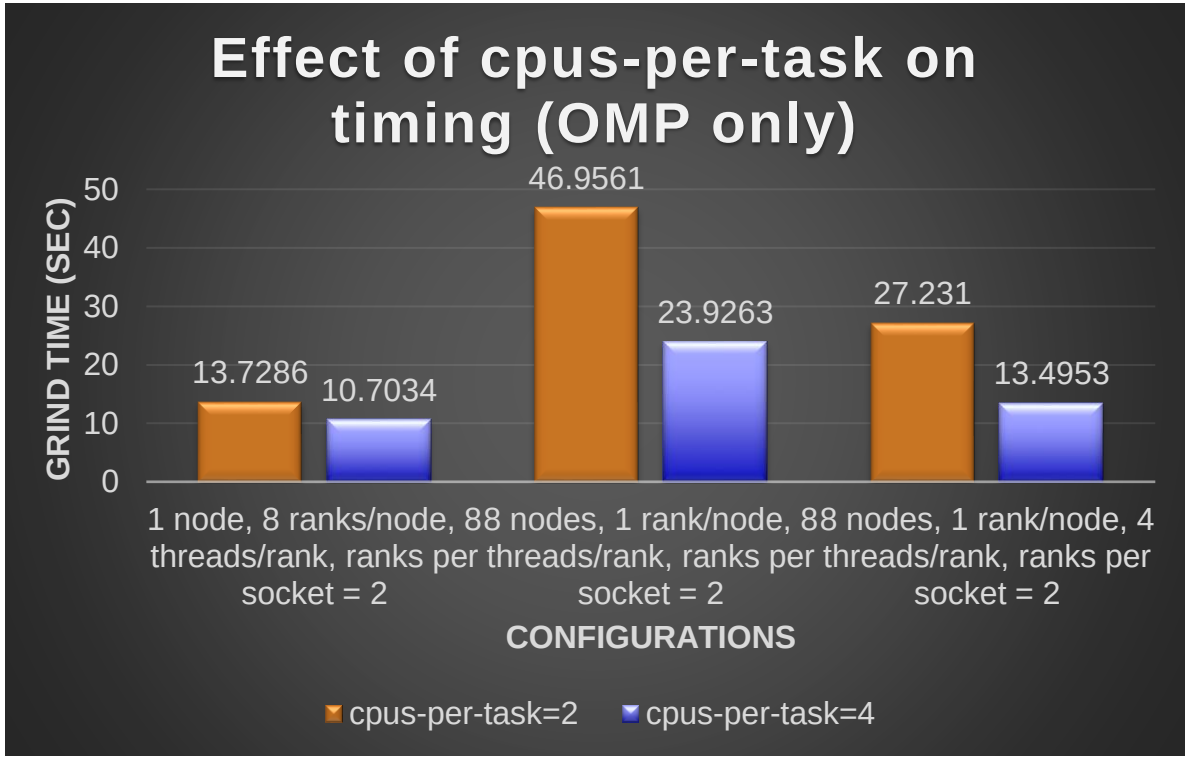
- Objective of DSE is to study the effect on performance by varying configuration (SLURM) parameters and choosing the optimal solution for an application (here LULESH).
- From two different scenarios (Hybrid and OMP only) where we vary only the "*ntasks-per-socket"* parameter between 1 and 2, the time 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.

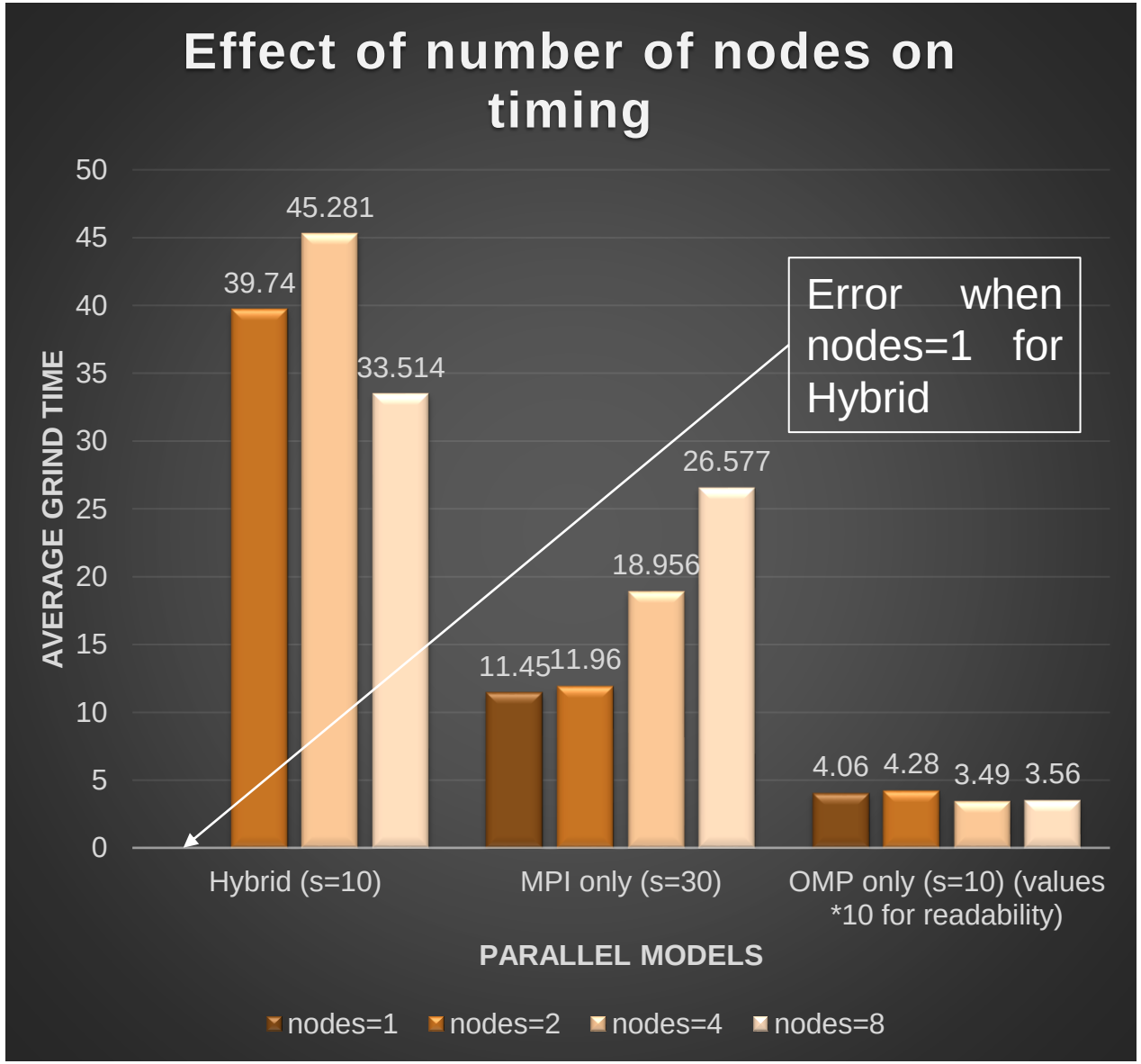




- Further, we observe that the *"number of nodes"* has an impact on timing as well.
- We consider the weighted average for different values of –nodes = 1, 2, 4 and 8.
- Note that the hybrid model throws an error if we run it on 1 node.

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.

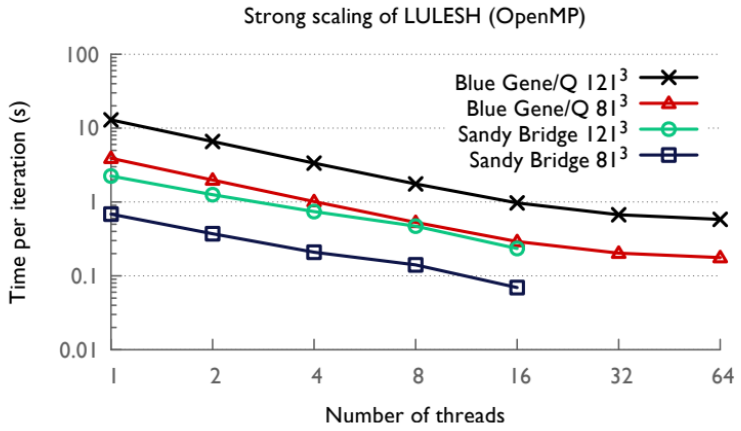


Supporting research paper findings

- 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 (current and next slide): https://dl.acm.org/doi/abs/10.1145/3149412.3149416

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

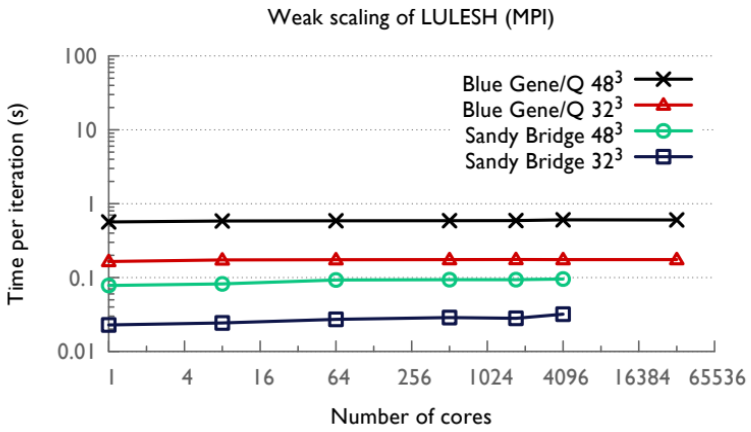


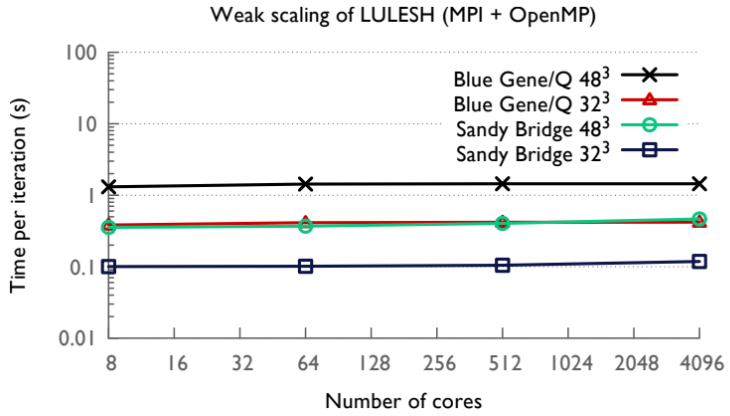
Our results: s=10, configuration = 2 nodes, 2 ranks/node, cpus per task = 2

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 number of threads increase from 1 to 4, exhibiting strong scaling. Time increases again as we further increase threads.

- Inference:
  - 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.
  - 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

- 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.

Summary

- Parallelism and load balancing in LULESH was analyzed with evidence from code.
- Serial, MPI, OMP, MPI+OMP and CUDA models were ported (executed) on HiPerGator.
- All CPU models were profiled with gprof and GPU (CUDA) model with nvprof.
- Time consuming functions were identified, and models were compared for performance.
- Profile data for all CPU models were converted to dot graph using gprof2dot.
- Memory consumption for all CPU models were analyzed using ScoreP and Scalasca.
- Modifications to code was suggested. One technique improved time performance whereas few techniques worsened memory consumption.
- Design space exploration was performed on MPI, OMP and hybrid to identify SLURM parameters for optimal performance.
- Data related to parallelism in LULESH were extracted from research papers and tests were executed to validate the findings.