Illinois Institute of Technology
14 December 2022

H. E. Greenblatt
A20324648

CS 597 Report

1 Introduction.

The purpose of this work is to further an ongoing study of performance and power profiling of workloads relevant to high-performance scientific computing on heterogeneous architectures. In particular, this report analyzes scaling, power, and GPU trace profiles of a proxy benchmark, AEFoam—which combines a traditional scientific workload (a computational fluid dynamics simulation) with machine learning tasks—on various configurations of a CPU/GPU system.

2 History.

Data collection and development were performed on the Argonne Leadership Computing Facility's ThetaGPU system[2]. Details of the ThetaGPU specifications, dependency management, environment configuration, and build process are detailed in the report of Spring 2022. None of this has changed (with one possible exception detailed in the Appendix).

2.1 AEFoam.

AEFoam is one of the three "Solver Example" benchmarks presented as representatives of PythonFOAM[3]. Like its sister solvers, AEFoam combines the OpenFOAM[4] C++ computational fluid dynamics toolkit (for simulation) with Python (for *in situ* data analysis): the C++ application initializes a Python interpreter and controls it through Python's C API. Unlike the other PythonFOAM solvers, however, AEFoam (whose name stands for *autoencoder*) makes use of the TensorFlow[5] machine learning framework, which implements GPU acceleration. It is therefore of interest as a proxy for AI-enabled scientific applications.

AEFoam's control flow is cyclical; a single cycle is depicted in Fig. 1. Specifically:

1. OpenFOAM simulates fluid flow numerically on the CPU, calculating the value of each field at each cell in its mesh for every iteration. The value of a field over the entire domain at a given iteration is called a *snapshot*. A parameter called `writeInterval` controls the treatment of these snapshots. Normally, OpenFOAM writes only the last of every `writeInterval` snapshots to disk and discards the rest (since the storage and I/O required to save and analyze them would otherwise soon grow prohibitive). However, AEFoam instead preserves them in memory, passing them to the Python interpreter.
2. After a block of `writeInterval` snapshots are collected, they are used to train an autoencoder. Since ThetaGPU's version of TensorFlow was compiled with GPU support, this happens on the GPU (if one is available).
3. OpenFOAM simulates another `writeInterval` timesteps. Each snapshot, as it is generated, is supplied to the Python interpreter, where it is encoded and decoded by the autoencoder, and the error of reconstruction calculated.

This cycle then repeats until the problem end time is reached. (N.B.: This description differs from that given in [3] because it is based on the code provided in the supplementary repository, which differs from that analyzed in the text.[6])

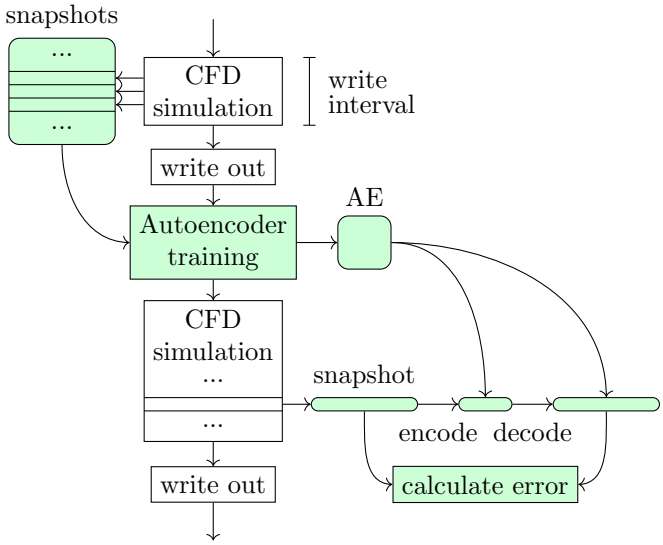

Figure 1: AEFoam control flow on a single subdomain. Operations (depicted with rectangular nodes) and data (rounded) in white occur in C++; those in green, Python. This figure is reproduced from [7].

Although previously studied only in serial, AEFoam was parallelized for this work in order to study its scaling qualities. OpenFOAM provides built-in parallelization capabilites, implemented in MPI, via *domain decomposition*: normal parallelized OpenFOAM workflows involve using OpenFOAM's `decomposePar` utility to divide the problem geometry into subdomains, running an OpenFOAM solver in parallel mode on the decomposed problem (with each processor assigned to one subdomain and exchanging data at the subdomain boundaries), and running the `reconstructPar` utility to reassemble the subdomain results into a single solution for the entire domain.[8] (This study concerns only performance, not output, so decomposition is not profiled and reconstruction is ignored completely.) This architecture means that each AEFoam process will have its own Python interpreter; although the interpreter *can* access the MPI communicator, it will train an individual autoencoder on its own subdomain only.

2.2 Modifications to AEFoam.

Several small changes were made to AEFoam, either to fix bugs or to improve data collection.

- A bug in handling Python references in C++ was fixed, preventing segfault-aborts under memory pressure:

```
    diff --git a/Solver_Examples/AEFoam/PythonComm.H b/Solver_Examples/AEFoam/
PythonComm.H
    index db20acc..543ff6d 100644
    --- a/Solver_Examples/AEFoam/PythonComm.H
    +++ b/Solver_Examples/AEFoam/PythonComm.H
    @@ -25,6 +25,7 @@ if (runTime.outputTime())
             clock_gettime(CLOCK_MONOTONIC, &tw1); // POSIX

             // Call autoencoder
    +        rank_val = PyLong_FromLong(rank);
```

```
            PyTuple_SetItem(autoencoder_args, 0, rank_val);
            (void) PyObject_CallObject(autoencoder_func, autoencoder_args);

    @@ -59,6 +60,7 @@ else
            clock_gettime(CLOCK_MONOTONIC, &tw1); // POSIX

            // Call encode
    +       rank_val = PyLong_FromLong(rank);
            PyTuple_SetItem(encode_args, 0, array_2d);
            PyTuple_SetItem(encode_args, 1, rank_val);
            PyArrayObject *pValue = reinterpret_cast<PyArrayObject*>
    @@ -81,6 +83,7 @@ else
            clock_gettime(CLOCK_MONOTONIC, &tw1); // POSIX

            // Call snapshot
    +       rank_val = PyLong_FromLong(rank);
            PyTuple_SetItem(snapshot_args, 0, array_2d);
            PyTuple_SetItem(snapshot_args, 1, rank_val);
            (void) PyObject_CallObject(snapshot_func, snapshot_args);
    diff --git a/Solver_Examples/AEFoam/PythonCreate.H b/Solver_Examples/AEFoam/
PythonCreate.H
    index 8920095..50af1b9 100644
    --- a/Solver_Examples/AEFoam/PythonCreate.H
    +++ b/Solver_Examples/AEFoam/PythonCreate.H
    @@ -52,7 +52,8 @@ volScalarField wpod_(U.component(vector::Z));
     volScalarField urec_(U.component(vector::X));

     // To pass rank to Python interpreter
    -PyObject *rank_val = PyLong_FromLong(Pstream::myProcNo());
    +int rank = Pstream::myProcNo();
    +PyObject *rank_val = PyLong_FromLong(rank);
     PyObject *array_2d(nullptr);

     int encode_mode = 0;
```

This bug was known from previous work on the sister solver PODFoam (see section 2.2, item 6 of the Spring report).

- Allocation of snapshot memory was moved from the stack to the heap:

```
    diff --git a/Solver_Examples/AEFoam/PythonCreate.H b/Solver_Examples/AEFoam/
PythonCreate.H
    index 50af1b9..fe8ef02 100644
    --- a/Solver_Examples/AEFoam/PythonCreate.H
    +++ b/Solver_Examples/AEFoam/PythonCreate.H
    @@ -63,6 +63,6 @@ int encode_mode = 0;

     // Placeholder to grab data before sending to Python
     int num_cells = mesh.cells().size();
    -double input_vals[num_cells][1];
    +auto input_vals = new double[num_cells][1];
```

This bug was also known from previous work (see section 2.2, item 10). This became necessary only for larger problem sizes (when the memory needed to store entire blocks of snapshots could no longer fit on the

stack), and some configurations were profiled using the original build. However, both general principle and empirical evidence suggest that the data remain comparable: memory management should not meaningfully differ based on virtual address, as long as access patterns are the same, and a scaling analysis for both 'stack' and 'heap' versions showed no apparent performance difference.

- The Python module was patched to pin each autoencoder to a specific GPU:

```
    diff --git a/Solver_Examples/AEFoam/Run_Case/python_module.py b/
Solver_Examples/AEFoam/Run_Case/python_module.py
    index d7e3423..bbe1374 100644
    --- a/Solver_Examples/AEFoam/Run_Case/python_module.py
    +++ b/Solver_Examples/AEFoam/Run_Case/python_module.py
    @@ -16,6 +16,15 @@
     from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
     from tensorflow.keras.models import load_model, Sequential, Model

    +import mpi4py
    +mpi4py.rc.initialize = False
    +mpi4py.rc.finalize = False
    +from mpi4py import MPI
    +if MPI.Is_initialized(): # we're not running in serial mode
    +    gpus = tf.config.list_physical_devices('GPU')
    +    if len(gpus) > 0:
    +        tf.config.set_visible_devices([gpus[MPI.COMM_WORLD.Get_rank() % len(
gpus)]], 'GPU')
    +
     # Custom activation (swish)
     def my_swish(x, beta=1.0):
         return x * K.sigmoid(beta * x)
```

TensorFlow's default behavior is to allocate almost all memory on all available GPUs, then run only on the GPU with the lowest ID[9]. Enabling the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable disabled greedy allocation, while calling `tf.config.set_visible_devices` here striped the processes across all available GPUs.

2.3 Modifications to the case definition.

The case definition (that is, the directory structure defining the particular boundary value problem AEFoam was solving) was modified in the following ways:

- The problem `endTime` was set to 0.035.

The original `endTime` of 10000 was far too long to work with (see section 2.2, item 11); a much shorter once was necessary. The specific value 0.035 was chosen purely for convenience (it made the original problem size finish within the maximum walltime of the debug queue).

- The geometry was re-meshed.

It was clear based on previous work that spatially larger problem sizes would be needed for good scaling data, and the problem had to be re-meshed in order to change the size. The original mesh generation tool was unavailable (due to affiliation and licensing issues). However, Dr. Maulik very kindly supplied the original geometry file, along with instructions for using the open-source alternative Gmsh[11]:

1. Set the `refinement` parameter in the geometry file to the desired scale (smaller values are finer meshes).

2. Run `gmsh -3 -format msh2 -o <out>.msh bfs.geo` to generate a 3D mesh from the geometry file `bfs.geo`.
3. Run `gmshToFoam <out>.msh -case <case directory>/` to generate an OpenFOAM mesh from an existing OpenFOAM case directory and a Gmsh mesh (`gmshToFoam` is supplied by OpenFOAM).
4. Edit `<case directory>/constant/polyMesh/boundary`, with reference to the original `boundary` file, to restore the proper `type` and `inGroups` parameter values.

Because of the re-meshing, it was also necessary to replace one initial condition. One of the flow-field variables, `nut`, was originally defined over the `internalField` as a `nonuniform` list of explicit values (derived from the output of "another solver that assisted with accuracy/time-to-solution"[10]). Because this list was not the correct length for the re-meshed geometry, it was replaced with a `uniform` value of 1×10^{-4}.

2.4 Data collection.

Performance and power data were collected using version 2 of Mantis, a test harness[12] for automatically running applications in multiple configurations through multiple profiling tools and compiling the results in a single format. Where in Spring, benchmarks were run with 'plugins' taking the form of Bash scripts, Mantis is now a Python library which can provide its own CLI or be called from other Python code. (Most of the work on Mantis took place during Summer 2022 and is therefore outside the scope of this report.) This AEFoam runscript does the latter:

```
import mantis_monitor
import itertools
import subprocess
import os

class AEFoam(mantis_monitor.benchmark.benchmark.Benchmark):
    cwd = '/lus/grand/projects/SEEr-Polaris/Run_Case_G'
    env = None # don't overwrite externally-set env vars

    @classmethod
    def generate_benchmarks(cls, arguments):
        return [
            cls({'gpu_count': gpu_count, 'ranks': ranks, 'nodes': nodes})
            for (gpu_count, ranks, nodes)
            in itertools.product(arguments['gpu_counts'], arguments['ranks'],
    arguments['nodes'])
        ]

    def before_each(self):
        print(f'Starting run with {self.gpu_count} GPUs')
        os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(str(i) for i in range(self.
    gpu_count)) # os.putenv doesn't update os.environ
        print('Cleaning output', flush=True)
        subprocess.run('rm *.h5', shell=True, cwd=self.cwd)
        subprocess.run("find . -maxdepth 1 -regex
    '\./\([1-9][0-9]*\|[0-9][0-9]*\.[0-9][0-9]*\)' -exec rm -r {} \;", shell=True,
    cwd=self.cwd)
        subprocess.run('rm -r processor*', shell=True, cwd=self.cwd)
        if self.ranks > 1:
            print('Decomposing mesh', flush=True)
            subprocess.run('decomposePar', shell=True, cwd=self.cwd)
```

```
        def get_run_command(self):
            aefoam_bin = '~/OpenFOAM/hgreenbl-8/platforms/linux64GccDPInt32-spack/bin/
        AEFoam-heap'
            if self.ranks == 1:
                return f'{aefoam_bin}'
            if self.nodes == 1:
                return f'mpiexec -np {self.ranks} {aefoam_bin} -parallel'
            env_string = ' '.join(f'-x {var}' for var in os.environ.keys())
            ranks_per_node = self.ranks // self.nodes
            return (
               f'mpiexec -hostfile $COBALT_NODEFILE {env_string} -display-map '
               f'-np {self.ranks} -npernode {ranks_per_node} {aefoam_bin} -parallel'
            )

        def __init__(self, arguments):
            self.gpu_count = arguments['gpu_count']
            self.ranks = arguments['ranks']
            self.nodes = arguments['nodes']
            self.name = f'AEFoam_{self.gpu_count}gpu_{self.nodes}node_{self.ranks}rank'

    mantis_monitor.monitor.run_with(AEFoam)
```

The AEFoam class defines what confguration parameters Mantis should accept for this benchmark, how to set up or tear down each run (in this case, cleaning up old output and generating a fresh decomposition), and how to actually run AEFoam (taking the configuration into account). Mantis itself handles actually running each profiling tool and all data collection.

3 Results.

Data were collected for the following problem sizes:

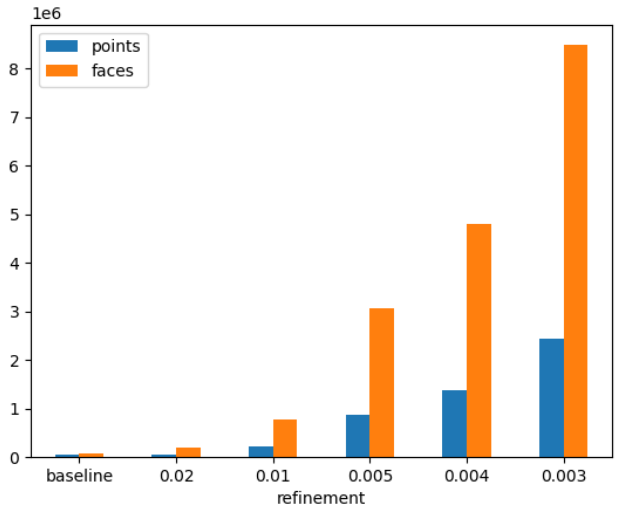

Figure 2: Numbers of points and faces in the problem sizes profiled. "baseline" refers to the original mesh geometry.

The ‘heap’ version of AEFoam was required for refinements 0.004 and 0.003 due to their greater memory requirements. However, a comparison of the ‘stack’ and ‘heap’ builds shows very little performance difference:

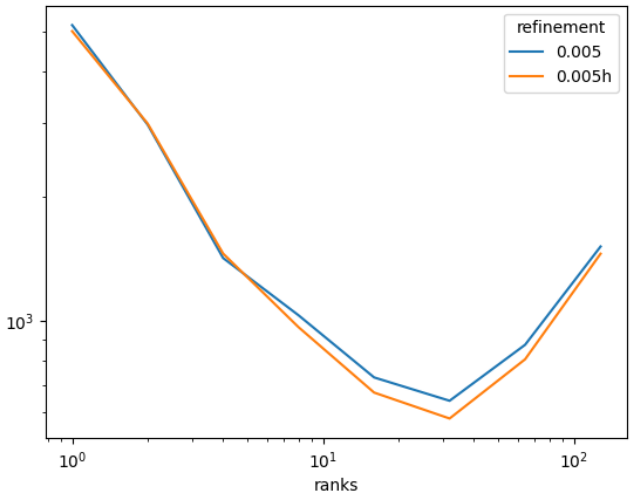

Figure 3: Time to completion (s) vs ranks (#) for ‘stack’ (0.005) and ‘heap’ (0.005h) AEFoam (8 GPUs, 1 node). All scaling plots are log-log.

Refinements below 0.003 were not collected due to their even greater memory requirements. (An attempt at 0.002 was OOM-killed.)

Overall scaling results for single-node tests are shown in Fig. 4. Scaling was very weak at smaller problem sizes and better at larger ones, but never showed consistent improvement all the way to maximum ranks; even the largest problem sizes peaked at just 32 ranks (rather than the 128 available per node).

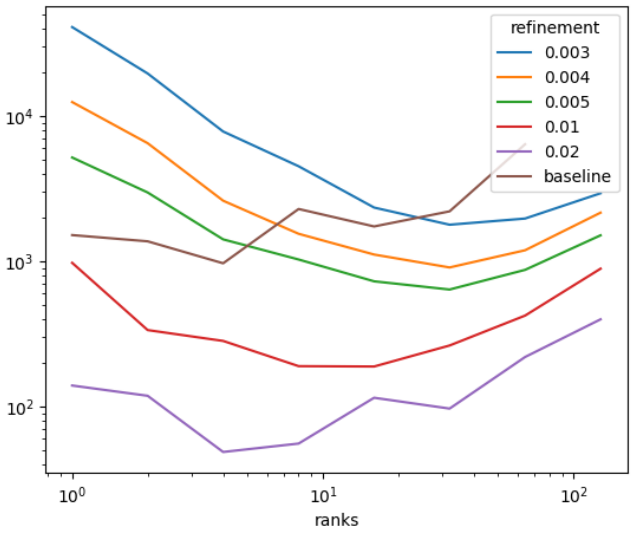

Figure 4: Time to completion (s) vs ranks (#) for various refinements (8 GPUs, 1 node).

Surprisingly, the re-meshed problem ran much faster than the baseline at comparable problem sizes. Inspection of the output showed this to be due to the solver adopting a much greater timestep—approximately 1×10^{-4} as opposed to the previous 3×10^{-6}. This appears to be due to the change in mesh strategy rather than the change in initial conditions, as keeping the baseline mesh but switching to a uniform `nut` made a comparatively tiny difference:

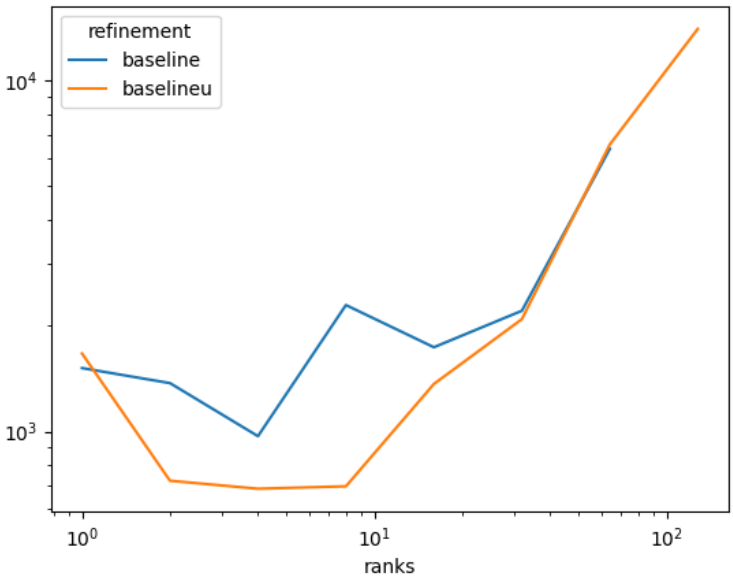

Figure 5: Time to completion (s) vs ranks (#) for 'baseline' and 'baseline-uniform' (baselineu) AEFoam (8 GPUs, 1 node).

The poor performance at small problem sizes was expected, as previous work had made it clear that the baseline case definition was quite small. PODFoam, supplied with a functionally identical case directory, did not scale past 8 ranks (see section 2.2, item 7 of previous report). Poor performance at high rank counts was less expected.

This may be in part an artifact of the 'simple' decomposition method used by OpenFOAM coping poorly with Gmsh's meshes: multiple subdomains at higher rank counts contained *no cells at all*—and this was almost completely independent of problem size!

Table 1: Number of empty subdomains, by refinement and ranks

baseline	1	1	0.02	1	0	0.01	1	0	0.005	1	0	0.004	1	0	0.003	1	0
baseline	2	1	0.02	2	0	0.01	2	0	0.005	2	0	0.004	2	0	0.003	2	0
baseline	4	1	0.02	4	0	0.01	4	0	0.005	4	0	0.004	4	0	0.003	4	0
baseline	8	1	0.02	8	0	0.01	8	0	0.005	8	0	0.004	8	0	0.003	8	0
baseline	16	1	0.02	16	0	0.01	16	0	0.005	16	0	0.004	16	0	0.003	16	0
baseline	32	2	0.02	32	1	0.01	32	1	0.005	32	1	0.004	32	1	0.003	32	1
baseline	64	4	0.02	64	3	0.01	64	3	0.005	64	3	0.004	64	3	0.003	64	3
baseline	128	8	0.02	128	9	0.01	128	9	0.005	128	9	0.004	128	9	0.003	128	9
									0.005	256	18				0.003	256	18
									0.005	512	36				0.003	512	36

Except for the baseline problem, with its different mesh, the number of empty subdomains depended solely on the number of subdomains total, and not at all on the problem size. Naïvely, one might expect that as refinement increased, cells would grow smaller everywhere, and the number of empty subdomains would decrease, but this is not the case. Inspecting the rendered mesh (for example, in the Gmsh GUI) reveals that there are significant differences in cell size between the center of domain and the area around its edges; it is possible that this overwhelms the general decrease in cell size. Using a different decomposition method (such as the 'Scotch') might or might not ameliorate this. Regardless, the presence of multiple empty subdomains implies that many more have only a few cells, further decreasing the effectiveness of decomposition.

However, the poor performance is certainly also due to GPU contention.

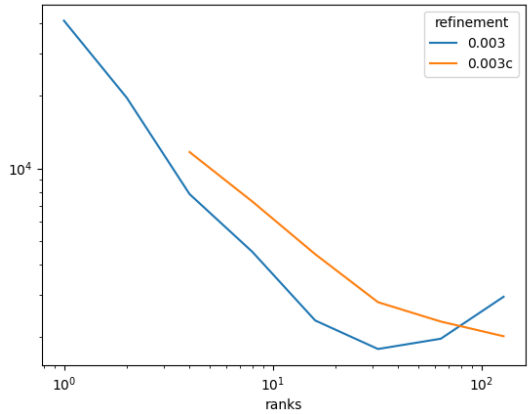

Figure 6: Time to completion (s) vs ranks (#) for 8-GPU (0.003) and 0-GPU (0.003c) AEFoam (refinement 0.003, 1 node).

Disabling GPU usage actually resulted in better scaling at the high end: although 8-GPU AEFoam performed better than 0-GPU AEFoam through 64 ranks, 0-GPU was better—not just relatively, but absolutely—at 128 ranks.

This is supported by the results for scaling performance across multiple nodes. (`mpiexec`'s `-npernode` option maps ranks to nodes by block, so the striping of ranks to GPUs still distributes them appropriately to all GPUs on a given node. This is confirmed by `-displaymap` and GPU usage metrics such as power draw.)

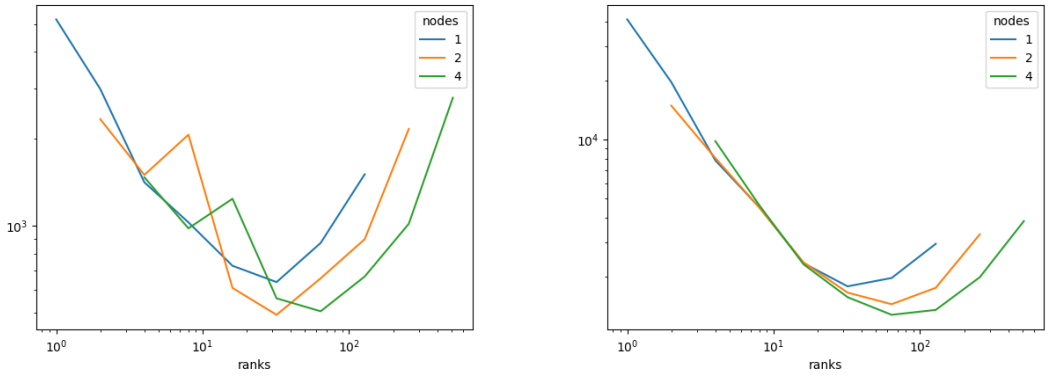

Figure 7: Time to completion (s) vs ranks (#) for 1, 2, and 4 nodes (8 GPUs, refinements 0.005 and 0.003).

Increasing the number of nodes for a given rank count provides little or no benefit up to 16 ranks (there is little or no detriment, either, suggesting that data transfer between nodes is not a significant issue—not too surprising, since this need only occur at subdomain boundaries and is therefore limited), but does show some benefit at 32 ranks and above, although not enough to improve scaling past 64. This suggests that contention becomes an issue at approximately 2–4 autoencoders per GPU, which is consistent with preliminary results from 1-GPU AEFoam:

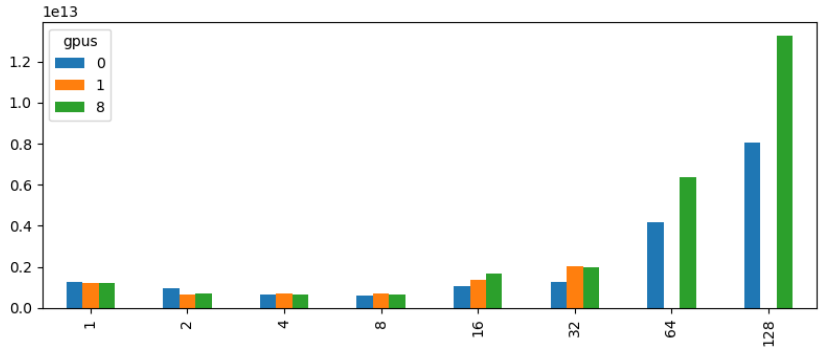

Figure 8: Time to completion (s) vs GPUs (#) for 1–128 ranks (refinement baseline, 1 node). Data could not be collected for 64 or 128 ranks on 1 GPU because the amount of memory required exceeds that available on a single GPU.

The size of the training dataset for each autoencoder should approximately halve for each doubling of the rank count, but this evidently does not translate to maintaining a constant training speed.

This is also illustrated by GPU power draw data.

On one node, from 1 to 8 ranks, the power spikes of the training intervals not only come closer together (as the CFD simulation on the CPU speeds up) and are themselves shorter (as the total complexity of training decreases with the input width), but draw less power (as the input decreases in size).

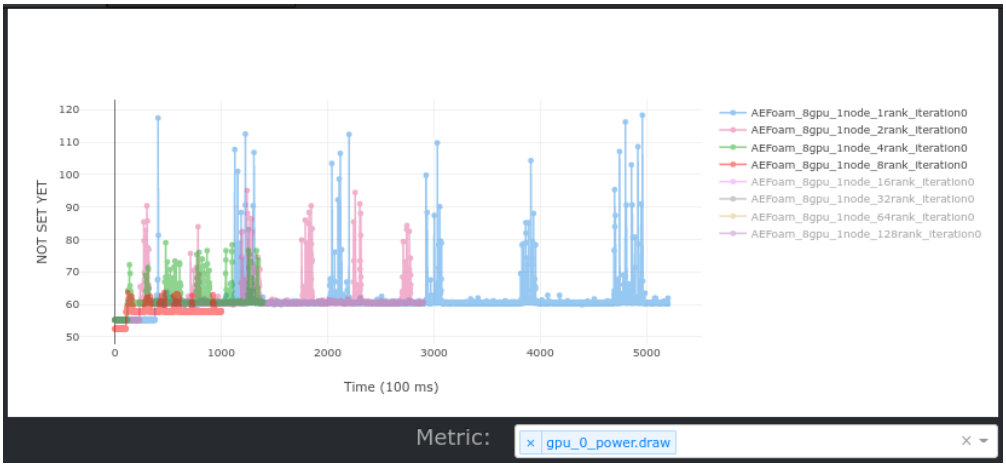

Figure 9: Power draw (W) vs time (100 ms) for 1–8 ranks. All power data was collected at refinement 0.005. GPU 0 is shown as representative. Measurement error may result in large baseline differences between time series.

From 8 to 16 ranks, the spikes still come closer together (as additional resources are still being added on the CPU side), but the improvements in duration and amplitude are less clear.

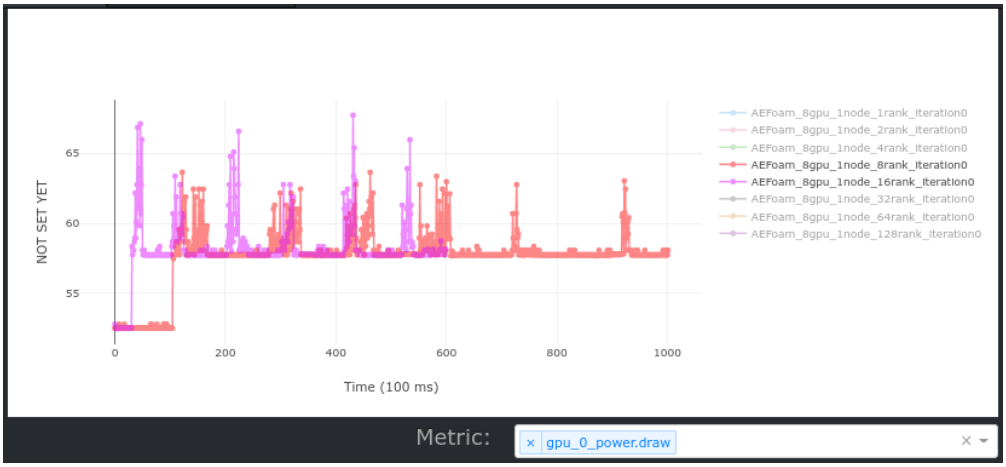

Figure 10: Power draw (W) vs time (100 ms) for 8–16 ranks.

Above 16 ranks, the spikes are increasingly smeared out, appearing to decrease in amplitude but not area-under-curve. (Unfortunately, this power data is likely too noisy to calculate total draw accurately; collection additional data at higher time resolutions could prove helpful here.) This cannot be explained by training intervals desyncing; neighboring processes must communicate during each iteration to exchange data at subdomain boundaries, and AEFoam's output confirms that all ranks always enter and leave the training phases together. It is a genuine increase in the duration of the training itself.

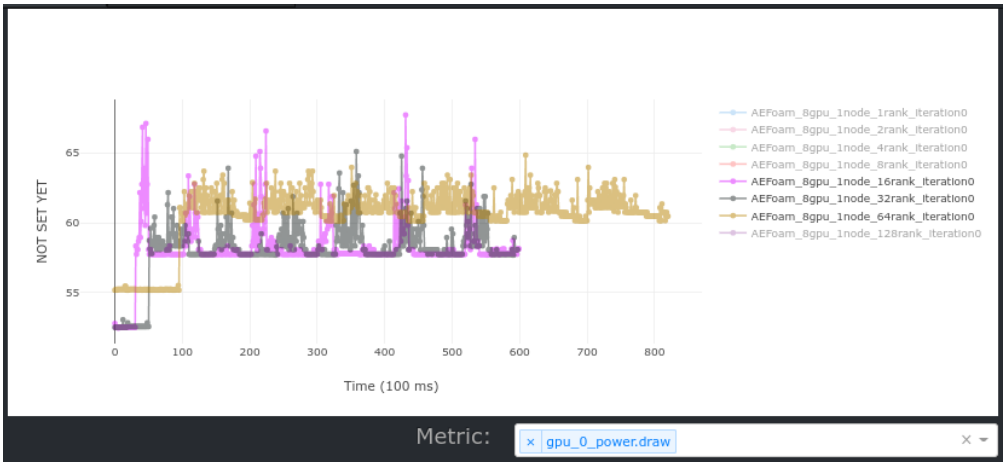

Figure 11: Power draw (W) vs time (100 ms) for 16–64 ranks.

On two and four nodes, behavior similar to that of one node is apparent at equivalent rank-per-node counts: at first a rapid increase in speed and decrease in power consumption, giving way to stagnation at approximately 16 ranks per node (again, 2–4 autoencoders per GPU), followed by large increases in training duration.

Figure 12: Power draw (W) vs time (100 ms) for various ranks (2 and 4 nodes).

Distributing the same number of ranks across increasing numbers of nodes decreases power draw per GPU—as expected, since although the size of the training datasets are unchanged, fewer of them are being processed on each GPU.

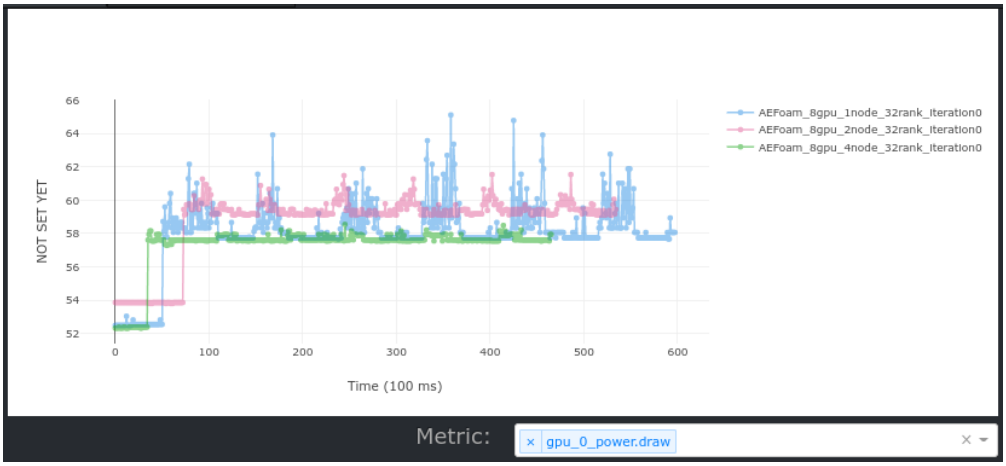

Figure 13: Power draw (W) vs time (100 ms) for 1–4 nodes (32 ranks).

Perhaps more surprisingly, distributing the same number of ranks *per GPU* across increasing numbers of nodes still decreases power draw per GPU: it seems that the decrease in dataset size alone is sufficient to reduce demand.

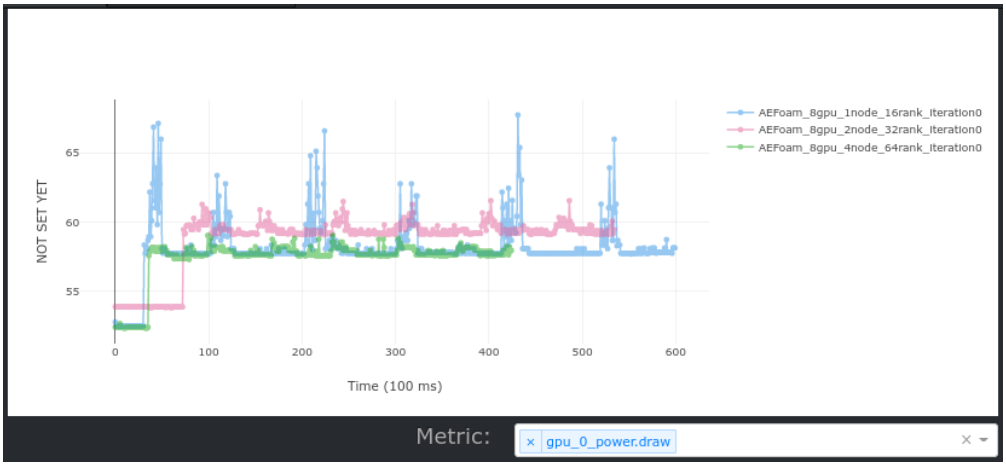

Figure 14: Power draw (W) vs time (100 ms) for 1, 2, and 4 nodes (16, 32, and 64 ranks).

Of course, the effect of increased decomposition on *total* power draw is unlikely to be as good as constant, never mind better (even assuming that GPUs are never turned off if not in use). Again, the noisiness of these data make it difficult to measure this accurately.

Finally, GPU trace data were collected for 1–8 ranks on 8 GPUs, refinement 0.005, 1 node. (Attempts to trace at higher rank counts hanged for unclear reasons.)

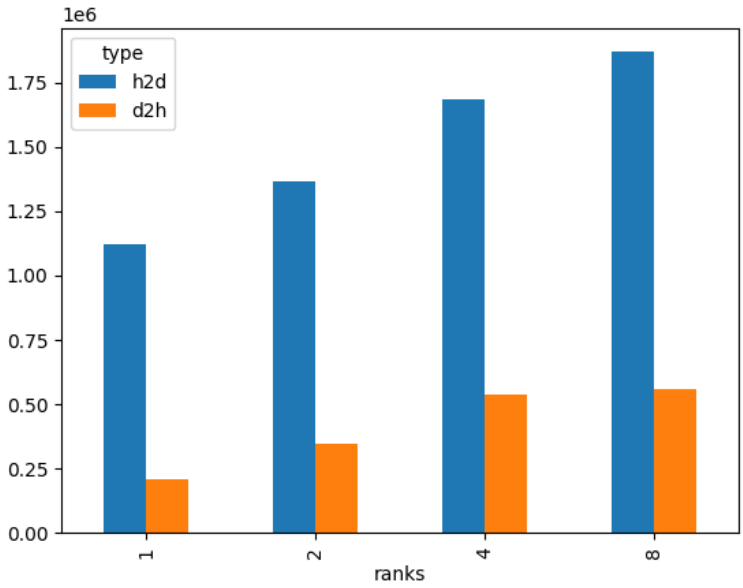

Figure 15: Data transferred host-to-device and device-to-host (MB), by ranks.

Device-to-device movement was negligible. The steadily decreasing ratio of host-to-device to device-to-host

movement is expected: the total input size summed over all autoencoders remains the same (so the host-to-device movement increases only by the overhead of additional autoencoders), whereas the total output size is a multiple of the number of autoencoders (so the device-to-host movement increases almost linearly, less any overhead). However, the cause of the faltering of this pattern at 8 ranks is unclear.

The memory time summary showed results similar to the size summary. The CUDA API summary had host-to-device and device-to-host memory copies as the second and third biggest runtime contributors, respectively, behind `cuEventRecord` at #1 (typical of a TensorFlow workload); together these three calls made up 96.1% of the CUDA API call runtime. The kernel summary was much more long-tailed, with TensorFlow's `ApplyAdamKernel` at just 18.6% of total kernel runtime.

Complete datasets are available in the Mantis format.

4 Future work.

The immediate next step is the collection of more targeted metrics (likely using both the perf[13] and Nsight Systems[14] Mantis collectors) to better understand the costs of data transfer between hosts and devices, among hosts, and between cache levels.

Beyond that, one possible future direction is a comparison of the current AEFoam architecture, with its isolated autoencoders, to a distributed ML model. Although TensorFlow itself implements distributed training strategies for multiple GPUs, and third-party wrappers like Horovod[15] allow training to be distributed over multiple nodes, neither of these features can easily be grafted onto AEFoam, because OpenFOAM's domain decomposition paradigm is a poor fit for the type of parallelism useful in distributed training. Each OpenFOAM process has a small piece of every snapshot; a distributed learning model would rather have a small sample of complete snapshots. However, this is not insurmountable. mpi4py[16] provides the primitives necessary to redistribute data, and AEFoam could be rewritten to train one model per GPU, one model per node, or—with the use of Horovod—one model *in toto*. (The technical problem of getting OpenFOAM to run under the MPI threading level required to safely combine Horovod with mpi4py is complex but solved.) Although it may have seemed like a poor fit *a priori*, the results presented here show that AEFoam is not making the best possible use of its GPUs.

Alternatively, the *very* tight Python-C++ coupling—efficient, but awkward to prototype and difficult to generalize—might be compared to an application using one of the emerging libraries designed for simulation/ML interaction, such as SmartSim[17]. Such tools have inevitable overhead, but programmers have been trading efficiency for expressive power since the assembler. The future of AI-enabled science is quite likely to involve general-purpose frameworks such as these, and quantifying the costs involved will be crucial for good architectural decision-making in software and hardware alike.

Appendix: Bus errors

Some runs of AEFoam ended in bus errors like the following:

```
    [thetagpu05:3649915:0:427126] Caught signal 7 (Bus error: nonexistent
physical address)
    ==== backtrace (tid: 427126) ====
     0  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(ucs_handle_error+0x77) [0x7fdcd513c5fc]
     1  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(+0x363b6) [0x7fdcd513c3b6]
     2  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(+0x364b2) [0x7fdcd513c4b2]
     3  /lib/x86_64-linux-gnu/libpthread.so.0(+0x14420) [0x7fdcde78b420]
     4  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(+0x14480) [0x7fdcd511a480]
     5  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(+0x3dafd) [0x7fdcd5143afd]
     6  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucs.so.0(+0x3f308) [0x7fdcd5145308]
     7  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucm.so.0(ucm_event_dispatch+0x4d) [0x7fdcdc00e282]
     8  /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1_cuda-11.4_gcc-9.4.0/lib/
libucm.so.0(ucm_vm_munmap+0xf0) [0x7fdcdc00e958]
     9  /lus/theta-fs0/software/thetagpu/openmpi/openmpi-4.1.4_ucx-1.12.1_gcc
-9.4.0/lib/libopen-pal.so.40(opal_mem_hooks_release_hook+0x7c) [0x7fdcdcc7d3dc]
    10  /lus/theta-fs0/software/thetagpu/openmpi/openmpi-4.1.4_ucx-1.12.1_gcc
-9.4.0/lib/libopen-pal.so.40(+0x813d0) [0x7fdcdccd03d0]
    11  /lib/x86_64-linux-gnu/libpthread.so.0(+0x88aa) [0x7fdcde77f8aa]
    12  /lib/x86_64-linux-gnu/libc.so.6(clone+0x43) [0x7fdcdf8bb133]
    =================================
    [thetagpu05:3649915] *** Process received signal ***
    [thetagpu05:3649915] Signal: Bus error (7)
    [thetagpu05:3649915] Signal code:  (-6)
    [thetagpu05:3649915] Failing at address: 0x89f70037b17b
    [thetagpu05:3649915] [ 0] /lib/x86_64-linux-gnu/libpthread.so.0(+0x14420)[0
x7fdcde78b420]
    [thetagpu05:3649915] [ 1] /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1
_cuda-11.4_gcc-9.4.0/lib/libucs.so.0(+0x14480)[0x7fdcd511a480]
    [thetagpu05:3649915] [ 2] /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1
_cuda-11.4_gcc-9.4.0/lib/libucs.so.0(+0x3dafd)[0x7fdcd5143afd]
    [thetagpu05:3649915] [ 3] /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1
_cuda-11.4_gcc-9.4.0/lib/libucs.so.0(+0x3f308)[0x7fdcd5145308]
    [thetagpu05:3649915] [ 4] /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1
_cuda-11.4_gcc-9.4.0/lib/libucm.so.0(ucm_event_dispatch+0x4d)[0x7fdcdc00e282]
    [thetagpu05:3649915] [ 5] /lus/theta-fs0/software/thetagpu/ucx/ucx-1.12.1
_cuda-11.4_gcc-9.4.0/lib/libucm.so.0(ucm_vm_munmap+0xf0)[0x7fdcdc00e958]
    [thetagpu05:3649915] [ 6] /lus/theta-fs0/software/thetagpu/openmpi/openmpi
-4.1.4_ucx-1.12.1_gcc-9.4.0/lib/libopen-pal.so.40(opal_mem_hooks_release_hook+0
x7c)[0x7fdcdcc7d3dc]
    [thetagpu05:3649915] [ 7] /lus/theta-fs0/software/thetagpu/openmpi/openmpi
-4.1.4_ucx-1.12.1_gcc-9.4.0/lib/libopen-pal.so.40(+0x813d0)[0x7fdcdccd03d0]
```

```
    [thetagpu05:3649915] [ 8] /lib/x86_64-linux-gnu/libpthread.so.0(+0x88aa)[0
x7fdcde77f8aa]
    [thetagpu05:3649915] [ 9] /lib/x86_64-linux-gnu/libc.so.6(clone+0x43)[0
x7fdcdf8bb133]
    [thetagpu05:3649915] *** End of error message ***
    -------------------------------------------------------------------------
    Primary job  terminated normally, but 1 process returned
    a non-zero exit code. Per user-direction, the job has been aborted.
    -------------------------------------------------------------------------
    -------------------------------------------------------------------------
    orterun noticed that process rank 17 with PID 0 on node thetagpu05 exited on
signal 7 (Bus error).
    -------------------------------------------------------------------------
```

This was extremely puzzling, particularly as it had no apparent precipitating event, was nondeterministic, and appeared suddenly in mid-November despite no deliberate changes to the software or system.

However, I now suspect that it is related to a change in environment configuration. The builds of AEFoam studied here (both the 'stack' and the 'heap' version) were linked against `openmpi/openmpi-4.1.1_ucx-1.11.2_gcc-9.3.0`; this is confirmed by the Spack configuration files and the Spring report. The Spring report further states explicitly that this is the version of `openmpi` that the `conda/2021-11-30` module depends on. However, the current module file for `conda/2021-11-30` specifies that it depends on `openmpi/openmpi-4.1.4_ucx-1.12.1_gcc-9.4.0`, a newer version. I cannot prove that this was the case, but I believe that the conda module had its Open MPI dependency changed without being itself given a version bump.

I have no specific evidence that the bus errors are related, but the coincidence is suspicious and I can conjecture no other cause. (In the case of a segfault, I might contemplate an error in OpenFOAM or even—given that this is still very new hardware—Open MPI, but a *bus error* is simply not a typical class of bug.)

Due to time limitations, OpenFOAM and AEFoam were not recompiled against the newer Open MPI; failed jobs were merely re-run. I am hopeful that the version mismatch does not otherwise affect the validity of the results presented here.

References

[1] https://github.com/argonne-lcf/PythonFOAM

[2] https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview

[3] https://arxiv.org/abs/2103.09389

[4] https://cfd.direct/openfoam/user-guide-v8/

[5] https://www.tensorflow.org/

[6] R. Maulik, private communication, Aug. 17, 2022.

[7] H.E. Greenblatt, et al, "Performance and Power Characterization of AI-enabled Applications on Heterogeneous System". Unpublished.

[8] https://cfd.direct/openfoam/user-guide/v8-running-applications-parallel/

[9] https://www.tensorflow.org/guide/gpu

[10] R. Maulik, private communication, Nov. 8, 2022.

[11] https://gmsh.info/

[12] https://github.com/mseryn/mantis-monitor

[13] https://perf.wiki.kernel.org/index.php/Main_Page

[14] https://docs.nvidia.com/nsight-systems/UserGuide/index.html

[15] https://horovod.readthedocs.io/en/stable/

[16] https://mpi4py.readthedocs.io/

[17] https://www.craylabs.org/docs/overview.html