Performance Debugging and Tuning of Flash-X with Data Analysis Tools

Kevin Huck
University of Oregon
Eugene, Oregon, USA
Email: khuck@cs.oregon.edu

Xingfu Wu
Argonne National Laboratory
Lemont, Illinois, USA
Email: xingfu.wu@anl.gov

Anshu Dubey
Argonne National Laboratory
Lemont, Illinois, USA
Email: adubey@anl.gov

Antigoni Georgiadou
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
Email: georgiadoua@ornl.gov

J. Austin Harris
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
Email: harrisja@ornl.gov

Tom Klosterman
Argonne National Laboratory
Lemont, Illinois, USA
Email: tklosterman@anl.gov

Matthew Trappett
University of Oregon
Eugene, Oregon, USA
Email: mtrappet@uoregon.edu

Klaus Weide
Argonne National Laboratory
Lemont, Illinois, USA
Email: kweide@anl.gov

***Abstract*—State-of-the-art multiphysics simulations running on large scale leadership computing platforms have many variables contributing to their performance and scaling behavior. We recently encountered an interesting performance anomaly in Flash-X, a multiphysics multicomponent simulation software, when characterizing its performance behavior on several large-scale platforms. The anomaly was tracked down to the interaction between the use of dynamic allocation of scratch data and data locality in the cache hierarchy. In this paper we present the details of unexpected performance variability we encountered, the extensive analysis using the performance measurement tool TAU to collect the data and Python data analysis libraries to explore the data, and our insights from this experience. In the process, we discovered and removed or mitigated two additional performance limiting bottlenecks for performance tuning.**

I. Introduction

Simulations for grand challenges in science often require multiphysics capabilities in the software that needs to run on some of the largest computing platforms. Many degrees of freedom exist on both sides of these requirements. Such software is complex because it has many components that need to interoperate with one another, and large scale platforms are complex because of their size and the increasing heterogeneity. Because of such complex platforms, the system software stack in general, and compilers in particular, have challenges of their own. The optimization search space is huge, and compilers make conservative assumptions about optimizability of code sections. As a result, sometimes, what may seem like an obvious outcome of a hand-optimization may not turn out to be true.

We recently encountered such instance of anomalous performance behaviour in optimizing Flash-X, a newly-developed code that has been derived from a multiphysics multicomponent code FLASH [1]–[3] which has been a community code for multiple science domains for many years. The infrastructural aspects of Flash-X have been built from ground up to be compatible with the increasing heterogeneity in the hardware and the increasing complexity because of higher fidelity models within the code. The base discretization in Flash-X is Eulerian, and the code has the ability to switch between three flavors of management for the discretized mesh. One of them is simple mesh where spacing between discrete points is uniform throughout the domain, while the other two use adaptive mesh refinement (AMR) [4], [5], where spacing between points is dictated by the resolution needs of the corresponding section of the domain. The two AMR packages, AMReX [6], and Paramesh [7] have very different approaches to data management and distribution (see II-A for details), which makes one-on-one comparison between them interesting.

During the course of this comparison we found some redundant communications in Paramesh. Eliminating this communication resulted in a very unexpected result. The communication cost went down, but the cost in some of the purely computational sections of the physics solvers went up by more than 20% in some instances. Further investigations led to uncovering a few other non-obvious performance implications. In this paper we discuss our experiments and finding, and conduct performance debugging and tuning of Flash-X with data analysis tools. Note that although accelerators play an important role in the overall performance of Flash-X for science, they are not relevant for this study. The performance anomaly that we describe was found in the CPUs, and the use of the network in scaling is the biggest performance concern of AMR in Flash-X, hence those are the performance aspects discussed

in this paper. For better understanding of the performance anomaly of Flash-X, we use the performance measurement tools TAU (Tuning and Analysis Utilities) Performance System [8] and PAPI [9] to collect the data and develop data analysis tools (a set of Jupyterlab [10] Python scripts) that utilize Pandas [11] and Plotly [12] to automate the generation of distribution and correlation visualizations. In this way, we discover and remove or mitigate two additional performance limiting bottlenecks for performance tuning.

The remainder of the paper is organized as follows: Section II describes Flash-X and its precursor FLASH along with the AMR packages supported in Flash-X, and Section III provides a description of the systems used in this study. The observed performance anomaly, and corresponding performance debugging are discussed in Section IV. Section V discusses the related work. Section VI gives our conclusions.

II. Flash-X

Flash-X framework is built with a combination of adapted architectural features from FLASH, and new features built from the ground up to support growing heterogeneity along orthogonal axes of hardware and application software. Hardware because of proliferation of special purpose devices such as accelerators and deeper memory hierarchy, and software because of higher fidelity models, need more various solvers.

A. Different Flavors of AMR

AMR is effectively a compression methodology for computations that do not need uniform resolution throughout the domain. It reduces both the memory footprint and the amount of computation. The technique has been around for more than 30 years and many flavors of its implementation have evolved over the years [13]. Two of them, Paramesh and AMReX, are of interest for our purpose because of being supported in Flash-X. Both use a logically rectangular collection of *cells* spanning a subsection of the domain as their basic abstraction. Paramesh and FLASH call this collection a *block*, while AMReX calls it a *grid*. Flash-X continues to the FLASH nomenclature, therefore, in this paper we call it a block.

Paramesh is the only available AMR support in FLASH and has been there since the inception of its code. It assumes that all blocks are identical in the number of cells along each dimension, and that the blocks are organized in an octree. Without any loss of generality we call the coarsest level of tree as the top level. Note that the top level can have more than one block, where the organization effectively becomes a collection of trees. Nodes from different trees can be adjacent to one another if their corresponding blocks are adjacent in the physical domain. Blocks at consecutive levels have parent child relationship; upon refinement a d-dimensional block generates 2^d child blocks with identical number of points, but half the spacing between the points. The union of all child blocks occupies the same area of the domain as the parent block. Figure 1 shows an example of an AMR mesh with its corresponding octree. Supported parallelism in Paramesh so far is exclusively MPI [14]. Load distribution uses Morton space filling curve where each block is assigned a weighting factor and a Morton number. Blocks are sorted and divided between MPI ranks based on their Morton numbers. The information about nearest neighbors and their refinement level for each local block is maintained in each MPI rank. Paramesh owns and manipulates the state data in a set of data structures that support cell-centered, face-centered and edge-centred discretizations.

AMReX is natively a patch based method. The fundamental difference between an octree and a patched based AMR is where a block can be placed relative to those in at the adjacent coarser level. AMReX does not have a parent-child relation between blocks and the only constraint on a finer level block is that it should be fully enclosed by a region at the adjacent coarser level, with at least on layer of coarser cells surrounding it. Note that the enclosing coarse region can be comprised of more than one block as long as they are next to each other. AMReX's data structures and algorithms follow a level-by-level approach. That is, the data structures for state variables span a level at a time. Thus for an n level mesh there will be n instances of space variables data structures. Similarly, space filling curve and load distribution are applied on each level independently of other levels. In Flash-X, AMReX mimics the appearance of an octree, that is, it accepts the constraints on having equally sized (in terms of data points) blocks and where they can be placed in the physical domain. However, it does not follow the parent-child relations and continues to manage all operations on a level-by-level basis.

In the context of using AMR in Flash-X it is obvious that even if the domain decomposition is identical in terms of blocks and their resolutions, everything else about the mesh management is different. The memory layout, the communication patterns, the load distribution, etc. are completely different between the two packages. Because of this Flash-X presents a unique opportunity to understand the behaviour of level-by-level vs the whole tree for producing identical scientific results.

Similar to FLASH, in Flash-X we continue to differentiate between the infrastructure, which knows about the platform details and manages all the data movement and layout, from physics, which primarily applies the numerical method on the data that it is handed. The separation is enhanced by further hiding any knowledge of data layout from physics through use of smart iterators. The hierarchy of decomposition is deeper in Flash-X through support of tiles in both the AMR packages. The framework has also been given enhanced interfaces to allow the AMR packages to be used in plug-and-play mode with the physics. The details of these modifications and the attendant refactoring of the code have been discussed in [15], [16].

B. Application Setup

All experiments are conducted for this study using the Sod [17] shock tube problem. It is a purely hydrodynamics application, where the domain is initialized with a discontinuity in density and pressure. In the configurations we used, the

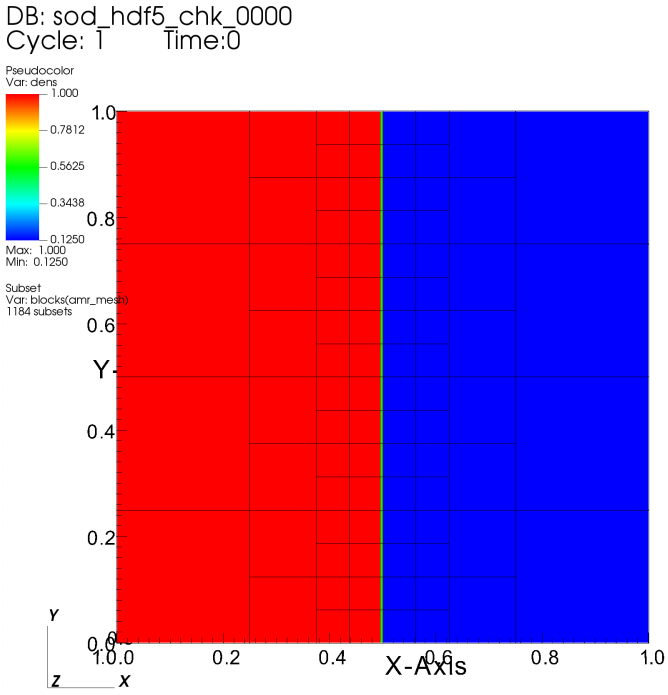

Fig. 1. A 2D slice of the initial 3D mesh for the Sod shock tube problem. Each of the outlined squares represents a block of 16^3 computational cells.

location of discontinuity is always a plane oriented perpendicular to one of the Cartesian coordinate directions. One half of the domain is at higher pressure and density than the other half, with a sharp drop in pressure at the interface. When the evolution begins, the discontinuity gives rise to a shock wave that travels perpendicular to the plane of discontinuity. The Sod problem is perfect for devising a weak scaling study for AMR because replicating the domain along an axis parallel to the plane of discontinuity as shown in Figure 2 increases the amount of work proportional to the replication factor. It is otherwise difficult to devise a weak scaling study for AMR codes, because increasing the work by a predetermined factor cannot be guaranteed by tweaking other parameters such as levels of refinement, change in domain size, etc.

III. Leadership Computing Platforms

We conducted our experiments on the IBM® POWER9™ heterogeneous system Summit [18] of approximately 200 petaflops peak performance at Oak Ridge National Laboratory and the Cray® XC40 Theta [19] of approximately 12 petaflops peak performance at Argonne National Laboratory. In this section, we briefly describe their specifications.

Summit has 4,608 IBM POWER System AC922 nodes. Each node contains two IBM POWER9 processors with the total 42 compute cores and six NVIDIA Volta V100 accelerators. Each POWER9 processor is connected via dual NVLINK bricks, each capable of a 25 GB/s transfer rate in each direction. Nodes contain 512 GB of DDR4 memory for use by the POWER9 processors and 96 GB of high-bandwidth memory (HBM2) for use by the accelerators. Additionally,

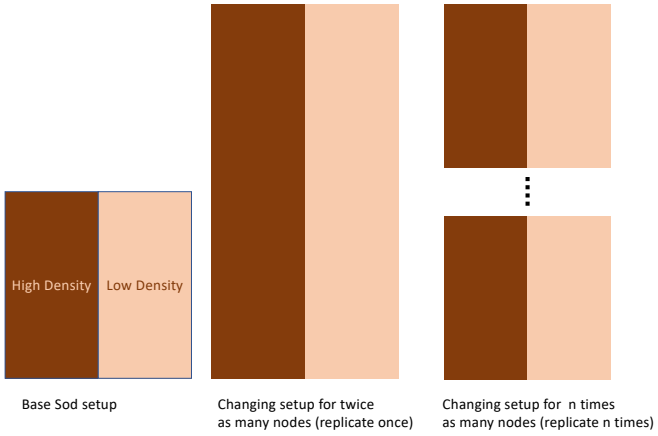

Fig. 2. Setting up the Sod shock tube problem for weak scaling by replicating the domain along one of the axes perpendicular to the axis of discontinuity.

each node has 1.6 TB of nonvolatile memory that can be used as a burst buffer. Summit is connected to an IBM Spectrum Scale filesystem providing 250 PB of storage capacity with a peak write speed of 2.5 TB/s.

Theta has 4,392 Cray XC40 nodes. Each node has 64 compute cores (one Intel Phi Knights Landing (KNL) 7230), shared L2 cache of 32 MB (1 MB L2 cache shared by two cores), 16 GB of high-bandwidth in-package memory, 192 GB of DDR4 RAM, and a 128 GB SSD. The Cray XC40 system uses the Cray Aries dragonfly network with user access to a Lustre parallel file system with 10 PB of capacity and 210 GB/s bandwidth. Theta provides two memory modes: cache and flat. Based on our experiments, we find that using cache mode outperforms using flat mode for Flash-X. In this paper, we conduct all experiments using cache mode on Theta.

IV. Performance Anomaly

The Paramesh Grid implementation used by Flash-X contains a subroutine, `mpi_amr_boundary_block_info`, which collects some information about the faces of blocks that coincide with the boundary of the simulation domain, and makes this global information available as a heap-allocated array on each rank. This subroutine is, by default, called once every time the AMR grid has changed. We noticed that the output of this subroutine, apparently intended to allow some specialized handling of boundary conditions, was not actually used anywhere, by either Paramesh or Flash-X. We therefore decided to remove calls to this routine (called from here on, for brevity, the "optional routine"), in order to eliminate any negative effect on weak scaling behavior from it. We found that rather than improving performance as expected, removing the calls had the effect of *slowing down* the code on some platforms. This is what we call "the anomaly".

The optional routine allocates scratch space that is some multiple of the number of local blocks that have any of their neighboring blocks on the boundary, and there are collective operations to share this information globally. In total, six scratch arrays are allocated, some of which persist through

the evolution step. Thus the relevant portions of the routine are allocations and global collective operations. Unlike most scratch spaces allocated in Flash-X which tend to be multiples of powers of 2 (because block sizes tend to be powers of 2), these allocations are sized by the number of blocks, therefore can have odd sizes.

The performance anomaly manifested itself on Theta when the calls to this routine were removed. While the time spent in communication decreased as expected, the overall evolution time increased non-trivially, in some instances by as much as 20% or more. Even more surprisingly, the degradation in performance was observed in the routine that computes Riemann states. This is a completely local routine, essentially an expensive stencil calculation, that has nothing whatsoever to do with the communication. Even more surprisingly, the behavior was reproducible not only on the same number of MPI ranks and therefore the same problem, but also across the entire weak scaling study. Similar behavior to a less dramatic extent was observed on the Bebop [20] cluster at Argonne National Laboratory. On Summit the effect is more subtle, yet exists as well. We could, therefore, eliminate the possibility that the observed behavior was due to variability inherent in the platforms because of workload differences at different times.

A. Performance Debugging

1) Approach: Based on the symptoms of the problems, we hypothesized that there was a side effect related to the allocation and de-allocation of memory in the optional routine. To test this hypothesis, we constructed an experiment in which Flash-X was configured with and without the optional routine call, labelled *orig* and *nocall* for the remainder of this paper. To minimize variability as much as possible, the two configurations were to be executed on Summit using identical input configurations and using the same allocation of hardware nodes. That is, we ran the simulations consecutively in the same job submission script. While Flash-X has built-in high level timers, we decided to use a performance measurement and analysis tool to gain deeper insight. Flash-X can be optionally integrated with the TAU (Tuning and Analysis Utilities) Performance System [8], in which the built-in application timers are replaced with TAU instrumentation calls. Because of this existing integration, we chose to use the TAU measurement library to collect the performance profile data.

TAU is a portable profiling and tracing toolkit for performance analysis of parallel programs and is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements, as well as event-based sampling. Both can be used at the same time, as a form of hybrid profiling [21]. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java Virtual Machine, or manually using the instrumentation API. As mentioned previously, TAU has been manually integrated into the Flash-X source code.

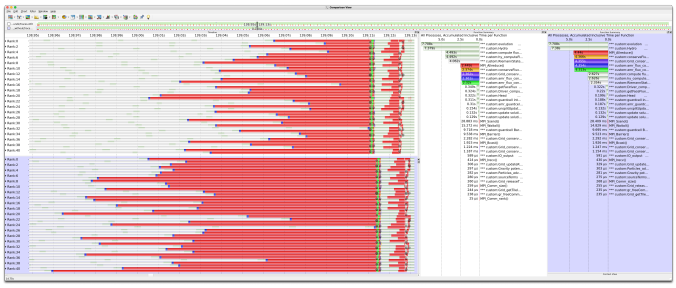

Fig. 3. Trace comparison between *orig* (above timeline, left profile) and *nocall* (bottom timeline, right profile) Flash-X on Summit, visualized in Vampir. The red regions represent time spent in `MPI_Allreduce`.

TAU's profile visualization tool, ParaProf, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. TAU also includes PerfExplorer [22], a data analysis framework for large scale profiles and for performing parametric studies such as ours. In addition, TAU can generate event traces that can be displayed with the Vampir [23], Paraver [24], or JumpShot [25] trace visualization tools. In addition to collecting performance traces or profiles of the application timers, TAU also provides the capability of measuring MPI calls using the MPI interposition wrappers, and is integrated with PAPI [9] to provide hardware counter measurements for detailed single-node analysis.

While the analysis tools that come with TAU and commercial trace analysis tools like Vampir are quite useful in many performance analysis cases, we wanted to explore the flexibility and performance provided by modern data analysis libraries in Python, such as Pandas [11] and Plotly [12] to understand the data distributions across the multi-dimensional profile data we were collecting. We were also trying to tease out relationships in the data that might lead us to the cause of the side-effect, so correlation analysis seemed to be a natural fit. We also needed the ability to re-run automated analysis in a predictable and portable way. For those reasons, we developed JupyterLab [10] analysis notebooks to assist in understanding dataset variability and to compare results between different configurations. For a majority of the analysis in this section we used these notebooks, leveraging the TAU profile parser library for Python that comes with the TAU installation to parse profile data into Pandas dataframes to visualize it.

To get an understanding of the problem, we began our analysis with interactive performance data visualizers. Initial comparisons of Flash-X compiled with PGI v19.9 on Summit showed that the differences were subtle. When directly comparing the mean times between the two runs in ParaProf, it appeared that the *nocall* version was roughly 2% slower than the *orig* version. Profiling with TAU showed that the performance difference seemed to be reflected by the `MPI_Allreduce` call in the `conserveFlux` call immediately after the computation in `custom:RiemannState`.

Figure 3 is a zoomed-in region of two different runs in

Vampir, focusing on the `conserveFlux` call and all of its children, including `MPI_Allreduce`. The *orig* run has a white background (upper timeline, left profile) and the *nocall* run has a lavender background (bottom timeline, right profile). Clearly, the `conserveFlux` call is "taking longer" because of a slight load imbalance in the main computation. A vast majority of the time is in the first `MPI_Allreduce` call during `conserveFlux`, and experienced MPI programmers know that a long time spent in the collective is likely due to a late arrival to the global collective operation.

Based on these early results, we concluded that something in the `mpi_amr_boundary_block_info` call is causing a side effect in the memory subsystem. We suspected that it is due to the memory allocations and de-allocations happening in that function, and has no relation to the MPI calls. For the Sod shock wave simulation, the peak memory usage without the call is roughly 6MB less than with the call, per process.

The next step was to capture cache-related hardware counters with PAPI and TAU to see if there was some correlation between the code in `mpi_amr_boundary_block_info` and overall cache performance, or whether the performance imbalance was due to an unlikely workload imbalance. To eliminate the possibility of a workload imbalance, the total instructions counter was also captured.

We added some additional code instrumentation to focus on what is happening to cause the imbalance in the main computation phase. Specifically, we added some instrumentation around the Riemann state calculation in `hy_getRiemannState.F90` (other regions showed no difference). What we discovered is that the call to `mpi_amr_boundary_block_info` is somehow making the L1 cache more efficient during this phase of the computation - for some ranks. Without that call, some ranks have a disproportionately higher number of L1 load misses, which causes an imbalance, which leads to a slightly longer synchronization time. For the worst rank, the L1 data cache misses were 60% more.

Figure 4 shows a comparison of the actual performance distribution of the `custom:RiemannState` subroutine between the *orig* and *nocall* configurations using 2688 MPI ranks on Summit. The charts were generated by our notebook analysis scripts. The *nocall* run was about 2% slower (452 seconds, instead of 443 seconds) overall. Figure 5 shows that there is a slight, but not significant difference in the number of instructions issued during `custom:RiemannState`, rejecting the possibility that the slowdown is caused by some kind of explicit work imbalance. Figure 6 does provide a confirmation that the L1 cache performance is related to the slowdown. Figure 7 shows that the `MPI_Allreduce` time distribution does go up in time (as does instructions), because MPI busy-waits at synchronization points. This is also confirmed by Figure 8, showing a negative correlation between the time spent in `custom:RiemannState` and `MPI_Allreduce`.

At this point, rather than adding more instrumentation of the sets of tight loops in the `custom:RiemannState` computation which could potentially perturb the measurement, we

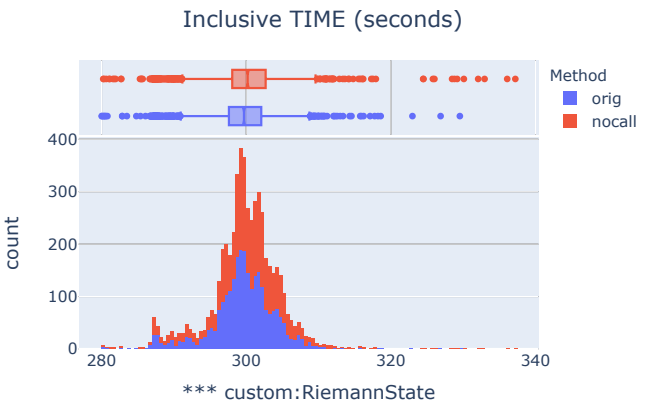

Fig. 4. Inclusive Time stacked histogram of the `custom:RiemannState` routine in Flash-X Sod simulation using 2688 ranks on Summit, with associated box plot. The *nocall* configuration has a longer tail towards outliers, suggesting an imbalance.

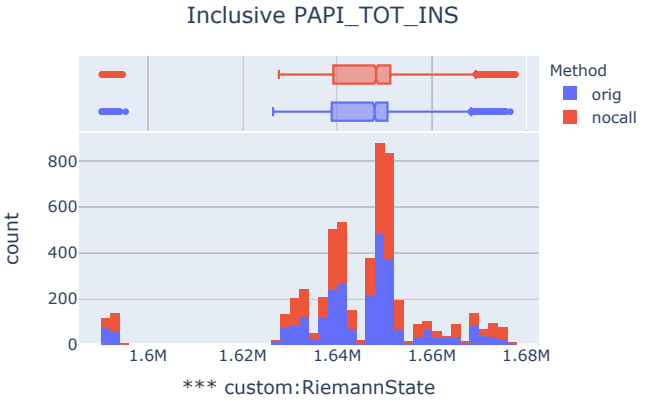

Fig. 5. Inclusive total instructions completed stacked histogram of the `custom:RiemannState` routine in Flash-X Sod simulation using 2688 ranks on Summit, with associated box plot. The distributions are quite similar, suggesting a balanced work distribution.

enabled sampling in TAU, collecting a hybrid profile of timers and samples. Figures 9 and 10 show a comparison between the hybrid measurements of *orig* and *nocall* in ParaProf, using a "tree table" view to provide context for the sample locations. The tree table is sorted by inclusive time. Even though there is some instruction skid (sampling is not always perfect), the full data shows that the relatively higher cache misses (both actual values and misses per instruction) are on lines 135, 136, 144, and 145 of `hy_upwindTransverseFlux.F90` - collectively shown in this view as a summary of the samples in that function. Those lines access the data in `allocatable` arrays declared in `hy_getRiemannState.F90` as `sig`, `lambda`, `leftEig` and `rghtEig`.

What is additionally concerning is the amount of memory management activity happening in this region of code. Nearly 163 seconds of the 251 seconds of the time is spent allocating and freeing (the top three sample locations in both runs)

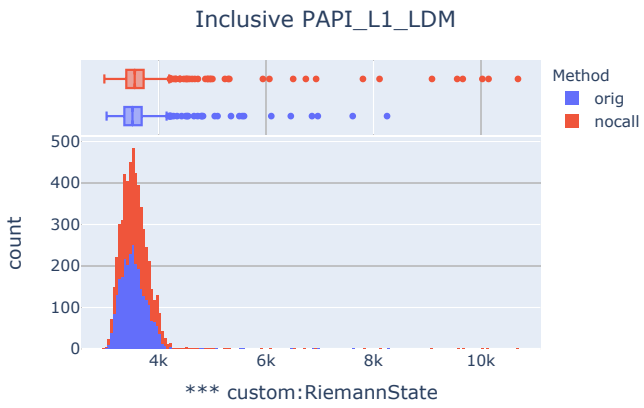

Fig. 6. Inclusive level 1 cache data load miss stacked histogram of the `custom:RiemannState` routine in Flash-X Sod simulation using 2688 ranks on Summit. The *nocall* configuration has a longer tail towards outliers, suggesting a possible cause for the imbalance.

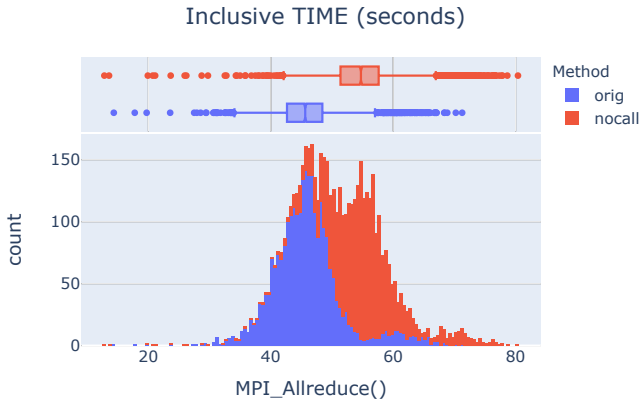

Fig. 7. Inclusive time stacked histogram of the `MPI_Allreduce` routine in Flash-X Sod simulation using 2688 ranks on Summit. The *nocall* configuration has a wider and shifted distribution relative to the *orig* configuration, indicating a negative correlation with extra time spent in `custom:RiemannState`.

`allocatable` arrays. We determined that it is worth to explore whether the arrays could be allocated once, and reused.

The results to this point led us to conclude that this particular scenario is taxing the memory subsystem, regardless of architecture. It is possible that a large number of synchronous processes per node (one per core), executing in lockstep, making frequent allocation requests at the same time would burden the OS kernel somewhat, regardless of architecture. It is also possible that the L1 cache misses are a misleading symptom, but are correlated with the true contention and are a helpful proxy. In general, we suspect that there are so many frequent, small allocation/deallocations happening that Flash-X is taxing the system allocator. In the next section, we analyze the performance of our code change on Theta, using the same Jupyter notebook scripted analysis. The performance differences between *orig* and *nocall* also exist on Theta, where

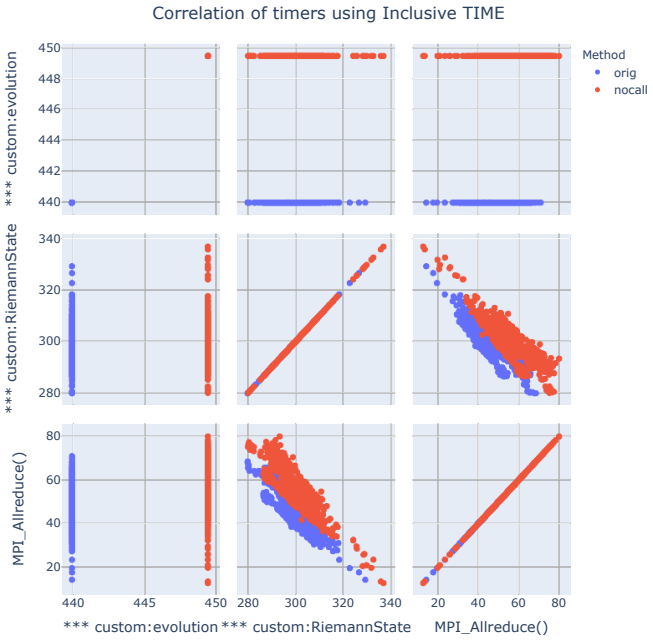

Fig. 8. Inclusive time scatterplot matrices of the top level timer (`custom:evolution`), `custom:RiemannState`, and `MPI_Allreduce` routines in the Flash-X Sod simulation using 2688 ranks on Summit. The *nocall* configuration has a wider distribution of `MPI_Allreduce` and `custom:RiemannState` relative to the *orig* configuration, and the matrix confirms the negative correlation between `MPI_Allreduce` and `custom:RiemannState`.

Flash-X is compiled with GCC v9.3.0. Having concluded that the performance anomaly is related to single-node properties, we limit our further experiments in this section to just one node.

2) Analysis: Having concluded that the increase in processing time was related to memory behavior, and having determined that the main computation was aggressively using `ALLOCATABLE` array variables and `ALLOCATE/DEALLOCATE` operations, we decided to address the possibility that concurrent processes could be contending for access to the memory subsystem. After some manual code analysis and discussion, we determined that the dynamically allocated array variables in the main computations of `hy_getRiemannState` and `hy_computeFluxes` were of a consistent size per process. That is, the dimensions of the variables do not change over time, so the memory could potentially be allocated once and reused. Three source files were modified to implement this change, replacing ALLOCATABLE array variables by global arrays of fixed size. We refer to this version of the code as *static*.

Figure 11 shows the distribution of `custom:RiemannState` for the *orig*, *nocall*, and *static* versions of Flash-X when executed with 64 ranks on one Theta node. The *static* implementation shows a considerable performance improvement in the reduction of outliers, but the mean performance across ranks has only reduced by less than 8%, from 151.48 seconds to 139.81 seconds. However,

Name	Inclusive TIME ▽	Inclusive PAPI_L1_LDM	Inclusive (PAPI_L1_LDM / PAPI_TOT_INS)
*** custom:hy_getRiemannState.calculating	254.807	2,729,301,359	0.002
[CONTEXT] *** custom:hy_getRiemannState.calculating	251.571	4,132,851,393	0.003
[SAMPLE] __GI___libc_free	68.855	1,083,900,360	0.003
[SAMPLE] __GI___libc_malloc	65.834	1,022,906,177	0.003
[SAMPLE] _int_malloc	29.298	456,492,454	0.003
[SUMMARY] hy_upwindtransverseflux_	24.84	402,237,951	0.003
[SUMMARY] hy_datareconstructnormaldir_mh_	13.62	220,462,336	0.003
[SUMMARY] hy_datareconstonestep_	9.42	146,145,069	0.003

Fig. 9. Hybrid profile of *orig* configuration, displayed in ParaProf (TAU). The main computation spends a considerable amount of time in memory management routines.

Name	Inclusive TIME ▽	Inclusive PAPI_L1_LDM	Inclusive (PAPI_L1_LDM / PAPI_TOT_INS)
*** custom:hy_getRiemannState.calculating	261.464	5,293,958,279	0.004
[CONTEXT] *** custom:hy_getRiemannState.calculating	257.779	5,990,283,496	0.004
[SAMPLE] __GI___libc_free	67.794	1,577,579,079	0.004
[SAMPLE] __GI___libc_malloc	67.32	1,404,165,584	0.004
[SAMPLE] _int_malloc	29.07	665,616,994	0.004
[SUMMARY] hy_upwindtransverseflux_	27.75	646,351,039	0.004
[SUMMARY] hy_datareconstructnormaldir_mh_	13.5	287,406,289	0.004
[SUMMARY] hy_datareconstonestep_	9.24	196,263,205	0.004

Fig. 10. Hybrid profile of *nocall* configuration, displayed in ParaProf (TAU). Relative to the *orig* configuration, the hy_upwindtransverseflux_ routine is taking longer, and reflects a 60% increase in L1 data cache misses, and a higher miss rate per instruction.

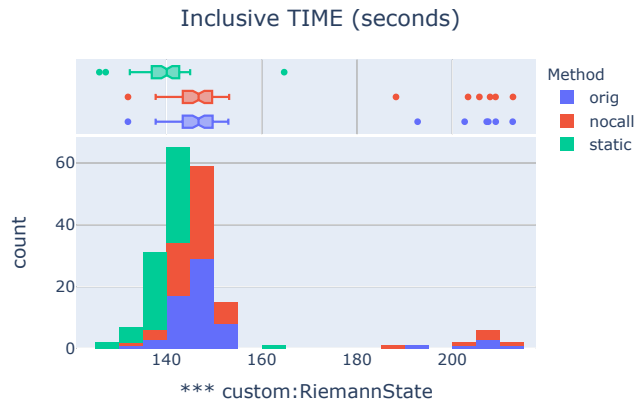

Fig. 11. Inclusive time comparison of custom:RiemannState in *orig*, *nocall*, and *static* configurations on Theta. The *static* configuration has a more compact distribution with fewer outliers.

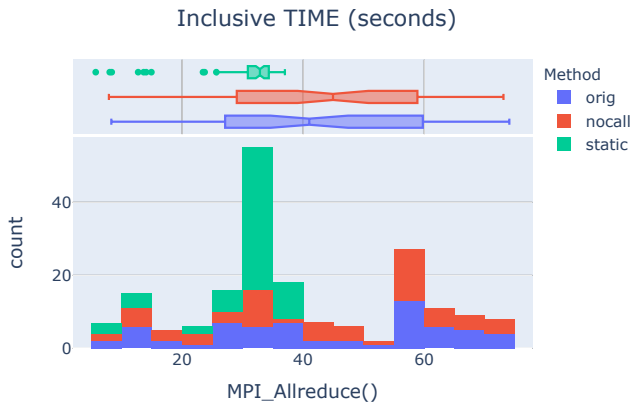

Fig. 12. Inclusive time comparison of `MPI_Allreduce` in *orig*, *nocall* and *static* configurations on Theta. The *static* configuration has a more compact distribution with fewer outliers, reflecting less synchronization delays.

the elimination in outliers reduced `MPI_Allreduce` mean time by nearly 31%, from 43.66 seconds to 30.18 seconds, as shown by Figure 12. Other MPI routines showed similar reductions in mean times. By eliminating the outliers, the overall runtime was reduced by 12.64% percent, from 385.3 seconds to 336.6 seconds. Figure 13 shows that the reduction in outliers is not due to a significant reduction in work performed in the `custom:RiemannState` subroutine. Instead, Figures 14 and 15 show that the overall reduction in time is correlated with a reduction in L1 data cache misses and a reduction in resource stalls.

B. Inferences

Figure 16 shows scatterplot matrices of inclusive time for the main function in Flash-X, `custom:RiemannState`, and `MPI_Allreduce` when executed with the three different configurations of *orig*, *nocall*, and *static*. While the result is encouraging, one visual detail sticks out in Figures 11, 14, 15, and 16. There is still one outlier with a long tail of time, L1 data cache misses and resource stalls. That outlier also happens to always be MPI rank 0. In fact, when running on multiple nodes, the lowest rank on each node shows a similiar behavior.

Our next hypothesis was that system noise on Theta is perturbing rank 0 (or more accurately, the lowest MPI rank on each node), causing it to be an outlier and increasing overall time in ReimannState compute. To test this hypothesis, we executed Flash-X with 2 nodes on Theta, enabling all PAPI (preset) counters at a high level (monitoring them with TAU periodically at a process level). Of the 128 total ranks, both

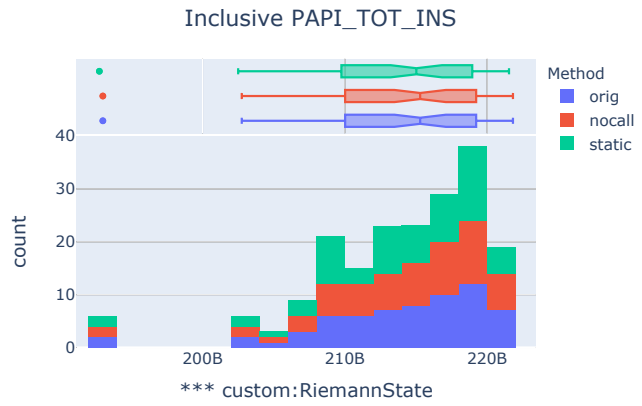

Fig. 13. Inclusive instruction comparison of `custom:RiemannState` in *orig*, *nocall*, and *static* configurations on Theta. The three configurations have roughly equivalent work distributions.

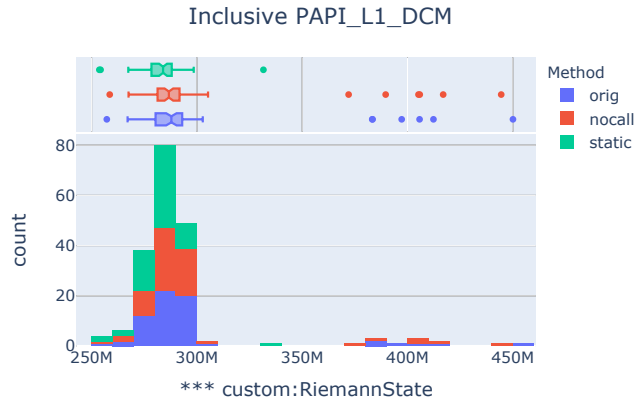

Fig. 14. Inclusive L1 data cache miss comparison of `custom:RiemannState` in *orig*, *nocall* and *static* configurations on Theta. The *static* configuration has a more compact distribution with fewer outliers.

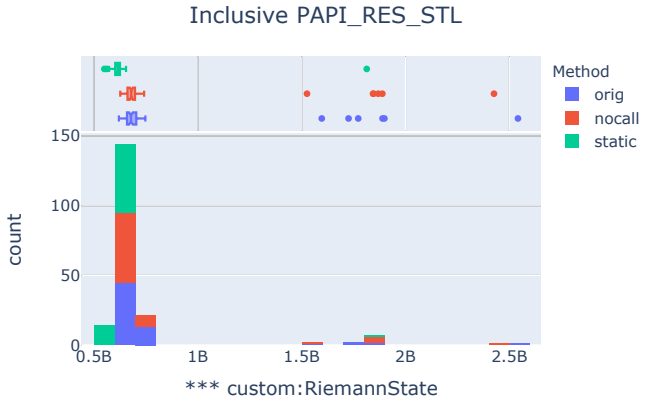

Fig. 15. Inclusive resource stall comparison of `custom:RiemannState` in *orig*, *nocall* and *static* configurations on Theta. The *static* configuration has a more compact distribution with fewer outliers.

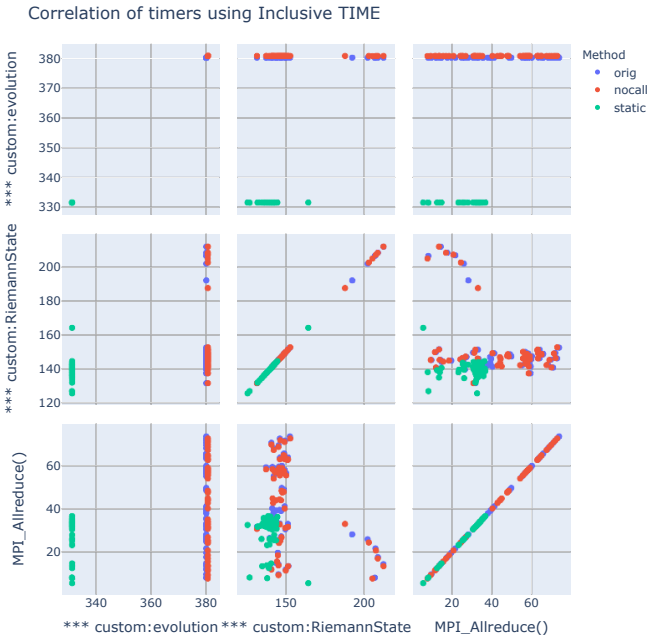

Fig. 16. Inclusive time scatterplot matrices of the top level timer (`custom:evolution`), `custom:RiemannState`, and `MPI_Allreduce` routines in the Flash-X Sod simulation using 64 ranks on Flash-X. The *static* configuration has a tighter distribution relative to the *orig* and *nocall* configurations, however there is still an outlier showing a negative correlation between `custom:RiemannState` and `MPI_Allreduce`.

ranks 0 and 64 were the worst performers. Evaluating the counters showed that L1 instruction cache misses, L1 data cache misses, and cycles without an instruction issue were all relatively higher for ranks 0 and 64, suggesting some cache pollution, likely due to contention for core 0.

Following instructions about specialization in [26], we configured our job on Theta to reserve one core of the node for system processes ("specialization"), and ran with 1 fewer MPI rank. We also configured a job to run with 1 fewer rank without reserving a core for system processes. The results show that `custom:RiemannState` is very well balanced now, and performs even faster - the total run time was reduced 18.5% to 314.09 seconds. The result is even more impressive when we factor in the fact that the 63 ranks of the *static* run each have 1/64 more work to do with 1 fewer rank in the allocation than the 64 rank *orig* run. Also interesting to note is that the *orig* and *nocall* configurations did not run faster when reserving a core for system processes - they ran in roughly the same amount of time. The job configuration with 63 ranks but did not use specialization was also faster than the original run, but only marginally.

As shown in Figure 18, the main takeaway is that on Theta, using 63 ranks per node and reserving one core for "specialization" (see `aprun` man page for the `-r` argument) improves Flash-X performance overall. The mean performance of `custom:RiemannState` is 1/64 worse, but the removal of the final outlier trims approximately 22 seconds off the

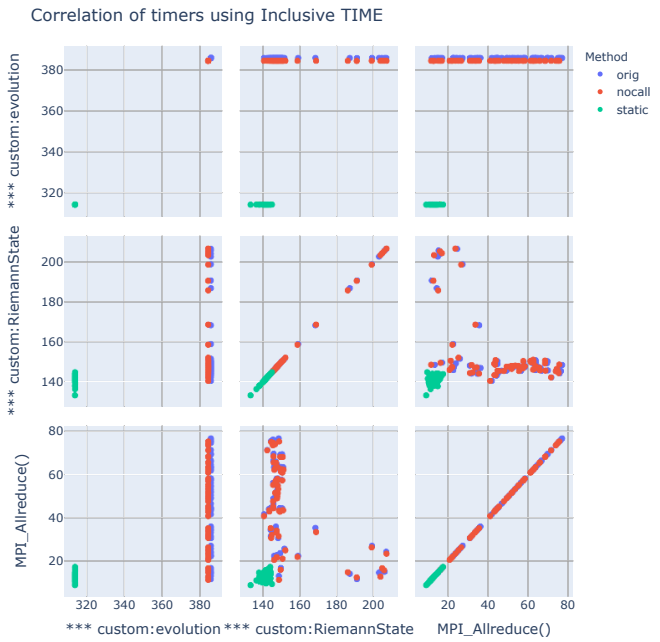

Fig. 17. Inclusive time scatterplot matrices of the top level timer (`custom:evolution`), `custom:RiemannState`, and `MPI_Allreduce` routines in the Flash-X Sod simulation using 63 ranks on Flash-X, reserving one core for system processes. The *static* configuration has a much tighter distribution relative to the *orig* and *nocall* configurations, and significant outliers are gone.

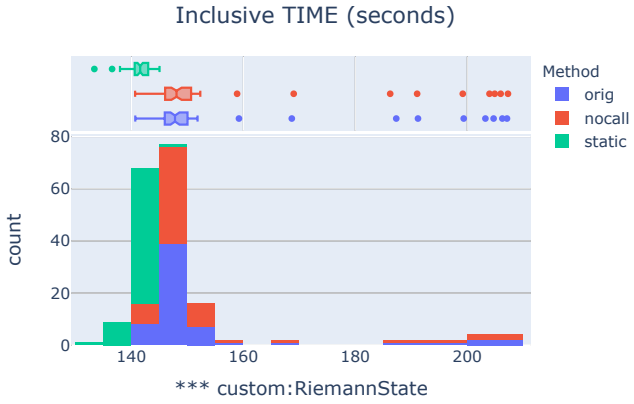

Fig. 18. Inclusive time comparison of `custom:RiemannState` in *orig*, *nocall* and *static* configurations on Theta, using 63 ranks and reserving a core for system processes. Compared with Figure 13, the *static* configuration no longer has the long tail outlier, reflecting even less synchronization delays. Interestingly, the *orig* and *nocall* versions did not benefit from reserving a core for system processes.

runtime.

V. Related

Ever since Python libraries and extensions like Pandas [11], [27], Plotly [12], [28], and Jupyter [29] have been made widely available, there is no shortage of users who have taken advantage of them to rapidly prototype analysis tools to tease information out of large multidimensional data sets. That said, there is specific performance analysis related and prior work to mention. The Java-based PerfExplorer [22] tool initially utilized the Octave [30] and later the Weka [31] data mining libraries to perform clustering and correlation analysis on large multidimensional performance data. The PerfExplorer GUI had some predefined analyses as well as a 'custom' chart interface, and an embedded – but limited – Python interpreter allowed for scripted analysis, but the workflow lacked flexibility and required an expert programmer to create new analyses or outputs. The Hatchet [32] library provides a Python library for parsing performance data from many different formats, including TAU. The Hatchet library introduced the *GraphFrame* data model and specializes in performance analysis on the individual nodes of an application's calling context tree. The GraphFrame provides a Pandas DataFrame entry for each unique node in the tree. While it could have been used as a data model, the standard Pandas dataframe import library that comes with TAU was sufficient for this analysis. The analysis developed in our Jupyter notebooks could easily and will likely be modified to also use GraphFrame data.

Finally, load imbalance has been a known problem since the introduction of parallel programming, and many tools have been developed to detect it, including EXPERT [33], CUBE [34], Paradyn [35], KappaPi [36], JavaPSL [37], and HPCToolkit [38], among others. The approach in this paper is not fundamentally different from those tools, however the visualization and analysis capability of Jupyterlab provides a new era of flexibility and extensibility.

VI. Conclusions

The extensive fine-grained performance analysis investigated in this paper was triggered by the non-intuitive outcome of an optimization effort. Our findings re-emphasize the notion that modern supercomputers and their software stacks have many moving parts that may not always interact with one another efficiently in a straightforward way. Increasing degrees of freedom in the optimization space make it much harder to explore, and reasoning about correlation between code modification and its performance change can become very difficult. Since there is no way to avoid continuing increase in platform and application complexity, the only available option is to facilitate meaningful communication between the system and the optimizer. For this reason, we developed a set of Jupyterlab Python scripts that utilize Pandas and Plotly to automate the generation of distribution and correlation visualizations for better understanding of performance anomaly of Flash-X. The scripts and associated software are open source, and freely available on GitHub [39].

Acknowledgements

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself,

and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan. This work was also supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) under contract DE-SC0021299.

REFERENCES

[1] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide, "Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code," *Parallel Computing*, vol. 35, no. 10-11, pp. 512–522, 2009.

[2] A. Dubey, K. Antypas, A. Calder, C. Daley, B. Fryxell, J. Gallagher, D. Lamb, D. Lee, K. Olson, L. Reid, P. Rich, P. Ricker, K. Riley, R. Rosner, A. Siegel, N. Taylor, F. Timmes, N. Vladimirova, K. Weide, and J. ZuHone, "Evolution of FLASH, a multiphysics scientific simulation code for high performance computing," International Journal of High Performance Computing Applications*, 28(2):225–237*, 2013.

[3] A. Dubey, L. B. Reid, and R. Fisher, "Introduction to FLASH 3.0, with application to supersonic turbulence," *Physica Scripta*, vol. T132, 2008, topical Issue on Turbulent Mixing and Beyond, results of a conference at ICTP, Trieste, Italy, August 2008.

[4] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[5] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[6] AMReX, https://amrex-codes.github.io/, Version 21.01-47, 2020.

[7] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, no. 3, pp. 330–354, 2000.

[8] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[10] P. Jupyter, "Project jupyter — home," http://jupyter.org, 2021.

[11] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, no. 9, pp. 1–9, 2011.

[12] Plotly, "Plotly: The front end for ml and data science models." http://plotly.com, 2021.

[13] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. V. Straalen, and K. Weide, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001178

[14] Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[15] J. O'Neal, K. Weide, and A. Dubey, "Experience report: Refactoring the mesh interface in FLASH, a multiphysics software," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, 2018, pp. 1–6.

[16] A. Dubey, J. O'Neal, K. Weide, and S. Chawdhary, "Distillation of Best Practices from Refactoring Flash for Exascale," SN Computer Science*, 1(4):223*, 2020.

[17] G. A. Sod, "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws," *Journal of Computational Physics*, vol. 27, no. 1, pp. 1–31, 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999178900232

[18] Summit, "IBM Power9 with NVidia V100 Heterogeneous System," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, 2020.

[19] Theta, "Cray XC40 System," https://www.alcf.anl.gov/theta, 2020.

[20] Bebop, https://www.lcrc.anl.gov/systems/resources/bebop/, 2020.

[21] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and implementation of a hybrid parallel performance measurement system," in *2010 39th International Conference on Parallel Processing*. IEEE, 2010, pp. 492–501.

[22] K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "Knowledge support and automation for performance analysis with perfexplorer 2.0," *Scientific programming*, vol. 16, no. 2-3, pp. 123–134, 2008.

[23] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[24] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1. Citeseer, 1995, pp. 17–31.

[25] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[26] S. Chunduri, "Run-to-run variability on theta and best practices for performance benchmarking," https://www.alcf.anl.gov/files/slides-chunduri-alcf-developer-session-2018-09.pdf, 2018.

[27] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, no. 1. Austin, TX, 2010, pp. 51–56.

[28] P. T. Inc. (2015) Collaborative data science. Montreal, QC. [Online]. Available: https://plot.ly

[29] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier, "The jupyter/ipython architecture: a unified view of computational research, from interactive exploration to communication and publication." in *AGU Fall Meeting Abstracts*, vol. 2014, 2014, pp. H44D–07.

[30] J. W. Eaton, "Gnu octave," https://octave.org, 2022.

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[32] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: pruning the overgrowth in parallel profiles," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–21.

[33] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An algebra for cross-experiment performance analysis," in *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE, 2004, pp. 63–72.

[34] F. Wolf and B. Mohr, *Automatic performance analysis of SMP cluster applications*. Citeseer, 2001.

[35] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[36] J. Jorba, T. Margalef, and E. Luque, "Performance analysis of parallel applications with kappapi 2." in *PARCO*. Citeseer, 2005, pp. 155–162.

[37] T. Fahringer and C. S. Júnior, "Modeling and detecting performance problems for distributed and parallel programs with javapsl," in *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. IEEE, 2001, pp. 38–38.

[38] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[39] K. Huck, "Python scripts for automating the analysis of tau profile data." https://github.com/UO-OACISS/analysis-scripts, 2022.