Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone ADCD-504-11

David Richards[1], Omar Aaziz[2], Jeanine Cook[2], Jeffery Kuehn[3], Gregory Watson[4],
Peter McCorquodale[5], William Godoy[4], Jenna Delozier[4], Mark Carroll[4],
Courtenay Vaughan[2], and The ECP Proxy App Team[6]

[1]Lawrence Livermore National Laboratory, Livermore, CA
[2]Sandia National Laboratories, Albuquerque, NM
[3]Los Alamos National Laboratory, Los Alamos, NM
[4]Oak Ridge National Laboratory, Oak Ridge, TN
[5]Lawrence Berkeley National Laboratory, Berkeley, CA
[6]https://proxyapps.exascaleproject.org/team

March 2021

LLNL-TR-821021

Contents

1 Executive Summary

The ECP Proxy Application Project has an annual milestone to assess the state of ECP proxy applications. Our FY21 milestone (ADCD-504-11) proposed to:

> Assess the performance and fidelity of proxy applications, including those in the ECP Proxy App Suite, relative to the ECP Application workload on heterogeneous platforms. Use proxy applications and selected ECP applications to assess the utility of critical elements of the Exascale toolchain, especially tools used to collect performance data. Identify gaps in coverage and/or common situations in which proxies may fail to adequately represent ECP applications.

To satisfy this milestone, the following specific tasks were completed:

- Study ability of MACSio to represent I/O workloads of adaptive mesh codes
- Prepare report on assessed performance and debugging tools
- Perform cosine similarity study on Intel and P9 and submit paper
- Complete LDMS/GPU subcontract, complete and receive report
- Complete validation of LDMS data collection for GPUs and submit paper
- Perform cosine similarity study on GPUs

This report presents highlights of these efforts.

Section 2 report on the evaluation of MACSio for the block structured AMR code, Castro. We found that output characteristics of our sample problem were significantly sensitive to various mesh-related input parameters. We also found that we can simulate some aspects of the non-linear trends in output size using the MACSio `data_growth` parameter. A more faithful representation of multi-step, multi-rank I/O patterns, especially with signifcant refinement enabled will require modification to MACSio.

Sections 3 and 4 describe a survey of ECP developers regarding the use of debugging and performance tools and our initial assessment of such tools. The survey results highlight the importance of performance and debugging tools to ECP developers. The results of the assessment are subject to NDA agreements. We are seeking permission from the corresponding vendors to publish our results.

Section 5 reports our cosine similarity study of 20 proxies and parent applications on Intel and P9 platforms. We find that most proxies show good similarity to their parents on both Intel and P9 CPUs with miniQMC/QMCPACK and XSBench/OpenMC as significant outliers. We also examine similarity of subgroupings of performance data such as cache-related or instruction mix-related counters. Curiously, these subgroups show high sensitivity to platform. For example, on Intel Skylake there is very little difference in instruction mix across the entire suite of applications studied. The same is not true on P9.

Section 6 describes the development and validation of an LDMS-based GPU performance sampler. This work proved considerably more challenging than initially anticipated and exposed bugs in the LDMS core as well as significant shortcomings in vendor documentation. We believe that sample provides accurate data, however we continue to observe some unstable behavior. Work to harden the tool as well as to extend capabilities to non-Nvidia GPUs is ongoing.

Finally, Section 7 presents our initial examinations of cosine similarity on GPUs. We find that both both the ExaMiniMD/LAMMPS and the sw4lite/sw4 pairs show poor similarity. This is surprising since the opposite is true on CPUs. However, these are very initial results and considerable additional effort will be needed to fully understand the root cause of the observed differences.

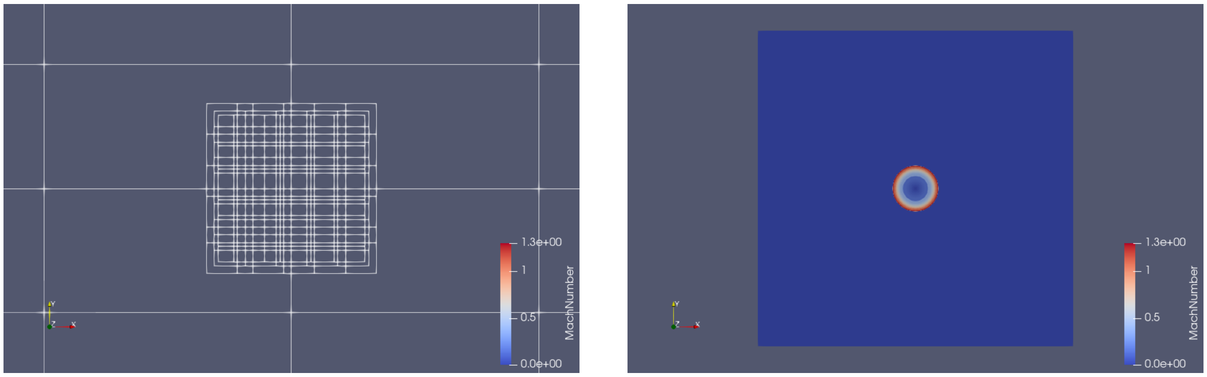
2 Evaluating MACSio as a Proxy Application for Block Structured AMR Codes

A typical approach to the numerical solution of partial differential equations (PDEs) is to divide the domain into a grid or mesh and estimate the values of the unknowns by solving the equations at each grid point. The grid spacing affects a variety of different properties of the calculation however, including the accuracy of the result and the number of calculations that need to be performed. In many situations there are regions of the grid where local properties require more computations to produce an acceptable result. The naive approach to this is to simply reduce the grid spacing until it is fine enough that the errors are within acceptable tolerances. However this can have a large impact on the overall computation since many additional calculations are being carried out in regions where they are not beneficial. Adaptive Mesh Refinement (AMR) was introduced to address this problem. It works by determining areas that require additional grid resolution and then creating finer subgrids over these regions. This process can continue until and acceptable level of refinement has been achieved for the region. The result is that additional computations are only required for areas where the refinement will have most benefit.

When characterizing the I/O behavior of an application, we typically look at the I/O generated as the result of data aggregation in the form of plots or other data files, or where intermediate computational results are stored so that the entire run does not have to be repeated if the application terminates for some reason (known as checkpoint/restart). In the case of regular grids, the I/O behavior (file size, number of files, read/write performance, etc.) would most likely be determined by a combination of the application size (number of grid points) and the data element size at each grid point. As an I/O proxy application, MACSio [20] can replicate the behavior of regular grids fairly easily since it has the ability to choose how data is distributed over the grid, how the grid is distributed over the cores/ranks of a parallel system, etc. However, it is not clear if I/O generated by applications employing AMR can be so easily be replicated, since as the refinement progresses, data is accumulated in certain areas of the grid, and computation is undertaken in a non-uniform manner. If the computation proceeds in this way, it seems reasonable to expect that the I/O behavior would also be non-uniform. If MACSio is to be used as a proxy application for I/O, it will be important that it has the capability to replicate behavior of this type.

Although there has been considerable research into the characterization of I/O performed by HPC applications, this is usually done using I/O benchamarking tools, I/O kernels, or a small number of real applications. It has been difficult to find any large-scale study that systematically characterizes I/O across a broad range of HPC applications, for example all of the Exascale Computing Project (ECP) applications. Even for the research that has been undertaken, there is disagreement as to what the typical characteristics are. For example, Shan *et al.* [29] determined that HPC application I/O is dominated by append-only writes, however Kim *et. al.* [16] found that there is only a marginal difference between read and write workload.

Although ECP applications utilize a number of AMR frameworks (AMReX [34], PARAMESH [19], Chombo [1], etc.) we chose to focus only on AMReX for this study as it is the most broadly used framework and is the focus of the Block-Structured AMR Co-Design Center in ECP. AMReX provides a general framework for solving PDEs with block-structured AMR algorithms using data containers and iterators for mesh-based data. The framework also provides native I/O support for writing plot files and checkpoints. AMReX enables a variety of parameters such as number of cells, refinement levels, number of time steps, gridding/regridding, plotfile/checkpoint frequency, etc. to be specified at runtime via a master configuration file. This makes it very convenient to perform a parameter study across a variety of input parameters. To further simplify the parameter study,



(a)

(b)

Figure 1: Sedov hydro case 2D cylinder in Cartesian coordinates pivot case used as a benchmark in this study showing (a) AMR mesh showing 4 levels, and (b) solution for the Mach number after 20 timesteps.

we also focused only on the Castro [3] astrophysical hydrodynamics code using a range of physics problem types.

We initially had three objectives for this study: analyze the I/O generated by a series of AMR-based ECP applications under a variety of input conditions and application sizes; establish a model (or models) of this behavior in order to determine the capabilities that would be required to accurately replicate the I/O behavior using a tool such as MACSio; and assess the ability of MACSio to reliably replicate the I/O patterns. The analysis and results of the study that was undertaken are presented in the following sections.

2.1 Methodology

In this section we describe the methodology and the model assumptions we used to characterize the output portion of the I/O from parallel runs of a typical AMR application. All test runs were performed on the Summit system operated by the Oak Ridge Leadership Facility (OLCF).

The goal is to first understand the role of I/O in an AMReX simulation rather than the complexity of the simulation and/or computation. In that regard, the Sedov 2D cylindrical case in Cartesian coordinates [1] was selected, as the test is readily available in the Castro suite of examples and shows a physical symmetry that we can use to isolate the AMR effects on the I/O on a simple problem. The AMR mesh levels used by Sedov are shown in Figure 1a, while Figure 1b shows the the Mach number after 200 timesteps illustrating the final symmetry of the solution.

The overall methodology we used for the study can be summarized in the following steps:

1. Run the existing Castro simulations as a point of reference for I/O generated by the AMReX infrastructure
2. Identify the important input parameters that drive the I/O simulation outputs to determine the basis for a variability study
3. Characterize the output generated given a variety of input parameters in Castro's configuration file

[1] https://github.com/AMReX-Astro/Castro/blob/main/Exec/hydro_tests/Sedov/inputs.2d.cyl_in_cartcoords

```
AMReX Castro Simulation Output
 ├── sedov_2d_cyl_in_cart_plt00000 (step directory - one per plot interval)
 │    ├── Header
 │    ├── job_info
 │    ├── Level_0 (level directory - one per level)
 │    │    ├── Cell_D_00000 (cell data file - one per rank)
 │    │    ├── Cell_D_00001
 │    │    ├── ...
 │    │    ├── Cell_D_0000N (N = number of MPI processes)
 │    │    └── Cell_H (mesh metadata file)
 │    ├── Level_1
 │    ├── Level_2
 │    ├── ..
 │    └── Level_L
 ├── sedov_2d_cyl_in_cart_plt00020
 ├── ..
 └── sedov_2d_cyl_in_cart_pltMMMMM (final step output)
```

Figure 2: Castro file plot analysis output structure for the Sedov 2D cylinder in Cartesian coordinates case.

4. Evaluate the I/O behavior of Castro and determine what functionality (if any) is missing from MACSio in order to replicate this behavior

2.1.1 Baseline Data Generation

Listing 1 in Appendix A shows a typical input file that contains the user input parameters used to drive the simulation. AMReX uses a "namespace" for input parameters, which are of the form `namespace.parameter`. Parameters in the `amr` namespace (including those without any namespace) are those affecting the main AMR functionality. Parameters in the `castro` namespace are specific to the Castro application.

The typical plot data output structure can be seen in Figure 2 showing a N-to-N pattern (i.e., each MPI rank writes to a separate file), where N is the number of MPI ranks, for the data produced at each refinement level of the simulation. Additional metadata is also produced at the top level in a file called `Header` and in each level directory in files called `Cell_H`. Note that a data file is only produced if there is data on a particular rank at the corresponding level. AMReX also supports the generation of checkpoint-restart data in a similar manner, but we focused on only the plot files for this study.

2.1.2 Data Model

In order to model the output of our selected baseline case we assume that data production can be characterized by the total number of dumped "domain sizes" objects. To illustrate this hypothesis, we assume that a problem input with size $Nx\,Ny$, where N is the number of cells per direction, and a certain *output_counter* (usually a function of the frequency *amr.plot_int* and maximum number of time steps *max_step* shown in Listing 1.) The rationale is that any data (and metadata) output must be proportional with the size of the problem at each requested output step. As a result, the independent variable in our model is expressed as a function of independent variables (x), and the

`amr.max_step`	maximum expected number of steps if not convergence
`amr.n_cell`	number of cells at level 0 in each direction
`amr.max_level`	maximum level of refinement allowed
`amr.ref_ratio`	ratio of coarse to fine grid spacing between subsequent levels
`amr.regrid_int`	how often to regrid
`amr.blocking_factor`	block factor in grid generation
`amr.max_grid_size`	maximum number of cells in each direction
`amr.plot_int`	frequency of plot outputs
`castro.cfl`	CFL condition

Table 1: Parameters used in the parameter study

dependent output sizes (y):

$$\begin{aligned} x &= \mathit{output\_counter} \times \mathit{total\_cells} \\ \mathit{output\_counter} &= 1, ..., \mathit{max\_step} \\ \mathit{total\_cells} &= Nx\,Ny \end{aligned} \tag{1}$$

$$\begin{aligned} y &= \mathit{data\_output}_i \\ i &= \texttt{time step, level, rank} \end{aligned} \tag{2}$$

2.1.3 Output Characterization

In order to understand the output behavior it is necessary to generate multiple runs of Castro with varying input file configurations. We performed this parameter analysis on the Summit [24] supercomputer using the message passing interface (MPI) [9] for parallelization. We used the input configuration file in Listing 1 as our baseline to understand the structure and size of the resulting output and to determine the input parameters that influence the output generation. Of particular interest are the parameters that influence the size of the problem, such as those in the REFINEMENT / REGRIDDING section and the Courant–Friedrichs–Lewy (CFL) [21] condition. Table 1 shows the parameters that we focused on for this study.

Each run generates corresponding data that can be used to quantify the cumulative output sizes at each requested time interval, output level, and rank (see Equation 2), as described in the hierarchy shown in Figure 2. The selected granularity allows for understanding the data production at the lowest single file level, while also giving an idea of how balanced (or not) the output is in a simple AMReX application.

2.1.4 Post-Processing Software: jexio

As part of this effort, we built software infrastructure to automate the post-processing data extraction and analysis from our parametric study. We were particularly interested in using the Julia Language [5] and its novel open-source language and integrated ecosystem designed for scientific computing and data science.

We called our project “Julia Extractor for eXtreme Input/Output” or “jexio”. The goal is to provide a reusable framework as we add more applications as described in Section 2.5. Details of

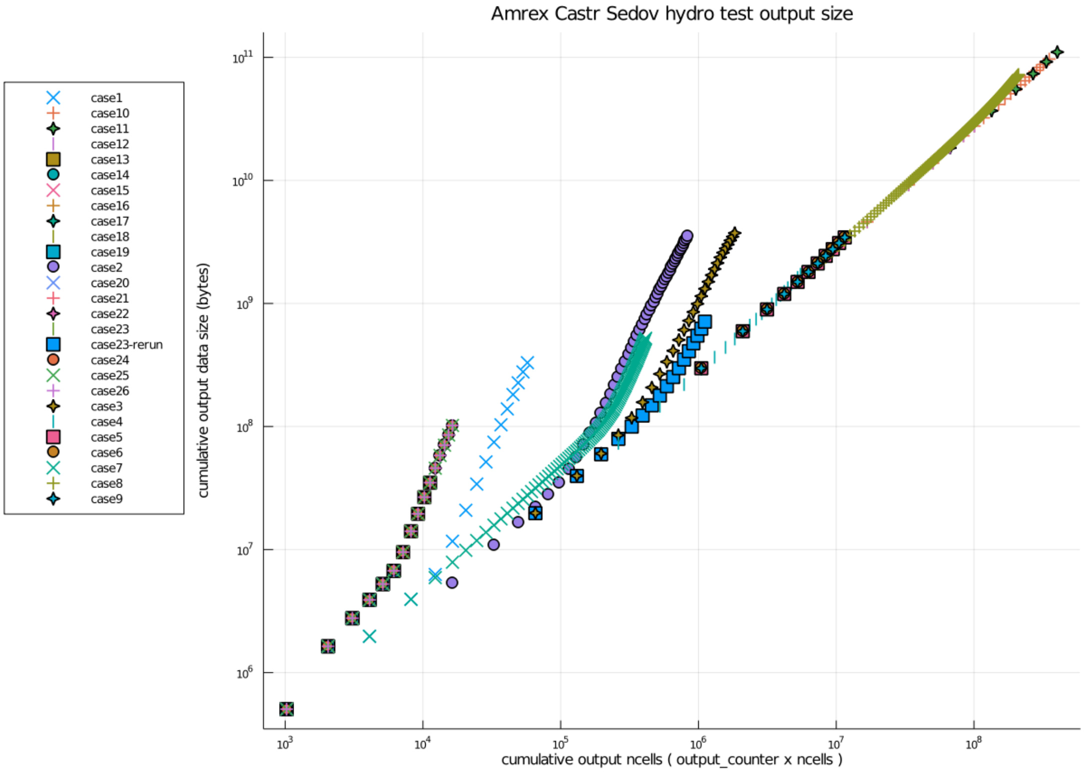


Figure 3: Cumulative output size as a function of the cumulative number of output cells as defined in Eqs. (1) and (2) for the Sedov 2D case.

the code and architecture can be found in our private GitLab repository [2] which is accessible upon request.

Once the data is generated, the "jexio" script is run to inspect a subset or all the outputs and generate a reduced comma separate (CSV) summary file that can be easily manipulated and analyzed in a DataFrame context. The data presented in Section 2.2 has been post-processed with this tool.

2.2 Results

The Castro Sedov hydro test generates a straight-forward I/O pattern in which each MPI rank outputs the data for each region, at each level, at each requested time interval in the simulation. Based on the resulting outputs, we can infer that this is done in a "burst buffer" [17] traditional pattern: the computation runs for some time, then the output is generated in a single "burst" for each time step requested in the input configuration file. Production of data is fairly regular on a step basis. Since AMReX provides two methods for writing plot file data, `WriteSingleLevelPlotfile` and `WriteMultiLevelPlotfile` it would appear that the latter is being used by Castro.

The cumulative output sizes as a result of varying the parameters in Table 1 are shown in Figure 3 for a variety of cases as a function of the produced output step (counter) multiplied by the total number of cells. It can be seen that several runs follow a very linear trend as expected for cases in which the mesh is constant. However there is another set of runs that deviate from this linear behavior that prompted further introspection and understanding of the output.

[2] https://code.ornl.gov/wfg/jexio

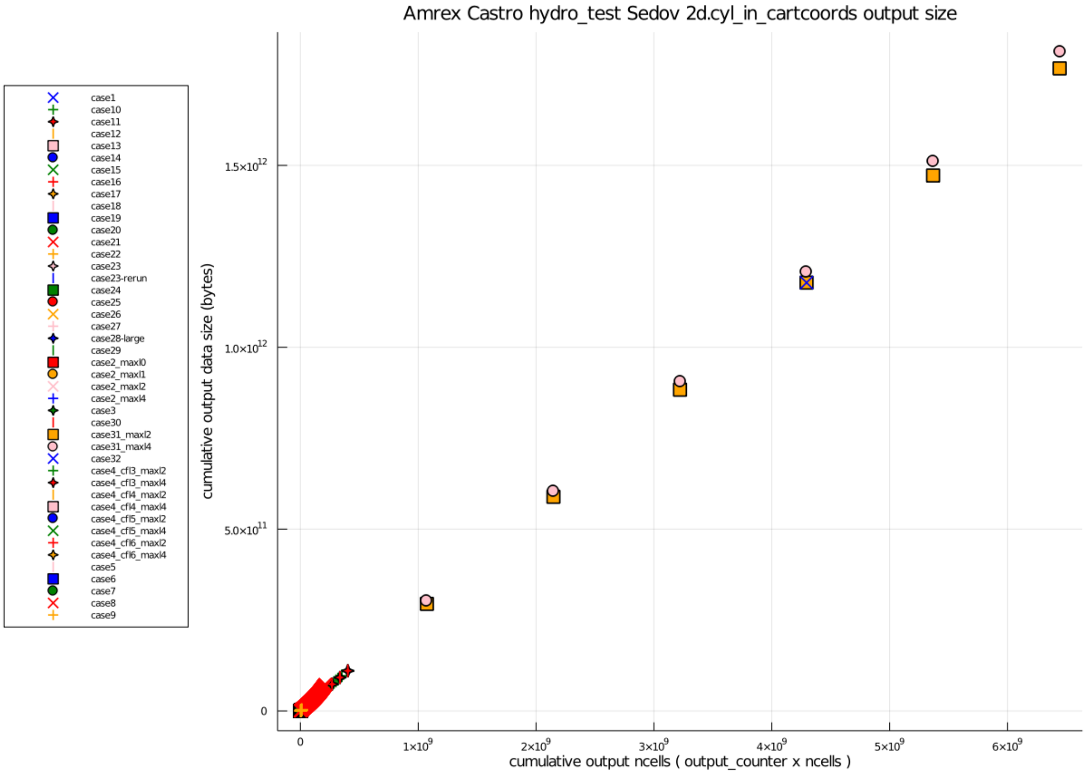


Figure 4: Cumulative output size for larger cases as a function of the cumulative number of output cells as defined in Eqs. (1) and (2) for the Sedov 2D case.

To confirm the observed trends in Figure 3 at larger scales we attempt to run extra "hero" runs. The results are shown in Figure 4 which confirm the linear trend of the smaller cases also shown in this figure.

We selected one of the cases as a pivot, labelled as case4 in our runs, with $N_x = 512$ and $N_y = 512$ containing 20 outputs. As shown in Figure 5, it is observed that while the CFL parameter has some influence on the overall output size, the number of AMR levels has a larger effect and explains the behavior on the other cases that deviates from the observed linear trend in Figure 3.

Once the effect on the overall data size output is inferred, further analysis is needed to understand how each point in the step output is distributed among AMR Levels and MPI processes. This is shown in Figure 6 for the pivot case (case4). As expected, the base (L0) level remains almost constant as it's mainly a function of the number of cells. Subsequent levels in the AMR refinement (L1, L2) are more sensitive as they are driven by physics (e.g., larger gradients) and the stability of the meshing algorithm (e.g., CFL condition).

The next level of granularity is to understand the output load as a function of the number of MPI processes. The latter will give an indication of the AMR effects on load balancing and the predictability of the I/O using tools like MACSio. Figure 7 illustrates the output variability across ranks for a large Sedov case generating outputs at 5 timesteps. As seen, AMR effects result is unbalanced loads at all 4 levels of the resulting mesh. Further investigation is needed to understand the relationship within AMR levels and MPI decomposition algorithms in AMRex and possible predictability, even in simple configurations such as the Sedov case.

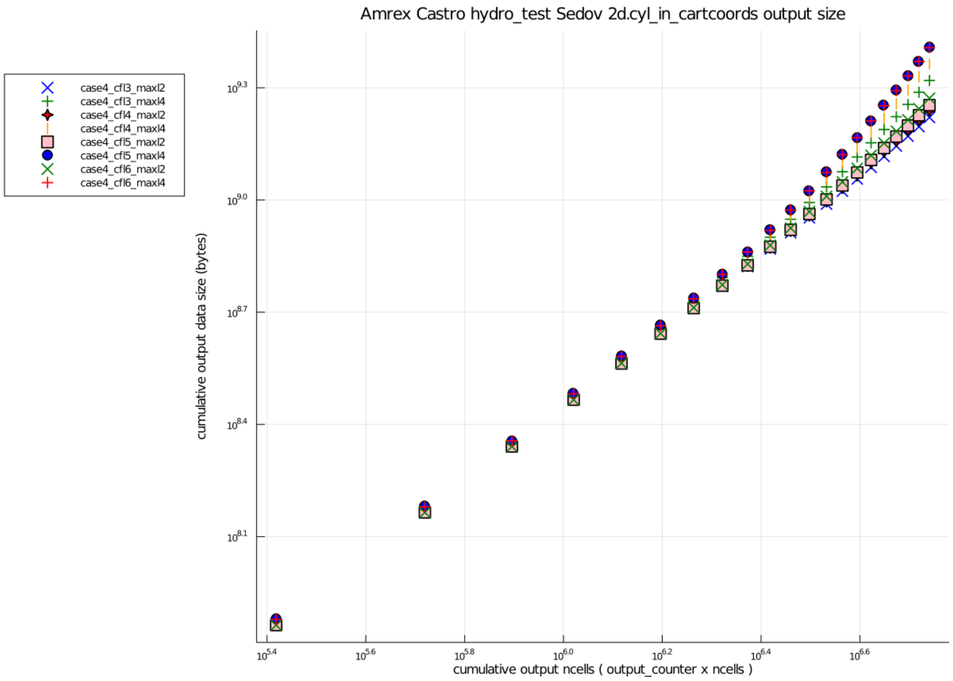


Figure 5: Dependency of the CFL and AMR Levels Cumulative output size as a function of the cumulative number of output cells for the Sedov 2D case.

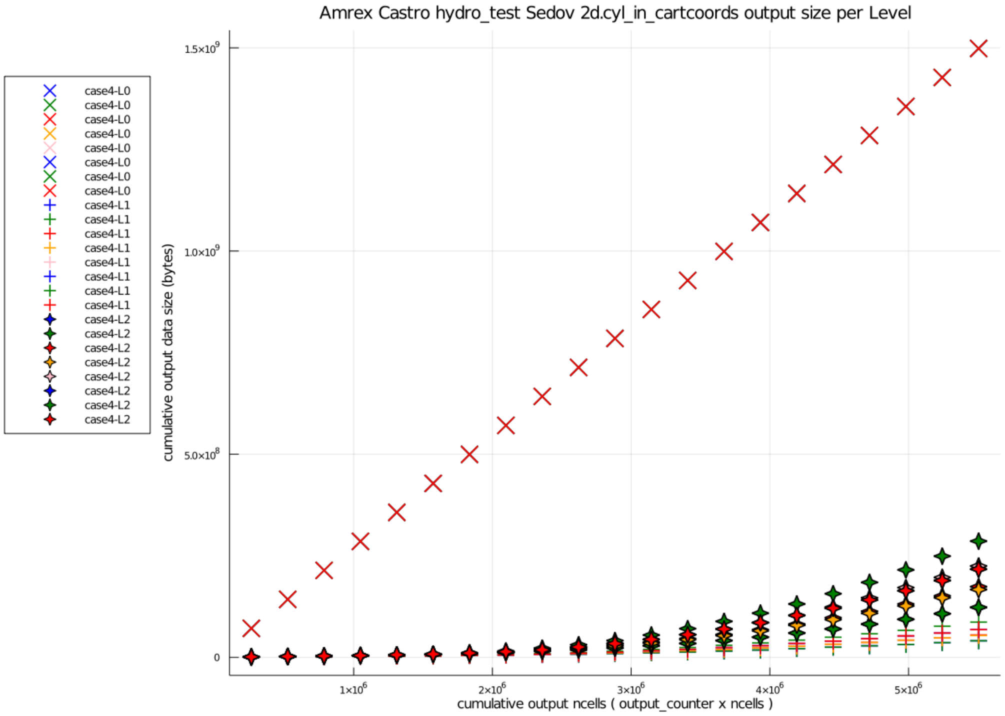


Figure 6: Dependency of the CFL and AMR Levels Cumulative output size for each AMR level (L0,L1,L2) as a function of the cumulative number of output cells for the Sedov 2D case.

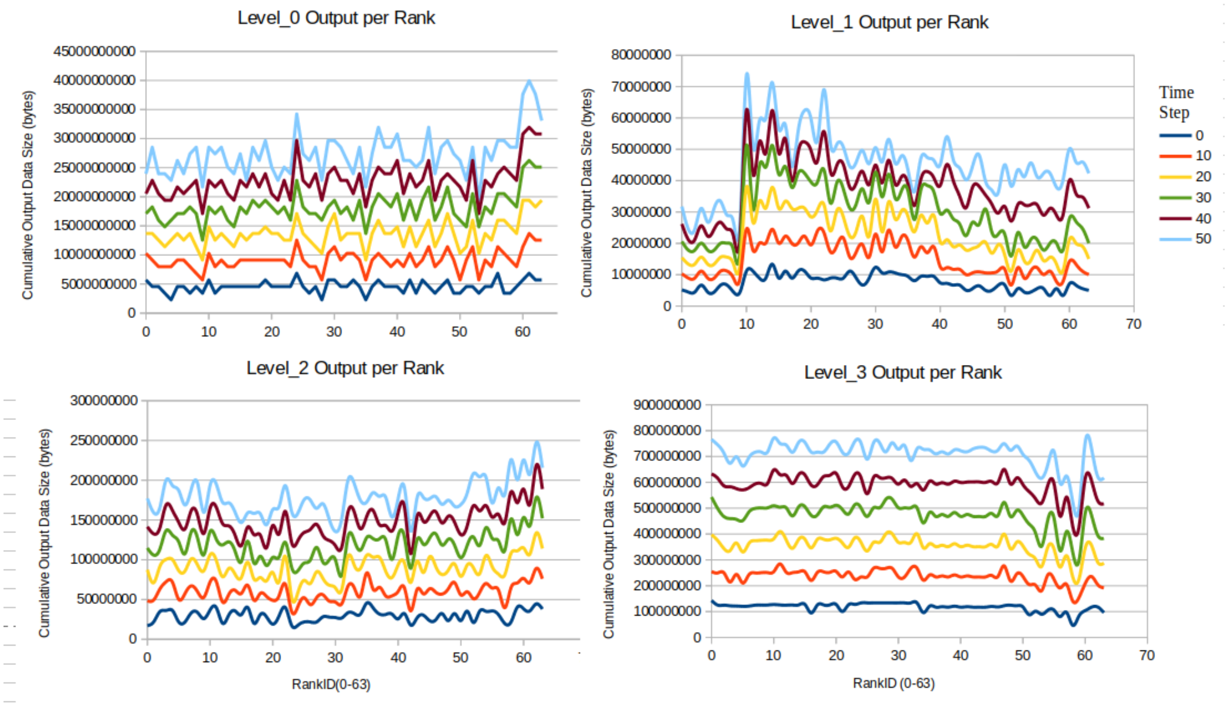


Figure 7: Output size distribution across different AMR Levels and MPI processes marked as RankID, for a large Sedov case using 64 processes.

2.3 Analysis

Our findings indicate there are several parameters that significantly influence the output characteristics. These parameters include the number of cells (`amr.n_cell`), output frequency (`amr.plot_int`), maximum level allowed (`amr.max_level`), and the Courant–Friedrichs–Lewy (CFL) condition (`castro.cfl`). Some of these parameters are more influential than others, with maximum level allowed influencing the output trend most and CFL condition influencing the output trend least (yet sill noticeably). When we compare the output sizes of a case with several different maximum level values, we observe just how significantly this parameter impacts the output sizes generated at each step of the run. The trends begin to approach a linear pattern as we decrease the allowed maximum level. Overall, as we refine the level of granularity we noticed a strong variability in the load balancing across MPI processors due to the evolving nature of AMR.

2.4 MACSio as Proxy I/O

After characterizing the AMReX I/O behavior, the next step was to provide a simple way to simulate the observed I/O patterns using a Proxy application. We used MACSio [20] due to its versatility for high-performance simulations. AMReX applications generate an I/O pattern that is dependent on timestep, refinement level, and output type, however MACSio only provides enough granularity to generate output on a per timestep basis. Initially, we sought to understand if MACSio was capable of providing an approximate solution that could simulate deterministic characteristics such as data size, computational overhead, and I/O patterns. This would allow us to construct a model that could help practitioners understand other random characteristics such as bandwidth, scalability, prioir to running a full AMReX application on different hardware platforms.

MACSio Argument	Description
`--interface`	output type hdf5, json (miftmpl), silo
`--parallel_file_mode`	File Mode: multiple independent, single
`--num_dumps`	number of dumps to marshal (buffer)
`--part_size`	per-task mesh part size
`--avg_num_parts`	average number of mesh parts per task
`--vars_per_part`	number of mesh variables on each part
`--compute_time`	rough time between dumps
`--meta_size`	additional metadata size per task
`--dataset_growth`	multiplier factor for data growth (1 is constant)

Table 2: MACSio command line arguments used to model AMReX-Castro outputs.

MACSio provides a simple command-line interface that captures and generates several types of I/O characteristics [20]. We used the N-to-N default output generation from the AMReX-Castro Sedov cases shown in Figures 3 and 5 to determine if MACSio can provide a valid approximation to AMReX-Castro I/O patterns. Table 2 shows the MACSio parameters we used in this study to fine-tune the data generation. We found the most important parameters were the interface, `parallel_file_mode`, `part_size`, and `dataset_growth`. This latter parameter enables MACSio to approximate non-linear data generation as some of the cases shown in Figure 3. The ultimately challenge, however, was to fine-tune this parameters to create a proxy I/O model to a level of granularity that is helpful in identifying I/O characteristics using MACSio.

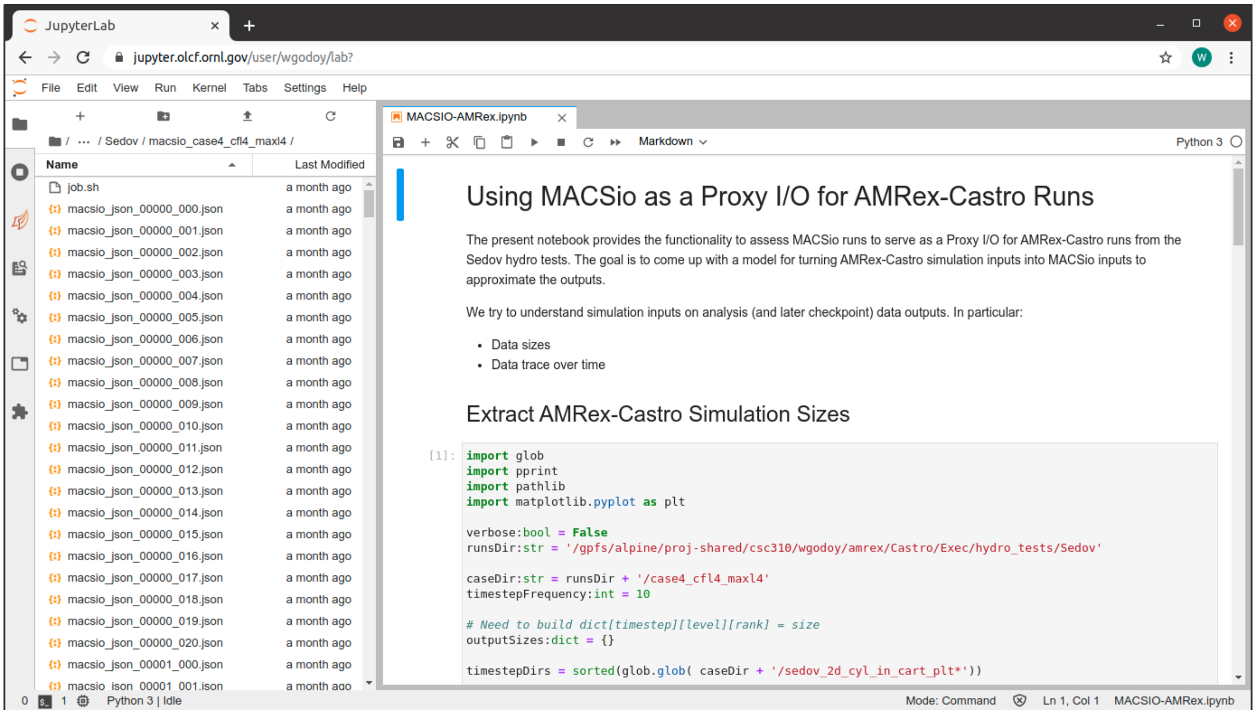
2.4.1 Implementation on JupyterHub

To build the MACSio-based model, we implemented a Jupyter Notebook on OLCF's recently available JupyterHub service [23]. JupyterHub allowed us to have scripts that can access the data directly in an interactive web-based graphical interface for data analysis. It also reduceed the need to move large amounts of data from our test cases on the Summit supercomputer. As can be seen in Figure 8, data was readily available to the notebook on the right hand side, so we were able interact with new MACSio runs as they become available in our parameterized study.

2.4.2 Non-linear data generation

In order to use MACSio to approximate non-liner behavior, we conducted a parameterized study using the case depicted in Figure 5 as an initial pivot for modeling data outputs per step with MACSio. Initial calibration was done to approximate the data sizes produced by AMReX-Castro, and the relevant parameters (listed in Table 2) were set to the initial data size at the first step. During this task we found that calibrated parameter `num_dumps` needed to be modified to match the observed data sizes in AMReX-Castro instead of just taking it as face value. In addition, the `data_growth` parameter was adjusted to approximate the final data sizes characteristics. This process is illustrate in Figure 9 in which the `data_growth` factor was optimized to match the resulting step based data output sizes in AMReX-Castro. It can be seen that while not a perfect fit, a value of `data_growth` = 1.013075 provides the "best non-linear fit" balances over and under predictions as observed in all MACSio simulations.

For completeness, the final data fitting is shown in Figure 10, where it can be seen that using MACSio as a proxy for I/O can provide oscillation around actual data size values. It is important to mention that while MACSio does not represent a perfect fit to AMReX-Castro, current capabilities



Name	Last Modified
job.sh	a month ago
macsio_json_00000_000.json	a month ago
macsio_json_00000_001.json	a month ago
macsio_json_00000_002.json	a month ago
macsio_json_00000_003.json	a month ago
macsio_json_00000_004.json	a month ago
macsio_json_00000_005.json	a month ago
macsio_json_00000_006.json	a month ago
macsio_json_00000_007.json	a month ago
macsio_json_00000_008.json	a month ago
macsio_json_00000_009.json	a month ago
macsio_json_00000_010.json	a month ago
macsio_json_00000_011.json	a month ago
macsio_json_00000_012.json	a month ago
macsio_json_00000_013.json	a month ago
macsio_json_00000_014.json	a month ago
macsio_json_00000_015.json	a month ago
macsio_json_00000_016.json	a month ago
macsio_json_00000_017.json	a month ago
macsio_json_00000_018.json	a month ago
macsio_json_00000_019.json	a month ago
macsio_json_00000_020.json	a month ago
macsio_json_00001_000.json	a month ago
macsio_json_00001_001.json	a month ago

Using MACSio as a Proxy I/O for AMRex-Castro Runs

The present notebook provides the functionality to assess MACSio runs to serve as a Proxy I/O for AMRex-Castro runs from the Sedov hydro tests. The goal is to come up with a model for turning AMRex-Castro simulation inputs into MACSio inputs to approximate the outputs.

We try to understand simulation inputs on analysis (and later checkpoint) data outputs. In particular:

- Data sizes
- Data trace over time

Extract AMRex-Castro Simulation Sizes

```
[1]: import glob
     import pprint
     import pathlib
     import matplotlib.pyplot as plt

     verbose:bool = False
     runsDir:str = '/gpfs/alpine/proj-shared/csc310/wgodoy/amrex/Castro/Exec/hydro_tests/Sedov'

     caseDir:str = runsDir + '/case4_cfl4_maxl4'
     timestepFrequency:int = 10

     # Need to build dict[timestep][level][rank] = size
     outputSizes:dict = {}

     timestepDirs = sorted(glob.glob( caseDir + '/sedov_2d_cyl_in_cart_plt*'))
```

Figure 8: Data analysis implementation on OLCF's JupyterHub [23] comparing AMReX-Castro Sedov case and several MACSio parameterized runs as a Proxy for I/O.

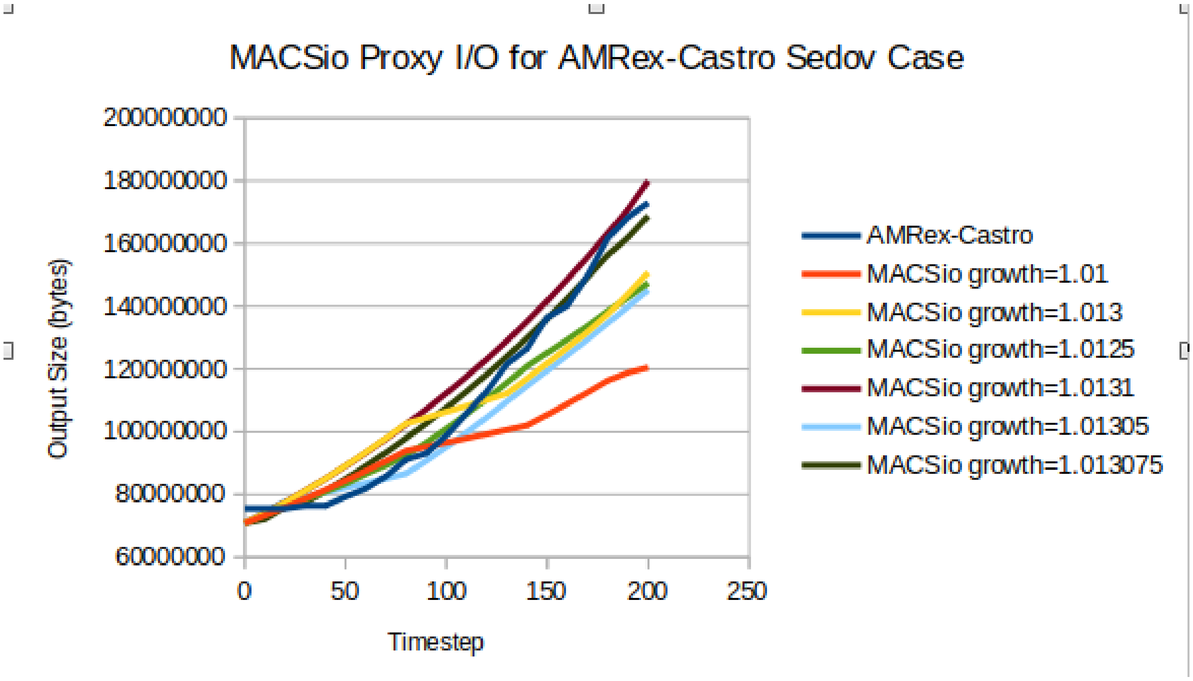


Figure 9: Data growth optimization of MACSio simulation matching the AMReX-Castro case in Figure 5 for step level data size outputs.

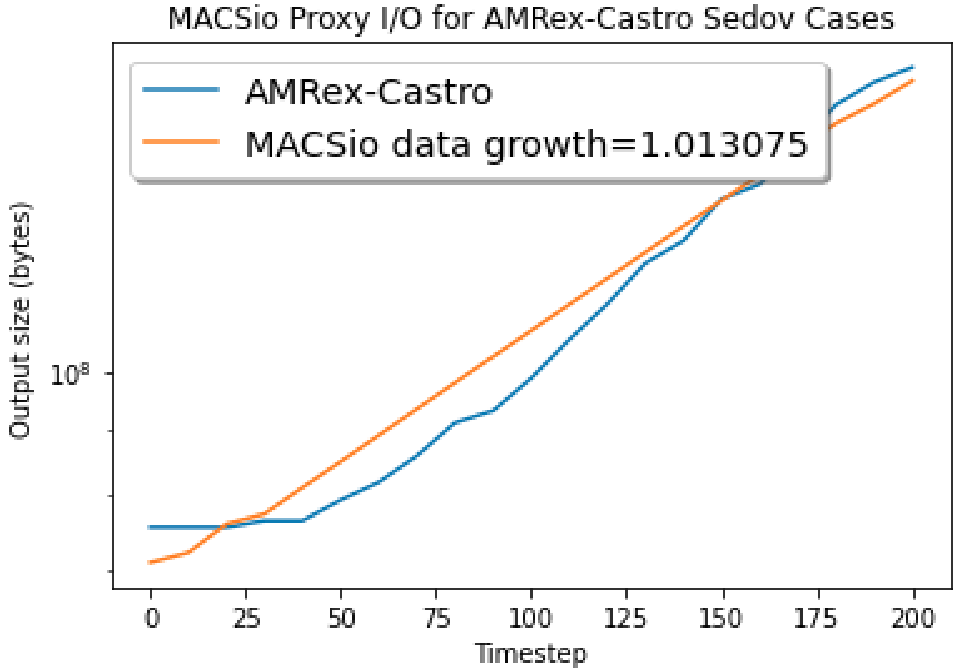


Figure 10: Final MACSio simulation “best fit” to the AMReX-Castro case in Figure 5 for step level data size outputs.

provide a level of confidence to simulate non-linearities observed in the Sedov case illustrated in Figure 1. Therefore, MACSio provides a readily available capability to simulate AMReX I/O patterns of similar characteristics.

The parameters used by the MACSio run for the final fit shown in Figure 10 are provided in Listing 2 in Appendix A. Once the parameters are set for data output sizes, we run an optimization on the `data_growth` value to fit the non-linear growth of AMReX-Castro data outputs shown in Figure 9. This value needs to be greater than 1 in order to create a non-linear data growth characteristic.

2.5 Future Work

Future work includes extending the present methodology to include checkpoint-restart data generation. In addition we hope to include other physics models provided by Castro, as well as other Exascale AMReX based applications (such as IAMR, MAESTROeX, MFiX-Exa, Nyx, PeleC, PeleLM, WarpX), in addition to other non-AMReX AMR-based codes. In terms of methodology, we hope to leverage the current “jexio” infrastructure and apply machine learning (ML) algorithms to train the characterization models as more datasets become available. We expect a low-effort at the higher-granularity level as these have shown more predictable patterns, while the low-level granularity levels show stochastic I/O load balancing requiring further study. Furthermore, extending MACSio as a Proxy I/O solution requires more testing to guarantee the coverage of a potential model that relates AMReX inputs to those of MACSio for a simple way to predict and study I/O trade-offs without the need to run full AMR simulations.

2.6 Conclusion

In this study, we looked at the data output characteristics of a single AMR-based application, Castro, under a variety of input conditions. The results show that MACSio in its current state can provide certain proxy I/O capabilities to a "per-step, per-rank" output size. MACSio would need to be modified if more granularity is required to model outputs corresponding to more refined AMR levels, although this becomes more complex as it is problem-dependent including domain decomposition strategy and mesh partition.

To extend MACSio to support this functionality, the following activities are required:

- gain a better understanding the output characteristics of a broader range of AMR-based applications
- determine if AMR output characteristics could be modelled in a simplistic manner, or if the output is largely driven by the computation being undertaken

Nevertheless, MACSio provides enough capabilities for initial studies of AMReX I/O characteristics without running a full AMR simulation. The latter is useful as storage systems evolve and could potentially enhance the decision-making when producing data at scale using AMR-based simulations.

3 Survey of Tools for Performance Analysis and Debugging

The Exascale Computing Project (ECP) is a multi-year, multi-institution project to develop the next generation of hardware and software required to solve some of the world's most complex scientific problems. The large number and complexity of the different codes, and the cutting edge nature of the computing hardware, creates an incredibly challenging environment for the performance analysis and debugging tools used by application developers. To identify some of the bottlenecks and issues facing developer's who work on these codes, we conducted a survey of Application Development teams, asking them to rate the importance of various functionalities, and their satisfaction with, the existing performance analysis and debugging tools they are currently using. Our hope is that this information can be used to address deficiencies in the development tools that are available to the scientific computing community.

3.1 Methodology

This survey was done on SurveyMonkey over two iterations, the first in November of 2020, and the second in January of 2021. The full survey form is shown in Appendix B of this report.

3.1.1 Initial Survey

What we sent: The first iteration of the survey was sent via a common link emailed to the `ecp-ad-pis` list by a list administrator. The members of this list are the Principal Investigators (PIs) of the Application Development (AD) teams of the Exascale Computing Project (ECP). The email, shown in Figure 11, asked PIs to forward the survey link to members of their project teams. The email was sent on November 13, and the survey closed on November 30. No reminders were sent.

The response: Only five responses were received. Since there are hundreds of ECP AD team developers, we figured that the low response rate indicated that few of these developers had even seen the survey, likely because not many PIs forwarded the surveys to their team members. It is impossible to know which PIs forwarded the survey, or how many did. However, the five who did respond reported being on different AD teams in Question 1 of the survey (Section 3.3.1 of this report). We do not know the identities of those who responded to the survey, but SurveyMonkey does give us their IP addresses.

```
The ECP Proxy Applications team is seeking input from ECP applications
developers to determine how well current and emerging performance
analysis tools and debugging tools meet ECP application requirements,
and to identify deficiencies and gaps.

So in order for us to obtain feedback from as broad a group as
possible, could you please forward this survey link to the members of
your ECP application development project team?  The survey closes on
November 30, 2020:
https://www.surveymonkey.com/r/CTP2KSH

Peter McCorquodale and Greg Watson, for the ECP Proxy Applications team
```

Figure 11: Initial email sent to `ecp-ad-pis` on November 13, 2020.

3.1.2 Second Survey

After the low response rate of the first survey, which relied on PIs fowarding it to their teams, we decided to send the same survey directly to the team members. To do this, we scraped email addresses from the the rosters listed for the 33 AD teams on the ECP website[3], where such rosters were available. When a team's pages on the website did not list contact information for members (such as LatticeQCD) we took email addresses from ECP 2020 annual meeting posters. In the case of the Proxy Applications project, we obtained email addresses from the team's own external website.

What we sent: We ended up with 613 email addresses that were provided to SurveyMonkey. SurveyMonkey sent a different link to each recipient, so responses could be tracked individually. The survey was sent on January 11, with automatic reminder emails sent out on January 25, and closed on January 30. The email had the same wording as the previous email in Figure 11, with the updated deadline date, and ending with: "`If you have already completed this survey, then thank you for your responses.`"

The response: According to SurveyMonkey, 27 (4.4%) of the 613 emails bounced, and 10 (1.6%) opted out of all emails from my SurveyMonkey account, so that left 576 email addresses that could possibly have responded. Out of these 576, there were 36 responses (6.3%).

3.2 Combined Responses

We consider it unlikely that anyone who completed the first survey would have also completed the second survey, so for the purposes of this report we have combined the 5 responses from the first survey with the 36 responses from the second, for a total of 41 responses. This supposition is supported by the fact that none of the IP addresses of those who completed the second survey matched any of those who completed the first survey.

Table 3 breaks down by Internet domain the email addresses that were taken from the ECP Confluence website and used as targets for the January survey, and the number of these targets who responded to the survey. The breakdown by the three major categories of Office of Science labs, National Nuclear Security labs, and Other is very similar among those surveyed (53.5%, 24.0%, and 22.5%) and among those who responded (58.3%, 22.2%, and 19.4%, respectively). The number of respondents by each individual domain is too small to be able to draw conclusions about representativeness, but there was at least one response from every lab that had 10 or more targets.

3.3 Results

In this section, we present a summary of the responses that were obtained for each of the survey questions.

3.3.1 Question 1: Project Identification

The first question on the survey asked:

> 1. Which ECP Application Development project(s) are you working on? Check all that apply.

There were 32 choices, as shown in the Appendix. Table 4 shows the distribution of responses to Question 1 on AD team affiliation, classified according to ECP's six application categories. The second-last column in the table gives the number of email addresses scraped from the team's web

[3] `https://confluence.exascaleproject.org/`

		targets		respondents	
	domain	#	%	%	#
Office of Science Laboratories:		**328**	**53.5%**	**58.3%**	**21**
Argonne National Lab	anl.gov	87	14.2%	8.3%	3
Brookhaven National Lab	bnl.gov	18	2.9%	5.6%	2
Lawrence Berkeley National Lab	lbl.gov	78	12.7%	16.7%	6
Oak Ridge National Lab	ornl.gov	78	12.7%	11.1%	4
Pacific Northwest National Lab	pnnl.gov	35	5.7%	11.1%	4
Princeton Plasma Physics Lab	pppl.gov	22	3.6%	5.6%	2
SLAC National Accelerator Lab	slac.stanford.edu	9	1.5%	0.0%	0
Thomas Jefferson National Accelerator Facility	jlab.org	1	0.2%	0.0%	0
National Nuclear Security Administration Laboratories:		**147**	**24.0%**	**22.2%**	**8**
Lawrence Livermore National Lab	llnl.gov	55	9.0%	11.1%	4
Los Alamos National Lab	lanl.gov	48	7.8%	8.3%	3
Sandia National Laboratories	sandia.gov	44	7.2%	2.8%	1
Other:		**138**	**22.5%**	**19.4%**	**7**
National Energy Technology Lab	netl.doe.gov	6	1.0%	0.0%	0
National Renewable Energy Lab	nrel.gov	21	3.4%	2.8%	1
National Institutes of Health	nih.gov	3	0.5%	0.0%	0
National Institute of Standards and Technology	nist.gov	1	0.2%	0.0%	0
	.edu (not slac.stanford.edu)	77	12.6%	13.9%	5
	.com	27	4.4%	2.8%	1
	.net	2	0.3%	0.0%	0
	.org (not jlab.org)	1	0.2%	0.0%	0
TOTAL:		**613**			**36**

Table 3: Distribution by Internet domain of targets and respondents to the January survey.

pages on the ECP Confluence website and used as January survey targets. Their total over all teams (or over all teams in an application category) exceeds the number of survey targets, because many people are on more than one team. Although Applications Assessment was not listed as a choice on the survey, because that project has ended, it is included in the table because our January survey targets included email addresses from its web pages on the ECP Confluence website.

From the number of team members who were sent the January survey, and the number who responded and clicked on the box for each team, we are able to calculate a response rate for each team on the January survey, and this rate is given in the last column of Table 4. No respondent checked any of the boxes for the three National Security applications. Response rates for the other categories were relatively low for Data Analytics and Optimization (4.2% of 144 emails sent) and Earth and Space Science (4.7% of 85), and relatively high for Chemistry and Materials (8.7% of 104) and Energy (9.8% of 153). Co-Design, the largest category with 203 email addresses over eight teams, was in the middle with a 7.4% response rate.

Question 1 also included a box for "Other: specify below." There were three responses given here, one each for ExaWorks and PETSc/TAO, which are ECP Software Technology teams, and one for Uintah, which is not an ECP project.

Discussion

The results in Table 4 show that the survey respondents are broadly representative of the ECP AD teams, except for National Security applications. Of the 29 non-National-Security teams, 25 were checked at least once over the two surveys, the four exceptions being EXAALT, ExaSky, Subsurface, and Urban. In the rest of this report, we have chosen not to break down survey responses by AD team or category, because the number of responses for each AD team is far too small, and even grouping by the five represented application categories, the applications within a category often use very different methods, so any conclusions drawn from breaking down responses by category are likely to be spurious.

	Total	November	———— January ————		
	responses	responses	responses	emails sent	response rate
Chemistry and Materials:	11	2	9	104	8.7%
LatticeQCD	1	0	1	10	10.0%
NWChemEx	3	1	2	21	9.5%
GAMESS	1	0	1	5	20.0%
EXAALT	0	0	0	9	0.0%
ExaAM	4	0	4	38	10.5%
QMCPACK	2	1	1	21	4.8%
Energy:	16	1	15	153	9.8%
ExaWind	2	0	2	33	6.1%
Combustion-Pele	2	0	2	26	7.7%
ExaSMR	2	0	2	14	14.3%
MFIX-Exa	2	0	2	11	18.2%
WDMApp	6	1	5	50	10.0%
WarpX	2	0	2	19	10.5%
Earth and Space Science:	5	1	4	85	4.7%
ExaStar	1	0	1	8	12.5%
ExaSky	0	0	0	25	0.0%
EQSIM	1	0	1	6	16.7%
Subsurface	0	0	0	13	0.0%
E3SM-MMF	3	1	2	33	6.1%
Data Analytics and Optimization:	7	1	6	144	4.2%
Urban	0	0	0	23	0.0%
ExaSGD	2	0	2	33	6.1%
CANDLE	3	0	3	49	6.1%
ExaBiome	1	0	1	14	7.1%
ExaFEL	1	1	0	25	0.0%
National Security:	0	0	0	7	0.0%
ATDM LANL	0	0	0	6	0.0%
ATDM LLNL	0	0	0	0	—
ATDM SNL	0	0	0	1	0.0%
Co-Design:	16	1	15	203	7.4%
Proxy Applications	1	0	1	19	5.3%
Application Assessment	0	0	0	11	0.0%
CODAR	4	0	4	39	10.3%
COPA	2	0	2	33	6.1%
AMREX	2	0	2	16	12.5%
CEED	2	0	2	32	6.3%
ExaGraph	1	0	1	4	25.0%
ExaLearn	4	1	3	49	6.1%
Other, named by respondent:	3	0	3		
ExaWorks (ST)	1		1		
PETSc/TAO (ST)	1		1		
Uintah (non-ECP)	1		1		
Total responses given:	58	6	52	696	7.5%
Number of respondents:	41	5	36	576	6.3%

Table 4: Table of survey responses for Question 1 on AD teams. The first column gives the number who checked the box for each team in either November of January, broken down in the second column for November and the third column for January. The fourth column shows the number of emails sent that were scraped from the ECP Confluence website for the January survey, and the last column shows the response rate of the January survey, as defined as the number of respondents who clicked the box for the AD team divided by the number of email addresses for that team.

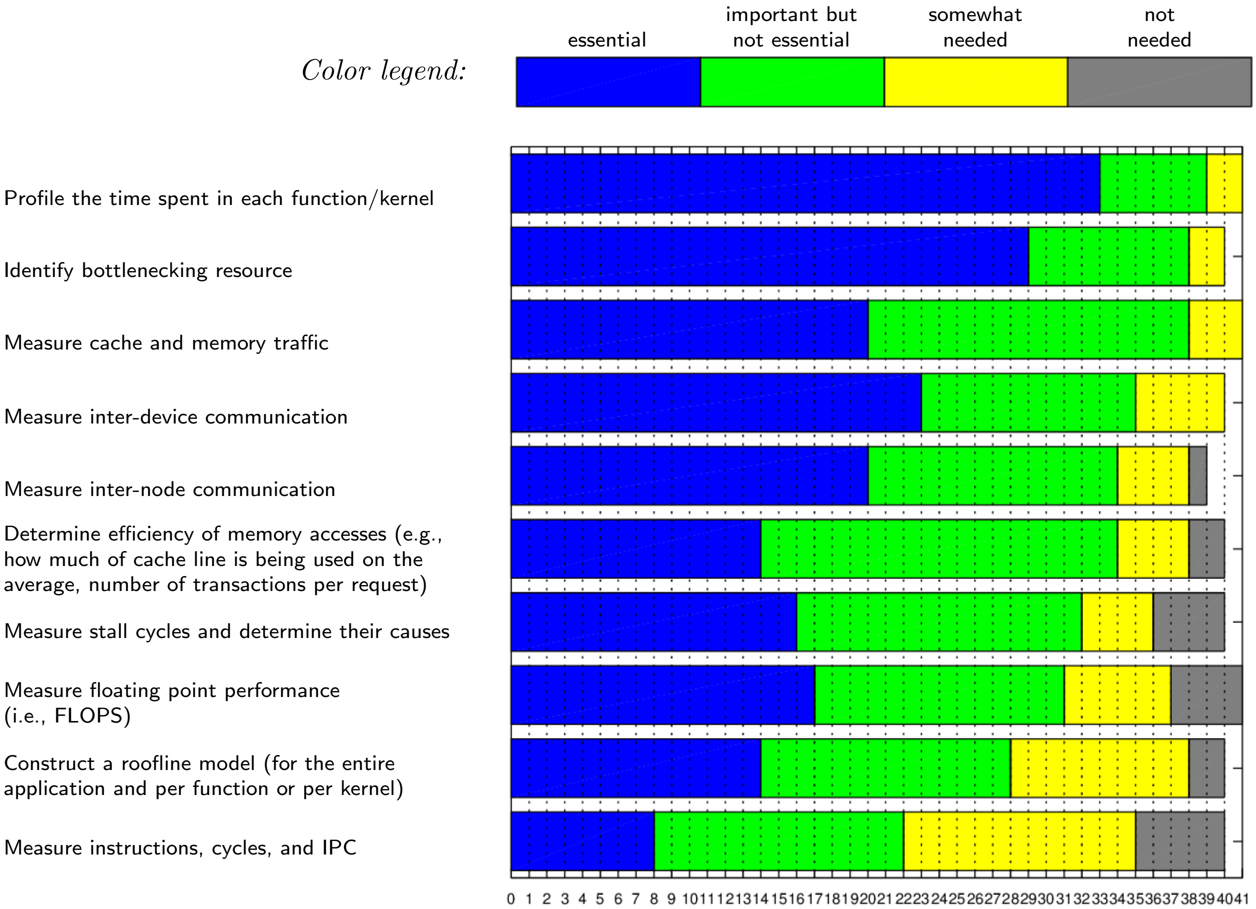


Figure 12: Distribution of responses to Question 2 on the importance of functionalities of performance analysis tools, listed from top to bottom in decreasing order of number of responses of either "essential" or "important but not essential", ties broken by number of responses of "essential".

3.3.2 Question 2: Performance Tool Functionality

The second question on the survey was:

> 2. Performance analysis tools: How important is it for your application development to have tools with each of these functionalities?

There were ten functionalities listed, in the order shown in the Appendix, and the options given for each were "essential", "important but not essential", "somewhat needed", and "not needed". There was also a space for respondents to write in other functionalities.

Figure 12 shows the responses checked on Question 2, with functionalities listed in order of number of responses of either "essential" or "important but not essential", with the highest at top. The number of respondents who checked a box for a given functionality varied from 39 to all 41 survey respondents. In the discussion below, when we report the fraction of respondents who gave a rating for any one functionality, we take it as a fraction of the total number, 41, inferring that a respondent who does not check a box for a particular functionality finds that functionality to be not needed.

At the bottom of Question 2, the survey asked:

Please list any other tool functionalities for performance analysis that you need but are not listed above.

Four respondents gave answers here:

- "The performance analysis tools need to work well with python – and with python applications which are parallelized using MPI."
- "Besides performance, need to see scheduling of kernels as well as memory copies. e.g. See that two kernels are running simultaneously."
- "It would be nice to use an accessor API of the performance analysis tools to feed the data they collect into our logging system as the program runs. Otherwise connecting the data we collect with the data the performance analysis tools is painful post-processing."
- "Memory footprints on host and device. Memory bandwidth. % usage of host versus device."

Discussion

The results in Figure 12 show us that:

- *Every* functionality on the list was rated as either essential or important by the majority of respondents (22 to 39 out of 41).
- *Every* functionality was rated as at least "somewhat needed" by at least 85% of respondents (35 of 41).
- The two functionalities that were rated as essential by the largest majorities were:
  - "Profile the time spent in each function/kernel" (80%); and
  - "Identify bottlenecking resource" (71%).

  These two functionalities were rated as at least important, but not essential by at least 93% of respondents.
- Other functionalities rated as essential by around half of respondents were:
  - "Measure inter-device communication" (56%);
  - "Measure cache and memory traffic" (49%); and
  - "Measure inter-node communication" (49%).

  All three of these were rated as at least important, but not essential by at least 83% of respondents.

3.3.3 Question 3: Performance Tool Satisfaction

The third question on the survey was:

3. Performance analysis tools: How do you rate the tools you currently use in terms of how they meet your requirements for each of these functionalities?

The same ten functionalities as in Question 2 were listed, and in the same order, as shown in the Appendix. The options given for each were "very good", "good", "fair", "poor", "I don't have this, but would like", and "I don't need this". Then there was a space for respondents to enter answers to the question, "Which tools, if any, do you currently use to do this?"

Figure 13 shows the multiple-choice responses received on Question 3, where functionalities are listed not in the order in which they appeared on the survey, but as in Figure 12 in Section 3.3.2, which is by decreasing order of importance as indicated by responses to Question 2. The number

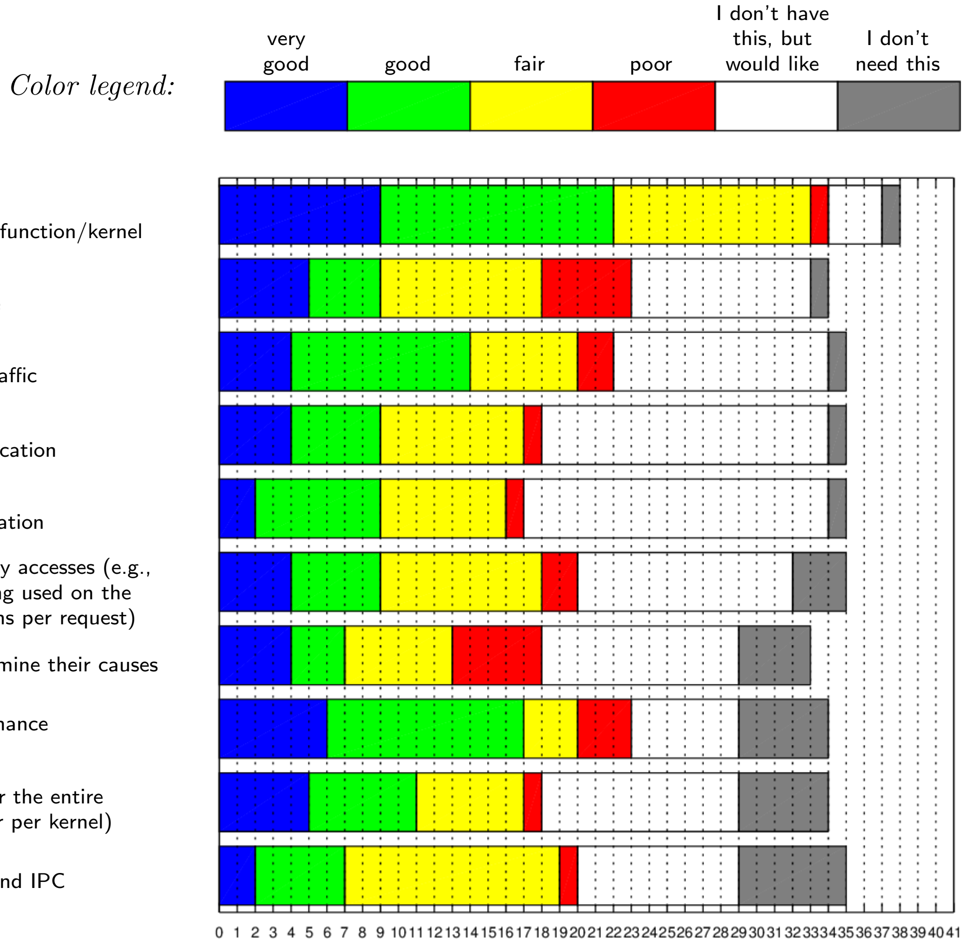


Figure 13: Distribution of responses to Question 3 on satisfaction with current performance analysis tools, listed in the same order as in Figure 12 for responses to Question 2, where functionalities have been ranked from most important down to least important.

of respondents who checked a box for a given functionality varied from 33 to 38, of the 41 total respondents to the survey.

Discussion

Recall that "Profile the time spent in each function/kernel" was rated by respondents to Question 2 as the most important of the ten functionalities that were listed, with 80% classifying it as "essential". On Question 3, 38 respondents gave ratings of their current tools on this functionality, compared to 34 or 35 for the other functionalities. And the results displayed in Figure 13 show us that this is the only functionality with a majority of respondents rating their current tools as either "very good" (24%) or "good" (34%). Of the rest, 29% gave ratings of "fair", 3% as "poor", 8% opted for "I don't have this, but would like", and 3% for "I don't need this". It was on this functionality that those last two options received the fewest responses of all, indicating that respondents definitely need to do time profiling, and almost all of them have tools to do it.

There was much less satisfaction with performance analysis tools on the other nine functionalities on the list. As can be seen by a glance at the adjacent red and white bars in Figure 13, of those who reported a rating or that they "would like" a functionality, other than time profiling, roughly half the respondents rated their current tools as either "poor" or "I don't have this, but would like", with the combined fraction who gave those two responses ranging from 31% for "Measure floating point performance (i.e., FLOPS)" up to 55% for "Measure stall cycles and determine their causes".

The two functionalities to which respondents gave the highest fraction of "I don't have this, but would like" answers were measuring communication: 50% for "Measure inter-node communication" and 47% for "Measure inter-device communication", in both cases excluding one response each of "I don't need this".

In the answer box asking respondents to name the tools they currently use for each functionality, the ones that were mentioned the most were NVIDIA's Nsight Compute and Nsight Systems, or sometimes just "Nsight". These tools received more mentions than all others put together, so it is worth including brief descriptions of them from NVIDIA's website:

> *NVIDIA® Nsight™ Compute is an interactive kernel profiler for CUDA applications. It provides detailed performance metrics and API debugging via a user interface and command line tool. In addition, its baseline feature allows users to compare results within the tool. Nsight Compute provides a customizable and data-driven user interface and metric collection and can be extended with analysis scripts for post-processing results.* [4]

> *NVIDIA® Nsight™ Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, help you identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs; from large server to our smallest SoC.* [5]

In the rest of this section on Question 4, we count Nsight Systems and Nsight Compute together, because some respondents entered simply "Nsight" and did not specify either Nsight Systems or Nsight Compute.

In Figure 14, we show the reported satisfaction levels of performance tools on nine of the ten listed functionalities on the survey, excluding responses of either "I don't have this, but would like" or "I don't need this", and separating responses that mentioned Nsight from those that did not.

[4] https://developer.nvidia.com/nsight-compute

[5] https://developer.nvidia.com/nsight-systems

	Allinea MAP	caliper	CUPTI	EP Analytics' PEBIL	gprof	HPCToolkit	Intel Advisor	IPM	kokkos profiling	mpip	nvprof	nvvp	PAPI	PSiNSTracer	cProfile and Pyinstrument	rocprof	TAU	timemory	valgrind	VTune	application timers	vendor tools
Profile the time spent in each function/kernel	G	F	G	G	F	G G			V		V G G	F			F	P	V G F			G	V V G F F	
Identify bottlenecking resource											P				V			V		V	V F P	
Measure cache and memory traffic				F							G P		F					G		G		
Measure inter-device communication															F			F			F	
Measure inter-node communication								G		G				G	G F		G F	G F			V G	
Determine efficiency of memory accesses (e.g., how much of cache line is being used on the average, number of transactions per request)											G		G							G		F
Measure stall cycles and determine their causes																					P	F
Measure floating point performance (i.e., FLOPS)				F		G	G						V			P		G			V G	
Construct a roofline model (for the entire application and per function or per kernel)							V															F
Measure instructions, cycles, and IPC						F							F						P			G

Table 5: Table of mentions by respondents to Question 3 of all performance analysis tools other than NVIDIA's Nsight tools. There is a letter entry in the table for each rating of a functionality of performance analysis tools of **V**ery good, **G**ood, **F**air, or **P**oor.

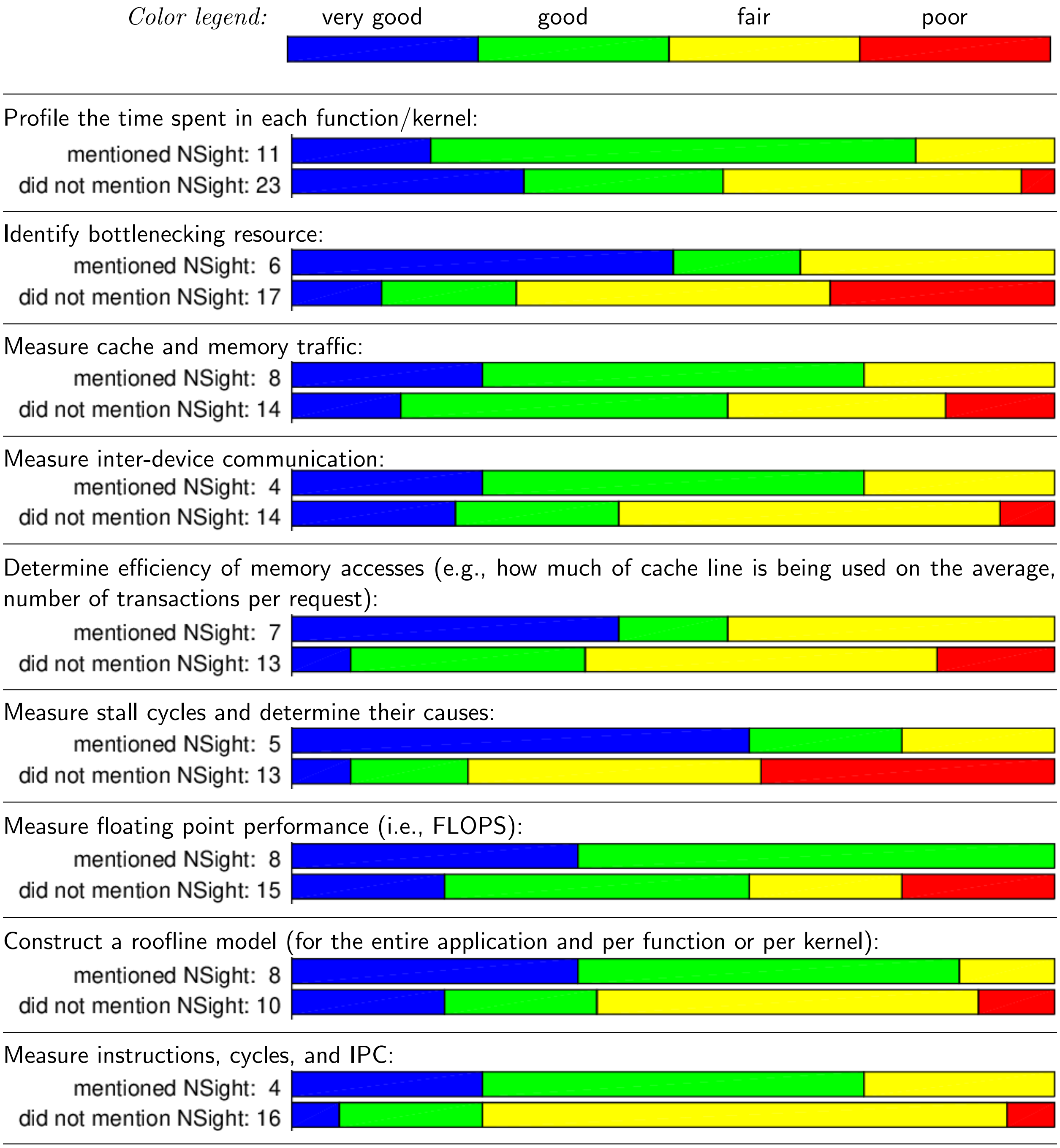


Figure 14: Comparison of satisfaction levels (for those who gave one) of those respondents who mentioned using Nsight tools, against those who did not mention Nsight tools. Functionalities are listed in the same order as in Figures 12 and 13, except that "Measure inter-node communication" is omitted, because no respondent reported using Nsight tools for that. These are not specifically ratings for Nsight tools; the ratings next to "mentioned NSight" are those given by all who mentioned using Nsight tools for the listed functionality, but some of them mentioned also using other tools in addition to Nsight.

In all cases, those who mentioned Nsight reported greater satisfaction (ratings of "very good" or "good") with performance analysis tools than those who did not mention Nsight. No one who mentioned using Nsight tools for any functionality reported a satisfaction level of "poor" on their tools for that functionality.

As shown in Figure 14, the number of respondents who mentioned Nsight performance analysis tools depended on which functionality was being asked about, and varied from 4 to 11 out of the 38 who responded to Question 3. The high levels of satisfaction reported by those who mentioned Nsight tools, relative to those who did not, may be simply a result of people who are satisfied with their tools being more likely to name those tools on the survey.

For completeness, Table 5 lists all mentions by respondents of performance analysis tools other than Nsight tools, with the rating given for current tools on each functionality. Since many respondents named multiple tools, the ratings shown in the table should not necessarily be interpreted as indicating satisfaction level with a particular tool. There are too few mentions of non-Nsight tools to draw broader conclusions.

3.3.4 Question 4: Debugging Functionality

The fourth question on the survey was:

> 4. Debugging tools: How important is it for your application development to have tools with each of these functionalities for HPC applications?

There were five functionalities listed, in the order shown in the Appendix, and just as in Question 2 on performance analysis tools, the options given for each were "essential", "important but not essential", "somewhat needed", and "not needed".

Figure 15 shows the responses received on Question 4, with functionalities listed in order of the number of responses of either "essential" or "important but not essential", with the highest at top. The number of respondents who checked a box for a given functionality was 38 or 39, out of 41 total respondents to the survey. In the discussion below, when we report the fraction of respondents who gave a rating for any one functionality, we take it as a fraction of the total number, 39, inferring that a respondent who does not check a box for a particular functionality finds that functionality to be not needed. We do not include the two survey respondents who skipped the whole question.

At the bottom of Question 4, the survey asked:

> Please list any other tool functionalities for debugging HPC applications that you need but are not listed above, other than basic debugger functions such as setting breakpoints and displaying memory contents.

Two respondents gave answers here:

- "gdb backtraces output from gasnet and/or frozen application (generally on a single node)"
- "Monitoring where and how an application is running (i.e., which cores, threads, GPUs, etc). Synchronous GPU kernel execution for debugging."

Discussion

The main takeaways from Figure 15 are:

- At least 87% of respondents consider these functionalities to be either essential or important:
  - "Detect memory errors (e.g., overwriting, leaks)": 72% essential;

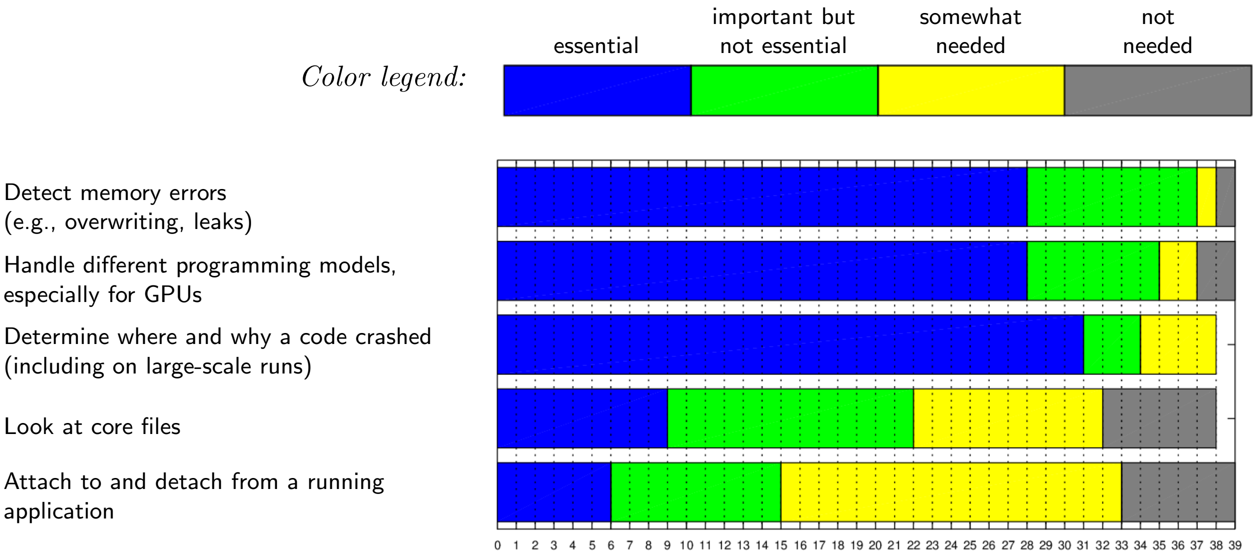


Figure 15: Distribution of responses to Question 4 on the importance of functionalities of debugging tools, listed from top to bottom in decreasing order of number of responses of either "essential" or "important but not essential".

  - "Handle different programming models, especially for GPUs": 72% essential;
  - "Determine where and why a code crashed (including on large-scale runs)": 79% essential.

- The remaining two functionalities are still rated as at least somewhat needed by 82% or more respondents, but not as important:
  - "Look at core files": 23% essential, 56% at least important;
  - "Attach to and detach from a running application": 15% essential, 38% at least important.

3.3.5 Question 5: Debugging Tool Satisfaction

The fifth and final question on the survey was:

> 3. Debugging tools: How do you rate the tools you currently use in terms of how they meet your requirements for each of these functionalities?

The same five functionalities as in Question 4 were listed, and in the same order, as shown in the Appendix. The options given for each were "very good", "good", "fair", "poor", "I don't have this, but would like", and "I don't need this". Then there was a space for respondents to enter answers to the question, "Which tools, if any, do you currently use to do this?"

Figure 16 shows the multiple-choice responses received on Question 5, where functionalities are listed by decreasing order of importance as indicated by responses to Question 4, as in Figure 15 in Section 3.3.4. The number of respondents who checked a box for a given functionality varied from 36 to 39, of the 41 total respondents to the survey. The results shown in Figure 16 suggest widespread dissatisfaction with current debugging tools on some functionalities: if we exclude those who checked "I don't need this" or did not check any box, 52% of respondents gave responses of either "poor" or "I don't have this, but would like" on the second-most important functionality of Handle different programming models, especially for GPUs (16 of 31), and also on the functionality rated least important on the list, Attach to and detach from a running application (13 of 25).

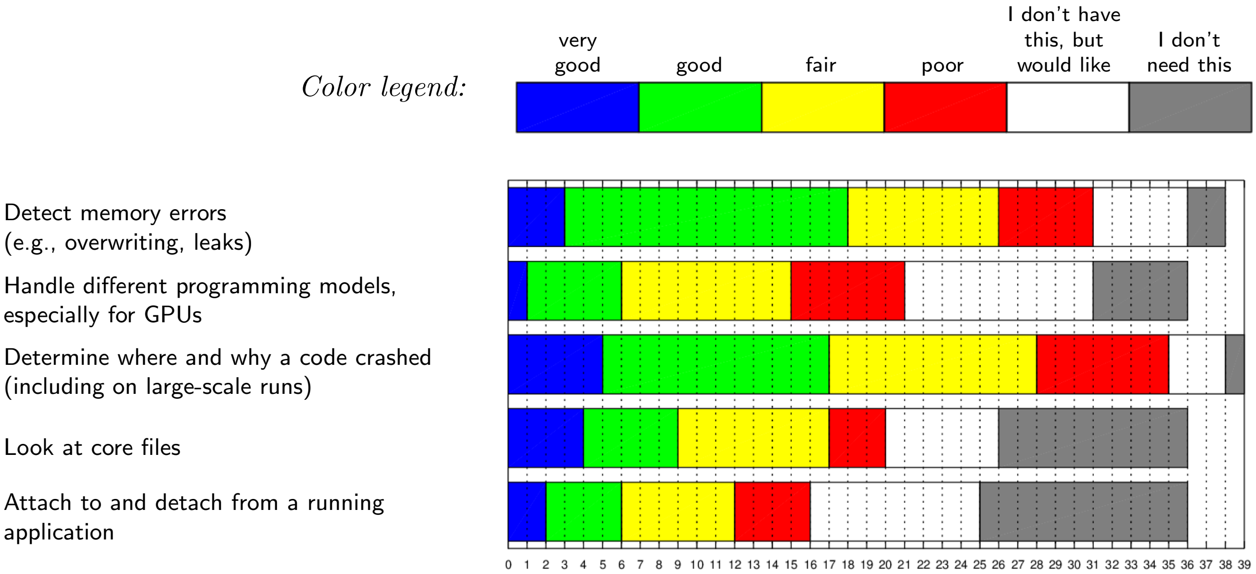


Figure 16: Distribution of responses to Question 5 on satisfaction with current debugging tools, listed in the same order as in Figure 15 for responses to Question 4, where functionalities have been ranked from most important down to least important.

Figure 17 shows satisfaction ratings for three of the functionalities on which respondents named particular debugging tools five or more times, compared with respondents who gave a rating for the same functionality but did not name any of these tools.

- For detecting memory errors, the majority (17 of 31) of respondents mentioned using Valgrind, but their reported satisfaction ratings were not significantly different from those of respondents who did not mention Valgrind.
- For determining where and why a code crashed:
  - Arm Forge received 8 mentions, and 6 of those who mentioned it rated their debugging tools to be at least good on this functionality;
  - There was a similar rating profile for TotalView, mentioned 5 times, with 4 of those who mentioned it rating their debugging tools on this functionality as good (none very good), and 1 fair;
  - there may be less overall satisfaction with gdb, as it was mentioned 8 times, with only 3 of these respondents rating their debugging tools on this functionality as good and 5 as fair.
- Of the 20 respondents who gave a rating to their debugging tools on looking at core files, 11 mentioned gdb, of which 7 rated their tools on this functionality as at least good, compared to only 2 out of the 9 who gave a rating but did not mention gdb.

Table 6 lists all mentions by respondents of debugging tools other than those listed in Figure 17 for the functionalities listed in that figure. Again, the ratings are of debugging tools as used for each functionality, and should not be interpreted as ratings of the tools, because several respondents mentioned using more than one tool for the same functionality.

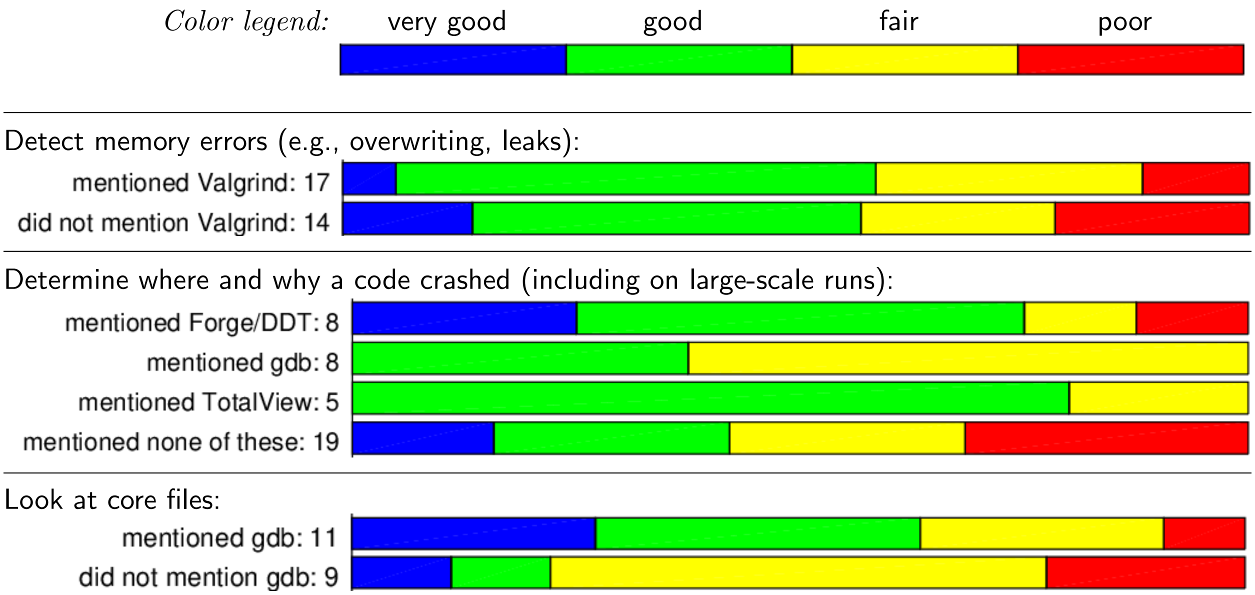


Figure 17: Comparisons of satisfaction levels, by those who gave them, of debugging tools applied to three different named functionalities on the survey, each broken down by those responses that mentioned the most frequently named debugging tools and those that did not. The ratings are not specifically for the tools mentioned; each is an overall rating of debugging tools for the functionality in question, by those who mentioned that tool. Several respondents named more than one debugging tool that they use for a given functionality. "Forge/DDT" refers to Arm Forge mentioned either by this name or by its former name of Allinea DDT.

	ATP (by Cray)	asan Address Sanitizer	assertion checking	cuda-memcheck	cuda-gdb	gasnet crash reports	gdb4hpc (by HPE)	hip	kokkos	lldb (by LLVM)	NVIDIA Compute Sanitizer	Nvidia-gdb	openss	printf	Python toolchain	rocm	Valgrind	vendor tools
Detect memory errors (e.g., overwriting, leaks)		V G		G F	G			P			G		G			P		
Handle different programming models, especially for GPUs					G				G			P						F
Determine where and why a code crashed (including on large-scale runs)	V V		P			F	V V			G F F				G	F		G	
Look at core files					V					V								
Attach to and detach from a running application																		

Table 6: Table of mentions by respondents to Question 5 of all debugging tools other than those listed in Figure 17 for three of the functionalities. There is a letter entry in the table for each mention of a tool and the accompanying rating of debugging tools in general on a functionality as being **V**ery good, **G**ood, **F**air, or **P**oor.

3.4 Conclusion

The primary conclusions we were able to draw from the survey are as follows.

Performance Tools

1. Every performance tool functionality was considered either essential or important by the majority of respondents, with function/kernel profiling and bottleneck identification being the most important.
2. Existing tools appear to address the function/kernel profiling requirement adequately.
3. The remaining functionalities we rated as “poor” or unavailable by a significant portion of the respondents indicating that tools either do not provide this capability, it is complex to use, or the documentation is not adequate.
4. Nsight was the most popular tool that was mentioned and received a high level of satisfaction from those who mentioned it.

Debugging Tools

1. Detection of memory errors, the ability to handle different programming models, and the ability to determine where and why a code crashed were considered essential requirements by most respondents.
2. Existing tools appear to address the detection of memory errors and determining where and why a code crashed adequately; however, there was considerable dissatisfaction with support for different programming models, so this would be a good target for future research and development activities.
3. The remaining functionalities receive a mixed response, so it is difficult to draw any conclusions on these.

4 Early Access Tools Assessment

The purpose of the Proxy Applications Tools Assessment effort is to gain an understanding of the availability and quality of development tools available on the ECP early access systems.

4.1 Test Systems

Our evaluation was undertaken on two systems, the A21 Testbed and Tulip, which are described in more detail below. The A21 Testbed is a test system for the future Argonne Leadership Computing Facility (ALCF) Aurora system. Tulip is a test system for the future Oak Ridge Leadership Computing Facility (OLCF) Frontier system.

4.1.1 A21 Testbed for Aurora

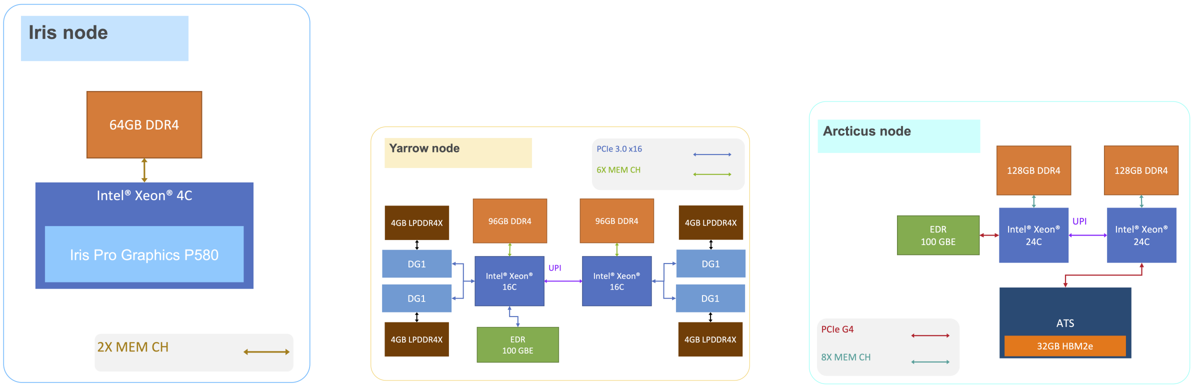
This system is a testbed for the Intel A21 hardware. It comprises a variety of Intel Gen9 (pre-A21) nodes as well as future Intel hardware, and Nvidia P100 and K80 hardware. The primary node types are called "Iris", "Yarrow", and "Arcticus". The system provides access to 20 Iris nodes, 20 Yarrow nodes, and 5 Arcticus nodes. A high-level architecture of these nodes is shown in Figure 18.

4.1.2 Tulip test system for Frontier

This system provides access to a variety of node hardware configurations. Each node has AMD EPYC CPUs with 4 GPUs, 6 nodes with AMD MI60 GPUs and 2 nodes with Nvidia V100 GPUs. There are further nodes with next-generation AMD GPUs; broadly they worked similarly to the MI60 nodes. The node configuration is shown in Figure 19.

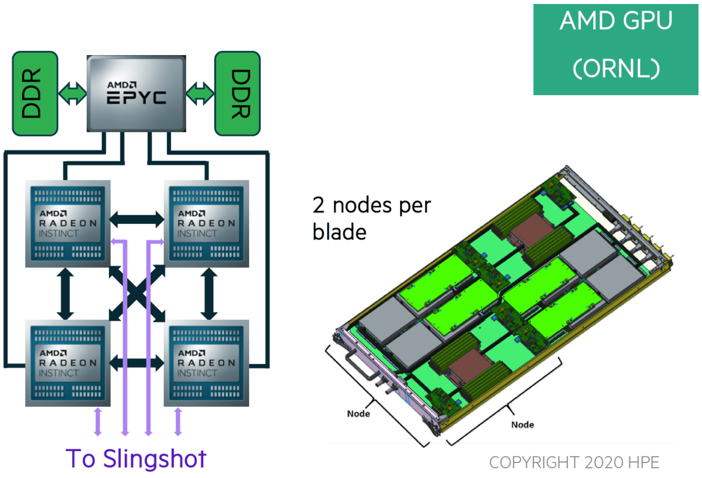
4.2 Methodology

Our approach to this task was to take the same steps that a developer would normally take in order to develop, debug and optimize application codes on these systems. We started by choosing a code that we know is representative of a scientific application, and building it on the systems. We then ran the application to ensure that the compiler toolchain was working correctly.



(a) (b) (c)

Figure 18: A21 Testbed nodes



(a)

Figure 19: Tulip nodes

Once the application was demonstrated to run correctly on the system, we utilized a series of checklists to assess the state of the debugging and performance tools available on the systems. Example checklists are shown in Appendix C.

4.2.1 Test Code

Our initial assessment used XSBench, which is a neutron transport modeling software from Argonne National Laboratory. It is a component of the ECP Proxy Applications Suite[6] and is a good choice for exploring computing environments because it offers various build options, the makefiles are easily adjusted for the environment, and the code is clear enough to ease debugger testing. The builds that were chosen for experimentation are:

- OpenMP Threading: for testing multicore CPUs without GPU use
- OpenCL: for testing AMD GPUs
- CUDA: for testing Nvidia GPUs

4.2.2 Performance Tools

Due to the NDA nature of the testbeds, the performance tools used in this assessment are presented in a separate document.

4.2.3 Debugging Tools

Due to the NDA nature of the testbeds, the debugging tools used in this assessment are presented in a separate document.

4.3 Results

Due to the NDA nature of the testbeds, the results of this assessment are presented in a separate document.

[6] https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/

5 Cosine Similarity of Proxy/Parent Application Performance on CPUs

A major thrust for assessing proxy applications comprises developing a quantitative understanding of the fidelity of proxies when compared to their parents with respect to the underlying behavior of and effect on hardware performance. The hardware components that we are interested in are the network, processor, and memory. Our efforts this year focus on processor and memory behavior, for CPUs and GPUs.

One of our primary deliverables this year was to do a gap analysis, which means identifying where we have applications that are not represented by any proxy application. To facilitate this, we expanded the number of proxy/parent pairs and other applications that we use in analysis from a total of 10 to include 20 proxies and parent applications. We also expanded the systems we use for this study from various Intel-based systems to include Sierra, which is an IBM Power9 processor with Nvidia Tesla V100 GPUs. This not only required porting 20 applications to this system, but also required appropriately adjusting input sizes and decks for this system and additional measurement infrastructure to accommodate the very different performance monitoring unit.

From the beginning, we cast the problem of quantifying proxy application fidelity as a similarity problem. If the proxy is similar to its parent with respect to hardware performance, then we have some fidelity in the proxy. We initially used statistical clustering techniques as similarity measures, but we found these difficult to interpret with respect to similarity between clusters, which is based typically on Euclidean distance. This motivated the use of the cosine similarity method where similarity is measured by the angle between two vectors, which proved to be more intuitive to reason about. When implementing clustering and statistical techniques for similarity measurements, we used PCA (principal component analysis) on a more constrained set of metrics that are commonly used in these types of studies. With cosine similarity, we use a vector of between 600 and 700 events depending on processor architecture. This required a very large number of application runs since we can only collect around 20 hardware performance counter events per execution due to accuracy loss from multiplexing.

The balance of this section describes our cosine similarity methodology, identifies the proxy/parent pairs we considered, and explains what we have learned about proxy/parent similarity on two different CPU platforms.

5.1 Cosine Similarity

Cosine similarity (COS) measures the similarity of vectors projected in multi-dimensional space by the cosine of the angle between them. An angle of zero degrees indicates that the vectors are, modulo a uniform constant factor, identical, while an angle of 90 degrees represents orthogonal behavior. This method is applicable to high dimensional data, and we collect a very large vector of performance data from each application without attempting to pre-select “important” metrics and thus be pre-biased. Using quantitative data obtained from application execution and COS, we can identify which proxies faithfully model the behavior of their parents. Further, with this methodology we can identify where proxy behavior matches or does not match that of the parent with respect to cache, memory, pipeline, etc., and subsequently through this analysis we can identify which specific proxies to use for memory, communication, and node studies. Through redundancy analysis, we can eliminate proxies that exhibit similar behavior and choose a set of proxies that uniquely cover the entire range of parent application behavior. For example, out of ten proxies, there may be four that span the behavior of the suite of parent applications. Additionally, we can identify if there are no proxies that cover certain application behaviors, which we call “gaps” in the proxy suite.

Our fundamental question in this work is how similar or different are applications to each other in the way they utilize system resources, with a focus in this work on node computation and memory resources. Hardware performance counters give us a large array of measurements that we can consider as informing this issue. Rather than pre-selecting a small set of counters over which to compare applications, we collect essentially all available counters, giving us a large vector of measurements for each application. We can think of this vector as an application execution's fingerprint on that hardware.

It would be impractical to try to evaluate the similarity of this data manually, and in the past we have done measurements such as correlation over data reduced by principal components analysis (PCA). Another option is to borrow an idea from text analysis and compare the angle between the vectors by virtue of properties of the vector inner product in vector spaces of two or more dimensions—a technique dubbed *cosine similarity.* The inner product can be conceptualized as the projection of one vector in the direction of the other vector. The calculation relies on the two complementary definitions (algebraic and geometric) for computing the inner product:

- Algebraic Inner Product: $x \bullet y = \sum_{i=1}^{n} x_i y_i$
- Geometric Inner Product: $x \bullet y = \|x\| \|y\| \cos\theta$
- Cosine Similarity: $\cos\theta = \frac{\sum_{i=1}^{n} x_i y_i}{\|x\| \|y\|}$

This approach has several interesting characteristics. It compares the direction of the vectors (which depends on the relative ratios of consumption/stress on system resources), but not their magnitudes (which can depend on run time). Because the metric data collected is generally non-negative, the cosine varies from 1.0 (identical vector direction) toward 0.0 (orthogonal vectors) providing a simple sortable metric for comparisons. It is mathematically forgiving of missing, extraneous, or redundant characteristics, because non-distinguishing components are naturally suppressed as they are outside the plane of the angle. This avoids the need for a principal component analysis, allowing data collection to cast a wide net without concern for corrupting results. The method easily extends to both different kinds of metrics and multiple systems/software stacks. However, the inner product computations do place a firm requirement the pre-processing the data: the units on the x_i and y_i elements must be consistent since their products will be summed. The advantages and requirements stated above, are a reasonably good match to most of the hardware event counters built into extant computer hardware.

While other cosine similarity metrics proposed by the machine learning community, such as Square Root Cosine Similarity and Improved Square Root Cosine Similarity [30] have been considered, neither reflects identity in the case of comparing a vector with itself, nor does either present a clear geometric understanding of the comparison.

5.2 Application of Cosine Similarity

In this section, we present our methodology for collecting data and how we use that data to produce a cosine similarity (COS) matrix. We use two system platforms with different architectures to collect data on 20 proxy and parent applications that span several scientific domains.

5.2.1 Proxy/Parent Application Suite

In this work, we use a suite of 20 total applications that are a combination of proxy/parent pairs with some extra proxies and parents that are not paired. The set of proxy/parent pairs is not

Table 7: Proxy/Parent version information

Proxy	Version	Parent	Version
AMG2013	2013_0	N/A	N/A
ExaMiniMD	1.0	LAMMPS	17 Aug 2017
Laghos	3.0	N/A	N/A
miniQMC	0.4	QMCPACK	3.8
miniVite	1.0	Vite	30 Sept 2020
Nekbone	17.0	Nek5000	19.0
PENNANT	0.9	N/A	N/A
PICSARlite	16 July 2020	PICSAR	16 July 2020
SNAP	1.09	N/A	N/A
SW4lite	2.0	SW4	2.0
SWFFT	1.0	HACC	1.0
N/A	N/A	Castro	20.07
XSBench	19.0	OpenMC	0.11.0

complete because several of the proxies have export-controlled parents, which has complicated data collection, delaying their addition to our suite. Also, Castro is the parent of a proxy called Thornado [8], but at this point, we are not using Thornado because of I/O issues that have caused problems with data collection; we intend to resolve this in future work. [7].

We use the vendor-specific compiler on each platform to compile all of the applications except OpenMC. On the Intel Skylake system, we used *icc* 20.0.2.254 and *OpenMPI* 4.0.3, on the IBM Power9 system we used *xl* V16.1.1 with IBM Spectrum MPI. OpenMC required us to use GNU compilers, 8.3.1 on Skylake and 8.2.1 on Power9.

For each proxy/parent pair, we use the same input problem and/or parameters where possible. In cases where we cannot run the same problem, we use the closest matching problem available and we size both proxy and parent application problems in all cases to use about 50% of the available memory.

Table 7 contains all of the proxy/parent pairs and other applications and the specific versions that we use in this work. If a date is given, it is the latest code available in the repository at that date.

AMG2013 [13] is a proxy application for BoomerAMG and is a parallel algebraic multigrid solver for linear systems arising from unstructured grid problems. We ran the default Laplace problem with a custom resizing. We did not run BoomerAMG because we did not have the expertise to ensure that we treated it fairly, but we intend to use it in the future. LAMMPS [26] is a classical molecular dynamics code, with particles ranging from a single atom to a large composition of material. It implements mostly short-range solvers, but does include some methods for long-range particle interactions. ExaMiniMD [31], which is a proxy for LAMMPS, implements limited types of interactions, and only short-range ones. Laghos [6] is a proxy application that is a high-order Lagrangian hydrocode meant to represent several compressible shock hydrocodes, including BLAST. We did not run BLAST because it is export controlled and we are still working with LLNL to run its experiments on their systems. QMCPACK [15] is a quantum Monte Carlo package for computing the electronic structure of atoms. MiniQMC [27] covers QMCPACK's essential computational kernels. The computational themes of miniQMC and QMCPACK are particle methods, dense and sparse linear algebra, and Monte Carlo methods. Vite [11] is an implementation of the Louvain method for (undirected) graph clustering or community detection. MiniVite [10] is a proxy application for Vite that implements a single phase of the Louvain method in distributed

Table 8: Proxy/Parent Problems/Input Sizes

Problem Inputs for Intel Skylake	
Proxy / Parent	Problem/Input size
AMG2013 / (N/A)	Default Laplace, -pooldist 1 -r 62 62 62 -P 4 2 2
ExaMiniMD / LAMMPS	in.snap.Ta.mod w/ region box block 0 512 0 512 0 512
Laghos / (N/A)	Sedov blast wave with partial assembly, -p 1 -m data/cube_27_hex.mesh -rs 6 tf 0.00001 -pa
miniQMC / QMCPACK	-r 0.99 -g ’2 2 2’/NiO-example.in.xml,NiO -fcc -supertwist111 -supershift000-S64.h5
miniVite / Vite	generated random graph w/ 1.6384E8 vertices
Nekbone / Nek5000	example2 w/ 24980 to 25000 elements per rank; eddy w/ 70 x 70 x 70 elements
PENNANT / (N/A)	leblancx128.pnt w/ tstop 0.03; meshparams 6400 57600 1.0 9.0;dtinit 0.25e-4
PICSARlite / PICSAR	homogeneous_plasma_lite w/ 400 x 400 x 880 grid
SNAP / (N/A)	inh0004t1a out4a w/ nthreads=1;npey=8;npez=16; 1536x32x32;tf=0.2;nsteps=20
SW4lite / SW4	gaussianHill.in w/ grid x=1.27 y=1.27 z=19.99 h=0.003
SWFFT / HACC	15 2048/ indat.params w/ NG 1024; NP 1024; TOPOLOGY 8x4x4
(N/A) / Castro	inputs.sc w/ amr.n_cell = 752 752 752
XSBench / OpenMC	-s small -g 135000 -l 2000; pincell problem w/ 4000 x 4000 tallies

Specific Problem Inputs for IBM Power9 (if not listed, the same inputs as above were used)	
Nekbone / Nek5000	example2 w/ 12480 to 12500 elements per rank; eddy w/ 60 x 60 x 60 elements
PICSARlite / PICSAR	homogeneous_plasma_lite w/ 480 x 480 x 960 grid
XSBench / OpenMC	-t 1 -s small -g 160000 -l 1000 ; pincell problem w/ 5000 x 5000 tallies

memory for community detection. Nek5000 [22] is a spectral element computational fluid dynamics solver while its proxy application Nekbone solves the Poisson equation with a spectral element multigrid preconditioned conjugate gradient solver. PENNANT serves as a proxy application for rad-hydro physics-based algorithms on an unstructured mesh, modeling the computation and memory access patterns typical to rad-hydro applications. It is modeled on, and thus serves as a proxy for, the LANL code FLAG. PICSAR [33] is Particle-In-Cell solver, while its proxy application, PICSARlite, is a subset of the actual codebase. SNAP serves as a proxy application for discrete ordinates neutral particle transport, modeling the computation and memory access patterns typical to neutral particle transport applications. It is modeled on, and thus is a proxy for, the LANL code PARTISN. SW4 [25] is a geodynamics code that solves 3D seismc wave equations with local mesh refinement. SW4lite [7] is a scaled-down version of SW4 that has limited seismic modeling capabilities, but does solve the elastic wave equation and uses some of the same numerical kernels as those implemented in SW4. The Hardware Accelerated Cosmology Code (HACC) [12] is an N-body framework that simulates the evolution of mass in the universe, with both short and long range interactions. The long-range solvers implement an underlying 3D FFT. SWFFT [7] is the 3D FFT that is implemented in HACC. Since this FFT accounts for a large portion of the HACC execution time, SWFFT serves as a proxy for HACC. Castro [4] is an adaptive mesh, astrophysical

Table 9: Hardware Characteristics of Intel Skylake and IBM Power9 Platforms

Component	Skylake	Power9
L1 data cache (private)	32 KB, 8-way	same
L1 instr. cache (private)	32 KB, 8-way	same
L2 cache	1 MB, 16-way per core	512KB, 8-way per core pair
L3 cache (shared)	24.75MB, 11 way	120MB, 20-way 12, 10MB banks
Memory (per node)	192 GB DDR4- MHz	256GB DDR4-MHz
Cores/threads	18/36	24/48
Sockets/node	2	2
Total nodes	1488	54
Interconnect	Omnipath	Mellanox EDR Infiniband
Max Memory BW (per processor)	GB/sec	170 GB/sec
Memory channels (per socket)	6	8

radiation hydrodynamics simulation code. OpenMC [28] is a Monte Carlo particle transport code. XSBench [32] is a proxy application for OpenMC and represents the continuous energy macroscopic neutron cross section lookup kernel, which is a key computational kernel of Monte Carlo particle transport.

5.2.2 System Platforms

We want to look at proxy/parent behavior similarity on significantly different platforms, so we chose an Intel Skylake and an IBM Power9 system. Some basic characteristics of these systems are shown in Table 9. The IBM platform does implement graphics processing units (GPUs) on each socket. However, since this work focuses on CPU behavior, we do not include these characteristics in Table 9. The Intel system runs the RHEL7.8 operating system (OS); RHEL7.6 OS runs on the IBM system.

From Table 9 these architectures seem relatively similar. The Intel architecture is a CISC (complex instructions set computer) and the IBM is a RISC (reduced instruction set computer) which is fundamentally different. However, this difference does not manifest in a fundamental difference in the execution pipeline. They have similar pipeline depths, numbers of execution units, and issue widths. The differences are primarily in the memory subsystem and in SIMD width. The Skylake processor supports up to 512-bit SIMD where the Power9 only supports 128-bit. For about half of the applications we observed similar execution times on the two platforms. For the other applications, we observed significant slowdowns on the IBM architecture. We believe (and IBM agreed) that these applications benefit from the wide SIMD on Intel Skylake and could not attain the same performance on Power9 given the 128-bit wide SIMD support.

We run all of our applications in MPI-only mode and the collection runs are done using 128 ranks, one rank per core, on four nodes. We chose this configuration because it is small enough to feasibly run large numbers of experiments relatively quickly, yet it is large enough to capture important communication behavior.

5.2.3 Data Collection and Analysis

We use LDMS [2] as the collection infrastructure in all of our experiments. LDMS implements a plug-in architecture, where plug-ins are often engineered to collect data for a particular component or piece of the system. In this work, we use the PAPI sampler, which implements the PAPI API within the sampler to connect to every process (rank) in each application for collection of node-related performance counter data. We carefully examined all of the available performance events on each hardware platform. We functionally tested to ensure that at a minimum, plausible data was returned for each event. We eliminated events that were clearly returning no data or unstable data (i.e., vastly varying) across application runs.

We collect on the order of 700 events on each platform for each application; the set of events that are used as input to the cosine similarity algorithm is called a vector. Several runs are required to collect the complete set of data, since the hardware has limited performance counter resources, and if software multiplexing of these resources is too extreme, accuracy can be lost. Although the number of events collected per run is application specific (i.e., depends on application behavior), we experimentally determined that if the number of events collected per experiment was 35 or less, the effect on accuracy was negligible. We were able mostly to create these subgroups so that they did (partially) represent hardware components or features, but a few event subgroups did have a mix of event types in them. Because we also aim to understand where the proxy is and is not a good model of the parent in terms of node components such as cache, TLB, branch predictor, and pipeline, we further grouped these subgroups into architectural concept groups, including cache, branch prediction, pipeline, instruction mix, memory, virtual memory, and others. We then evaluated COS in these subgroups and concept groups.

Data is collected from each application process (i.e., each MPI rank) and we compute an average for each event across all ranks so that all data reflects an average rank value. Before an application vector is input into COS, we normalize the event counts by cycles executed, which is also a collected event. So all events input into COS are $eventcount/cycle$.

Our analysis tools that compute cosine similarity and principal components are written in python and use several of the standard math libraries (e.g., math, numpy) and scikit-learn facilities (cosine-similarity). We have also developed an extensive infrastructure to automate data collection experiments.

5.3 Results and Analysis

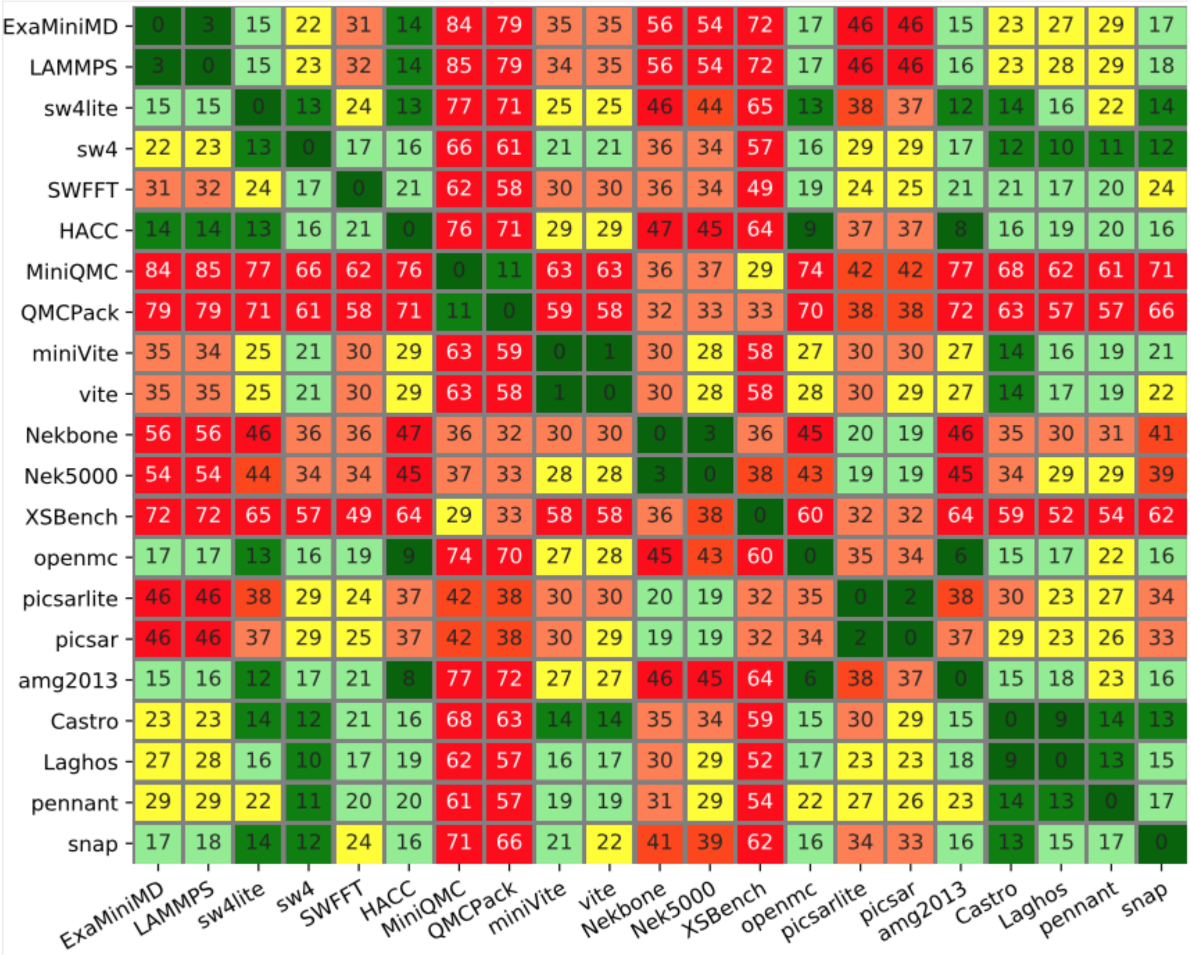
Here we present results from the cosine similarity studies on both the Intel and IBM platforms. We present these results separately, then also report the COS analysis results of unifying the data vectors from both machines for each application. As the reader will see, there is some inconsistency in proxy/parent similarity between different architectures. This makes it very difficult to really understand if a proxy is indeed a good or acceptable model of its parent. By unifying the vectors, we can either validate or invalidate these inconsistencies.

5.3.1 Overall Cosine Similarity: Per Platform and Cross-Platform

Before looking at per-component node data, we first look at the cosine similarity between all the applications across all of the data (i.e., data vector contains about 700 events for each' platform).

The similarity threshold scale we use for all data is defined on the vector difference angle θ:

- Highly similar, $0 < \theta \leq 10$ degrees (dark green)
- Moderately similar, $10 < \theta \leq 15$ degrees (light green)
- Similar, $15 < \theta \leq 22$ degrees (light, light green)
- Not similar or dissimilar, $22 < \theta \leq 30$ degrees (yellow)
- Dissimilar, $30 < \theta \leq 38$ degrees (light orange)



	ExaMiniMD	LAMMPS	sw4lite	sw4	SWFFT	HACC	MiniQMC	QMCPack	miniVite	vite	Nekbone	Nek5000	XSBench	openmc	picsarlite	picsar	amg2013	Castro	Laghos	pennant	snap
ExaMiniMD	0	3	15	22	31	14	84	79	35	35	56	54	72	17	46	46	15	23	27	29	17
LAMMPS	3	0	15	23	32	14	85	79	34	35	56	54	72	17	46	46	16	23	28	29	18
sw4lite	15	15	0	13	24	13	77	71	25	25	46	44	65	13	38	37	12	14	16	22	14
sw4	22	23	13	0	17	16	66	61	21	21	36	34	57	16	29	29	17	12	10	11	12
SWFFT	31	32	24	17	0	21	62	58	30	30	36	34	49	19	24	25	21	21	17	20	24
HACC	14	14	13	16	21	0	76	71	29	29	47	45	64	9	37	37	8	16	19	20	16
MiniQMC	84	85	77	66	62	76	0	11	63	63	36	37	29	74	42	42	77	68	62	61	71
QMCPack	79	79	71	61	58	71	11	0	59	58	32	33	33	70	38	38	72	63	57	57	66
miniVite	35	34	25	21	30	29	63	59	0	1	30	28	58	27	30	30	27	14	16	19	21
vite	35	35	25	21	30	29	63	58	1	0	30	28	58	28	30	29	27	14	17	19	22
Nekbone	56	56	46	36	36	47	36	32	30	30	0	3	36	45	20	19	46	35	30	31	41
Nek5000	54	54	44	34	34	45	37	33	28	28	3	0	38	43	19	19	45	34	29	29	39
XSBench	72	72	65	57	49	64	29	33	58	58	36	38	0	60	32	32	64	59	52	54	62
openmc	17	17	13	16	19	9	74	70	27	28	45	43	60	0	35	34	6	15	17	22	16
picsarlite	46	46	38	29	24	37	42	38	30	30	20	19	32	35	0	2	38	30	23	27	34
picsar	46	46	37	29	25	37	42	38	30	29	19	19	32	34	2	0	37	29	23	26	33
amg2013	15	16	12	17	21	8	77	72	27	27	46	45	64	6	38	37	0	15	18	23	16
Castro	23	23	14	12	21	16	68	63	14	14	35	34	59	15	30	29	15	0	9	14	13
Laghos	27	28	16	10	17	19	62	57	16	17	30	29	52	17	23	23	18	9	0	13	15
pennant	29	29	22	11	20	20	61	57	19	19	31	29	54	22	27	26	23	14	13	0	17
snap	17	18	14	12	24	16	71	66	21	22	41	39	62	16	34	33	16	13	15	17	0

Figure 20: Application Cosine Similarity, Skylake, All Data

- Moderately dissimilar, $38 < \theta \leq 45$ degrees (red/orange)
- Highly dissimilar, $\theta > 45$ degrees (red)

Determining the threshold of similarity is complicated because deeming two applications to be similar is somewhat subjective based on how this data is to be used. For example, if one is using this data to choose proxy applications for a memory study, the constraints on similarity may be different if the study is on cache. Because memory behavior can have a much larger impact on overall performance than cache behavior, small differences in cache behavior between proxy and application will likely have a smaller impact on overall performance than small proxy/parent divergence in memory behavior. In contrast, if one is using this data to decide if the proxy can be used to examine refactoring the code for improved memory performance, a small difference between proxy and parent in memory or cache behavior may not matter. Because of this complexity and the lack of prior work on this issue in this domain, we have imposed our own threshold scale on the COS data. We are investigating better techniques to do this and this will be part of future work. However, from experimentation, we observe that this is an acceptable starting point.

Figure 20 shows the cosine similarity matrix in degrees using all of the events for each application on the Intel Skylake platform. Note that all matrices are symmetric on the diagonal. All proxies that have parents are listed first on the axes (top on y; left on x) and each parent is listed directly after its corresponding proxy. The five miscellaneous applications (either a proxy with no parent or a parent with no proxy: AMG2013, Castro, Laghos, PENNANT, SNAP) are listed at the bottom on the y-axis and on the right on the x-axis.

Thus down the diagonal, through PICSAR, one would hope to see 2x2 green blocks that show high similarity between proxy and parent. Looking at the figure, we see that miniVite and Vite are the most similar proxy/parent pair with an angle of one degree between them. PICSARlite and PICSAR, Nekbone and Nek5000, and ExaMiniMD and LAMMPS are also highly similar. SW4 and SW4lite and miniQMC and QMCPACK are moderately similar. XSBench and OpenMC are highly dissimilar, with an angle of 60 degrees between them. This is probably because XSBench is only the cross-section lookup portion in Monte Carlo neutron transport, which is a highly data-intensive kernel. OpenMC implements the full neutron transport code so likely has more opportunity to hide poor cache/memory behavior with other computation. Near the end of the diagonal, the five unpaired applications, oddly, show quite a bit of similarity among them, and with Castro, a real application, being very similar to Laghos, a proxy for a different application. Castro is also quite similar to PENNANT and SNAP.

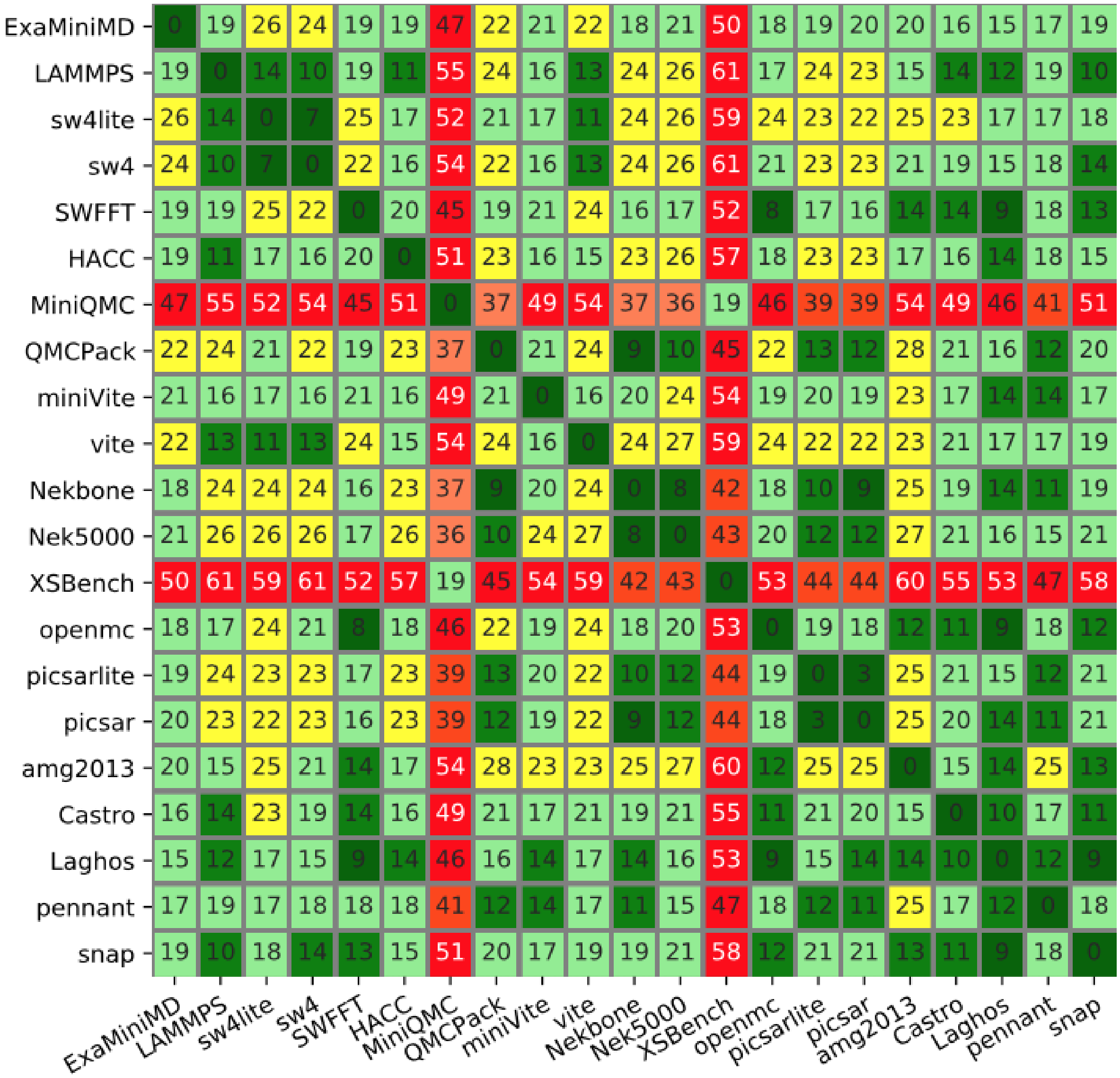
Another thing to note from Figure 20 is that miniQMC and QMCPACK are different than all other applications with the exception of miniQMC being not similar or dissimilar (yellow) to XSBench. XSBench, Nekbone, Nek500, PICSARlite and PICSAR are also outliers in that their behavior shows significant differences compared to most other applications. Nekbone and Nek5000 are not similar to anything except PICSARlite and PICSAR. AMG2013, Castro, Laghos, PENNANT, SNAP, SW4lite, SW4, HACC and SNAP show good similarity.

Figure 21 shows COS matrix in degrees using all of the events for each application on the IBM Power9 system. Most notable here is the changes in similarity between proxy/parent pairs. Now not only is XSBench not a good proxy of OpenMC, but miniQMC diverges enough from QMCPACK that they are dissimilar. ExaMiniMD/LAMMPS and miniVite/Vite are also noticeably less similar, but the rest of the proxy/parent pairs remain fairly consistent in their degree of similarity. Another thing to note is that overall, on the Power9 system, there is less unique behavior (more green than red) across all applications. MiniQMC and XSBench are most dissimilar to everything else, while QMCPACK, Nekbone/Nek5000, and PICSARlite/PICSAR are no longer outliers compared to most other applications. The five unpaired applications are still quite similar to each other.

Figure 22 shows the cosine similarity of the applications when we concatenate the SKX and P9 data vectors for all of the data. What we see here is that the application similarity relationships look very similar to those for the Skylake system in Figure 20. Notice that the four outer corners are characterized by mostly green blocks of good similarity. But then we again see that MiniQMC/QMCPACK, miniVite/Vite, Nekbone/Nek5000, XSBench/openMC, picsarlite/picsar all showing relatively unique overall behavior.

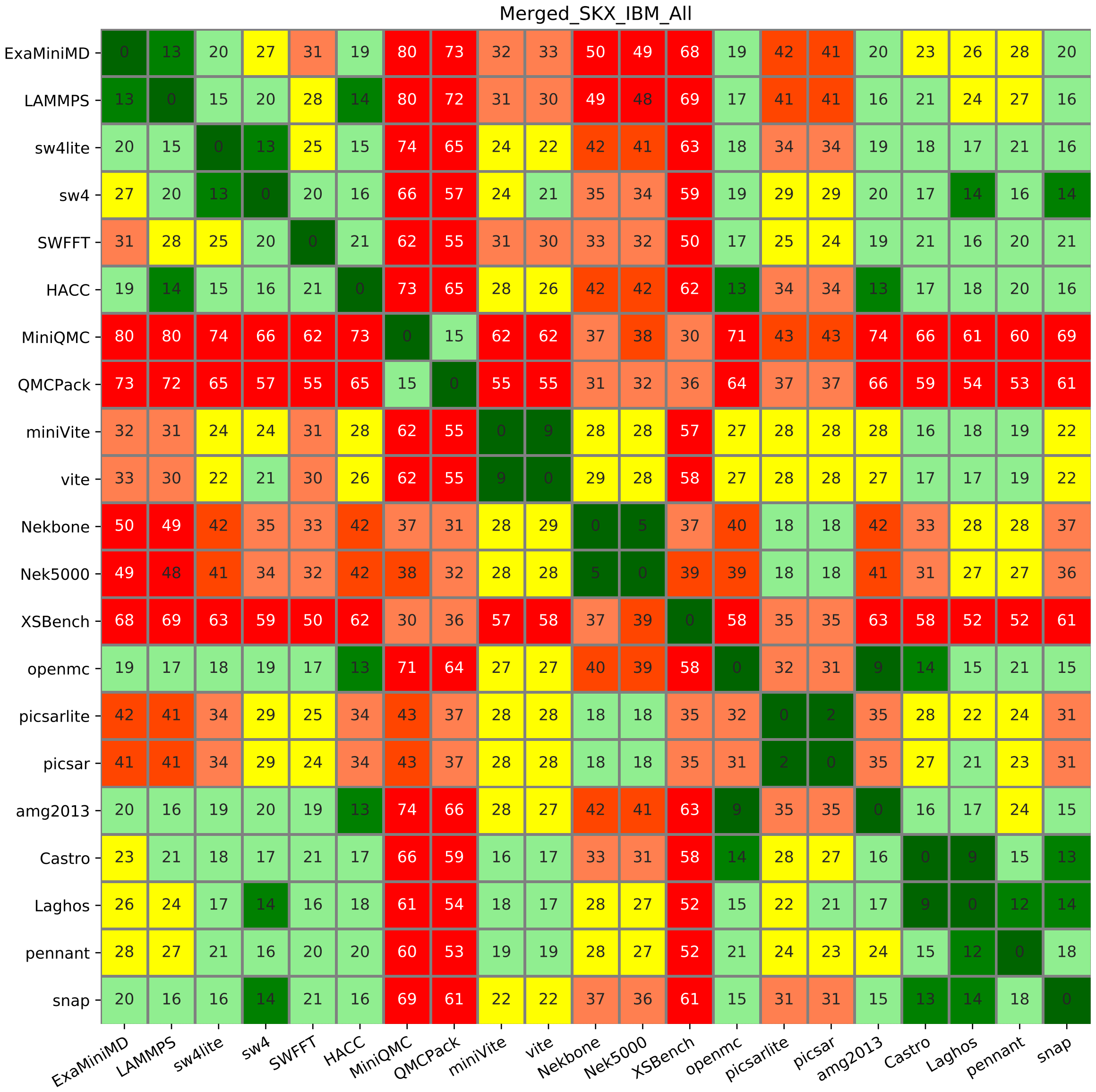
Table 10 shows the percent difference of the combined cosine similarity of SKX+P9 from the COS of each of the architectures individually. To compute this difference, for each application we use the average angular distance between it and every other application. From this data we see quantitatively that Skylake similarity behavior dominates the similarity results of the combined SKX+P9 vector. The P9 behavior has the effect of reducing the angles between all applications and other applications in this combined case.

Summary: All proxies show good alignment with their parents with respect to overall node behavior across the Intel Skylake and IBM Power9 platforms except for miniQMC/QMCPACK and XSBench/OpenMC. For creating a broad system benchmark suite, the proxies MiniQMC, Nekbone, XSBench, and PICSARlite are most different from other proxies and applications.



	ExaMiniMD	LAMMPS	sw4lite	sw4	SWFFT	HACC	MiniQMC	QMCPack	miniVite	vite	Nekbone	Nek5000	XSBench	openmc	picsarlite	picsar	amg2013	Castro	Laghos	pennant	snap
ExaMiniMD	0	19	26	24	19	19	47	22	21	22	18	21	50	18	19	20	20	16	15	17	19
LAMMPS	19	0	14	10	19	11	55	24	16	13	24	26	61	17	24	23	15	14	12	19	10
sw4lite	26	14	0	7	25	17	52	21	17	11	24	26	59	24	23	22	25	23	17	17	18
sw4	24	10	7	0	22	16	54	22	16	13	24	26	61	21	23	23	21	19	15	18	14
SWFFT	19	19	25	22	0	20	45	19	21	24	16	17	52	8	17	16	14	14	9	18	13
HACC	19	11	17	16	20	0	51	23	16	15	23	26	57	18	23	23	17	16	14	18	15
MiniQMC	47	55	52	54	45	51	0	37	49	54	37	36	19	46	39	39	54	49	46	41	51
QMCPack	22	24	21	22	19	23	37	0	21	24	9	10	45	22	13	12	28	21	16	12	20
miniVite	21	16	17	16	21	16	49	21	0	16	20	24	54	19	20	19	23	17	14	14	17
vite	22	13	11	13	24	15	54	24	16	0	24	27	59	24	22	22	23	21	17	17	19
Nekbone	18	24	24	24	16	23	37	9	20	24	0	8	42	18	10	9	25	19	14	11	19
Nek5000	21	26	26	26	17	26	36	10	24	27	8	0	43	20	12	12	27	21	16	15	21
XSBench	50	61	59	61	52	57	19	45	54	59	42	43	0	53	44	44	60	55	53	47	58
openmc	18	17	24	21	8	18	46	22	19	24	18	20	53	0	19	18	12	11	9	18	12
picsarlite	19	24	23	23	17	23	39	13	20	22	10	12	44	19	0	3	25	21	15	12	21
picsar	20	23	22	23	16	23	39	12	19	22	9	12	44	18	3	0	25	20	14	11	21
amg2013	20	15	25	21	14	17	54	28	23	23	25	27	60	12	25	25	0	15	14	25	13
Castro	16	14	23	19	14	16	49	21	17	21	19	21	55	11	21	20	15	0	10	17	11
Laghos	15	12	17	15	9	14	46	16	14	17	14	16	53	9	15	14	14	10	0	12	9
pennant	17	19	17	18	18	18	41	12	14	17	11	15	47	18	12	11	25	17	12	0	18
snap	19	10	18	14	13	15	51	20	17	19	19	21	58	12	21	21	13	11	9	18	0

Figure 21: Application Cosine Similarity, Power9, All Data

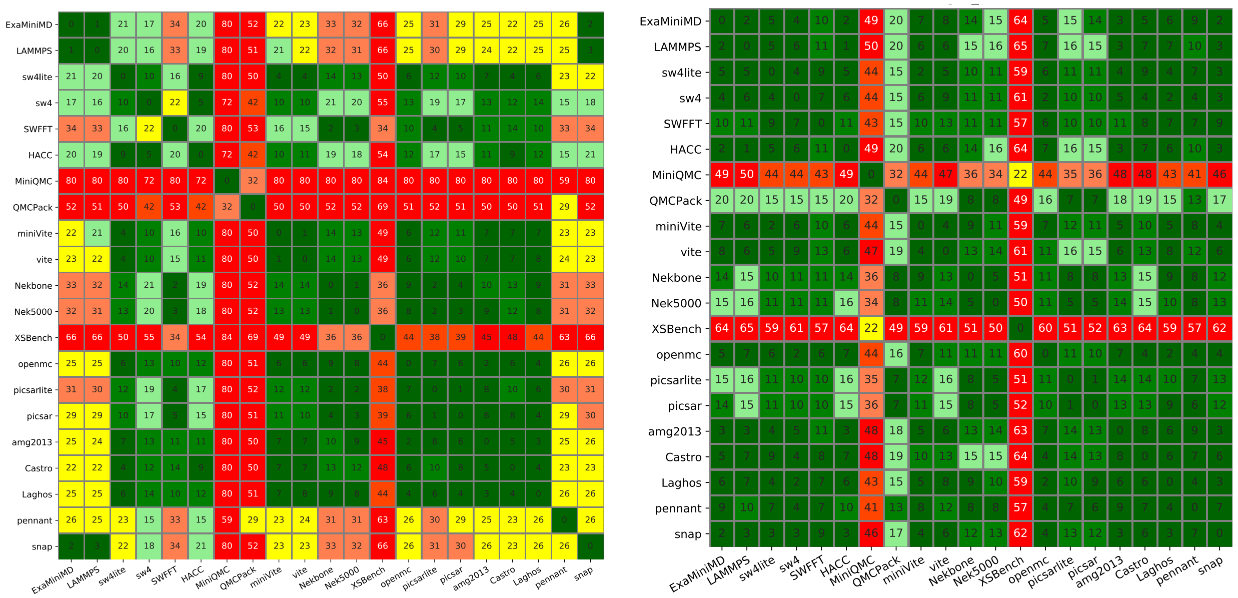


	ExaMiniMD	LAMMPS	sw4lite	sw4	SWFFT	HACC	MiniQMC	QMCPack	miniVite	vite	Nekbone	Nek5000	XSBench	openmc	picsarlite	picsar	amg2013	Castro	Laghos	pennant	snap
ExaMiniMD	0	13	20	27	31	19	80	73	32	33	50	49	68	19	42	41	20	23	26	28	20
LAMMPS	13	0	15	20	28	14	80	72	31	30	49	48	69	17	41	41	16	21	24	27	16
sw4lite	20	15	0	13	25	15	74	65	24	22	42	41	63	18	34	34	19	18	17	21	16
sw4	27	20	13	0	20	16	66	57	24	21	35	34	59	19	29	29	20	17	14	16	14
SWFFT	31	28	25	20	0	21	62	55	31	30	33	32	50	17	25	24	19	21	16	20	21
HACC	19	14	15	16	21	0	73	65	28	26	42	42	62	13	34	34	13	17	18	20	16
MiniQMC	80	80	74	66	62	73	0	15	62	62	37	38	30	71	43	43	74	66	61	60	69
QMCPack	73	72	65	57	55	65	15	0	55	55	31	32	36	64	37	37	66	59	54	53	61
miniVite	32	31	24	24	31	28	62	55	0	9	28	28	57	27	28	28	28	16	18	19	22
vite	33	30	22	21	30	26	62	55	9	0	29	28	58	27	28	28	27	17	17	19	22
Nekbone	50	49	42	35	33	42	37	31	28	29	0	5	37	40	18	18	42	33	28	28	37
Nek5000	49	48	41	34	32	42	38	32	28	28	5	0	39	39	18	18	41	31	27	27	36
XSBench	68	69	63	59	50	62	30	36	57	58	37	39	0	58	35	35	63	58	52	52	61
openmc	19	17	18	19	17	13	71	64	27	27	40	39	58	0	32	31	9	14	15	21	15
picsarlite	42	41	34	29	25	34	43	37	28	28	18	18	35	32	0	2	35	28	22	24	31
picsar	41	41	34	29	24	34	43	37	28	28	18	18	35	31	2	0	35	27	21	23	31
amg2013	20	16	19	20	19	13	74	66	28	27	42	41	63	9	35	35	0	16	17	24	15
Castro	23	21	18	17	21	17	66	59	16	17	33	31	58	14	28	27	16	0	9	15	13
Laghos	26	24	17	14	16	18	61	54	18	17	28	27	52	15	22	21	17	9	0	12	14
pennant	28	27	21	16	20	20	60	53	19	19	28	27	52	21	24	23	24	15	12	0	18
snap	20	16	16	14	21	16	69	61	22	22	37	36	61	15	31	31	15	13	14	18	0

Figure 22: Application Cosine Similarity, SKX+Power9, All Data

Table 10: Differences of Average Angular Distance Between Each App and All Other Apps

Application	% Diff from SKX	% Diff from P9
ExaMiniMD	−0.840	36.695
LAMMPS	−7.887	36.607
sw4lite	−0.503	21.477
sw4	4.909	18.364
SWFFT	−2.410	29.776
HACC	−2.211	25.510
miniQMC	−1.715	22.727
QMCPACK	−6.142	59.597
miniVite	−0.336	27.059
Vite	−1.871	20.578
Nekbone	−7.402	40.483
Nek5000	−5.513	33.538
XSBench	−0.384	2.495
openMC	−3.004	28.092
picsarlite	−5.802	30.887
picsar	−5.690	31.724
amg2013	−1.695	18.475
Castro	−0.385	21.002
Laghos	−3.320	29.253
pennant	−3.036	28.463
snap	−3.467	27.190
Total	−58.703	589.992



A) Skylake All Cache B) Power9 All Cache

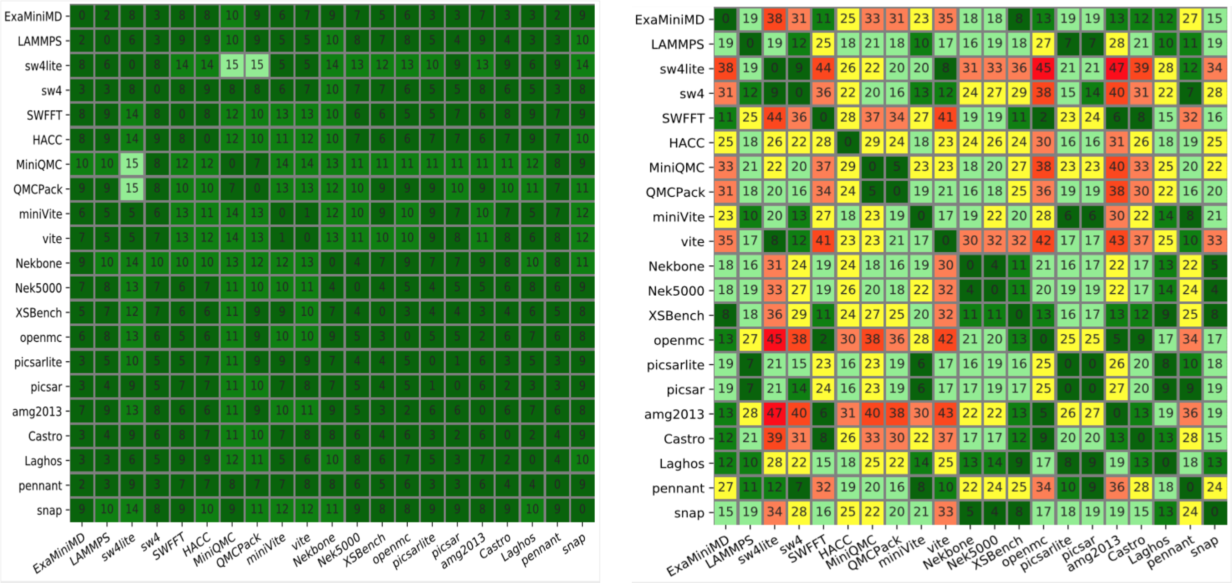
Figure 23: COS Matrices for All Cache Data

Figure 23 shows similarity only over cache-related performance counters. The first point to note is that there is more similar behavior between proxies, parents and other applications in the cache subsystem on Power9 than there is on Skylake. This is a good example of application response to relaxation of a system constraint. The Skylake has a much smaller cache subsystem than the Power9 and the Power9 has a very large L3 cache (120MB), and it is banked. MiniQMC, QMCPACK, and XSBench are known to be characterized by poor memory behavior because of both large data sets and indirect accesses [14, 18]. This can be seen in Figure 23 (A) for the Skylake. With the exception of QMCPACK being not similar or dissimilar to PENNANT in cache behavior, all of these applications are not similar to any of the others. We also see some unique cache behaviors in ExaMiniMD/LAMMPS and PENNANT and SNAP. This method identifies these differences, but studying the data structures in these codes (compared to those in other codes) is necessary to explain this in more depth.

Looking at Figure 23 (B) we immediately see more similarity (more green). The hypothesis here is that as the cache constraints are relaxed in the Power9, cache behavior improves. If applications are all enjoying good cache behavior, they will look similar. QMCPACK is no longer an outlier here, but now looks more similar in behavior to all of the other applications. It appears based on [15] that much refactoring effort has gone into QMCPACK to improve its memory behavior and accuracy, and which has not been implemented in miniQMC. This would explain why miniQMC and QMCPACK diverge here on both platforms and why miniQMC's behavior does not improve on this larger cache subsystem. XSBench still shows outlying behavior as expected because of its extremely large working set size.

Summary: The only proxies that do not represent their parent's cache behavior are miniQMC and XSBench, and they also diverge from all other applications. There are not other large cache behavior divergences; ExaminiMD, LAMMPS, PENNANT, and SNAP have reasonable divergent behavior from all other applications on the more constrained system (Skylake).

For brevity, we do not present COS analysis for branch behavior because for both platforms, all applications, with the exception of miniQMC, are very similar (all less than 22 degrees apart;



A) Skylake

B) Power9

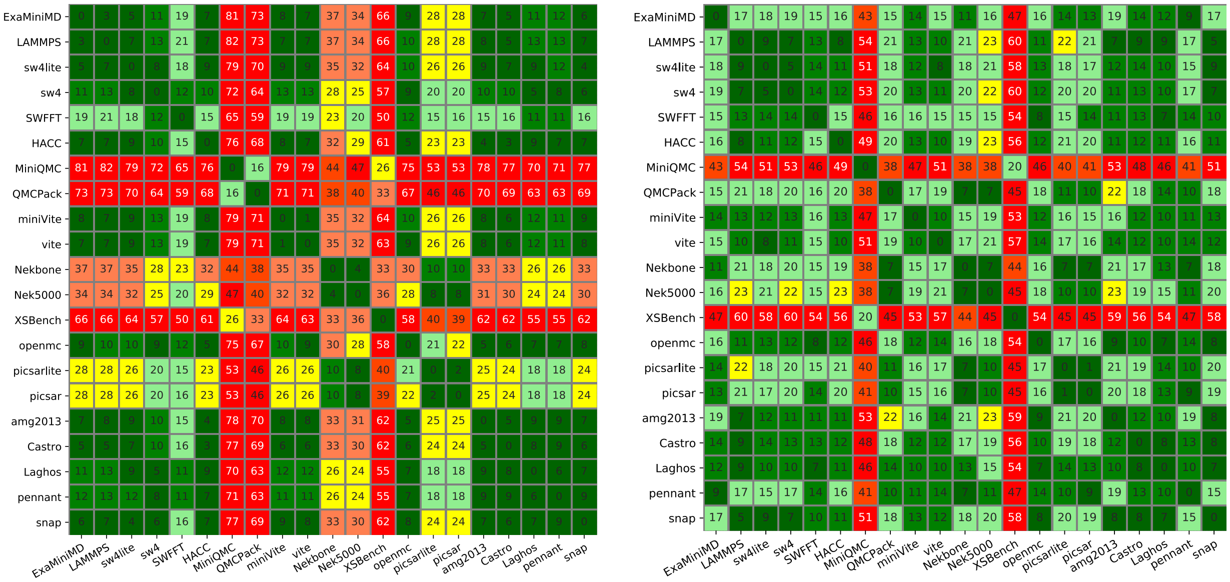
Figure 24: COS Matrices for Instruction Mix Data

varying shades of green). MiniQMC shows outlying branch behavior compared to every other application except for QMCPACK and SW4lite. We see this in the underlying data in that miniQMC seems to have fewer branches per instruction and higher branch prediction miss rates. **Summary:** All proxies are good representations of their parent's branch behavior.

Figure 24 shows instruction mix similarities across the two platforms, and obviously the two platforms differ significantly. On the Intel platform, all proxy/parent pairs, and all applications, show very high similarity in their instruction mix. On the Power9 platform, however, even though there is little red (highly dissimilar), there is still much more variation than on the Intel Skylake platform. Even here, though, proxies remain similar to their parents, except for a slight (yellow) difference between SWFFT and HACC, and the odd similarity between Castro and Laghos still holds. We explored the data to learn why the Skylake results show almost no variation; we believe it is because of the less detail in the floating point performance counters than what is captured on the Power9 platform. **Summary:** There are no large instruction mix divergences, but on Power9 SW4lite, Vite, OpenMC, AMG2013, and Castro have reasonable differences from other applications.

Figure 25 shows similarity over core pipeline-related performance counters only. On Skylake, proxies are parents are very similar except, again, for XSBench and OpenMC. However, there are also other large groups of similarity, in the upper left and lower right; The most different applications still remain the same: XSBench and the QMC, Nek, and PICSAR proxies and parents. On Power9, only MiniQMC and XSBench stand out as very different than the other applications. In our investigation this appears to be due to the effect of the similar difference in cache performance (Figure 23B), which is then causing pipeline stalls and other performance counters to vary. **Summary:** Pipeline performance similarities and differences are consistent with other prior views, and do not show anything significantly novel.

We look at many other subgroupings of data, and cannot describe it all in this paper. On Skylake, out of 29 subgroupings of performance counters, 22 matrices had no colors other than green or light green (no dissimilarity), one (virtual memory-related) had only XSBench as dissimilar, and



A) Skylake

B) Power9

Figure 25: COS Matrices for Pipeline Data

three others (pipeline related) had only XSBench, MiniQMC and QMCPACK as dissimilar. Thus three groupings, one cache related, one memory related and one pipeline related had more extensive dissimilarities, and these contributed most to the overall Skylake similarity matrix. On Power9, out of 33 subgroupings of performance counters, 14 were completely green (no dissimilarity), four had only miniQMC and/or XSBench as yellow, and another 13 had only miniQMC and/or XSBench as red. Thus two subgroupings, one instruction mix related and one pipeline related showed other significant dissimilarities, with instruction mix containing the most dissimilarity. Given that more than half of the subgroupings show miniQMC and/or XSBench (and generally both) as dissimilar, it is clear that these two proxies exercise Power9 hardware differently than the other applications, across the spectrum of subsystems (cache, pipeline, memory).

Finally, given that COS is evaluating data vector direction, it is natural to apply principal components analysis (PCA) to attempt to discern which feature metrics might be most dominant for the vector directions. A full PCA analysis is beyond this paper, and we are still thinking about how best to apply it, but one informal analysis we did was to take the top five principal components from a PCA for each hardware platform, and then look at the top ten feature metrics for each of those five PCs. On the Skylake platform the top five PCs explain about 70% of the variance, and on Power9 they explain about 60%. On both platforms PC6 explains about 5% of variance, so both have reasonably long tails of variance explanation. On Skylake, all five PCs are dominated by "off-core" metrics, which capture memory subsystem activity; a few L1, L2, and i-cache events also show up, but no core pipeline events. On Power9 the first two PCs are dominated by load-store unit, L2, and L3 cache metrics; PCs 3–5 still have some load/store, TLB, and fetch metrics, but also have quite a few pipeline events: various instruction stall counters and various floating point instruction counters (which probably, as mentioned, is also reflected in Figure 24B).

5.4 Conclusion

This work presented a broad study of using *cosine similarity* to evaluate how proxy and parent applications relate, to each other and to other applications. Although this work leads us to much more future work, mentioned throughout, some significant conclusions are evident in this study.

If the goal is to select proxies representative of parent applications that are important to the organization, then among the proxy/parent pairs that we evaluated, XSBench does not represent OpenMC very well, but all the others do, except that MiniQMC showed some dissimilarity to QMCPACK on Power9. Oddly, Laghos (a proxy) was quite similar to Castro (a real app) across a wide range of comparisons.

If the goal is to have a proxy application benchmark suite that generally balances system performance across a wide variety of application behaviors, from our data it seems best to select: MiniQMC, XSBench, Nekbone, PICSARlite, miniVite, ExaMiniMD, SWFFT, and SW4lite, in relative descending preference. It would also probably be good to have one of Laghos, PENNANT, or SNAP, but these three are quite similar to each other.

Our data is extensive and we are still analyzing it and thinking about different ways it can elucidate the issue of characterizing proxies and parents, and their relationships. Evaluation of communication behavior using the ideas presented here is another dimension of needed experimentation and analysis.

6 LDMS GPU Sampler: Implementation and Validation

Another significant project within the proxy application assessment effort this year was the development of a GPU sampler plugin for LDMS. This sampler is targeted at collecting hardware event counter data from GPUs rather than power-related data that other NVML-based samplers collect. Although we aimed to develop a generic GPU sampler, one that could be applied to AMD, Intel, or NVidia GPUs, the current sampler is only applicable to NVidia GPUs. As of today, there is no generic interface to GPU hardware performance counters like there is for CPUs through the PAPI (Performance Application Programming Interface) interface. Although we spent significant time looking at the existing PAPI GPU component and other GPU hardware performance counter tools like NVprof and NSight, we found that the optimal way to develop this tool was to use CUPTI (CUDA Profiling Tools Interface), which is essentially used by all tools today that implement a GPU performance counter interface. We found an issue in the PAPI CUDA component that precluded attachment of a measurement thread to a running GPU process from outside the GPU process that the PAPI team has been working on to fix; this also weighed in the decision to directly use CUPTI.

There are some advantages to implementing samplers within LDMS rather than using component-specific tools and interfaces. The LDMS core and its samplers have very low collection overhead and collection can be scaled typically to full-system size (i.e., 1000s of nodes). Data aggregators may need to be configured properly to support this, but it can and has been done. Another advantage of LDMS is that it doesn't integrate a GUI interface with collection so it's really easy and fast to run remotely. Then the user can choose whether to post-process and visualize data locally or do this real-time on a remote machine. Finally, LDMS does not require instrumentation of the application for data collection. It uses multiple samplers to identify the data associated with various code regions.

The advantage to implementing a GPU sampler within LDMS rather than using NVidia's NVprof profiling tool (the transition from NVprof to NSight happened during sampler development) is that we can simultaneously measure system, CPU, network, or filesystem activity as well as GPU performance. Additionally, by implementing a GPU sampler within LDMS, no code instrumentation is required and the tool learning curve is reduced. Although we have not implemented analysis and visualization within the LDMS framework for GPU data, we plan to do this in the future. This will be a relatively simple task since all of our analysis and viz is python-based as are the LDMS back-end facilities.

In the sections below, we present the GPU sampler implementation details, validation sampler including accuracy and overhead measurements, and implementation issues that currently remain and need to be addressed in the future.

6.1 LDMS GPU Sampler Implementation

The current GPU sampler is based on CUPTI, which is an API that supports dynamic profiling for NVidia GPUs. We aim to develop a sampler that satisfies the following requirements:

1. No instrumentation of the code to collect data pertaining to GPU-implemented kernels
2. Ability to collect data from GPU codes that run multiple contexts across multiple GPUs
3. Ability to associate data collected to a particular kernel executing on a particular GPU
4. Ability to sample data at user-defined intervals
5. Accuracy must be within 5% of standard GPU profiling tools and overhead must be either equivalent at a minimum or less than that of standard tools

CUPTI provides the profiling facilities needed to define event sets and read the performance counter events. The CUDA Driver API contains the functions needed to intercept kernel launch and enable obtaining context, kernel, and device IDs. We implement the functionality of the sampler in a library and use LD_PRELOAD to "inject" the functionality of the sampler into each executing GPU kernel.

Figure 26 presents the execution flow and basic function of all of the various objects involved in collecting hardware performance event data from a GPU application. Figure 27 shows this execution flow with more details on the functionality of the the tools and objects involved. The **Configuration** box in Figure 27 shows a configuration file and a meta data file. There are actually several configuration files read by LDMS that configure the basic functionality of the LDMS core (e.g., samplers to be used, daemon service port, transport option, authentication method), the functionality of the data aggregators (essentially same parameters as LDMS core), and the specifics about the samplers to be used (e.g., which samplers, sampling intervals, names). The meta data file contains the names of the hardware performance events to be collected.

When LDMS is started, it reads these configuration files, reads the meta data file, and creates a shared memory space for communication between it and the samplers. Next the GPU sampler/injector starts executing. LD_PRELOAD essential enables us to inject the GPU sampler library before linking that implements the functionality to use CUPTI and to intercept kernel launches for kernel, context, and device resolution during application execution.

We use the LD_PRELOAD constructor to initialize CUPTI and the LDMS shared memory, and the meta data file is read by LDMS and the sampler. When the application starts running, using the CUDA Driver API we are able to intercept every kernel launch and retrieve the information we need to associate the collected data with a specific kernel running on a specific GPU device. During kernel intercept, we also initialize the CUPTI event groups. Event groups are sets of hardware events that can simultaneously be measured. This is dictated by the implementation of the performance monitoring unit on the GPU. While collaborating with NVidia performance engineers during sampler development, we learned that no documentation of these event groups exists and we were advised to experimentally determine these, which we did (this took an enormous amount of time!). NVprof and NSight both use CUPTI to re-run kernels based on the events chosen by the user; our tool also implements this method. We also set the CUPTI collection mode and create a CPU pthread that actually performs the data collection before returning control to the runtime for kernel execution.

6.2 GPU Sampler Validation

When we did our initial overhead testing, we used NVprof for comparing accuracy and overhead because that was the current NVidia profiling tool. Since then, NVidia has released NSight, which is their new profiling tool that reportedly has greater functionality and is faster (i.e., probably less overhead). We are and have been doing this work on a system that only has NVprof available. We have requested NSight to be installed, but this is a low priority for the admin team and our request has yet to be granted. So for now, we do our validation comparisons using NVprof as the ground truth. In the future, we will repeat this testing using NSight.

To measure the accuracy of our tool, we collect data with the GPU sampler while running a workload and compare that to the same data collected using NVprof. We do not expect any accuracy issues since we use the same collection interface that NVprof does, which is CUPTI. Our workload comprises two applications, the Rodinia BFS benchmark and a CUDA VectorAdd kernel. We run each application 10 times on a single GPU and report the average for each event in the table below. For BFS we use graph1MW_6.txt, which represents a 1 Million node graph and is

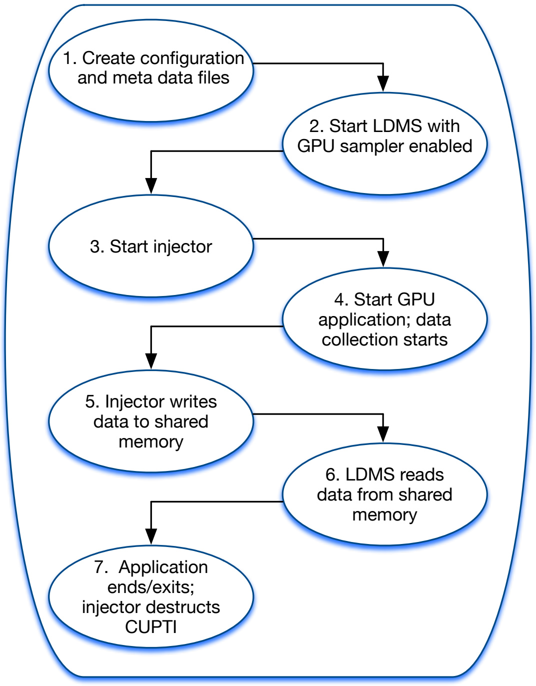

1. Create configuration and meta data files
2. Start LDMS with GPU sampler enabled
3. Start injector
4. Start GPU application; data collection starts
5. Injector writes data to shared memory
6. LDMS reads data from shared memory
7. Application ends/exits; injector destructs CUPTI

Figure 26: Sampling a GPU Application

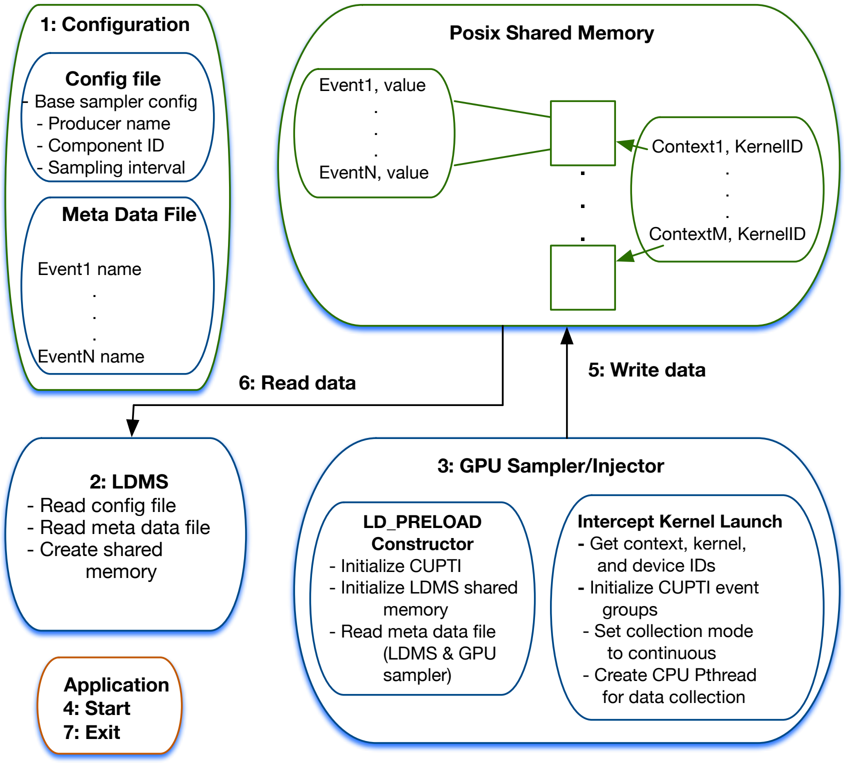


Figure 27: GPU Sampler Implementation

Table 11: Accuracy GPU Sampler vs NVprof, Rodinia BFS

Event Name	GPU Sampler	NVProf	Ratio
inst_executed	375187	375187	1
active_cycles	350547	357185	1.02
inst_issued	439578	439578	1
thread_inst_executed	12002790	12002790	1
warps_launched	31264	31264	1

Table 12: Accuracy GPU Sampler vs NVprof, VectorAdd Kernel

Event Name	GPU Sampler	NVProf	Ratio
inst_executed	281250000	281250000	1
active_cycles	366639640	366621830	1.00
inst_issued	229122701	229028849	1.00
thread_inst_executed	9000000000	9000000000	1
warps_launched	15625000	15625000	1

provided as an example by the Rodinia benchmark suite. For VectorAdd, we use a vector addition of 500000000 elements. Tables 11 and 12 show the results for these two applications, respectively. Although all of the results show very high accuracy, the differences in active cycles are the most curious. Fully investigating this will be part of future work.

To measure the timing overhead, we run a workload by itself, then we do individual runs for each of the cases with the GPU sampler and with NVprof. The sampler is currently being refactored often, so our overhead measurement methodology is based on using a single application and 5 event groups. In future work, we plan to do more exhaustive testing. We check the overhead each time we refactor the sampler.

We chose five event groups to use in overhead testing that represent a broad range of behaviors. These groups and their respective events are shown in Table 13. The *fb_* events count the number of read/write requests sent to different sub-partitions of the DRAM unit; the *l2_* events count the number of read/write misses from the L2 cache to sub-partitions of the DRAM. The *prof_trigger*

Table 13: NVprof/GPU Sampler Event Groups

Group	Events	Group	Events
Group 1	shared_ld_transactions shared_st_transactions elapsed_cycles_sm inst_executed_fp16_pipe_s0 inst_executed_fp16_pipe_s1	Group 3	prof_trigger_00 prof_trigger_01 prof_trigger_02 prof_trigger_03 active_warps active_cycles
Group 2	fb_subp0_read_sectors fb_subp1_read_sectors fb_subp0_write_sectors fb_subp1_write_sectors	Group 4	l2_subp0_read_sector_misses l2_subp1_read_sector_misses l2_subp0_write_sector_misses l2_subp1_write_sector_misses
		Group 5	active_cycles_sys elapsed_cycles_sys

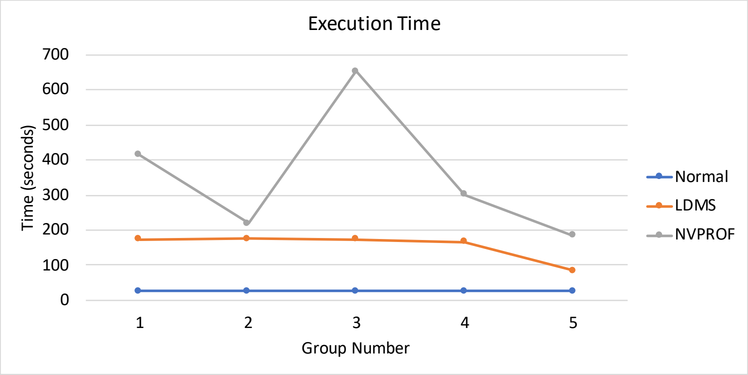


Figure 28: GPU Sampler Overhead: Time

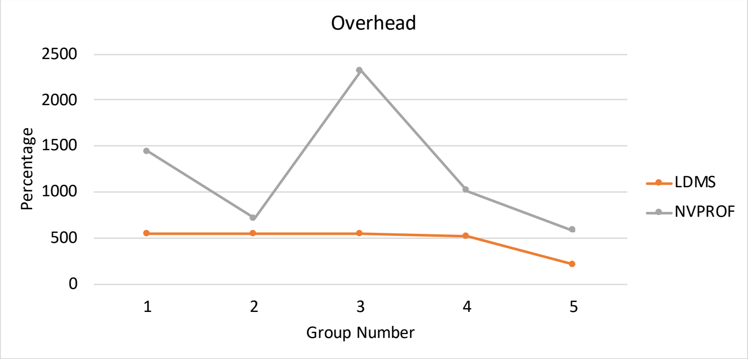


Figure 29: GPU Sampler Overhead: Percent

events are not meaningful here and are triggers used for user instrumentation of the kernel code. We use this group so we can measure the *active_warps* and *active_cycles* events. The remainder of the events in Table 13 are easily interpreted.

For this testing, we use the LAMMPS application with input as shown in Table 8. Figure 28 shows the overhead in seconds for each of the five event groups using both NVprof and the GPU Sampler for measurement. Figure 29 shows this overhead as a percentage of the base run time (without any performance measurement). As can be seen, the overhead of both tools is significant. This comes from the multiple runs of the kernel for the various event groups (i.e., to get event data from five different groups, five runs of each CUDA kernel are executed) and the fact that kernel launch is expensive. Multiple runs are required because of the lack of a hardware performance counter multiplexing feature implemented in CUPTI. Multiplexing in CPUs is done in a software layer (either the kernel or the hardware performance counter interface), but this has not been done yet for GPUs. We did verify this with Nvidia GPU performance engineers. We hypothesize that the overhead of the GPU sampler is smaller than that of NVprof only because we did not implement fixed delays for various functionality in the tool. For example, when a new thread reads context, we implement only the delay that's required to read that context but this time can be variable dependent on context size. Note the large increase in overhead for the group 3 measurements. We investigated this a bit but are still not sure why this is happening. Fully investigating this will also be part of future work. We are still improving and re-factoring the sampler to be more resilient, so the overhead may change in the future.

6.2.1 Conclusion, Lessons Learned and Future Work

In this work, we developed an initial GPU sampler to be used as a plugin with the LDMS monitoring core. This is the first GPU sampler for LDMS of its kind. This sampler currently has only been tested on IBM Power9-based systems and software stacks with NVidia GPUs.

Developing this tool proved to be far more difficult than we anticipated. During development, we exposed some bugs in the LDMS core that had to be fixed before proceeding. We have also exposed some issues in features of the LDMS core that need to be modified. All of the bugs and problems that have been exposed in LDMS through this work have been reported and either have been addressed/resolved or will be in the future.

We also encountered many problems with inconsistencies in software stacks across systems that are supposed to be identical. It's hard to even enumerate the systems issues that we helped resolve on multiple machines with system administrators over the past year, just through exercising these systems through tool development and validation. We also lost an enormous amount of time due to systems being out of operation either for service or for unexpected problems. Because of this, we have ported and adjusted inputs for all of the GPU applications we use for validation and testing across multiple platforms so if one system is down, we can use another. This turned out to be a really good strategy and we will continue doing this in the future in all projects.

The sampler works and collects accurate data, however, the tool still has some issues that we are working on. We believe that we have some timing and memory problems because at times, we do see some unstable behavior. We are working on a rigorous code review and debug cycle to identify and fix these issues.

To date, we have tested the tool on NVidia GPUs in platforms that only contain IBM processors. We plan to port the tool to an Intel-based system that has some nodes with NVidia GPUs. We also will port to an Arm-based system also with NVidia GPUs. Some of the problems that we see with the tool may be an artifact of the IBM software stack.

Finally, we will perform more rigorous validation when the tool matures to be more production-hardened. We will do overhead and accuracy comparisons using both NVprof and Nsight in the future.

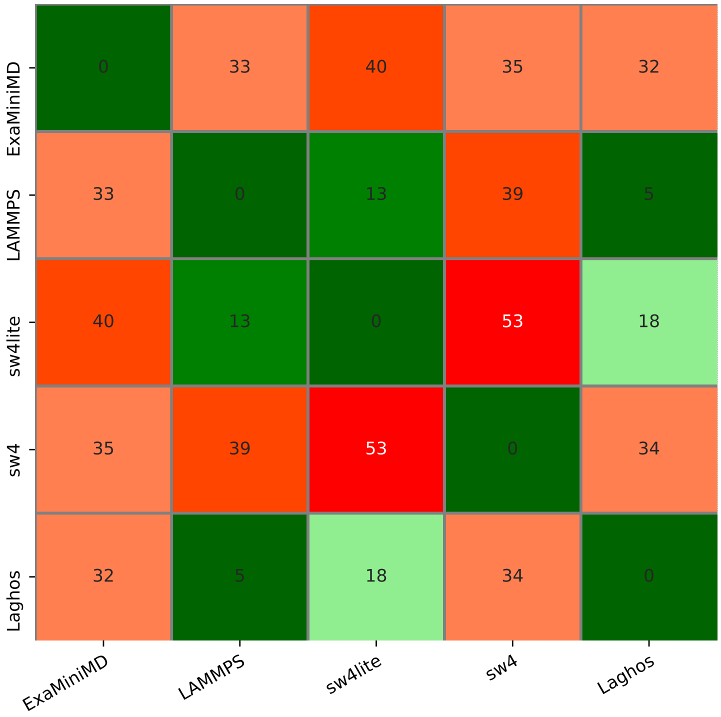


Figure 30: Cosine Similarity, NVidia Tesla V100

7 Cosine Similarity of Proxy/Parent Application Performance on GPUs

Cosine similarity applied to the comparison of GPU proxy and parent application implementations is synonymous with the CPU case. We run the applications on a Tesla V100 GPU and collect all available events with the exception of events that are already normalized. For example, CUPTI supports many "events" that are returned as a ratio. These include events such as *flop_dp_efficiency*, which is the ratio of achieved to peak double-precision floating-point operations. We remove events that are returned as a ratio because we want the flexibility to appropriately normalize the data if need be. The number of events collected is on the order of 100+. So for each application, the vector used for cosine similarity comprises these 100+ events.

We ran these applications on a single GPU rather than multiple GPUs for simplicity and speed. In the future, more extensive experimentation will be done. These are our initial experiments done using the initial version of the GPU sampler.

Figure 30 presents the initial cosine similarity results for a small suite of ECP applications. Not all of the ECP applications currently have GPU implementations. We aim to increase the number of GPU proxy/parent pairs and applications in general in the future. The threshold scale of similarity used here is the same as that presented in Section 5.3. Note that neither of the proxy/parent pairs, ExaMiniMD/LAMMPS and sw4lite/sw4. show good similarity here. In fact, the angles between them are large. LAMMPS and sw4lite show some similarity and they both show some similarity with Laghos. ExaMiniMD and sw4 are outliers, showing poor similarity with all of the other applications.

Because the year was mostly dedicated to developing the GPU sampler tool, these are very initial results for which we have not generated any additional performance data to help understand the similarities and differences between these applications.

7.1 Conclusion and Future Work

We present an initial GPU cosine similarity for two proxy/parent pairs and a single application. The results show that proxy/parent pairs are not similar, but there is some similarity between these applications.

Future work will include examining the actual GPU code to determine if proxy/parent pair implementations are fundamentally different. This could be the case for ExaMiniMD and LAMMPS as their CPU code base is significantly different. However, the results for sw4lite and sw4 are surprising.

In this next year, we will aim to obtain more GPU implementations of the ECP applications and their proxies. We obtained implementations that we could easily find in open source. We suspect that code teams have implementations that we may be able to get, but are not yet released to the public.

We also plan to collect data that supports the results from cosine similarity for GPUs. We do this in the CPU arena, where we extract probably around 100 metrics pertaining to cache, memory, pipeline behavior, instruction mix, and floating-point operations and we use these metrics to help validate the cosine similarity data.

For GPU analysis, we also need to group events according to architectural components as we do for CPUs so that we can understand where overall behavior between applications diverges. This will also aid in determining which proxy applications can be confidently used, for example, for GPU cache and memory studies, This and the other work mentioned above will be reflected in the proxy application assessment milestones for the next fiscal year.

8 Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC, for the U.S. Department of Energy Under contract DE-AC05-00OR22725.

Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under Contract No. 89233218CNA000001.

Argonne National Laboratory is managed by UChicago Argonne LLC for the U.S. Department of Energy under contract DE-AC02-06CH11357.

The Lawrence Berkeley National Laboratory is a national laboratory managed by the University of California for the U.S. Department of Energy under Contract Number DE-AC02-05CH11231.

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Input Configuration File for AMReX-Astro Castro Sedov

Listing 1: Input configuration file for the AMReX-Astro Castro Sedov `inputs.2d.cyl_in_cartcoords` case.

```
# ------------------ INPUTS TO MAIN PROGRAM -------------------
max_step = 500
stop_time = 0.1

# PROBLEM SIZE & GEOMETRY
geometry.is_periodic = 0 0
geometry.coord_sys = 0 # 0 => cart
geometry.prob_lo = 0 0
geometry.prob_hi = 1 1
amr.n_cell = 32 32

# >>>>>>>>>>>>> BC FLAGS <<<<<<<<<<<<<<<<
# 0 = Interior 3 = Symmetry
# 1 = Inflow 4 = SlipWall
# 2 = Outflow 5 = NoSlipWall
# >>>>>>>>>>>>> BC FLAGS <<<<<<<<<<<<<<<<
castro.lo_bc = 2 2
castro.hi_bc = 2 2

# WHICH PHYSICS
castro.do_hydro = 1
castro.do_react = 0

# TIME STEP CONTROL
castro.cfl = 0.5 # cfl number for hyperbolic system
castro.init_shrink = 0.01 # scale back initial timestep
castro.change_max = 1.1 # maximum increase in dt over successive steps

# DIAGNOSTICS & VERBOSITY
castro.sum_interval = 1 # timesteps between computing mass
castro.v = 1 # verbosity in Castro.cpp
amr.v = 1 # verbosity in Amr.cpp
#amr.grid_log = grdlog # name of grid logging file

# REFINEMENT / REGRIDDING
amr.max_level = 3 # maximum level number allowed
amr.ref_ratio = 2 2 2 2 # refinement ratio
amr.regrid_int = 2 # how often to regrid
amr.blocking_factor = 8 # block factor in grid generation
amr.max_grid_size = 256

# CHECKPOINT FILES
amr.check_file = sedov_2d_cyl_in_cart_chk # root name of checkpoint file
amr.check_int = 20 # number of timesteps between checkpoints

# PLOTFILES
amr.plot_file = sedov_2d_cyl_in_cart_plt
amr.plot_int = 20
amr.derive_plot_vars=ALL

# PROBIN FILENAME
amr.probin_file = probin.2d.cyl_in_cartcoords
```

Listing 2: MACSio parameters for final fit on Summit

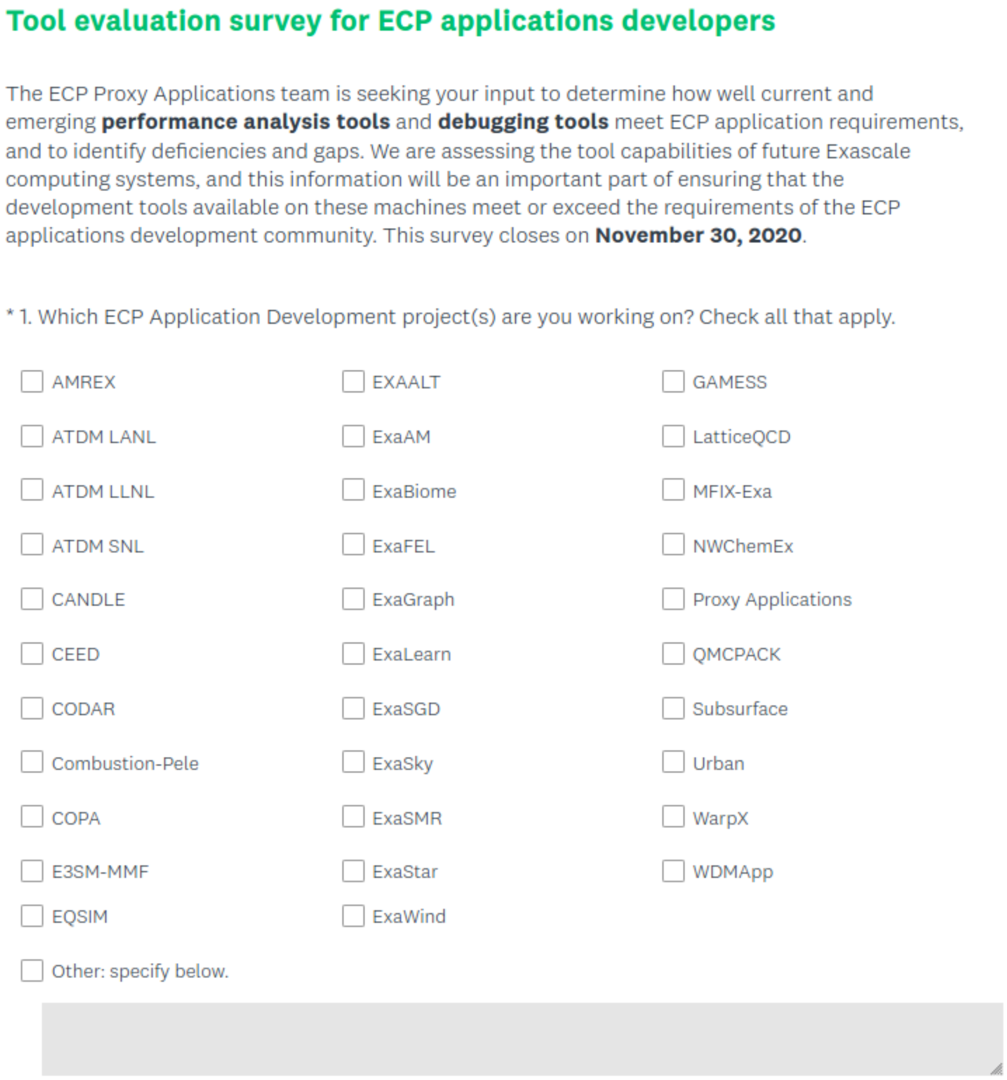
```
jsrun -n 32 ${MACSIO_EXEC} \
 --interface miftmpl \
 --parallel_file_mode MIF 32\
 --num_dumps 21 \
 --part_size 1600000 \
 --avg_num_parts 1 \
 --vars_per_part 1 \
 --compute_time 0.88 \
 --meta_size 0 51891 \
 --dataset_growth 1.013075
```

B Full Survey Form

The exact survey form as seen on the SurveyMonkey website is reproduced in this appendix.

B.1 Preamble and Question 1

This was for the November survey; the only difference between this form and the one sent in January was the closing date of the survey.



Tool evaluation survey for ECP applications developers

The ECP Proxy Applications team is seeking your input to determine how well current and emerging **performance analysis tools** and **debugging tools** meet ECP application requirements, and to identify deficiencies and gaps. We are assessing the tool capabilities of future Exascale computing systems, and this information will be an important part of ensuring that the development tools available on these machines meet or exceed the requirements of the ECP applications development community. This survey closes on **November 30, 2020**.

* 1. Which ECP Application Development project(s) are you working on? Check all that apply.

☐ AMREX
☐ ATDM LANL
☐ ATDM LLNL
☐ ATDM SNL
☐ CANDLE
☐ CEED
☐ CODAR
☐ Combustion-Pele
☐ COPA
☐ E3SM-MMF
☐ EQSIM

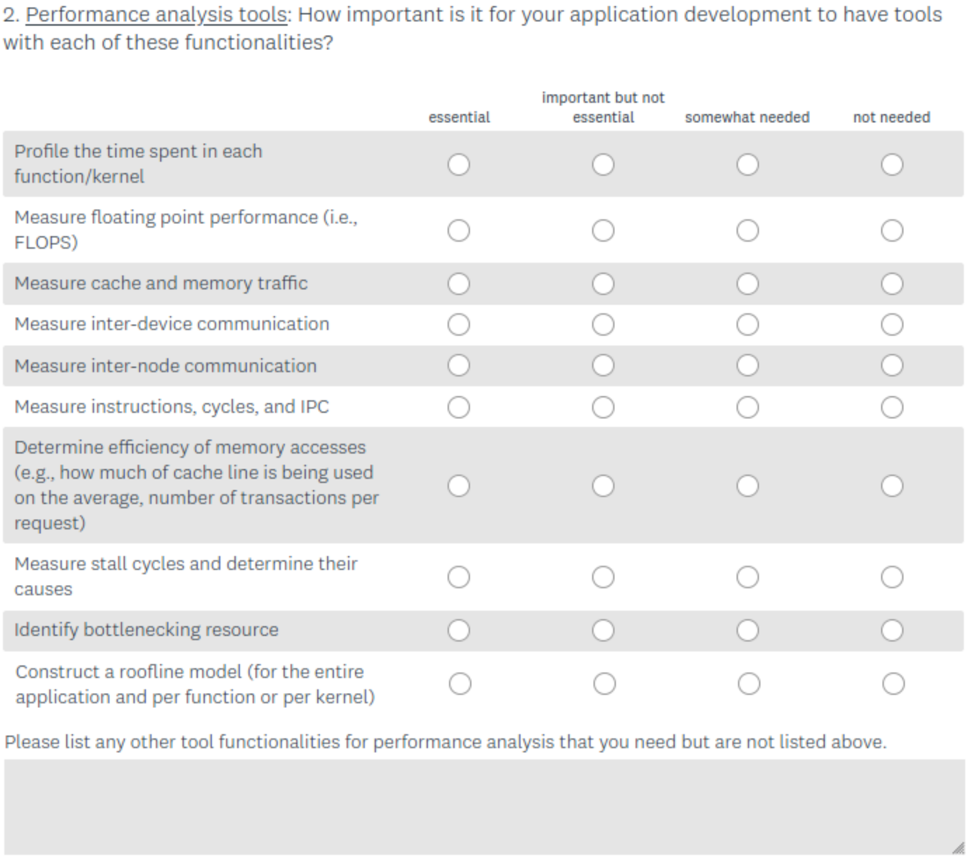
☐ EXAALT
☐ ExaAM
☐ ExaBiome
☐ ExaFEL
☐ ExaGraph
☐ ExaLearn
☐ ExaSGD
☐ ExaSky
☐ ExaSMR
☐ ExaStar
☐ ExaWind

☐ GAMESS
☐ LatticeQCD
☐ MFIX-Exa
☐ NWChemEx
☐ Proxy Applications
☐ QMCPACK
☐ Subsurface
☐ Urban
☐ WarpX
☐ WDMApp

☐ Other: specify below.

B.2 Question 2

The importance of functionalities of performance analysis tools



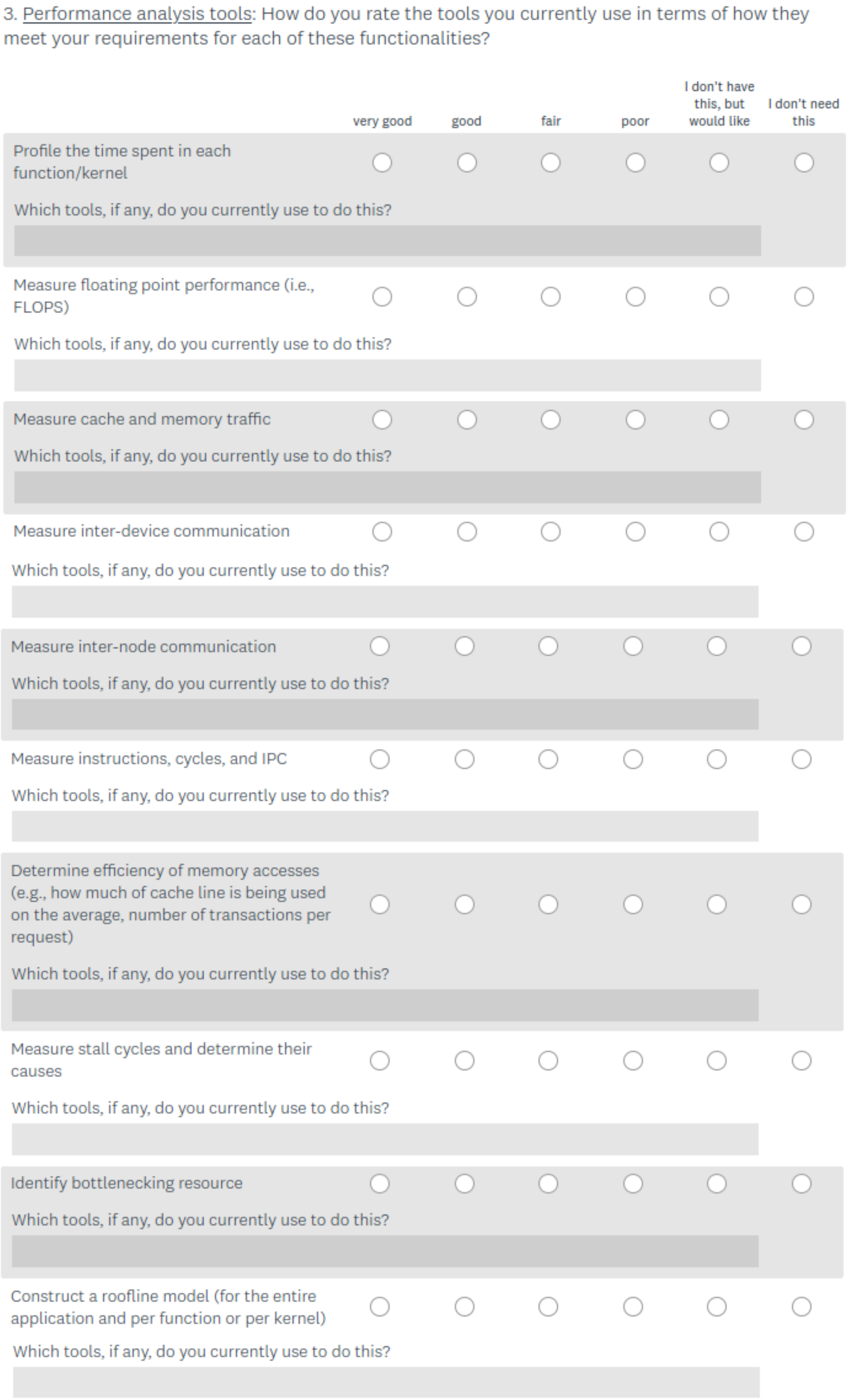
2. Performance analysis tools: How important is it for your application development to have tools with each of these functionalities?

	essential	important but not essential	somewhat needed	not needed
Profile the time spent in each function/kernel	○	○	○	○
Measure floating point performance (i.e., FLOPS)	○	○	○	○
Measure cache and memory traffic	○	○	○	○
Measure inter-device communication	○	○	○	○
Measure inter-node communication	○	○	○	○
Measure instructions, cycles, and IPC	○	○	○	○
Determine efficiency of memory accesses (e.g., how much of cache line is being used on the average, number of transactions per request)	○	○	○	○
Measure stall cycles and determine their causes	○	○	○	○
Identify bottlenecking resource	○	○	○	○
Construct a roofline model (for the entire application and per function or per kernel)	○	○	○	○

Please list any other tool functionalities for performance analysis that you need but are not listed above.

B.3 Question 3

The satisfaction with performance analysis tools.

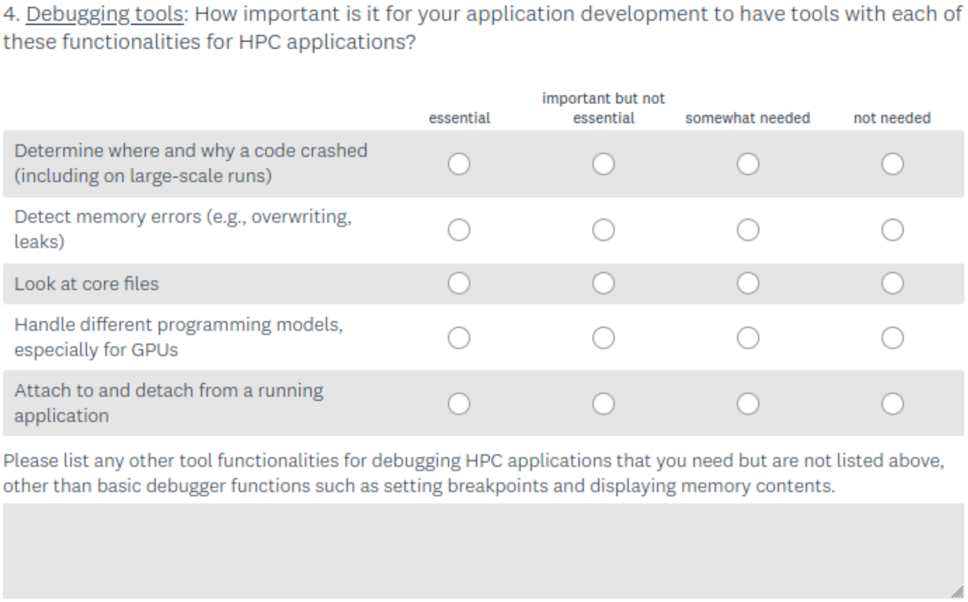


3. Performance analysis tools: How do you rate the tools you currently use in terms of how they meet your requirements for each of these functionalities?

	very good	good	fair	poor	I don't have this, but would like	I don't need this
Profile the time spent in each function/kernel	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure floating point performance (i.e., FLOPS)	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure cache and memory traffic	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure inter-device communication	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure inter-node communication	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure instructions, cycles, and IPC	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Determine efficiency of memory accesses (e.g., how much of cache line is being used on the average, number of transactions per request)	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Measure stall cycles and determine their causes	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Identify bottlenecking resource	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Construct a roofline model (for the entire application and per function or per kernel)	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						

B.4 Questions 4 and 5

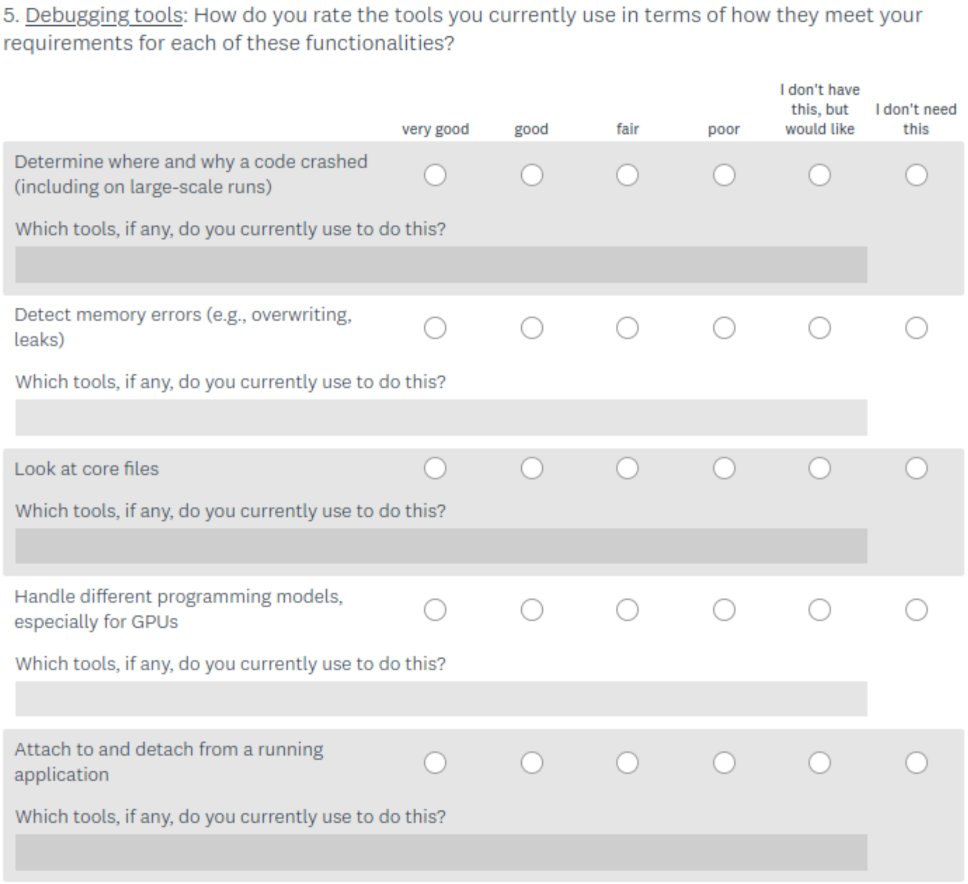
The importance of functionalities of debugging tools, and satisfaction with those tools



4. Debugging tools: How important is it for your application development to have tools with each of these functionalities for HPC applications?

	essential	important but not essential	somewhat needed	not needed
Determine where and why a code crashed (including on large-scale runs)	○	○	○	○
Detect memory errors (e.g., overwriting, leaks)	○	○	○	○
Look at core files	○	○	○	○
Handle different programming models, especially for GPUs	○	○	○	○
Attach to and detach from a running application	○	○	○	○

Please list any other tool functionalities for debugging HPC applications that you need but are not listed above, other than basic debugger functions such as setting breakpoints and displaying memory contents.



5. Debugging tools: How do you rate the tools you currently use in terms of how they meet your requirements for each of these functionalities?

	very good	good	fair	poor	I don't have this, but would like	I don't need this
Determine where and why a code crashed (including on large-scale runs)	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Detect memory errors (e.g., overwriting, leaks)	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Look at core files	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Handle different programming models, especially for GPUs	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						
Attach to and detach from a running application	○	○	○	○	○	○
Which tools, if any, do you currently use to do this?						

C Tool Checklists

C.1 Example Checklist for Performance Tools

Functionality	**Tool1**	**Tool2**	**Tool3**
Hardware performance counters			
CPU	□	□	□
GPU	□	□	□
Resource usage			
Disks	□	□	□
Network	□	□	□
Traffic			
Number of transactions per request (cache)	□	□	□
Number of transactions per request (memory)	□	□	□
Inter-process	□	□	□
Inter-device	□	□	□
Inter-node	□	□	□
Granularity			
Broad-brush modules, individual methods, etc.	□	□	□
Count how many times targets are called	□	□	□
Analysis			
Compare measurements, show largest differences between runs	□	□	□
Characterize how time was spent	□	□	□
Costly or poorly performing code	□	□	□
High memory allocation or many cache misses	□	□	□
Available resources but blocked, idle waits	□	□	□
Causes of stall cycles	□	□	□
Highlight bottlenecks, critical path	□	□	□
Cache utilization	□	□	□
Resource utilization	□	□	□
Timing	□	□	□
Storage allocation patterns	□	□	□
Core-aware analysis for NUMA	□	□	□
Resources consumed	□	□	□
Break out statistics across processes and threads	□	□	□
Accumulate timeline to show changes over run, e.g., with cache population	□	□	□
Aggregate across threads and nodes for resource usage summary	□	□	□
Construct roofline model			
For the entire application	□	□	□
Per function or per kernel	□	□	□
Generate required metrics for POP model			
For the entire application	□	□	□
Per function or per kernel	□	□	□
Application size			
Small test code	□	□	□
Large code (e.g. O(1000) tasks)	□	□	□
Full-scale code	□	□	□
Collection techniques			
Event-based profiling	□	□	□
Statistical profiling	□	□	□
Manual instrumentation	□	□	□
Automatic instrumentation	□	□	□
Tracing	□	□	□

C.2 Example Checklist for Debugging Tools

Functionality	**Tool1**	**Tool2**	**Tool3**
Process/thread control			
Multi-process support	□	□	□
Multi-thread support	□	□	□
Independently controllable process groups	□	□	□
Independently controllable thread groups	□	□	□
Control thread(s) independently of other processes / threads	□	□	□
Follow execution across threads / processes	□	□	□
Source code			
View code as disassembly	□	□	□
View code as source	□	□	□
Determine what a symbol refers to	□	□	□
Program state			
Show current status of processes and/or threads	□	□	□
Show call stack	□	□	□
Move up/down the call stack	□	□	□
Program data			
Display the contents of a variable	□	□	□
Evaluate an expression	□	□	□
Modify the contents of a variable	□	□	□
Program execution			
Stop execution	□	□	□
Start/continue execution	□	□	□
Step forward	□	□	□
Step backward	□	□	□
Terminate program execution	□	□	□
Breakpoints			
Set breakpoint on source line	□	□	□
Set breakpoint on function entry/exit	□	□	□
Run to breakpoint	□	□	□
Set conditions on breakpoints	□	□	□
Error conditions and asynchronous events			
Exceptions, including context from throw scope where possible	□	□	□
Signals, interrupts, process exit, etc.	□	□	□
Incorrect memory usage patterns (use of free, failure to free, etc.)	□	□	□
Monitor code execution			
Record execution	□	□	□
Replay execution	□	□	□
Watch values of variables	□	□	□
Programming model support			
Identify message-passing or threading libraries? (MPI, OpenMP, etc.)	□	□	□
Catch and show passed messages and data	□	□	□
Application size			
Small test code	□	□	□
Large code (e.g. O(1000) tasks)	□	□	□
Full-scale code	□	□	□
Application support			
Able to obtain interactive control of application	□	□	□
Attach to and detach from a running application	□	□	□
Load application state from core file	□	□	□
Combined GPU/CPU debugging	□	□	□

References

[1] M. Adams, P. Colella, D. T. Graves, J.N. Johnson, N.D. Keen, T. J. Ligocki, D. F. Martin, P.W. McCorquodale, D. Modiano, T.D. Sternberg P.O. Schwartz, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory, 1999. URL: https://commons.lbl.gov/download/attachments/73468344/chomboDesign.pdf?version=1&modificationDate=1554672305006&api=v2.

[2] Anthony Agelastos et al. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC'14: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 154–165. IEEE Press, 2014.

[3] A. S. Almgren, V. E. Beckner, J. B. Bell, M. S. Day, L. H. Howell, C. C. Joggerst, M. J. Lijewski, A. Nonaka, M. Singer, and M. Zingale. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. *The Astrophysical Journal*, 715:1221–1238, June 2010. arXiv:1005.0114, doi:10.1088/0004-637X/715/2/1221.

[4] A. S. Almgren et al. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. *Astrophysical Journal*, 715:1221–1238, June 2010. doi:10.1088/0004-637X/715/2/1221.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. URL: https://doi.org/10.1137/141000671.

[6] V. Dobrev, Tz. Kolev, and R. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34:B606–B641, 2012. URL: https://doi.org/10.1137/120864672.

[7] Exascale Proxy Application Suite, 2020. URL: https://proxyapps.exascaleproject.org.

[8] Eirik Endeve, Ran Chu, Anthony Mezzacappa, and Bronson Messer. Towards a discontinuous galerkin method for the multi-group two-moment model of neutrino transport. Technical Report ORNL/TM-2017/501, Oak Ridge National Laboratory, 2017.

[9] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[10] S. Ghosh et al. MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems. In *IEEE/ACM Perf. Modeling, Benchmarking and Sim. of High Perf. Computer Systems (PMBS)*, November 2018.

[11] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC, Canada, May 2018.

[12] S. Habib et al. Hacc: Extreme scaling and performance across diverse architectures. *Commun. ACM*, 60(1):97–104, December 2016. doi:10.1145/3015569.

[13] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Appl. Num. Math.*, 41:155–177, 2002.

[14] P. R. C. Kent, Abdulgani Annaberdiyev, Anouar Benali, M. Chandler Bennett, Edgar Josué Landinez Borda, Peter Doak, Hongxia Hao, Kenneth D. Jordan, Jaron T. Krogel, lkka Kylänpää, Joonho Lee, Ye Luo, Fionn D. Malone, Cody A. Melton, Lubos Mitas, Miguel A. Morales, Eric Neuscamman, Fernando A. Reboredo, Brenda Rubenstein, Kayahan Saritas, Shiv Upadhyay, Guangming Wang, Shuai Zhang, and Luning Zhao. Qmcpack: Advances in the development, efficiency, and application of auxiliary field and real-space variational and diffusion quantum monte carlo. *Chemical Physics*, 152(17), 2020.

[15] P. R. C. Kent et al. QMCPACK: Advances in the Development, Efficiency, and Application of Auxiliary Field and Real-Space Variational and Diffusion Quantum Monte Carlo. *J. Chemical Physics*, 152(174105), 2020. doi:10.1063/5.0004860.

[16] Youngjae Kim and Raghul Gunasekaran. Understanding I/O workload characteristics of a Peta-scale storage system. *Journal of Supercomputing*, 71(3), 11 2014. doi:10.1007/s11227-014-1321-8.

[17] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2012.

[18] Tianyu Liu, Noah Wolfe, Christopher D. Carothers, Wei Ji, and X. George Xu. Optimizing the monte carlo neutron cross-section construction code xsbench for mic and gpu platforms. *Nuclear Science and Engineering*, 185(1):232–242, 2017. doi:10.13182/NSE16-33.

[19] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330 – 354, 2000. URL: http://www.sciencedirect.com/science/article/pii/S0010465599005019, doi:https://doi.org/10.1016/S0010-4655(99)00501-9.

[20] Mark Miller. Design & Implementation of MACSio. Technical report, Lawrence Livermore National Laboratory (LLNL), 2015. URL: https://macsio.readthedocs.io/en/latest/_downloads/1f9c7922040985a619639fd5947d36ea/macsio_design.pdf.

[21] Carlos A. de Moura and Carlos S. Kubrusly. *The Courant-Friedrichs-Lewy (CFL) Condition: 80 Years After Its Discovery.* Birkhäuser Basel, 2012.

[22] Nek5000 version 19.0, December 2019. URL: https://nek5000.mcs.anl.gov.

[23] Oak Ridge Leadership Computing Facility. OLCF JupyterHub. URL: https://jupyter.olcf.ornl.gov/.

[24] Oak Ridge Leadership Computing Facility. Summit supercomputer. URL: https://www.olcf.ornl.gov/summit/.

[25] N.A. Petersson and B. Sjrogreen. Sw4 v2.0. computational infrastructure of geodynamics, 2017. doi:10.5281/zenodo.1045297.

[26] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, March 1995. doi:10.1006/jcph.1995.1039.

[27] D. Richards et al. FY18 Proxy App Suite Release. Milestone Report for the ECP Proxy App Project. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, 2018.

[28] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. Openmc: A state-of-the-art monte carlo code for research and development. *Ann. Nucl. Energy*, 82:90–97, 2015. URL: https://doi.org/10.1016/j.anucene.2014.07.048.

[29] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[30] Sahar Sohangir and Dingding Wang. Improved sqrt-cosine similarity measurement. *Journal of Big Data*, 4(1):25, 2017. doi:10.1002/cpe.3587.

[31] Aidan P. Thompson and Christian Robert Trott. A brief description of the kokkos implementation of the snap potential in examinimd. 11 2017. doi:10.2172/1409290.

[32] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. URL: https://www.mcs.anl.gov/papers/P5064-0114.pdf.

[33] H. Vincenti and J.-L. Vay. Detailed analysis of the effects of stencil spatial variations with arbitrary high-order finite-difference maxwell solver. *Computer Physics Communications*, 200:147, 2016.

[34] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, May 2019. doi:10.21105/joss.01370.