SAND2022-6149R

Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone ADCD-504-28

Jeanine Cook[1], Omar Aaziz[1], Si Chen[3], William Godoy[2], Amy Jo Powell[1], Gregory Watson[2], Courtenay Vaughan[1], Avani Wildani[3], and The ECP Proxy App Team[4]

[1]Sandia National Laboratories, Albuquerque, NM
[2]Oak Ridge National Laboratory, Oak Ridge, TN
[3]Emory University
[4]https://proxyapps.exascaleproject.org/team

April 2022

SAND-XXXXXX

Contents

1 Executive Summary

The ECP Proxy Application Project has an annual milestone to assess the state of ECP proxy applications and their role in the overall ECP ecosystem. Our FY22 March/April milestone (ADCD-504-28) proposed to:

> Assess the fidelity of proxy applications compared to their respective parents in terms of kernel and I/O behavior, and predictability. Similarity techniques will be applied for quantitative comparison of proxy/parent kernel behavior. MACSio evaluation will continue and support for OpenPMD backends will be explored. The execution time predictability of proxy apps with respect to their parents will be explored through a carefully designed scaling study and code comparisons.

Note that in this FY, we also have quantitative assessment milestones that are due in September and are, therefore, not included in the description above or in this report. Another report on these deliverables will be generated and submitted upon completion of these milestones.

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

- Study the ability of MACSio to represent I/O workloads of adaptive mesh codes.
- Re-define the performance counter groups for contemporary Intel and IBM platforms to better match specific hardware components and to better align across platforms (make cross-platform comparison more accurate). Perform cosine similarity study based on the new performance counter groups on the Intel and IBM P9 platforms.
- Perform detailed analysis of performance counter data to accurately average and align the data to maintain phases across all executions and develop methods to reduce the set of collected performance counters used in cosine similarity analysis.
- Apply a quantitative similarity comparison between proxy and parent CPU kernels.
- Perform scaling studies to understand the accuracy of predictability of the parent performance using its respective proxy application.

This report presents highlights of these efforts.

Section 2 reports on the evaluation of MACSio for block structured AMR codes.

Section 3 reports the new performance counter groups we defined for all of the available IBM Power9 and Intel Skylake performance events. We also present the results of the proxy/parent application similarity studies we did on the two system platforms using the new event groups.

Section 4 presents how we accurately process all of the data we collect for an application on all of its ranks to generate a single data vector to be used in cosine similarity analysis.

Section 5 reports the work done to quantify the similarity in hardware usage/behavior of proxy/parent primary kernels.

Finally, Section 6 describes the study and results that quantify the accuracy in which a proxy app predicts the execution time of its respective parent.

2 Modeling pre-Exascale AMR Parallel I/O Workloads via Proxy Applications

The work presented here satisfies the ADCD504-32 quantitative assessment milestone:
Continue MACSio evaluation on AMRex based codes. Submit results in a workshop/conference paper.

As we approach the exascale era in the next generation of supercomputers [22], it is crucial to understand high-performance computing (HPC) computational, communication and input output (I/O) workloads in massive, large-scale scientific applications in order to effectively utilize the power of these systems [9, 24]. This understanding becomes extremely complex and sophisticated due to the large number of factors involved in the end-to-end operation, from the existing algorithms in a scientific application or workflow, to the computing and communication patterns when fine-tuning the software stack parameters for a particular platform.

Adaptive Mesh Refinement (AMR) [10] is a powerful technique used for solving partial differential equations. AMR simulations running at scale are computationally and I/O intensive [25]. The I/O characteristics rely heavily on several aspects related to the parallelization, load balancing and domain decomposition strategies, user input specifying output frequency, and the problem-dependant configuration of the physics in a particular application. In addition, parallel I/O capabilities are often managed independently from the computational capabilities in large leadership facilities, thus applications can increasingly become I/O bound due to the different rates of improvements in hardware (parallel file system and storage throughput), and software (parallel I/O libraries) when compared to computational capacity [25].

We evaluated the use of the Multipurpose, Application Centric, Scalable I/O (MACSio) [27] framework as a lightweight proxy solution to model I/O characteristics of AMR simulations based on the well-established AMReX framework for massively parallel, block-structured AMR applications [41]. We built a simple "generalized" model based on data collected from AMReX-based inputs and analysis data outputs that can be translated to MACSio command-line parameters in order to replicate data production patterns observed in a simulation. The goal of this effort is to close the gap between the I/O characterization of specific use-cases in AMReX-based applications and using MACSio as a general, lightweight, kernel proxy I/O tool that can capture the non-linearities of the observed AMR data generation. Having a calibrated lightweight proxy application for I/O is a valuable tool in the codesign process for understanding I/O system dynamic characteristics and trade-offs across AMReX-based codes on exascale systems without having to undertake the deployment of full-scale applications.

2.1 Methodology

The methodology used to characterize the output portion of the I/O from parallel runs of an AMR application and the model scope and assumptions for the translation to a proxy application using MACSio input parameters are similar to those identified by Dickson *et al.* [11] in their work with MACSio. This can be summarized as follows:

1. Run existing simulations as a point of reference for the I/O generated by the AMReX Castro infrastructure
2. Identify the important input parameters that drive the I/O simulation outputs (checkpoint and analysis data) to determine the basis for a variability study

3. Characterize the output generated given a variety of input parameters in AMReX Castro's configuration file
4. Evaluate the I/O behavior of AMReX Castro and determine what functionality (if any) is missing from MACSio in order to replicate this behavior

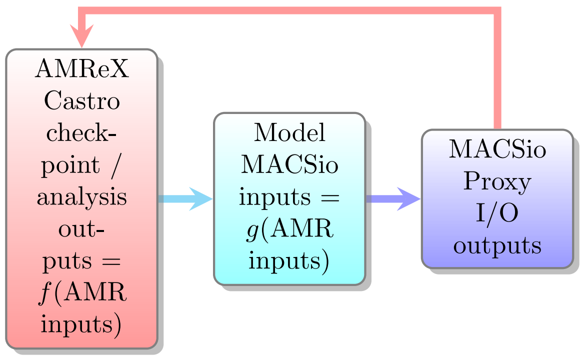

AMReX Castro checkpoint / analysis outputs = f(AMR inputs)
Model MACSio inputs = g(AMR inputs)
MACSio Proxy I/O outputs

Figure 1: High level flow description of the methodology used in this study to generate a MACSio Proxy I/O model application that captures AMReX based I/O. g represents the functional form of the proposed model based on MACSio functionality.

The flow is illustrated in Fig. 1. This shows a schematic representation of how AMReX Castro inputs and the generated output can be associated with a proxy I/O model. MACSio is used to simulate potential outputs using a set of "AMR inputs" typically given in a user configuration file. Since the goal is to first understand the role of I/O in an AMR simulation rather than the complexity of the simulation and/or computation, we chose to study the Sedov 2D cylindrical case in Cartesian coordinates for a typical input file[1]. This test is readily available in the Castro suite of examples and shows a physical symmetry and can be used to isolate the AMR effects on the I/O on a simple problem.

2.1.1 AMReX Castro Parameterized Runs

The output directory structure for the generated analysis data (in the form of plot files) can be seen in Fig. 2. This default output is done using a $N - to - N$ pattern, where each of N message passing interface (MPI) tasks writes to a separate file, 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 directory level in files called `Cell_H`. Note that a file is only produced if there is data generated on a particular task at the corresponding mesh level. AMReX also supports the generation of checkpoint-restart data in a similar manner, but we focused on only the plot files for this particular study.

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 Summit using MPI for parallelization. We used a standard input configuration file as our baseline to understand the structure and size of the resulting output and to determine the input parameters that influence the output generation. Table 1 shows the input parameters that we focused on for this study.

Aside from those input parameters expected to have direct influence on the analysis data generation in Fig. 2, such as the frequency of output file generation, the number of cells at the base mesh

[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 per-step
│   ├── Header metadata file
│   ├── job_info metadata file
│   ├── Level_0 per-level
│   │   ├── Cell_D_00000 per-task file
│   │   ├── Cell_D_00001
│   │   ├── ...
│   │   ├── Cell_D_0000N
│   │   └── Cell_H mesh metadata file
│   ├── Level_1
│   ├── Level_2
│   ├── ..
│   └── Level_L
├── sedov_2d_cyl_in_cart_plt00020
├── ..
└── sedov_2d_cyl_in_cart_pltMMMMM
```

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

`amr.max_step`	maximum expected number of steps
`amr.n_cell`	number of cells at Level 0 in each direction
`amr.max_level`	maximum level of refinement allowed
`amr.plot_int`	frequency of plot outputs
`castro.cfl`	CFL condition

Table 1: Subset of AMReX Castro input configuration file parameters varied to understand output behavior in the Sedov hydrodynamics baseline case.

```
MACSio data output
├── data
│   └── macsio_json_{taskID}_{stepID}.json
│       ├── macsio_json_00000_000.json
│       ├── macsio_json_00000_001.json
│       ├── ...
│       ├── macsio_json_00000_{nsteps}.json
│       ├── ...
│       └── macsio_json_{nprocs}_{nsteps}.json
└── metadata
    └── macsio_json_root_{stepID}.json
        ├── macsio_json_root_000.json
        ├── macsio_json_root_001.json
        ├── ...
        └── macsio_json_root_{nsteps}.json
```

Figure 3: MACSio's $N-to-N$ output pattern using the `miftmpl` interface ordered by task and output step, in which `nsteps` is the total number of steps, and `nprocs` is the number of MPI tasks.

(Level 0), and the maximum number of steps, we also focused on parameters that might impact on the mesh refinement. The two primary candidates for these were the CFL condition and the maximum number of refinement levels allowed, so were also considered in the set of parameterized runs.

Each run generated corresponding data that was used to quantify the cumulative output sizes at each requested time interval, refinement level, and task, as described in the hierarchy shown in Fig. 2. The selected granularity made it easier to understand the data production at the lowest single file level, while also giving an idea of how balanced (or not) the output was in a simple AMReX application. The results are discussed in more detail in section 2.2.

2.1.2 Modeling outputs with MACSio as a Proxy

After characterizing the AMReX Castro I/O behavior, the next step was to provide a simple way to simulate the observed I/O patterns using a proxy application. One of the immediate trade-offs when replicating AMReX I/O patterns with MACSio is the level of granularity of the generated output shown in Fig. 2. While AMReX generates outputs based on the tuple $(timestep, refinement\_level, MPI\_task)$, MACSio only provides enough granularity to generate outputs based on $(timestep, MPI\_task)$. Nevertheless, the simplicity of MACSio makes it a worthy candidate for assessing if an approximate solution could simulate deterministic characteristics such as data size, computational overhead, and I/O burstiness at different scales. The latter allows a model to be constructed that could help practitioners understand random and dynamic system characteristics such as bandwidth, file system variability, and scalability, prior to running full AMReX-based simulations on different hardware platforms.

MACSio provides a simple command-line interface that allows the user to specify how to capture and generate several types of I/O characteristics. We used the default $N-to-N$ output generation from the AMReX-Castro Sedov cases to determine if MACSio can provide a valid approximation to AMReX-Castro I/O patterns for data generation at each timestep. This pattern is shown in Fig. 3 for data and metadata files generated from the MACSio executable.

Table 2 shows the MACSio parameters we used in this study to fine-tune the data generation.

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

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

Listing 1: Proxy app model formulation for mapping MACSio executable to AMReX Castro inputs on Summit.

```
jsrun -n nproc
macsio
 --interface miftmpl
 --parallel_file_mode MIF nproc
 --num_dumps $\frac{amr.max\_steps}{amr.plot\_int}$
 --part_size $f_{(amr.n\_cell)}$
 --avg_num_parts 1
 --vars_per_part 1
 --compute_time $f_{(platform,all\_inputs)}$
 --meta_size $f_{(all\_inputs)}$
 --dataset_growth $f_{(amr.n\_cell,castro.cfl,amr.max_level,...)}$
```

We found the most important parameters were the `interface`, `parallel_file_mode`, `part_size`, and `dataset_growth`. The latter parameter enables MACSio to approximate non-linear data generation. The ultimate challenge, however, was to fine-tune these parameters to create a proxy I/O model to a level of granularity that is helpful in identifying I/O characteristics in AMRex-Castro.

When comparing the AMReX and Castro inputs in Table 1 and MACSio inputs in Table 2, it can be seen that MACSio parameters as such: `interface` `parallel_file_mode`, and `num_dumps` can be easily mapped to the simulation inputs. As such, the constructed model in MACSio will have the functional form shown in Listing 1.

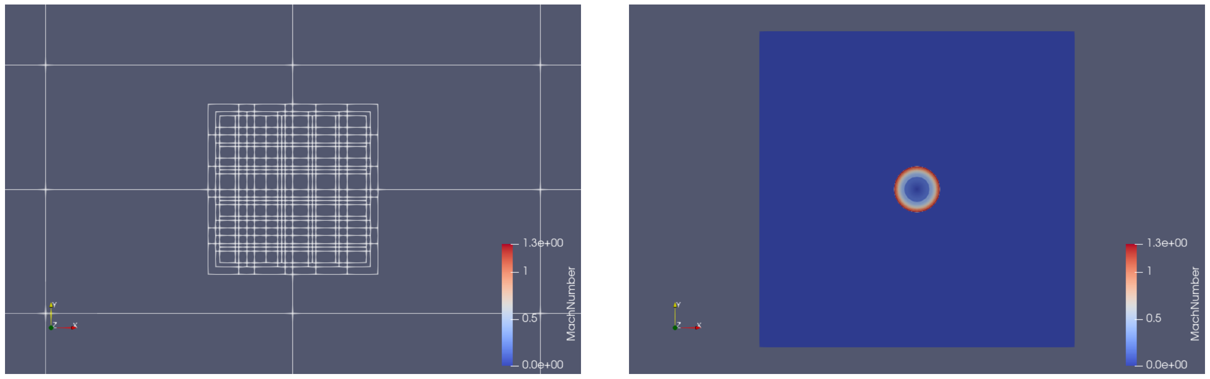
The remaining challenge then is how to determine the relationship between the `part_size` and the `dataset_growth` parameters as data is generated from AMR simulations. Other parameters that are "runtime" in nature such as `compute_time` and `meta_size` can be determined after collecting data runs. In particular, `compute_time` represents a degree of freedom that can be adjusted independently of "static" data size modeling for "dynamic" studies to fine-tune the I/O "burstiness" on a particular platform.

2.2 Results

This section applies the methodology shown in Section 2.1 on parameterized runs on the Castro solution of the 2D Sedov blast wave standard hydrodynamics test problem [36]. First, data output characteristic are presented for multiple configurations running on Summit. The next step is to construct a functional form of the model shown in Listing 1 via a minimization process varying the parameters in the MACSio executable. We also list the limitations, current scope and potential use of MACSio as a proxy application to model I/O in AMR simulations.

2.2.1 AMReX Castro Sedov Parameterized Outputs

The Castro Sedov hydro test generates a straight-forward I/O pattern in which each MPI task 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" 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. Since AMReX provides two methods for writing analysis data, `WriteSingleLevelPlotfile` and `WriteMultiLevelPlotfile`, we assumed that the latter is being used by Castro by default based on the file structure shown in Table 1. Figure 4 illustrates the AMR solution and expected physical results of the baseline symmetric Sedov test. It can be seen that the fine-grained refined levels are generated near the source terms of the hydrodynamics problem as expected in AMR formulations.



(a) (b)

Figure 4: Sedov hydro case 2D cylinder in Cartesian coordinates pivot case used as a benchmark in this study showing (a) AMR mesh showing the moving refined levels, and (b) solution for the Mach number after 20 timesteps. The lower levels follow the solution in the middle affecting overall load balancing, partitioning and I/O.

In total, 47 runs were performed on Summit at different scales while varying the parameters listed in Table 1 along with the number of files generated as a function of the number of MPI tasks (`nprocs`). The parameter ranges are summarized in Table 3 showing the scope of the present study, along with the `amr` and `castro` parameters that were used. The number of MPI tasks (`nprocs`) and the number of Summit nodes were also varied accordingly to run cases, from small mesh sizes of $32 \times 32 \approx 1\,K$ cells, to a large mesh size of $131,072 \times 131,072 \approx 17\,B$ cells using up to 512 Summit nodes, or equivalent to 1/9 of the 4,608 total system nodes. The expectation is that these ranges would provide enough information to understand the feasibility and scope of MACSio as a simple proxy application by isolating the computational aspects of the baseline AMReX Castro Sedov case.

To illustrate the non-regular output characteristics of the generated runs, the independent variable x is considered a function of the user-prescribed number of cells at the "L0" base level, `amr.ncells` in Table 1, and the count of the number of output events up to the maximum number of steps, `amr.max_step` in Table 1, thus resulting in a cumulative quantity. The rationale is that any data output sizes must be proportional to the size of the problem at each requested output event.

Parameter	Range
`amr.max_step`	40 - 1000
`amr.n_cell`	(32×32) - $(131,072 \times 131,072)$
`amr.max_level`	2 - 4 (1 to 3 levels)
`amr.plot_int`	1 - 20
`castro.cfl`	0.3 - 0.6
`nprocs`	1 - 1,024
`Summit nodes`	1 - 512 (1/9 total system)

Table 3: AMReX Castro input configuration file parameters range for the Sedov case running imulations to produce different output sizes.

As a result, the independent variable in our model is expressed as a function of the cumulative independent variable (x), and the dependent output size (y) at three hierarchical levels:

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

$$\begin{aligned} y &= data\_output_i \\ i &= \texttt{time step, level, task} \end{aligned} \tag{2}$$

The cumulative output sizes at each time step are shown in Fig. 5 for a subset of cases varying the parameters listed in Table 1. It can be seen the mixed linear and non-linear outputs characteristics for the selected range, while some of the larger cases are excluded for illustration purposes. It can be seen that several runs follow a near-linear trend as expected when the x variable in Eq. (1) grows with the L0 number of cells. However there is clearly another set of runs that deviate from this linear behavior, and this prompts further inspection in order to better understanding the nature of the generated output.

To understand the non-linear behavior in output sizes, one of the cases was selected as a pivot (case4) with $N_x = 512$ and $N_y = 512$ containing 20 outputs. As shown in Fig. 6, it is observed that while the CFL number 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 deviate from the observed linear trend in Fig. 5.

Further analysis is thus needed to understand how output is distributed among AMR levels, for each level in the output hierarchy ($timestep, level, task$) as illustrated in Fig. 2. The latter is shown in Fig. 7 for the pivot case (case4). As expected, the L0 level remains almost constant as it is mainly a function of the user-input number of cells. Subsequent levels in the AMR refinement (L1, L2) are more sensitive as they are driven by the physics (*e.g.* larger gradients) and the stability of the meshing algorithm (*e.g.* CFL number). An interesting aspect is that the overall "per-level" output shows a smooth variation. This opens the potential for creating a "kernel" solution like MACSio, by separating and superimposing the "linear" behavior from L0 and the "non-linear" smooth behavior of more refined levels (L1,L2).

Data generation as a function of each mesh level and task is shown in Fig. 8. The data produced by each MPI task gives an indication of the AMR effects on load balancing and the predictability of the I/O using a "kernel" tool like MACSio. Data is generated for 5 output steps, for a simulation

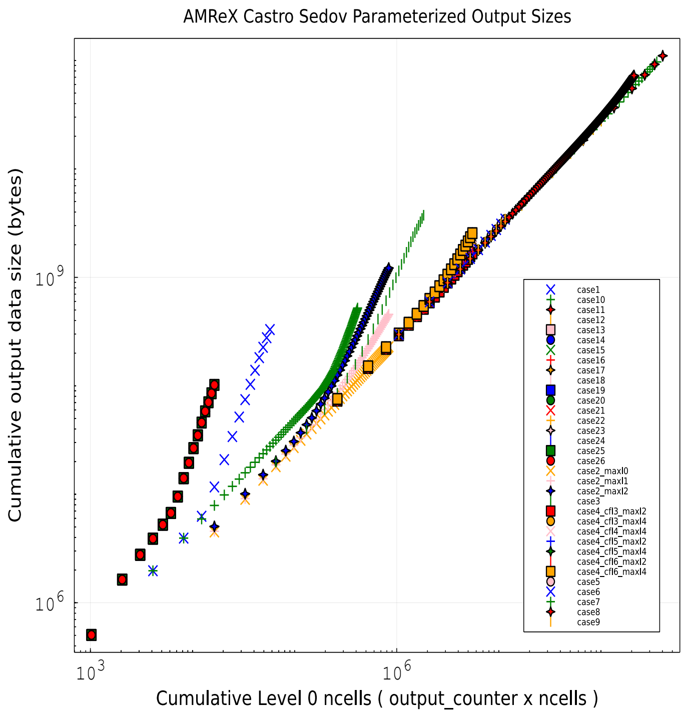


Figure 5: Cumulative output size per output step as a function of the cumulative number of output cells as defined in Eqs. (1) and (2) for the Sedov 2D case running on Summit.

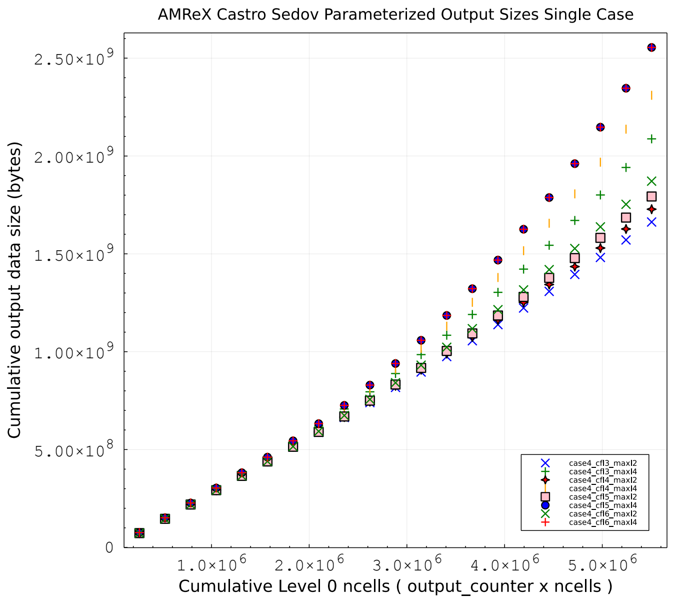


Figure 6: Dependency on the CFL number and AMR number of discretization levels for the cumulative output size for a single Sedov simulation run using 2 Summit nodes, 32 tasks and a L0 base mesh of 512×512 cells.

of the Sedov case (identified as case27) using 64 ranks on a $1,024 \times 1,024$ L0 base mesh. As can be seen, AMR effects result in unbalanced loads at all 4 levels of the resulting mesh hierarchy. Further investigation is needed to understand the relationship within AMR levels and MPI decomposition

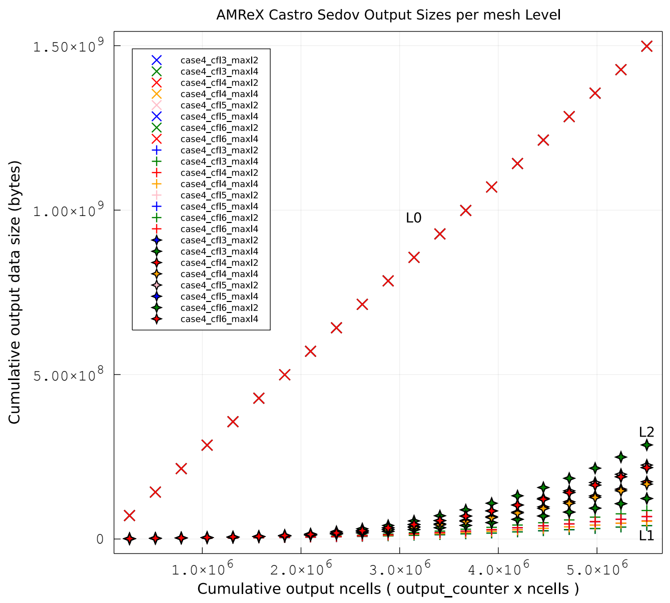


Figure 7: Dependency of the cumulative output size for each AMR level (L0,L1,L2) as a function of the cumulative number of output cells and CFL number for the Sedov 2D case.

algorithms in AMReX and possible predictability, even in simple configurations such as the Sedov case. Nevertheless, this is an indication that this level of granularity is highly volatile, even for a simple and symmetric problem. Therefore, the model construction using the current MACSio "kernel" characteristics can only simulate data output loads up to a mesh "level", but not at the "rank" level in the AMReX output. This current MACSio limitation is a consequence of its data output model described in Fig. 2.

2.2.2 MACSio Model

The next step is to provide a simple way to simulate the observed I/O patterns using a proxy application. As explained in Section 2.1, MACSio was used due to its command-line simplicity and versatility for high-performance simulations. While AMReX applications generate an I/O pattern that is dependent on timestep, refinement level, and output type, our goal was to demonstrate that MACSio could capture enough of the non-linearity behavior observed in Fig. 5 to provide an approximate solution that could simulate deterministic characteristics such as data size, computational overhead, and I/O patterns loads.

The AMReX Castro Sedov result (identified as case4) illustrated in Fig. 6 was selected as a baseline. The particular case for which the $cfl = 0.4$ using 4 levels was compared against several simulations using MACSio for which the `data_growth` parameter was calibrated to obtain a non-linear kernel trace for the generated output described in Fig. 3. The initial data size was calibrated against the simulated "expected" output size multiplied by a correction factor due to its approximate nature in MACSio as a result of constraints involved in creating a valid mesh topology. As a result, a first order approximation for MACSio's `part_size` factor in Listing 1 was estimated as:

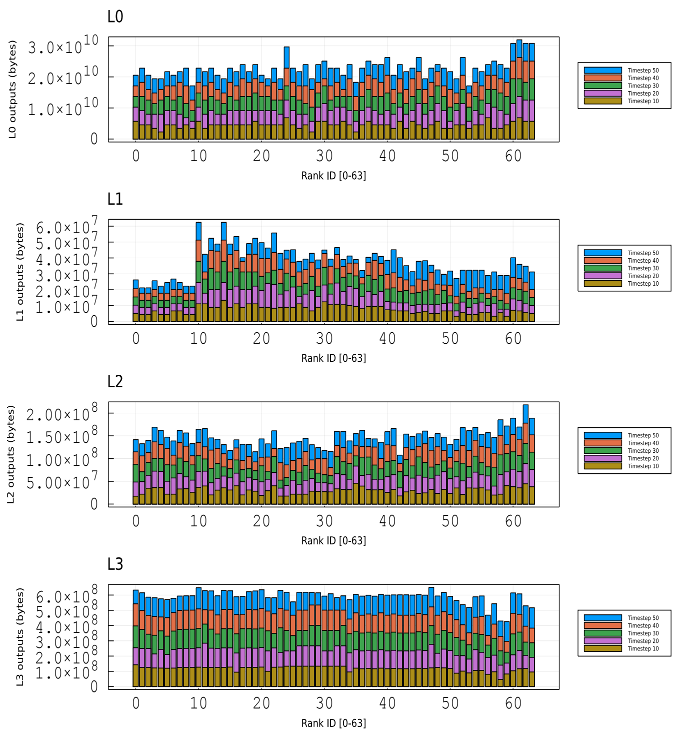


Figure 8: Output generation at each timestep per compute task (taskID) for 4 mesh levels in case27, for $1,024 \times 1,024$ L0 mesh for the AMReX Castro Sedov baseline.

$$\texttt{part\_size} = f \frac{8\,Nx\,Ny}{nprocs} \quad [bytes] \qquad (3)$$
$$f \approx [23 - 25]$$

where f is a correction factor due to the difference in nature of the MACSio json-based output and AMReX output file formats. The value of 8 accounts for the extra bytes in the double precision setup of the Castro executable. The results show that the empirical factor f for the Sedov cases is somewhere around 23 and 25, although this value might need to be reevaluated if the number of output fields or a different set of problems is used. Selecting a precise value for f in Eq. (3) depends on the focus of the approximation as a variational problem with two parameters. The latter is illustrated in Fig. 9 for which the "best" `data_growth` factor is optimized. It can be seen that keeping the initial data size in Eq. (3) fixed would lead to a single parameter optimization problem, which after a few runs shows that MACSio can provide a "kernel" approximation that is "close enough" to the output sizes generated with the AMReX Castro Sedov case. The final solution for `data_growth` $= 1.013075$ initially deviates from the simulation output sizes, however it becomes close to the correct value as the value as time steps increase, thus providing enough non-linear effect to model a similar behavior. As a result, if the intention is to model data workloads at each time step of the simulation, MACSio provides a simple interface to simulate "static" loads that can be a starting point for "dynamic" studies of more random system behaviors.

The proposed model in Eq. (3) is evaluated for the different cases shown in Fig. 5 by running MACSio several times while fine tuning the `data_growth` parameter to approximate the behavior of the measured data outputs from the Sedov case. To illustrate the validity of the MACSio model,

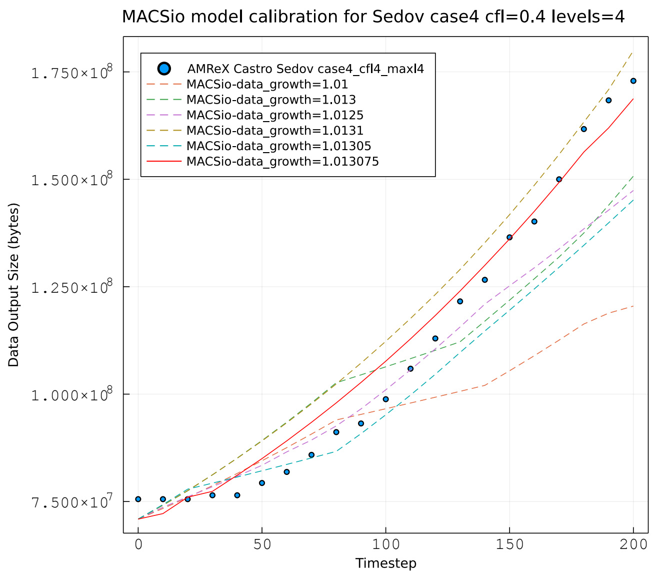


Figure 9: Modeling calibration for timestep outputs for the Sedov case4, $cfl = 0.4$, 4 AMR levels using MACSio non-linear kernel approach. Each curve represents a step in the convergence calibration.

Fig. 10 shows the resulting output sizes from the pivot simulation Sedov case (case4) and the comparison against the proposed MACsio model. It can be seen that after setting the initial data size from Eq. (3) to a constant value $= 1550000 \approx 23.65 \times 512^2 \times 8/32$ the `data_growth` becomes a function of the maximum number of levels and the CFL number. Still, this variation is smooth and choosing a small `data_growth` value below 1.02 (or 2%) based on CFL interpolation from these results, can be a good initial guess if further minimization is required depending on use-case (*e.g.* machine learning models, quick I/O evaluation, etc.).

Last, but not least we select the large case from Table 1 running on 64 nodes of the Summit supercomputer for a 8192×8192 L0 mesh. From Fig. 5, it can be seen that as cases become large the non-linearity introduced at the more refined levels becomes less dominant. Nevertheless, while the variation might be less smooth due to a natural reduction in the number of output steps for large scale runs, MACSio can still provide a first-order kernel approximation using the present model. This is illustrated in Fig. 11 in which the variation of the output at large scales is less smooth and a sudden jump in output size occurs as convergence of the solution is approached. MACSio can generate kernels that are in the vicinity of these values, while not necessarily providing an exact proxy for the observed non-smooth behavior. We argue that the simplicity of proxy applications, as shown with MACSio for the present study, still provides a reasonable trade-off between complexity and accuracy for modeling desired output workload characteristic at different scales for AMReX based simulations.

2.3 Conclusions

The data output characteristics of a AMReX-based Castro application under a variety of input conditions are analyzed and modeled using MACSio as a potential candidate for a simple "kernel"-based proxy I/O application. Results show that MACSio, in its current state of outputting a file for each rank and step, can provide proxy I/O capabilities that are able to model "per-step" and "homogeneous per-rank" output sizes of the Sedov hydrodynamic baseline case in AMReX-Castro

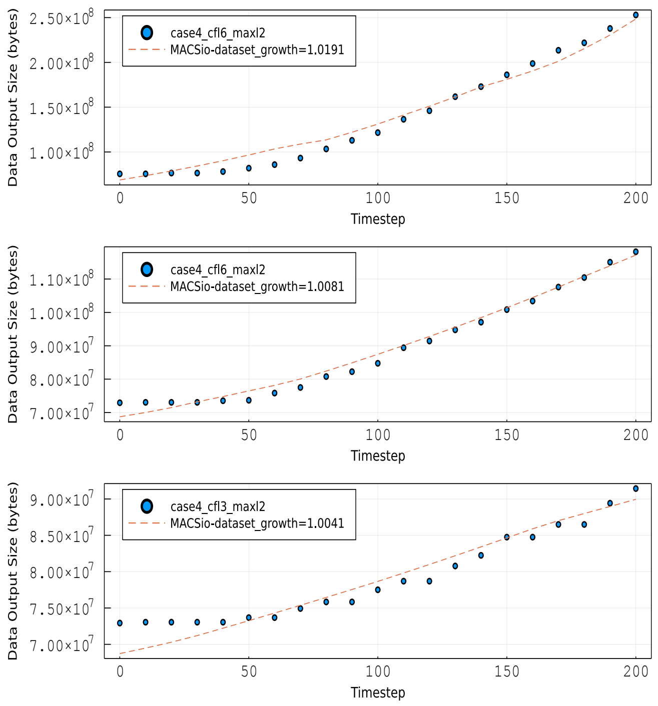


Figure 10: Comparison of the baseline Sedov case4 simulation outputs at each time step for different CFL numbers 0.3 (cfl3) and 0.6 (cfl6) and maximum number of mesh levels (maxl=2,4) against the proposed MACSio model.

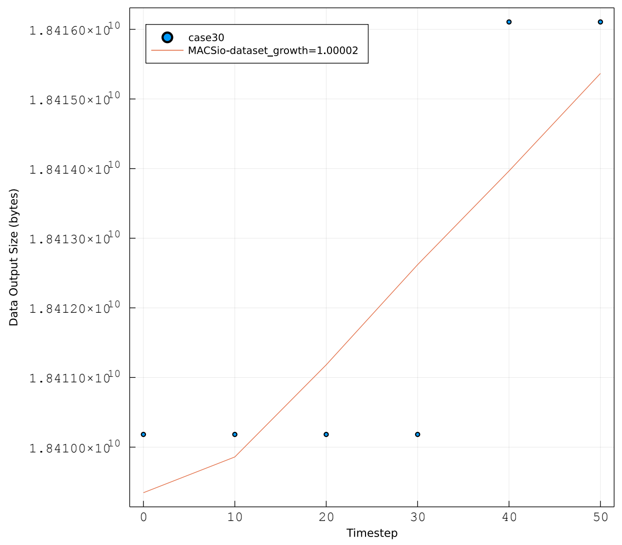


Figure 11: Comparison of a large L0 mesh = 8192 × 8192 Sedov non-smooth simulation output against the proposed MACSio kernel model.

running on the Summit supercomputer at different scales. In addition, a calibration methodology and an analytical model are provided that relate AMReX Castro inputs with those of MACSio, thus resulting in a lightweight proxy approach to conduct initial studies of AMReX I/O modeled characteristics without having to run a full simulation in parameterized studies. The latter simple analytical model becomes useful as storage systems evolve, along with the need to understand the co-design trade-offs when producing data at scale using AMR-based simulations. Furthermore, this simplified proxy “kernel”-based approach can be a good initial candidate for follow up studies on predictive I/O sizes, as well as dynamic random system characteristics that could potentially benefit from machine-learning approaches as more data becomes available. Since the ultimate goal is to improve the understanding of AMR output generation, proxy applications combined with simple modeling approaches can be a powerful predictive tool for autotuning more complex parallel I/O workload patterns in current pre and upcoming exascale supercomputing platforms. The latter autotuning aspect is something we would like to explore in subsequent studies.

3 Definition of More Accurate Performance Counter Groups

The work presented here satisfies the ADCD504-58 quantitative assessment milestone:
Re-define the performance counter groups for contemporary Intel and IBM platforms to better match specific hardware components and to better align across platforms (make cross-platform comparison more accurate), update extraction, processing, and automated data checking tools, then re-collect full data sets on the two platforms based on the new groups and repeat the similarity analysis.

Performance monitoring units are designed by vendors to be platform-specific. For example, some platforms support 4 registers for accumulating results and some support 6, and they all support different events with different event names, although often these events are semantically similar (e.g., cache miss events). Even within successive vendor platforms, the number of available registers and event names often change.

For the work that we've done on similarity analysis, we collect a performance vector comprising all of the valid performance events on each system of interest. For the Intel Skylake platform, this is around 500 total events (prior to event selection of important features). Because we want to understand how similarity and behavior of proxy/parent application pairs changes across architectures, we need to collect data that count similar events on each system architecture. To accomplish this, we inspected documentation, did some high-level validation, and from this, grouped events into primary and secondary categories for a particular architectural component. Examples of architectural components include L1, L2, and L3 cache, pipeline, and memory. Each primary category comprises events that represent behavior that is somewhat similar across platforms. Therefore, we only use component primary categories in our similarity analysis to determine if proxy/parent similarity remains constant across architectures. Secondary categories comprise architecture-specific events. These event sets can be used to understand proxy/parent similarity for a particular component on a specific system, but are not used when looking at similarity across different system architectures. This grouping of events into sets that pertain to specific architectural components is not perfect, but represents a first attempt to ensure a fairer comparison across very different architectures, such as the IBM Power9 and Intel's Skylake.

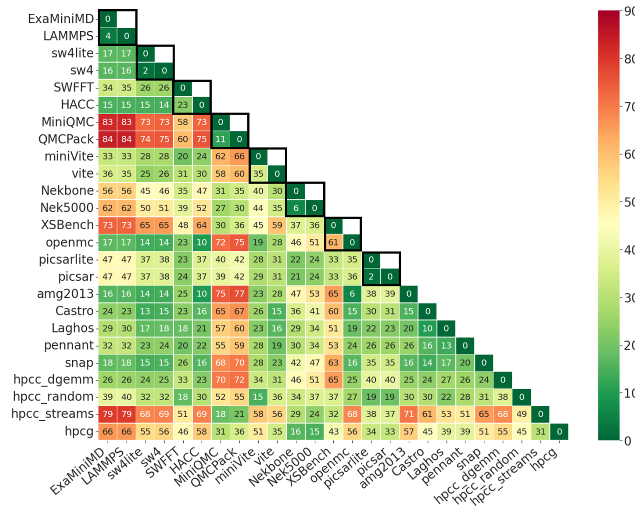


Figure 12: Cosine Similarity, Intel Skylake, Full Execution

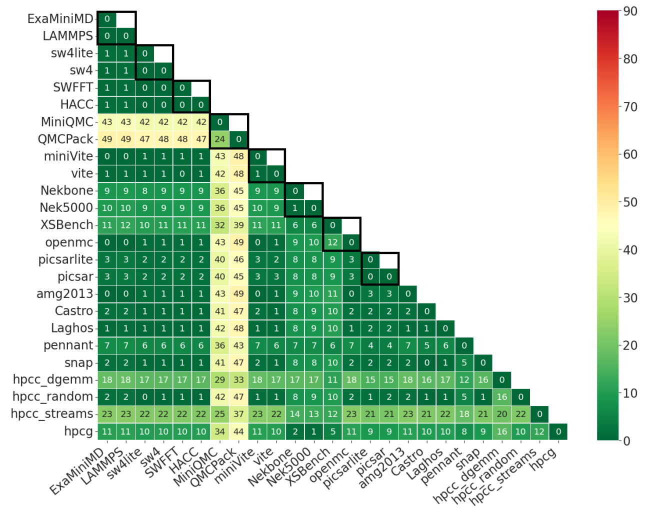


Figure 13: Cosine Similarity, Intel Skylake, L1 Cache

We categorize events according to architectural component because we want to understand where (i.e., for which component) a proxy exhibits behavioral similarity with a parent application. For example, a proxy might diverge from a parent in its total execution behavior, but its cache behavior may be similar, indicating that the divergent behavior is generated by a component other than cache. In this case, the proxy may be a good cache proxy of its parent, and can, therefore, be used in co-design efforts that are examining new cache architectures. Figure 12 shows similarity in angle (degrees) between proxy/parent pairs and a group of other applications (a mix of benchmarks, applications, and proxies) using the cosine similarity method on an Intel Skylake platform. The diagonal, which shows all 0 degrees, is self-similarity of an application with itself. The highlighted black boxes show the similarity between proxy/parent pairs. Note that *XSBench* and *openmc* are most divergent, with an angle of 61° between them that indicates that their overall behavior for their entire execution at the underlying hardware level is quite different. However, looking at Figure 13, which is the similarity of proxy/parent and other applications for L1 cache behavior, we see that *XSBench* and *openmc* have fairly similar cache behavior. In this case, *XSBench* can be used as a proxy for the L1 cache behavior of *openmc*. Their divergent behavior must be realized in some other component of the architecture.

We list all of the component groups and their events for the IBM Power9 and Intel Skylake systems here: https://github.com/sandialabs/proxy-parent-data/. We include some examples of component groups in Appendix A.

After completing re-grouping of performance counter events to better accommodate cross-system proxy/parent app similarity behavior, we updated all of our tooling infrastructure to reflect these new event sets. These include collection tools and data processing and analysis tools. We are currently working through the process to publicly release these tools on our Sandia organizational external github site.

We re-collected all of the data needed for proxy/parent pair similarity analysis on an IBM Power9 and Intel Skylake platform. In the FY21 report on quantitative assessment [33], we presented a methodology to compare proxy app to parent behavior for quantifying proxy fidelity using cosine similarity. We submitted a paper to ISC2021 based on this work. This paper was rejected primarily because the reviewers were unsure of the accuracy of the results. We subsequently performed much work to validate that the results returned by cosine similarity analysis are indeed accurate. Because we do not know a priori if the proxy application uses the underlying hardware in a similar manner compared to the parent application, we cannot know with absolute confidence that the similarity results are valid. We have no ground truth in this case, although we do have intuition through code and algorithm inspection as to which proxies should be more or less similar to their respective parents. Since we have no ground truth, we decided to apply many different ML-based, unsupervised (as is cosine similarity) algorithms to proxy/parent behavior comparison in an attempt to quantify the accuracy of cosine similarity. We also added some standard HPC benchmarks to our similarity analysis to serve as baseline measurements. Note that we did the exploration of similarity algorithms, added HPC benchmarks, and performed feature extraction on the Intel platform only. We describe this work below, much of which has also been submitted in a paper to SC22, but we also include the new cosine similarity analysis (based on new performance counter groups) for both the IBM Power9 and Intel Skylake platforms.

3.1 Comparison of Similarity Algorithms for Quantifying Proxy Application Fidelity

The three main research questions we address are: 1) how closely proxy behavior on a system represents parent behavior, 2) how similar proxies are to each other so that a co-design or procurement suite does not suffer from redundancy, and, 3) how to identify gaps, where there is no proxy that represents the behavior of a parent application. For this purpose, we introduce **SimEngine** to determine the similarity between proxy and parent applications. We generalize the engine for more comprehensive usage in the HPC area, as well as implementing several similarity algorithms. We demonstrate how to choose the best algorithm for a given (our) dataset through algorithmic comparison. We do this primarily because in many contexts, there is no ground truth (*i.e.*,, we do not know *a priori* which proxies are, or should be, representative of their parents' behavior). By comparing results from different similarity algorithms, we get best-case validation that the observations are accurate. **SimEngine** also includes algorithms to facilitate feature selection, as minimizing the number of features is very important to reduce the data collected and thus lower the number of application runs.

With the help of the quantitative similarity measurement we have developed in **SimEngine**, users are guided to choose proper proxy applications for particular uses. Besides quantifying fidelity of proxy applications, similarity measurement approaches in **SimEngine** can also be applied to various HPC problems, such as compiler optimization, code refactoring, and application input sensitivity.

3.2 SimEngine

To determine how similar proxy and parent applications behave in terms of resource utilization such as computation and memory, we explore the use of several ML similarity techniques, which are all unsupervised. These techniques work differently, making it difficult to determine which one reveals the real relationship. Therefore, we create **SimEngine** to interpret the level of agreement between these techniques and converge upon a measure of proxy fidelity.

We collect hardware performance counters from several proxy applications, parent applications, and benchmarks to provide a comprehensive collection of informational measurements that reveal application execution resource fingerprints to determine similarity. Figure 14 shows the workflow of **SimEngine**, where we use Lightweight Distributed Metric Service (LDMS) [7] as the data collection framework to collect PAPI hardware performance counters from HPC applications. This data is processed (see Section 3.5.4) and then input to the feature selection layer as a matrix X that has n rows of application vectors $x_1, \ldots, x_n$ and features $f_1, \ldots, f_d$ as the columns.

During feature selection, we rank the features by calculating importance scores and selecting the important k features using a correlation filter. The matrix X', which has k features as the columns, preserves the similarity structure of the matrix X. Finally, using the selected important features, we produce similarity matrices to compare the similarity between applications, and quantify the fidelity of proxy and parent applications.

3.3 Similarity and Distance

Evaluating the similarity between proxy and parents is one of the main goals of our work. For each application, we sample, by the second, the accumulated hardware counters, for the whole execution. Then we average the last 5 seconds of the application execution for each event across ranks. Thus, we get an application vector x_i, that contains a series of averaged hardware event counters (*i.e.*, x_{ik}). We define two applications as *similar* if the vectors that represent the applications are a short distance apart, which, in many classification tasks, corresponds to the vectors belonging to the same class. After calculating the pairwise distance between each application pair, we get a similarity matrix. Finally, we compare the results of four typical similarity metrics, introduced below, and choose the metric that most accurately correlates known proxy/parent pairs.

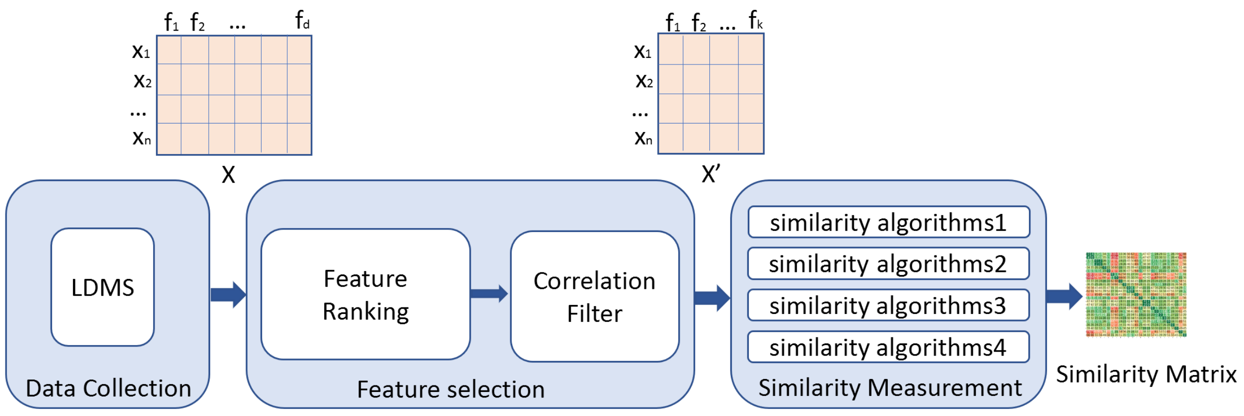


Figure 14: **SimEngine** Architecture

3.3.1 Cosine Similarity

Cosine similarity compares the angle between vectors in an inner product space, where the inner product can be conceptualized as the projection of one vector x_i in the direction of the other vector x_j. The calculation relies on two complementary definitions (algebraic and geometric) for computing the inner product:

- Algebraic Inner Product: $x_i \cdot x_j = \sum_{k=1}^{d} x_{ik}x_{jk}$
- Geometric Inner Product: $x_i \cdot x_j = \|x_i\|\|y_i\| \cos\theta$
- Cosine Similarity:

$$\cos(\theta) = \frac{\sum_{k=1}^{d} x_{ik}x_{jk}}{\|x_i\|\|x_j\|} \tag{4}$$

The cosine varies from 1.0 (identical vector direction) to 0.0 (orthogonal vectors), and the degree θ varies from 0° (similar) to 90° (dissimilar). If two applications have similar behaviors then we are expecting their cosine similarity angle to be closer to 0°.

3.3.2 Jensen-Shannon (JS) Divergence

Instead of comparing two vectors, we can normalize each vector and think of it as a probability distribution (each event turns into a probability), and then compare the two distributions. JS divergence [14] measures the distance between two probability distributions P and Q. JS divergence is a generalization of Kullback–Leibler (KL) divergence [23], the relative entropy from distributions Q to P, *i.e.*, the expectation of the logarithmic difference between the probability distributions Q and P.

$$\begin{aligned} KL(P|Q) &= \sum_x P(x) \log \frac{P(x)}{Q(x)} \\ &= -\sum_x P(x) \log Q(x) + \sum_x P(x) \log P(x) \\ &= \text{cross entropy} - \text{entropy} \end{aligned}$$

KL divergence is asymmetric and has no upper bound. Unlike KL divergence, JS divergence is symmetric and returns a value between 0 and 1, where 0 is similar and 1 is divergent. JS divergence is defined as

$$\mathrm{JS}(P\|Q) = \frac{1}{2}KL(P\|M) + \frac{1}{2}KL(Q\|M) \tag{5}$$

where $M = \frac{1}{2}(P + Q)$.

3.3.3 Wasserstein Distance

Wasserstein distance [35] can be interpreted as the minimum "cost" of transforming a probability distribution P into distribution Q, where "cost" is measured as the amount of distribution weight that must be moved, times the distance it has to be moved [4]. Unlike other statistical distances like JS divergence, Wasserstein distance multiplies the probability distribution and the distance, thus the order of the events matters. Since Wasserstein distance does not require both measures to be in the same probability space (*i.e.*, two vectors may have different lengths), it can be used to compare application performance across platforms that support a different number of hardware event counts. The p^{th} Wasserstein distance is defined as

$$W_p(P, Q) = \left(\inf_{J \in \mathcal{J}(P,Q)} \int \|x - y\|^p dJ(x, y) \right)^{1/p} \tag{6}$$

, where $\mathcal{J}(P, Q)$ denotes all joint distributions J for (X, Y) that have marginals P and Q. We use the first Wasserstein distance ($p = 1$) between two 1-dimensional distributions, which is also known as earth mover distance, to determine similarity. Wasserstein distance does not have an upper bound; 0 indicates similarity (equivalence), while increasing values indicate growing divergence.

3.3.4 Mahalanobis Distance

Mahalanobis distance [31] is an effective multivariate distance metric that measures the distance between a point (vector) and a distribution, or two points from the same distribution. The Mahalanobis distance between two vectors x_i and x_j from the same distribution is defined as

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^T S^{-1} (x_i - x_j)} \tag{7}$$

, where S is the covariance matrix of the data set. Geometrically, it transforms the data by whitening and normalizing the covariance, and computes the Euclidean distance for the transformed data. Since the Mahalanobis distance accounts for the variance of each variable and the covariance between variables, it is used in areas such as multivariate anomaly detection and imbalanced classification. Note that the Mahalanobis distance requires more samples than features in order to calculate the covariance matrix. Thus, we reduce dimensionality with principal components analysis (PCA) before applying Mahalanobis distance. Mahalanobis distance also does not have an upper bound; here, 0 indicates similarity (equivalence), while increasing values indicate growing divergence.

3.4 Feature Selection

A large number of hardware performance counter events are available on most HPC architectures, making it difficult to identify a comprehensive but minimal set. Thus, we exhaustively collect all the events available on the Intel Skylake platform (500 total events) for each application. However, our aim is to reduce irrelevant features, since these features may add noise, reduce model accuracy, increase latency and storage overhead, or extend the amount of data processing. Therefore, we have to find a suitable feature selection algorithm that is simple and efficient, in order to speed up our engine, find a concise event subset of uncorrelated features, simplify the data collection, and enable cross-platform comparison. There are two components in the feature selection layer in **SimEngine**: feature score, which is used to rank the important features, and the correlation filter, which is used to remove the correlated features. Again note that we perform feature selection for the Intel platform only and will perform this for the IBM Power9 feature data as future work.

3.4.1 Feature Score

Consider a data matrix X that has n rows of samples $x_1, ..., x_n$ with d features $f_1, ...f_d$. Our goal is to find the subset of k important features that preserve the similarity structure of the matrix X.

Since we are mainly interested in maintaining the similarity relationship between the proxy/parent application pairs, neighbor embedding is the perfect unsupervised method to reduce our feature set. We choose a graph-based feature ranking technology called Laplacian score [18] to compute the importance of features. Laplacian score builds a nearest neighbor graph for application points and seeks those features that respect local graph structure. In our case, these features help preserve the neighbor similarity between proxy and parent applications.

We express the Laplacian score of the r^{th} feature as:

$$L_r = \frac{\widetilde{\mathbf{f}}_r^T L \widetilde{\mathbf{f}}_r}{\widetilde{\mathbf{f}}_r^T \widetilde{\mathbf{f}}_r} \tag{8}$$

, or in a more understandable way:

$$L_r = \frac{\Sigma_{ij} \left(f_{ri} - f_{rj}\right)^2 S_{ij}}{\mathrm{Var}\left(\mathbf{f}_r\right)} \tag{9}$$

, where f_r is the r^{th} feature, and S_{ij} is the weight matrix.

$$S_{ij} = e^{-\frac{\left\|\mathrm{x}_i - \mathrm{x}_j\right\|^2}{2}} \tag{10}$$

An element S_{ij} will only have a nonzero value when i and j are neighbors, otherwise the value of that entry is zero. The denominator in Eqn. 9 indicates the variance of the r^{th} feature; larger values correlate to more information represented. The numerator indicates the sum of feature value differences within the points' neighbors; here, the smaller the Laplacian score is, the more important the feature is. Finally, we get the ranked features sorted by Laplacian score.

Since features are evaluated separately in the Laplacian score, if we select features with the smallest Laplacian score, some correlated features may be included. To further reduce the important feature set, we introduce the correlation filter.

3.4.2 Correlation Filter

When a group of features are highly correlated, we only need to include one in our selection. A broadly used correlation measure is the Pearson correlation coefficient (PCC). PCC measures the linear correlation between two variables f_i, f_j.

$$\rho_{f_i,f_j} = \frac{\mathrm{cov}(f_i, f_j)}{\sigma_{f_i}\sigma_{f_j}} \tag{11}$$

where cov is the covariance and $\sigma_{f_i}$is the standard deviation of f_i. PCC has a value between +1 and -1, where +1 is a total positive linear correlation, 0 is no linear correlation, and -1 is a total negative linear correlation. We choose a PCC threshold of 0.9 because we want to keep a reasonable number of important features while removing those with strong correlation.

We calculate the pairwise PCC for all ranked features and extract features sequentially from the ranked feature set and add them to our important feature subset. Each time we select one feature, we remove the redundant features with PCC > 0.9 or < -0.9 from the remaining features. We stop collecting features when there are no more features in the feature pool.

Note that the PCC can evaluate only a linear relationship between two continuous variables; we will investigate more correlation methods in future work.

3.5 Experimental Platform

In this section, we present our methodology for collecting data and how we use that data to perform similarity analysis using **SimEngine**. We use Intel Skylake and IBM Power9 production HPC systems to collect data on 21 proxy and parent applications that span several scientific domains. We also collect data on several standard HPC benchmarks and use these in our analysis to provide a baseline for aiding in understanding the results.

3.5.1 Application Suite

Of the applications used in this work, some are proxy/parent pairs and some are proxies or parents that are not paired. We also include several standard HPC benchmarks. Not all applications we use are proxy/parent pairs because some proxies have export-controlled parents for which data cannot be publicly released. Also, Castro is the parent of a proxy called Thornado [13], but at this point, we are not using Thornado because of I/O issues within the code that caused problems with data collection. Per developer documentation, all of the proxy/parent pairs in our suite comprise proxies that are intended to represent the computation, communication, and memory behavior of their respective parent.

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 4 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. The input that we use for all of the applications can be found at https://github.com/sandialabs/proxy-parent-data/tree/main/input_files.

3.5.2 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 5. 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 5. The Intel system runs the RHEL7.8 operating system (OS); RHEL7.6 OS runs on the IBM system.

From Table 5 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)

Table 4: Proxy/Parent version information

Proxy	Version	Parent	Version
AMG2013 [19]	2013_0	N/A	N/A
N/A	N/A	Castro [8]	20.07
ExaMiniMD [38]	1.0	LAMMPS [30]	17 Aug 2017
Laghos [12]	3.0	N/A	N/A
miniQMC [32]	0.4	QMCPACK [21]	3.8
miniVite [15]	1.0	Vite [16]	30 Sept 2020
Nekbone [20]	3.1	Nek5000 [28]	19.0
PENNANT [2]	0.9	N/A	N/A
PICSARlite [3]	16 July 2020	PICSAR [40]	16 July 2020
SNAP [5]	1.09	N/A	N/A
SW4lite [13]	2.0	SW4 [29]	2.0
SWFFT [13]	1.0	HACC [17]	1.0
XSBench [39]	19.0	OpenMC [34]	0.11.0
HPCG benchmark [6]	3.1	N/A	N/A
HPCC benchmark [1]	1.5.0	N/A	N/A

Table 5: 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 DDR4-2666	256GB DDR4-2667
Cores/threads	18/36	24/48
Sockets/node	2	2
Total nodes	1488	54
Interconnect	Omnipath	Mellanox EDR Infiniband
Max Memory BW (per processor)	20GB/sec	170 GB/sec
Memory channels (per socket)	6	8

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.

3.5.3 Data Collection

We use LDMS [7] 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. With LDMS, we use the Performance Application Programming Interface (PAPI) [37] sampler, which implements the PAPI API within the sampler to connect to every process (rank) in each application, in order to collect node related performance counter data. We carefully examined all of the available performance events on our hardware platform and functionally tested them to ensure that at a minimum, plausible data was returned for each event. We eliminated events that were returning no data or unstable (*i.e.*, vastly varying) data across application runs. Finally, we collected more than 500 and more than 700 hardware events for each application on the Intel Skylake and IBM Power9 platforms, respectively.

Several runs are required to collect the complete set of data since the hardware has limited performance counter registers, 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.*,, dependent 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 collect the data in subgroups according to the event categories suggested by experts. Thus, we have the following groups: Branch, DecodeIssue_Pipeline, Dispatch_Pipeline, Execution_Pipeline, Frontend, Instruction_Cache, Instruction_Mix, L1_D_Cache, L2_D_Cache, L3_D_Cache, Memory_Pipeline, Misc, Power, Retirement_Pipeline, and Memory. Because we also aim to understand the ways in which a proxy is or 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.

3.5.4 Data Pre-Processing

To account for performance variation and any performance counter variation, we run each event subgroup 5 times for each application, resulting in over 3000 data-collection runs. Further, we collect data from each application process (*i.e.*,, each MPI rank). For similarity analysis, we need a single vector for each application. Therefore, we (1) compute the average for each event across all ranks for each of the five runs, (2) compute the average for each event for each of the five runs, and finally (3) normalize the event counts by cycles executed. The result is a vector of length 500 or 700 (Intel or IBM) for each application, where each element is $event\_count/CPU\_cycles$.

Because irrelevant (noisy) features can hurt the performance of the cluster learning for unsupervised feature selection [26], we remove these features before they are ranked. Hardware events in our collection platform have a prefix (Table 8), which allows us to filter some events using domain knowledge. We observe, for instance on the Intel platform, that a large number of hardware events with the prefix 'OFFCORE_RESPONSE' always show extremely small values with little variance. We calculate the absolute degree difference, entry by entry, between cosine similarity matrices with or without these 'OFFCORE_RESPONSE' features. The sum of absolute difference is only 1.4 compared to 22078, which is the sum of all the entries in the cosine similarity matrix with all features. From this, we propose

Observation 1: *Events that begin with the prefix 'OFFCORE_RESPONSE' are irrelevant in similarity analysis.*

After excluding these features with the prefix 'OFFCORE_RESPONSE', we also eliminate 10 additional features on the Intel platform that do not have value for some applications. Therefore, prior to ML-based feature selection, the number of features is already reduced to 202.

Much feature selection work considers centralization (zero mean), normalization (norm 2 equals 1), or standardization (variance equals 1) as preprocessing steps. Because each feature in our data has an explicit physical meaning (hardware event counts per CPU cycle), if we preprocess the data as mentioned above, the feature scales and the relationship between features will be lost. Therefore, we keep the data scales as they are. Note that feature selection has only been done on the Intel platform and is presented in Section 4. Feature selection on the IBM platform is future work. Therefore, we show similarity results using all features for both platforms.

3.6 Results

Here we present the results that show the fidelity of the proxy applications as compared to their respective parents. We first present a comparison between similarity algorithms using data from the Intel platform only. We use the similarity algorithms outlined in Section 3.3 and evaluate the accuracy of the resultant proxy/parent pairs. We then present cosine similarity results for both platforms, followed by a presentation of some of the results for component subgroups.

3.6.1 Similarity Matrix Comparison

Since the similarity matrix is symmetric on the diagonal, we use the lower triangular heatmap (Figures 15, 16, 17, and 18) to visualize the similarity. The diagonal entries are zero because they indicate the distances between the applications and themselves. We adjust the scale of value in each figure to be within the same range to make them comparable. The spectrum colors that represent similarity are from a standard colormap, where dark green is highly similar and dark red is highly dissimilar. 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. Along the diagonal, eight 2×2 black border blocks circle the relationship between proxy and parent application pairs. One would expect to see the lower left of these blocks to be dark green, which shows a high similarity between proxy and parent. The nine miscellaneous applications (either a proxy with no parent or a parent with no proxy) are listed at the bottom on the y-axis and the right on the x-axis.

In Figures 15, 16, 17, and 18, we see that, generally, the four distance metrics we evaluate return similar correlations. Take for example Figure 15 which uses cosine similarity: PICSARlite and PICSAR, SW4lite and SW4, Nek5000 and Nekbone, and ExaMiniMD and LAMMPS, are highly similar proxy/parent application pairs. QMCPack and MiniQMC, and SWFFT and HACC, show good similarity. While these six pairs have similar behaviors, the other two pairs show some behavior gaps: miniVite and Vite are moderately similar, with an angle of 35° between them, and XSBench and OpenMC are highly dissimilar, with an angle of 60° between them. In this case, this is probably because of the difference in complexity between the parent and proxy; XSBench only performs the cross-section lookup portion of Monte Carlo neutron transport, which is a highly data-intensive kernel, whereas OpenMC implements the full neutron transport code so, likely has more opportunity to hide poor cache/memory behavior with other computation. The unpaired proxy applications (amg2013, Castro, Laghos, pennant, and snap) show relative similarity to each other, and note that many of these are closer in behavior to SWFFT than is HACC, its parent, for many of the similarity algorithms. The other four HPC benchmarks are not similar to each other because they aim to measure totally different memory or data patterns. But note that for all four algorithms, the behavior of hpcc-random generally most closely matches the behavior of all of our applications, which is expected since many of them are characterized by random access memory behavior. Similarly, all four algorithms show that hpcc-streams is most divergent in behavior from all of the applications, which is also expected. This is likely because it is a synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computational rate for a simple vector kernel, thus, it does not behave like standard scientific

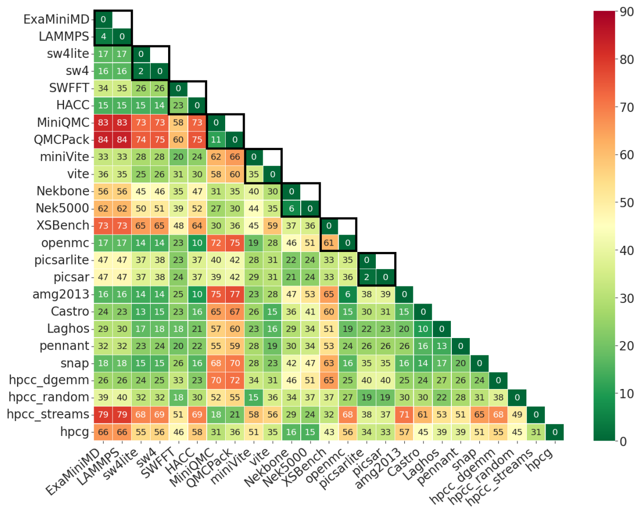


Figure 15: Cosine Similarity

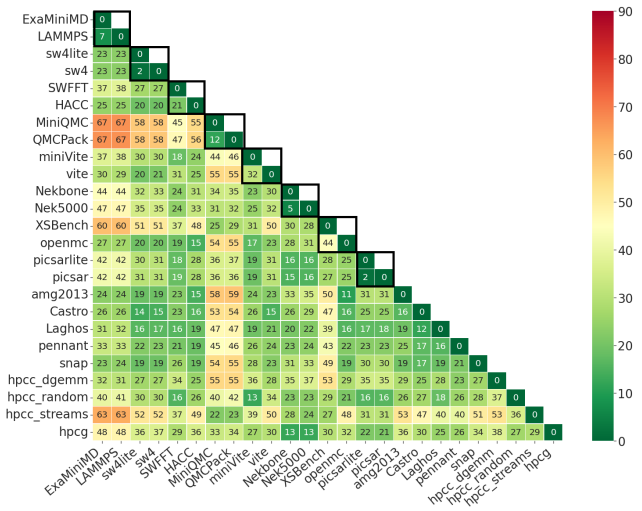


Figure 16: JS divergence (Values are multiplied by 100)

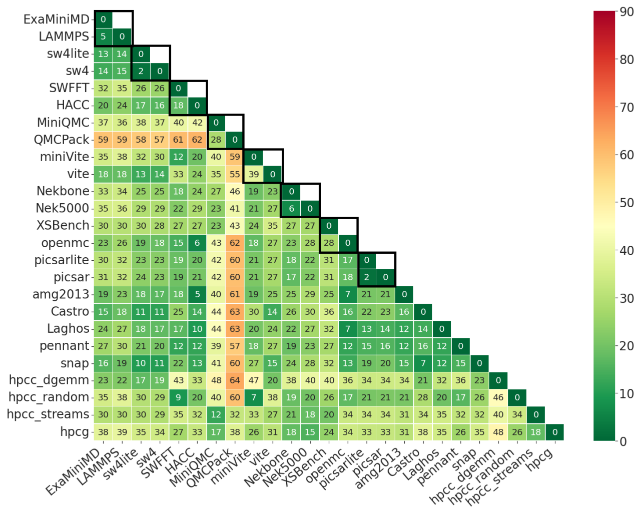


Figure 17: Wasserstein distance (Values are multiplied by 1000)

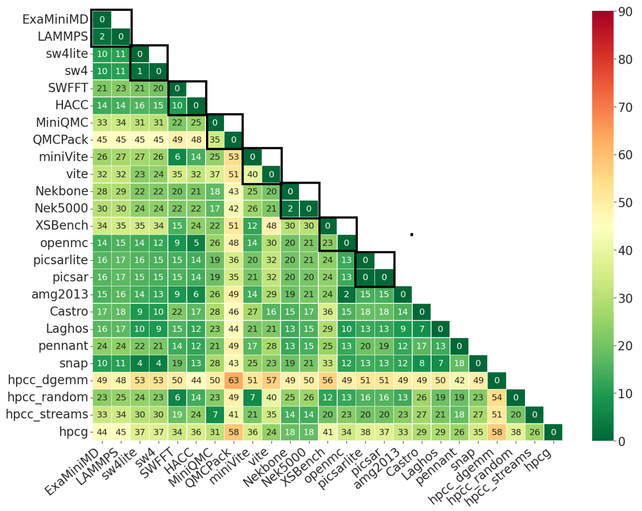


Figure 18: Mahalanobis distance (Values are multiplied by 10)

applications with random memory access and a much heavier computational workload. hpcg is designed to exercise computational and data access patterns that more closely match a different and broader set of important applications. In our suite, only Nekbone and Nek5000 are similar to hpcg. Another thing to note from Figure 15 is that application pairs QMCPack and MiniQMC are similar to one another but very different from other applications. XSBench, Nekbone, Nek5000, PICSARlite, and PICSAR are also outliers in that their behavior shows significant differences compared to most other applications.

We also notice that there is some diversity when applying these four similarity algorithms. When looking at the dark red areas, applications hpcc_streams and hpcg are highly dissimilar to other applications in terms of cosine similarity, JS divergence, and Mahalanobis distance, while not that dissimilar in Wasserstein distance. This may be due to Wasserstein distance being impacted by the feature order in the vector. If the order of the events in the vector changes, the Wasserstein distance would also change. For example, if the events that have divergent behavior are located far away in the vectors, the Wasserstein distance becomes larger. The application hpcc_degemm is extremely different from other applications in Mahalanobis distance but shows no obvious differences in other similarity algorithms. This may be because of the whitening process in Mahalanobis distance. QMCPack and MiniQMC diverge more in Wasserstein distance and Mahalanobis distance compared to the other two methods, and the differences mainly come from the cache and pipeline subgroups, which we will discuss in Sec 3.7. A significant refactoring effort has gone into QMCPack to improve its memory behavior and accuracy [21], but this has not been implemented in miniQMC. Overall, cosine similarity agrees most closely with JS divergence with respect to similarity/divergence results.

Observation 2: *Similar proxy/parent application pairs remain similar no matter what similarity algorithm is used, while dissimilarities differ depending on what algorithm is used.*

Since these distance methods show similar results for similar pairs, and the run-time difference of each method is negligible, we select cosine similarity for validating proxy/parent pairs due to its simplicity, performance, and ease of interpretability by geometric angle.

3.7 Cosine Similarity in Intel and IBM Platforms

The milestone work presented in this section was to re-group performance event counts to more closely represent the various processor component behaviors. In prior work, we had grouped events into component groups, but those groups were ill-defined and we had many events that were clearly in incorrect groups. We improved these issues, then performed the study that examined various similarity algorithms to determine which we should really use. This study concluded that because two out of four of the algorithms had a high level of agreement (cosine similarity and Jensen-Shannon divergence), we should use cosine similarity because its interpretation is intuitively simple. This section presents the cross-platform cosine similarity study that we did after re-grouping performance count events, which is the last part of the milestone. Results of this study are shown below.

We use the experimental platform as presented in Section 3.5. Figures 19 and 20 show the cosine similarity results for the Intel and IBM platforms, respectively, using the entire execution vector (i.e., all valid performance counter events). We see that the proxy/parent pair relationships remain essentially the same, with the exception that miniQMC and QMCPACK are more divergent on the IBM system. The difference in overall similarity is starkly noticeable, with a larger proportion of green in the plot of Figure 20. In Figure 20, we see that miniQMC and XSBench are outliers, showing little similarity with the rest of the apps. In contrast, on the Intel figure, we see more outlier

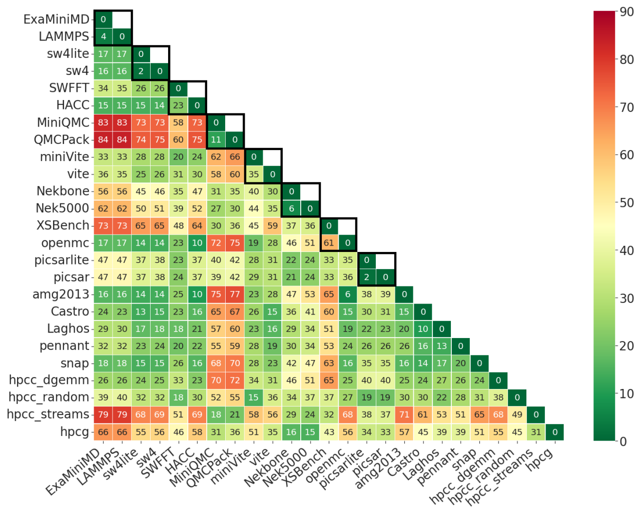


Figure 19: Cosine Similarity, Intel Skylake

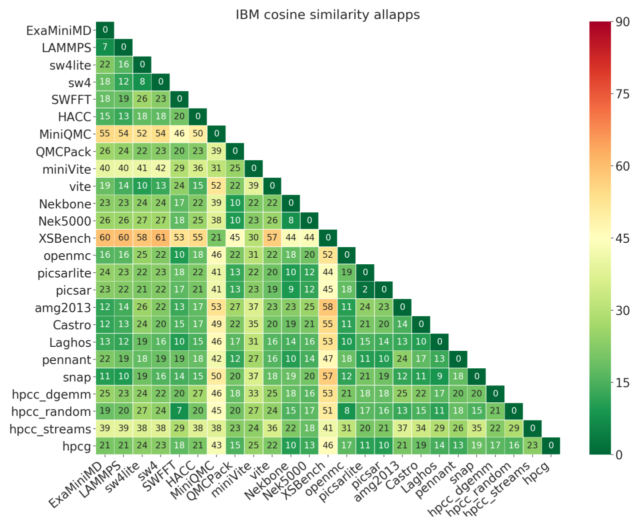


Figure 20: Cosine Similarity, IBM Power9

applications–QMCPack, Nekbone, Nek5000, hpcc_streams, and hpcg in addition to miniQMC and XSBench. Why? We believe that the answer is at least partially attributed to the difference in cache and memory subsystems in these two platforms. The Intel system is characterized by a smaller, more bandwidth-constrained memory subsystem in general. If any of the applications are constrained by these resources, there will be more activity or behavior observed in those components. And because every application probably strains these resources differently, it seems this might generate more divergent behavior. If we look at data from some of the memory subsystem component groups, we see this more clearly.

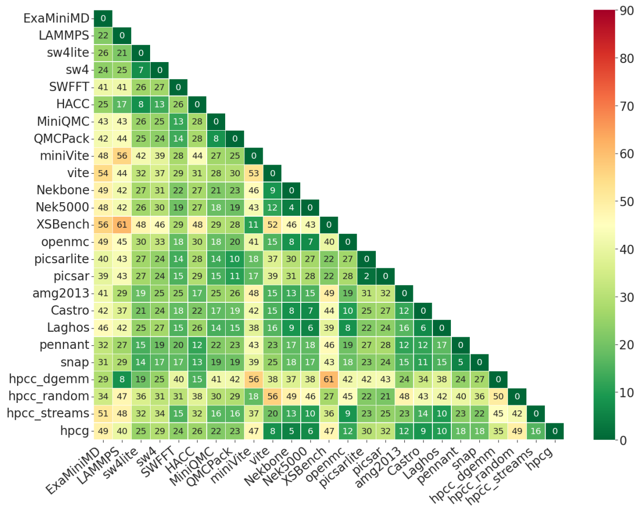


Figure 21: Cosine Similarity, Intel Skylake, L2 Cache

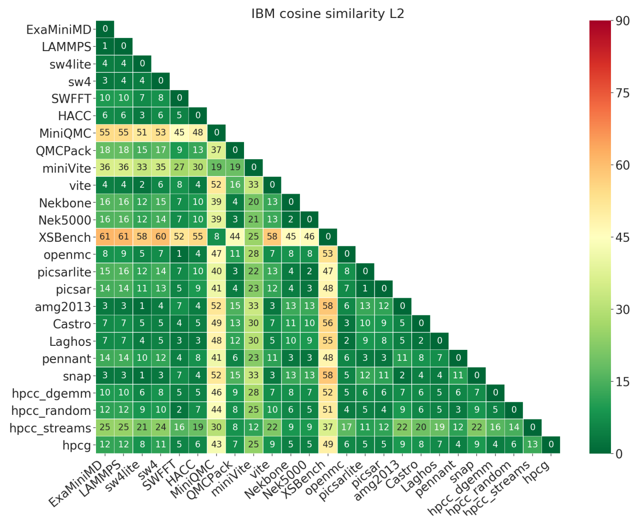


Figure 22: Cosine Similarity, IBM Power9, L2 Cache

Figures 21 and 22 show the cosine similarity of the L2 cache behavior for the Intel and IBM systems, respectively. In the Intel L2 Cache behavior plot, we see more divergence in Nekbone/Nek5000 and hpcc_streams and hpcg. QMCPack's L2 cache behavior is not really an outlier here, but rather is fairly similar to most of the cache behavior in the other applications. However, if you look at the L1 data cache similarity of QMCPACK on the Intel system in Figure 25, we see that its behavior, as well as miniQMC's, is quite distinct from the other applications. In the IBM system, Figure 22 looks similar to Figure 20, where the whole performance vector is used in the analysis, except hpcc_streams is not as divergent from all the other apps.

On the Intel platform, we also see QMCPack's outlying behavior in the pipeline. Figures 23 and 24 show the cosine similarity for the Dispatch and Execution stages of the pipeline, respectively.

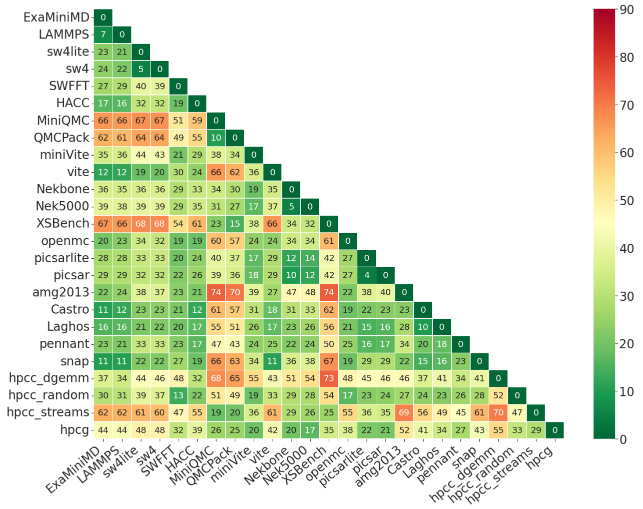


Figure 23: Cosine Similarity, Intel Skylake, Dispatch Pipeline

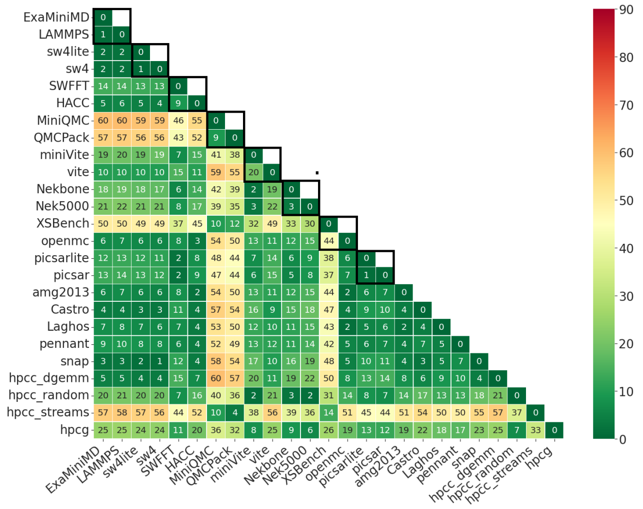


Figure 24: Cosine Similarity, Intel Skylake, Execution Pipeline

QMCPack's behavior really diverges from that of other applications; Nekbone/Nek5000 also outlies more from the other apps in its dispatch behavior.

On the IBM system, hpcc_streams behavior is distinct from other apps primarily in the memory stage of the pipeline, specifically in the LSU (load/store unit). We see this in Figure 26. Although we do not include all of the processor component data here, upon inspection of this data, this is likely what generates the distinct behavior for hpcc_streams in the overall cosine similarity analysis of Figure 20. This makes sense since this benchmark saturates memory bandwidth, and this should be observable through the behavior in the LSU.

3.8 Discussion and Future Work

Our results indicate, surprisingly, that the particularities of the similarity algorithms have less impact than predicted on the likelihood of correlating parent/proxy pairs. Since we use ECP pairs, we expect that this is a general result, but we cannot rule out the possibility that some domains will not have clearly separable parent/proxy relationships.

Run-time variance is inevitable in HPC. On the one hand, to collect the complete set of data, we run each application multiple times. Minor run-time differences occur among multiple runs. On the other hand, different applications vary in run-time, *e.g.*, most parent applications have longer run-time than their corresponding proxy applications. Since our current analysis is based on the final accumulated average of each hardware event count for each application, we temporarily do not

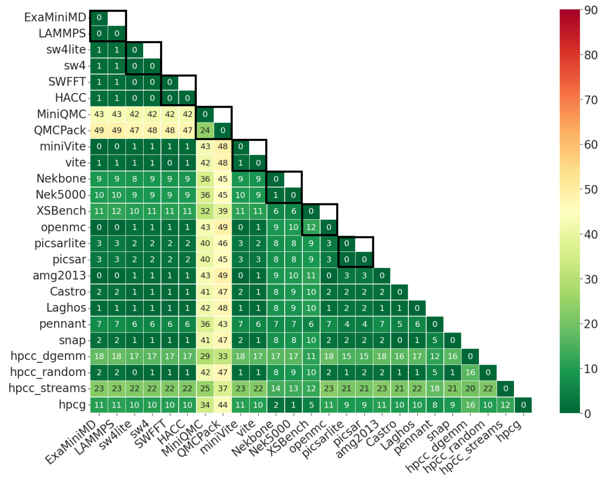


Figure 25: Cosine Similarity, Intel Skylake, L1 Dcache

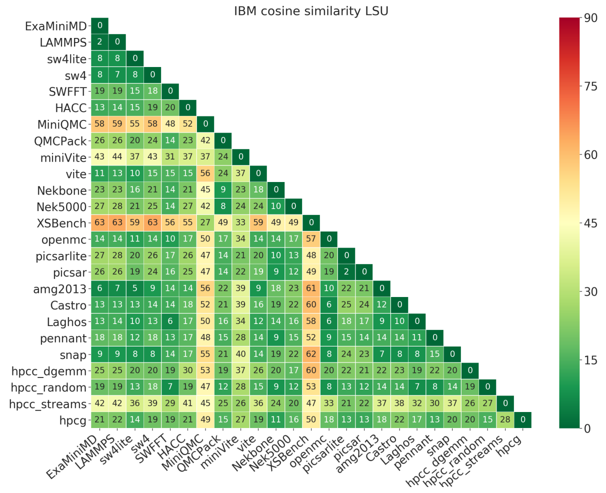


Figure 26: Cosine Similarity, IBM Power9, Memory Pipeline Stage (LSU)

take these run-time differences into account. However, if we want to keep the time phases for each application, we need to consider aligned data. Currently, we have implemented run-time alignment for each application with a weighted index for each run, but aligned data could be used for future time phase comparison. In the future, we will also evaluate important features on more platforms, including the IBM Power9 in particular.

4 Data Processing for Input to ML Stage

The work presented below satisfies the ADCD504-35 quantitative assessment milestone:
This work involves statistical and ML methods specifically to reduce the set of collected performance counters used in cosine similarity analysis. Deliverable: Report detailing analysis and results.

In this work, we perform two primary tasks: (1) mathematical analysis of our data that accurately combines multiple data sets from an application to average and align the phases, and (2) application of ML methods to reduce the set of performance counters collected. We first present the mathematical analysis that we developed to more accurately process our data prior to similarity analysis, then we present our feature selection method.

4.1 Mathematical Analysis: Data Pre-processing

We collect the hardware performance counters from the applications using LDMS, a sampling monitoring framework that runs on all the system nodes. We use the PAPI sampler in LDMS to sample the counters from the application processes (MPI ranks) and nodes and store the data in CSV files. The PAPI sampler can collect up to 30 hardware counters, so we divide our 500 counters into several groups and collect them individually. We subsequently post-process the data files by averaging the hardware counters, for each MPI rank then each node for each group, into one data set consisting of a time series of one-second interval sampled data. Table 6 shows a sample data

Table 6: miniVite Data set

Original Progress	**Scaled Progress**	**Appname**	**HWC1**	**HWC2**	**HWC3**
0.0	0.0	miniVite	1.975197e+07	7.117608e+05	5.811678e+05
1.0	1.0	miniVite	2.435274e+07	7.182168e+05	5.847572e+05
2.0	1.0	miniVite	3.445416e+07	7.317115e+05	6.006805e+05
3.0	2.0	miniVite	5.156964e+07	7.616228e+05	6.217035e+05
4.0	3.0	miniVite	7.627150e+07	8.073562e+05	6.467540e+05
...	...	...	...	...	...
359.0	358.0	miniVite	4.718202e+10	1.327342e+10	3.499148e+08
360.0	359.0	miniVite	4.718488e+10	1.327484e+10	3.499154e+08
361.0	360.0	miniVite	4.718853e+10	1.327642e+10	3.499168e+08
362.0	360.0	miniVite	4.718875e+10	1.327652e+10	3.499285e+08
363.0	361.0	miniVite	4.718910e+10	1.327674e+10	3.499363e+08

set of miniVite with a few hardware counters; we only display a portion of the counters due to the limit in page width. We also repeat the collection process for each group five times by running the applications using the same input to mitigate the effects of performance counter and runtime variation. The applications use the same input and are expected to run for the same amount of time, but variation in the execution run time exists, resulting in varying lengths of time series data for each experiment. Table 7 shows the different run times for the five run instances of LAMMPS.

We aim to normalize all execution runs so that we eliminate performance variation and align all data in time, thereby, accurately preserving application phases of execution. Our method to do this is as follows:

1. We calculate a scale factor for each of the five applications by dividing each application's run times by the average run time of those five instances. Table 7 shows the scale factor calculated

Table 7: Run time Variation

jobId	appname	CounterGroup	time	Scale
1	LAMMPS	L1_D_Cache	1805.095581	1.045169
2	LAMMPS	L1_D_Cache	1725.500933	0.999082
3	LAMMPS	L1_D_Cache	1725.074348	0.998835
4	LAMMPS	L1_D_Cache	1718.414612	0.994979
5	LAMMPS	L1_D_Cache	1723.570011	0.997964

for LAMMPS in collecting the L1 data cache hardware group five times. The set has a mean of 1727.085558, and the most variation shows in jobId=1 and jobID=4. The first run with jobId=1 has a longer run time than the average, and jobId=4 has a shorter run time.

2. Use the scale factor to create progress indices in seconds and milliseconds (i.e., eliminating the timestamp), which can be used to align the multiple runs of an application. This process will expand or shrink the time series for each application run to match the average run time for the five applications.
3. For applications that run faster than the average application time, we add more rows to increase the time series. The new rows are subsequently filled with data using interpolation from the previous and successor data points. We used the Scipy interpolate function for that purpose. For the longer runs, we use the new progress indices to enable shrinking the original time series by allowing some duplicate timestamps. For example, for miniVite, the average run time for the 5 run instances was 361 seconds. The execution of miniVite in Table 6 originally ran for 363 seconds. Therefore, the time series data for this run had to end at 361 seconds. So to do that, we change the time stamp to create the number of duplicates needed at random places to create a time series that ends at 361 seconds.

This work was initially implemented to understand how similar the proxy execution phases are to those in the parent application. Therefore, this process was necessary to align phase boundaries in time from the 5 executions prior to comparing the data in those phases. However, for the work in this report, we examine proxy/parent similarity using the whole application execution only, so we chose to use an average of the final 5 rows of hardware counter values in each application's performance vector. Here we aim to show that aligning the data does not affect the resulting similarity matrix. Figure 27 is the similarity matrix before the alignment, and Figure 28 shows the similarity matrix after the alignment. We show that the data alignment does not significantly change the similarity outcome when we compare the overall behavior of the application's when using an average of the last 5 rows of performance counter values.

4.2 Feature Selection

We describe our chosen feature selection algorithm in Section 3.4 above. Although we investigated many algorithms to do feature selection and extraction, we believe that Laplacian Score in conjunction with a correlation filter are the most accurate and suitable for the type of performance counter data that we collect for this work. We did this work only on the Intel platform. Feature selection on the IBM platform is included in future work.

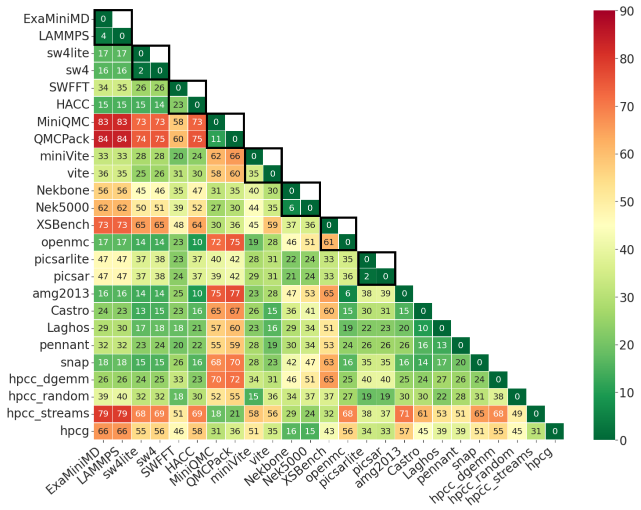


Figure 27: Cosine similarity before data alignment

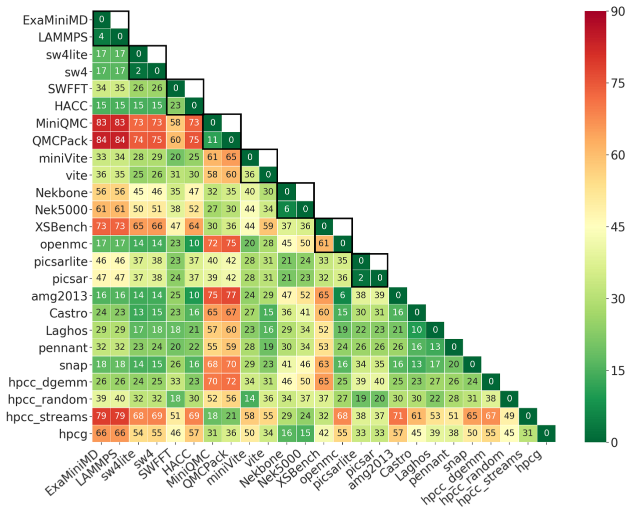


Figure 28: Cosine similarity after data alignment

4.2.1 Important features

Since we assume that each proxy and parent application pair show similar performance, we set the parameter of neighbor size to be 2 in the Laplacian score algorithm. After removing the correlated features via the correlation filter, we get 89 ranked features. Table 8 lists the top 20 features, and

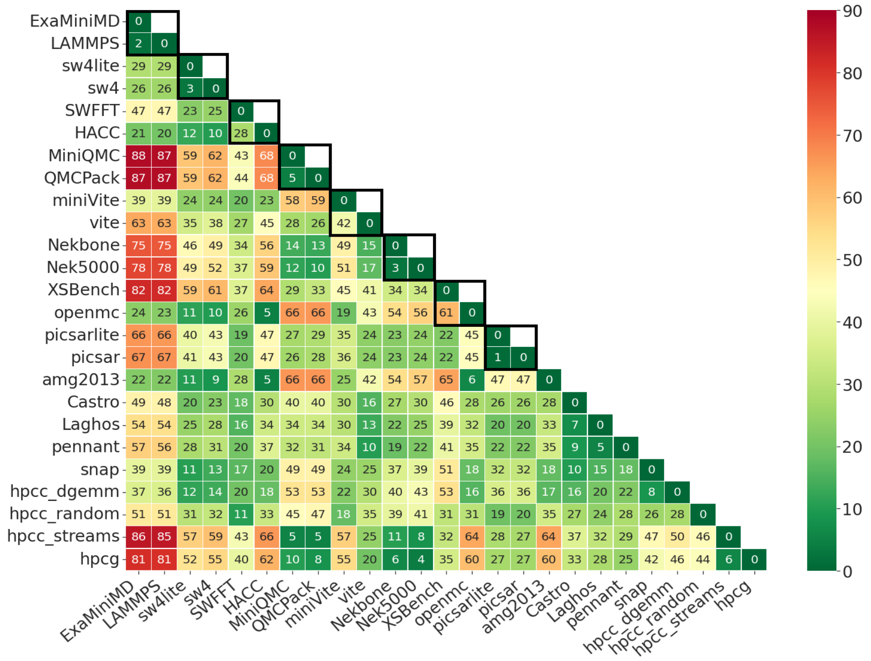


Figure 29: Cosine similarity with the top 20 important features

Figure 29 shows the cosine similarity matrix using these 20 features. Compared to using all 500 features as in Figure 19, using only the top 20 features produces a semantically more interpretable explanation of the similarity score of an application pair. As we can see in Figure 29, all 2×2 black

Table 8: Top 20 features

Rank	**Hardware event count names**
1	'MEM_LOAD_UOPS_L3_HIT_RETIRED:XSNP_NONE'
2	'OFFCORE_REQUESTS:DEMAND_DATA_RD'
3	'UOPS_EXECUTED:THREAD'
4	'OFFCORE_REQUESTS_OUTSTANDING:ALL_DATA_RD'
5	'OFFCORE_REQUESTS_BUFFER:SQ_FULL'
6	'MEM_LOAD_UOPS_L3_MISS_RETIRED:LOCAL_DRAM'
7	'MEM_LOAD_UOPS_RETIRED:HIT_LFB'
8	'CYCLE_ACTIVITY:CYCLES_L1D_MISS'
9	'OFFCORE_REQUESTS_OUTSTANDING:ALL_DATA_RD_CYCLES'
10	'CYCLE_ACTIVITY:CYCLES_LDM_PENDING'
11	'EXE_ACTIVITY:3_PORTS_UTIL'
12	'OFFCORE_REQUESTS_OUTSTANDING:L3_MISS_DEMAND_DATA_RD'
13	'BR_INST_RETIRED:NEAR_CALL'
14	'OFFCORE_REQUESTS:ALL_REQUESTS'
15	'ARITH:DIVIDER_ACTIVE'
16	'ICACHE_64B:IFTAG_HIT'
17	'MEM_UOPS_RETIRED:STLB_MISS_LOADS'
18	'OFFCORE_REQUESTS_OUTSTANDING:L3_MISS_DEMAND_DATA_RD_GE_6'
19	'UOPS_DISPATCHED:PORT_1'
20	'MEM_UOPS_RETIRED:SPLIT_STORES'

border blocks that show high similarity between proxy and parent are preserved. Unsurprisingly, applications that are not pairs remain dissimilar.

Observation 3: *75% of the proxies in our suite demonstrate highly convergent behavior concerning their parents, and, therefore, are faithful representations.*

4.2.2 Feature Standard Deviation

Besides selecting the important features to preserve the similarity of proxy and parent pairs, we were also curious about what contributes to the dissimilarity of the proxy and parent pairs. Therefore, we investigate the full-execution run-time time series data. Using the data of hardware event counts per second, we get the standard deviation for each application (although not all of them are normally distributed, we assume them so in order to simplify the analysis) on each feature (corresponding to hardware event counts). If one point is located within 2 standard deviations, we consider this point to be within the distribution.

We calculate the standard deviation of each feature for each parent application and check whether the accumulated mean of the same feature of a proxy application is within two standard deviations of the parents' accumulated mean within a normal distribution. Table 9 shows the feature numbers for each proxy/parent pair when the difference is greater than 2, 3, 4, or 5 standard deviations, respectively. The result is as we expect as in Figure 19. We can see that SW4lite and SW4, and PISCARlite and PISCAR, are the most similar proxy/parent pairs because, in each pair, the proxy only has one feature ('MOVE_ELIMINATION: SIMD_NOT_ELIMINATED' and 'UOPS_EXECUTED: X87' separately) deviating relatively far from the parent. The proxies that

Table 9: Dissimilarity feature source for proxy/parents pairs

Proxy and Parent pairs	**>2std**	**>3std**	**>4std**	**>5std**
ExaMiniMD / LAMMPS	10	8	8	4
SW4lite / SW4	1	1	1	1
SWFFT / HACC	17	12	11	8
miniQMC / QMCPACK	13	10	8	6
miniVite / Vite	72	38	23	21
Nekbone / Nek5000	11	6	2	2
XSbench / OpenMC	16	9	9	8
PICSARlite / PICSAR	1	1	1	0
Unique feature #s	99	64	49	38

have more features deviating farther from their parents are accordingly less similar. For example, miniVite and Vite have 72 features with large deviation, and they are moderately similar with 35° in the cosine similarity matrix of Figure 19. However, XSBench and OpenMC differ by 61°, so we expect that XSBench would have more features outside 2 standard deviations of the its parent distribution, OpenMC. But we do not see this in Table 9. We need to do some further investigation into this with more closer examination of the features and their actual magnitudes.

Observation 4: *A proxy application that has more event counts distributed outside of 2 standard deviations of the parent application is more divergent from its parent.*

5 Proxy/Parent Application Kernel Comparison

The work presented here satisfies the ADCD504-56 quantitative assessment milestone:
Develop a methodology and tool infrastructure to support collecting an LDMS time-stamped application kernel profile that enables alignment of kernel time-stamps with performance counter event time-stamps collected via LDMS. This may or may not require application instrumentation of kernels (preferably no instrumentation).
It also satisfies the ADCD504-30 milestone:
Apply a quantitative similarity comparison between proxy and parent CPU kernels. Deliverable: include study and results in final ECP report and/or submit a workshop/conference paper on the study and results.

Note that we did run into some technical issues and have additional work to do in this area. We will complete this milestone, but open a new milestone to reflect the work that we will be doing between now and the end of the FY.

Our aim with this work is to compare the behavior of the primary kernels implemented in respective proxy/parent application pairs. Through this, we aim to discover where in the full-execution of the application, similarity or divergence lies and which kernels are similar or different. This sounds fairly simple until these applications are profiled. Many proxies have very different implementations compared to their parents. In addition to profiling, detailed code inspection is required to determine kernels of similar functionality, since names and general code structure are not often similar between a proxy and its parent.

We first present the methodology by which we propose to collect this data. We then present and discuss the platform that we use and the proxy/parent profiles. We include a short section on the issues that we encountered during this work. We then present the data that we collected on two proxy/parent pairs. Finally, we conclude this section with a short discussion about future work.

5.1 Kernel Comparison Methodology

A recently developed capability within the Lightweight Distributed Metric Service (LDMS) makes it possible to use LDMS and our PAPI sampler plugin to collect performance counter data from instrumented kernels. LDMS Streams is an API for publishing and subscribing to support data event collection. Using this in conjunction with the PAPI sampler, we can collect performance counter data within distinct kernel regions.

5.1.1 LDMS and LDMS Streams

In order to receive published Streams data, LDMS daemons and plugins subscribe to a Streams *tag* and receive any published data events which match that tag. This is illustrated in Figure 30 (bottom). At publication, data is specified as being in either `string` or `JSON` format.

Synchronized data collection is achieved across LDMS sampler plugins through the use of a wake-up driven sampling process scheduled against the compute-node's local real-time clock. Synchronization errors in data acquisition across a cluster are the result of real-time clock skew, which is typically minimal on a well-managed HPC cluster, and of wake-up decisions by the OS kernel. In practice, variations of up to a few milliseconds are seen in production systems.

By instrumenting kernels with a *start/stop kernel name* specifier with the Streams API, we get a timestamp with these specifiers that can be correlated to the timestamped PAPI sampler plugin data. Through timestamp correlation, we can determine which PAPI data belongs to which kernel.

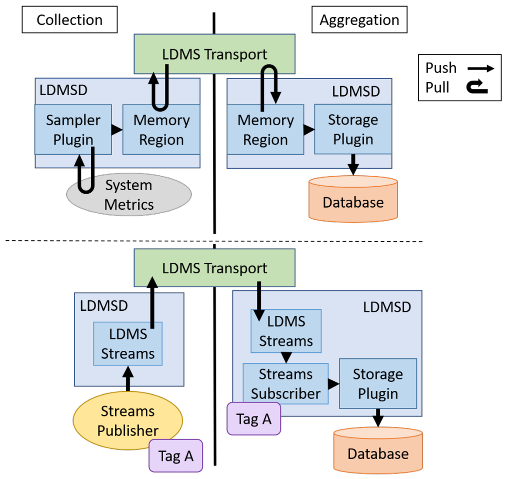


Figure 30: LDMS data collection and transport modes for system data (top) and event data (bottom). Black arrows indicate data flows and *Tag A* indicates a Streams name.

5.2 Experimental Platform

We use the Intel Skylake platform described in Section 3.5.2. Of the applications described in Section 3.5.1, we use the following proxy/parent pairs:

- miniQMC/QMCPACK
- miniVite/Vite
- sw4lite/SW4
- XSbench/openMC

5.3 Application Profiles

For each of the applications under investigation, we collect execution profiles using *gprof*, with the exception of openMC. For some reason, openMC fails with *gprof* so we had to use *CrayPAT* to generate its execution profile.

Two characteristics of *gprof* that we need to keep in mind when doing this work are:

1. it samples at 0.01sec granularity, so kernels that account for a large percentage of execution time, but execute a single instantiation faster than 0.01 sec will not be recorded.
2. it does not account for blocked time. For example, if a kernel blocks for communication, synchronization, or anything else, this time will not be accounted for in the execution profile.

Without further analysis, it is not fully clear that *gprof* not accounting for blocked time affects our analysis. For now, we assume that it doesn't. In future work, we'll look at this issue closer. Based on manual code inspection, we do not believe that the sampling period is resulting in missing data. The functions that we expect to account for large portions of total execution time do, although we'll do a more thorough code analysis in future work to ensure this is the case.

Tables 10 and 11 summarize the flat profile generated by gprof for miniVite and Vite, respectively. We say this is a summary because not all of the data from the profile is included here. We include enough data to show why generating call graphs is a better way to understand an execution profile.

These flat profiles are very difficult to read and can be confusing. For example, one might think

Table 10: Execution Profile, miniVite

% time	cum secs	self secs	calls	self s/call	total s/call	name
27.41	34.47	34.47	3840000	0.00	0.00	distGetMaxIndex
17.73	56.76	22.29	57204054	0.00	0.00	_Rb_tree::_M_get_insert_unique_pos
12.45	72.42	15.66	3840000	0.00	0.00	distBuildLocalMapCounter
10.98	86.23	13.81	3840000	0.00	0.00	distExecuteLouvainIteration
7.68	95.89	9.66	31183927	0.00	0.00	_Node_iterator::_M_insert
6.74	104.36	8.47	29591844	0.00	0.00	_Node_iterator::_M_emplace
4.46	109.97	5.61	3	1.87	15.66	fillRemoteCommunities
3.93	114.91	4.94	8007	0.00	0.00	_Rb_tree::_M_erase
3.92	119.84	4.93	1	4.93	7.06	exchangeVertexReqs
1.63	121.89	2.05	9863948	0.00	0.00	_Hashtable::_M_insert_unique_node
0.55	124.29	0.69	3	0.23	0.31	updateRemoteCommunities
0.45	124.86	0.57	72267964	0.00	0.00	Graph::get_owner
0.31	125.25	0.39	1	0.39	124.33	distLouvainMethod
0.07	125.64	0.09	1	0.09	0.09	BinaryEdgeList::read
0.06	125.71	0.07	1	0.07	0.10	distInitLouvain
0.02	125.74	0.03	2	0.02	0.02	std::vector::_M_default_append

that instrumenting the functions that account for the largest percentage of time is probably the correct action to take. However, most of these functions in the miniVite and Vite profiles are called so many times that the time for a single instantiation of that function is essentially 0.

With LDMS, we typically sample data every 1s. This is the most common sampling period we have ever used in any context of work. LDMS is advertised as being capable of sampling in the millisecond range, but we have never used this capability and have not validated that data collected at this granularity is valid. This brought us to look for kernels that execute for at least one second for a single instantiation.

Because the *gprof* flat profiles are tedious to sift through, we use a tool called *grof2dot* that generates a call graph with timing data imposed on the nodes. We found this to be a more efficient tool for identifying good instrumentation points.

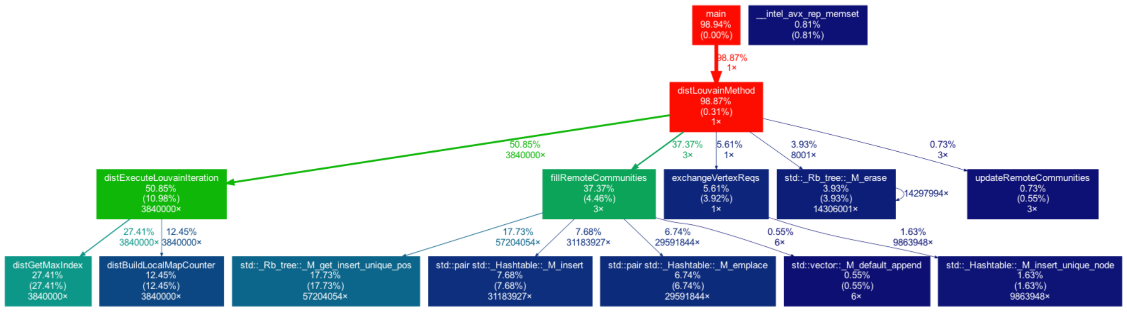


Figure 31: Execution Profile, miniVite

Figures 31 and 32 are graphic representations of the *gprof* flat profiles of miniVite and Vite, respectively. The text that appears in each box is:

- function name
- total time % (self time %)
- total calls

where *total time %* is the percentage of the running time spent in this function and all its children;

Table 11: Execution Profile, Vite

% time	cum secs	self secs	calls	self s/call	total s/call	name
30.19	718.53	718.53	27613312	0.00	0.00	distGetMaxIndex
23.11	1268.53	550.00	27613312	0.00	0.00	distBuildLocalMapCounter
12.13	1557.36	288.84	792277693	0.00	0.00	_Node_iterator::_M_insert
10.78	1813.88	256.52	609357178	0.00	0.00	_Rb_tree::_M_get_insert_unique_pos
5.59	1946.84	132.95	129	1.03	6.25	fillRemoteCommunities
5.11	2068.42	121.58	787817722	0.00	0.00	_Node_iterator::_M_emplace
2.79	2134.76	66.34	23965	0.00	0.00	_Rb_tree::_M_erase
2.33	2190.16	55.40	27613312	0.00	0.00	distExecuteLouvainIteration
1.37	2222.75	32.59	5	6.52	6.87	fill_newEdgesMap
1.12	2249.35	26.60	129	0.21	0.23	updateRemoteCommunities
0.83	2288.84	19.66	6	3.28	4.46	exchangeVertexReqs
0.80	2307.99	19.15	55585079	0.00	0.00	_Node_iterator::_M_emplace
0.49	2333.03	11.60	1257031360	0.00	0.00	DistGraph::getOwner
0.37	2341.90	8.87	258	0.03	0.03	_M_default_append
0.36	2350.59	8.69	43267420	0.00	0.00	_Hashtable::_M_insert_unique_node
0.21	2355.55	4.96	1	4.96	7.03	setUpGhostVertices
0.18	2359.85	4.30	26639769	0.00	0.00	_Rb_tree::find
0.18	2364.10	4.25	6	0.71	9.51	distReNumber
0.14	2367.51	3.41	5	0.68	18.54	distbuildNextLevelGraph
0.09	2372.72	2.07	6	0.35	375.64	distLouvainMethod
0.00	2380.44	0.00	1	0.00	375.64	distLouvainMethodWithColoring

self time % is the percentage of the running time spent in this function alone; *total calls* is the total number of times this function was called (including recursive calls). An edge represents the calls between two functions and has the following layout:

```
                    total time \%
                         calls
 parent --------------------------------> children
```

where *total time %* is the percentage of the running time transferred from the children to this parent (if available); *calls* is the number of calls the parent function called the children.

The color of the nodes and edges varies according to the total time % value. Functions where most of the execution time is spent (hot-spots) are marked as red, and functions where little time is spent are marked as dark blue. Functions where negligible or no time is spent do not appear in the graph by default.

Looking at Figure 31 for miniVite, we would want to instrument the kernels marked in a green color. As was shown in the flat profile, *distExecuteLouvainIteration* and *distGetMaxIndex* are called over 1 million times and since we cannot sample reliably right now at less than 1 second intervals, we cannot instrument these functions. If we go up a level in the call graph and instrument *distLouvainMethod*, we capture practically the entire execution, for which we already have performance counter data.

We ran into the problem noted above not only for miniVite and Vite, but also for several of the other proxy/parent applications pairs that we planned to target. We include the flat profile data and generated call graphs for all of the target proxy/parent pairs in Appendix B.

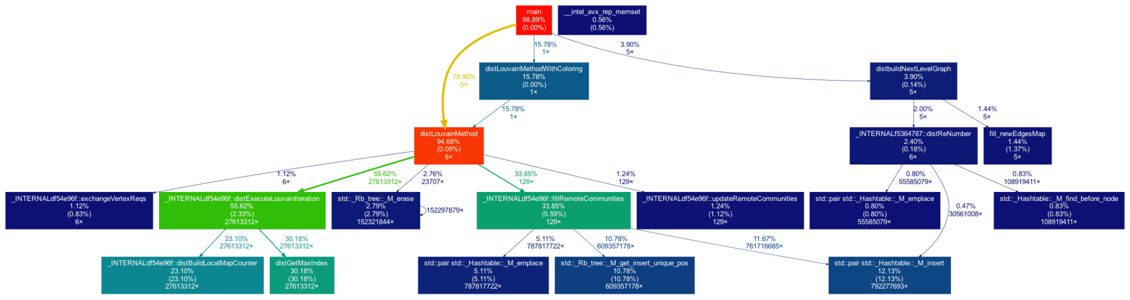

main 98.89% (0.00%)
__intel_avx_rep_memset 0.56% (0.56%)
15.78% 1×
3.90% 5×
78.90% 5×
distLouvainMethodWithColoring 15.78% (0.00%) 1×
15.78% 1×
distbuildNextLevelGraph 3.90% (0.14%) 5×
2.00% 5×
1.44% 5×
distLouvainMethod 94.68% (0.09%) 6×
_INTERNALf5364787::distReNumber 2.40% (0.18%) 6×
fill_newEdgesMap 1.44% (1.37%) 5×
1.12% 6×
55.62% 27613312×
2.76% 23707×
33.85% 129×
1.24% 129×
0.80% 55585079×
0.83% 108919411×
0.47% 30561008×
_INTERNALdf54e96f::exchangeVertexReqs 1.12% (0.83%) 6×
_INTERNALdf54e96f::distExecuteLouvainIteration 55.62% (2.33%) 27613312×
std::_Rb_tree::_M_erase 2.79% (2.79%) 152321844×
152297879×
_INTERNALdf54e96f::fillRemoteCommunities 33.85% (5.59%) 129×
_INTERNALdf54e96f::updateRemoteCommunities 1.24% (1.12%) 129×
std::pair std::_Hashtable::_M_emplace 0.80% (0.80%) 55585079×
std::_Hashtable::_M_find_before_node 0.83% (0.83%) 108919411×
23.10% 27613312×
30.18% 27613312×
5.11% 787817722×
10.78% 609357178×
11.67% 761716685×
_INTERNALdf54e96f::distBuildLocalMapCounter 23.10% (23.10%) 27613312×
distGetMaxIndex 30.18% (30.18%) 27613312×
std::pair std::_Hashtable::_M_emplace 5.11% (5.11%) 787817722×
std::_Rb_tree::_M_get_insert_unique_pos 10.78% (10.78%) 609357178×
std::pair std::_Hashtable::_M_insert 12.13% (12.13%) 792277693×

Figure 32: Execution Profile, Vite

5.3.1 Work Accomplished

Because we ran into the significant problem noted above, that we really need a tool that can sample at sub-second intervals due to the short execution time of most of our target application kernels, we had to modify our work plan. We planned to perform a kernel comparison on four proxy/parent pairs (miniQMC/QMCPACK, miniVite/Vite, sw4lite/sw4, XSBench/openMC). However, because we discovered this issue relatively late in this work, after we had profiled and instrumented all of the target codes, we actually accomplished the following:

- miniQMC/QMCPACK kernel comparison: For QMCPACK, we were able to instrument the DMC (diffusion Monte Carlo) and VMC (variational Monte Carlo) kernels separately. These kernels were at a level in the call graph that did not include the full execution. MiniQMC only implements DMC. Therefore, we chose to compare the DMC and VMC kernels in QMCPACK to the execution of DMC in miniQMC.
- sw4lite/sw4 kernel comparison: Although we have not validated the data collection capability of LDMS using Streams and the PAPI sampler plugin at millisecond sampling granularity, we instrumented the kernels that we identified as equivalent in sw4lite/sw4 and collected the performance counter data for these kernels. We use this performance counter data in our cosine similarity analysis to determine similarity/divergence in this kernel behavior.

The results for these experiments are presented in Section 5.3.2 below. Note that we will elaborate on how this limited study affects future milestones in Section 5.3.3.

5.3.2 Results

Figures 33–36 show the results of comparing the DMC and VMC kernels in QMCPACK to the full execution of miniQMC, which implements the DMC algorithm. We use cosine similarity in this analysis to quantify the angle in degrees between performance event vectors collected for each of the DMC/VMC kernels in QMCPACK. We compare these vectors to the performance event vector of miniQMC's full execution. We also present data for comparison between L1 data, L2, and L3 cache. From Figure 19 we see that miniQMC and QMCPACK are 11° apart. Figure 33 shows that miniQMC is the same distance from QMCPACK DMC and VMC. This is surprising, but note that QMCPACK DMC and VMC differ by only 3°. Figures 34 and 35 show miniQMC closer to QMCPACK VMC than DMC, which is counter-intuitive. However, looking at the execution profiles of these applications in Figures 41 and 42, the DMC::run call graph in QMCPACK does not look very similar compared to that of miniQMC–functions are quite different and have different timings. This supports our idea that differences between these two applications are primarily because QMCPACK has benefitted from significant development efforts that have not been implemented in

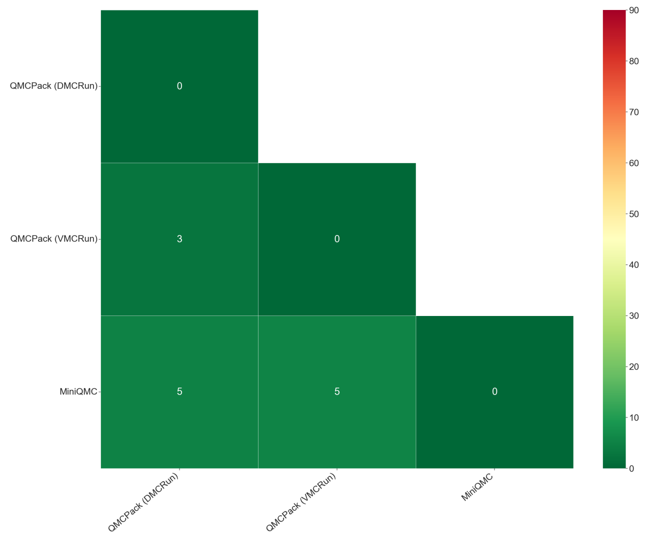


Figure 33: Cosine Similarity, QMCPACK DMC/VMC vs miniQMC full execution

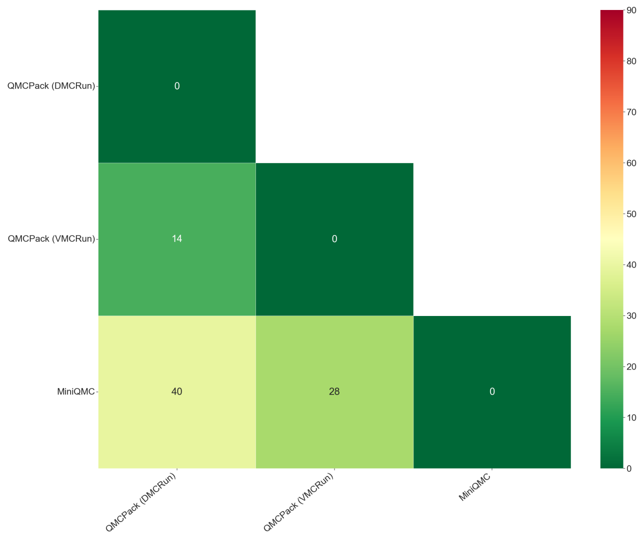


Figure 34: Cosine Similarity, QMCPACK DMC/VMC vs miniQMC full execution, L1 Data Cache

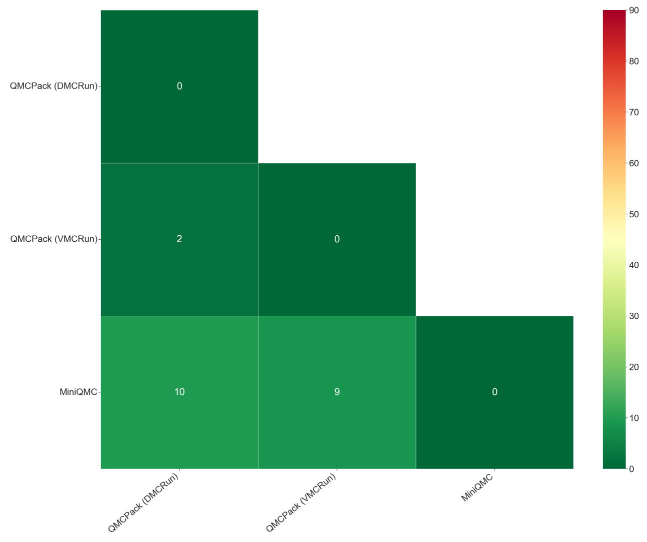


Figure 35: Cosine Similarity, QMCPACK DMC/VMC vs miniQMC full execution, L2 Cache

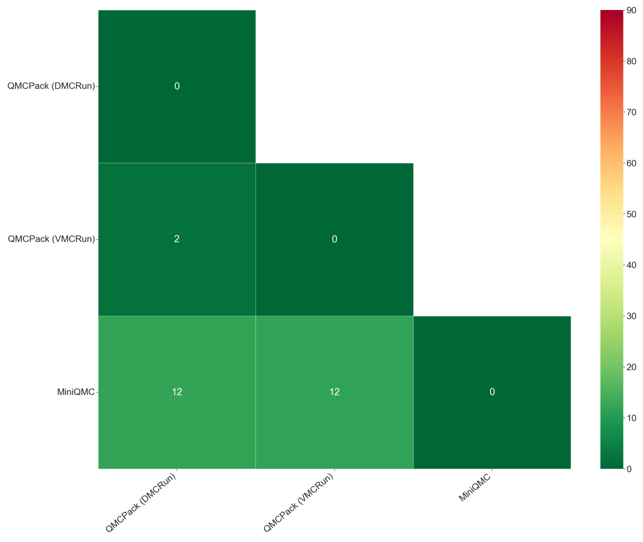


Figure 36: Cosine Similarity, QMCPACK DMC/VMC vs miniQMC full execution, L3 Cache

miniQMC.

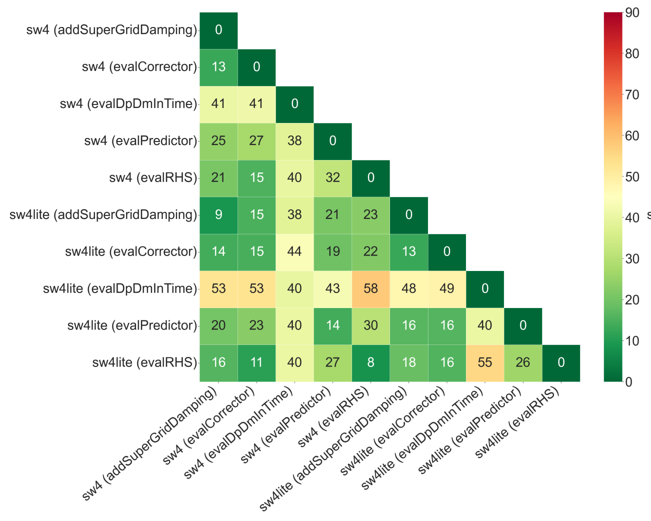


Figure 37: Cosine Similarity, sw4lite/sw4 Kernels

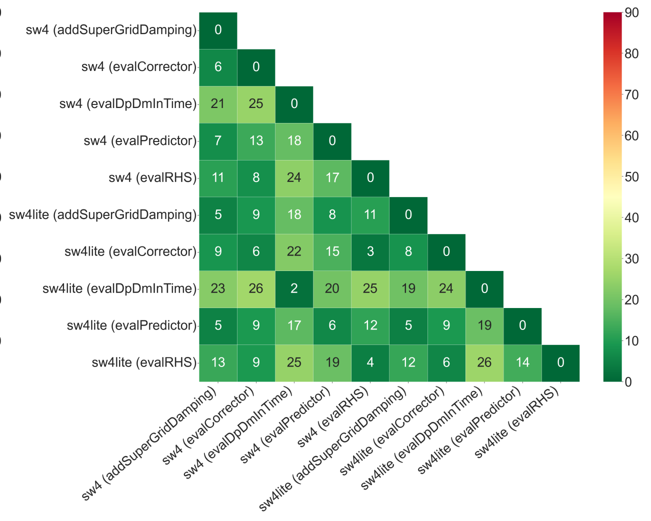


Figure 38: Cosine Similarity, sw4lite/sw4 Kernels, Memory Pipeline Stage

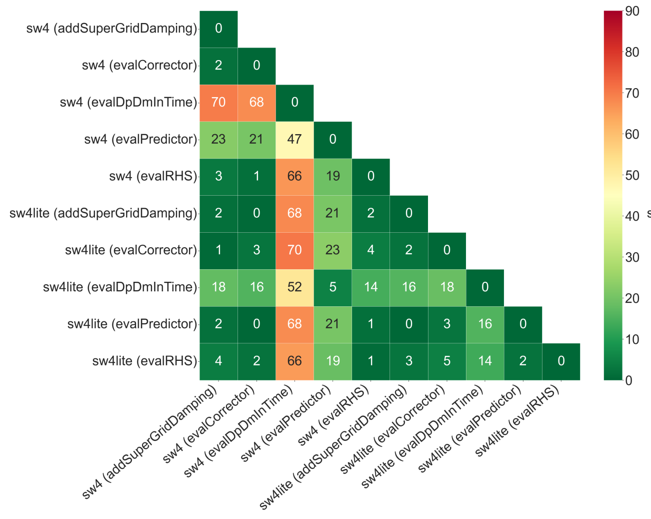


Figure 39: Cosine Similarity, sw4lite/sw4 Kernels, Decode/Issue Pipeline Stage

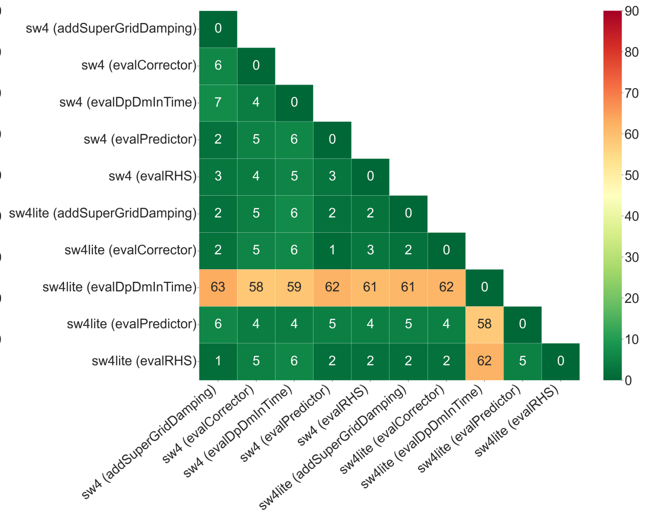


Figure 40: Cosine Similarity, sw4lite/sw4 Kernels, Execution Pipeline Stage

Figures 37–40 show cosine similarity kernel comparisons between sw4lite and sw4. Keep in mind that we have not validated our collection tool when sampling at the subsecond granularity that is required to obtain this data. Validating LDMS, LDMS Streams, and our plugin at this sampling granularity will be done in future work.

From Figure 19 we see that sw4lite and sw4 are only 2° apart. The code base of sw4 and sw4lite is very similar. The functionality implemented in some of the sw4lite kernels is simplified compared to that in sw4. Figure 37 shows the kernel comparison using the full performance event vector within these code kernels. The interesting thing here is all of the kernels between the two applications align fairly well with the exception of *evalDpDminTime*. However, this kernel accounts for only about 3.39% of around 95% of total execution time, which is not enough to affect divergence. The remaining figures show similarity between sw4lite and sw4 kernels for various components in the processor architecture. The primary thing to note here is that the outlier in all of these plots is the dissimilarity between the *evalDpDminTime* kernel in the two apps.

5.3.3 Future Work

Because we ran into the issue of un-validated millisecond sampling interval in our collection tool, we did not collect full data for a complete comparison of kernels between our target proxy/parent pairs. Therefore, we will create a new milestone to be delivered at the end of FY22 that includes:

1. Validation of sub-second sampling intervals using LDMS Streams and the PAPI performance counter plugin. We will validate the data we collect within kernels with a tool such as Vtune or HPCToolkit.
2. Similarity comparison between proxy/parent pairs including miniQMC/QMCPACK, miniVite/Vite, sw4/sw4lite, XSBench/openMC.

6 Proxy Apps: Do they Predict Performance of Their Parents?

We have done much work in this project to understand if proxy applications are representative of the underlying hardware behavior of their respective parents. Because proxy apps are used in system procurement, we began to ask whether proxy execution time is predictive of parent app execution time. We often use proxy applications as benchmarks for HPC system procurement. If proxies do not predict the execution time/scaling behavior of their respective parents, this could potentially be problematic.

We present our predictability studies for a subset of proxy/parent application pairs below. This study is performed on CPUs only, but we have a GPU predictability study milestone due at the end of the FY.

6.1 LAMMPS and ExaMiniMD

LAMMPS is a classical molecular dynamics simulation code and stands for Large-scale Atomic/-Molecular Massively Parallel Simulator. ExaMiniMD is a proxy application for molecular dynamics particle codes and is able to run a restricted set of LAMMPS inputs. A comparison of the run times for the two on Mutrino is in Table 12 and on Stria is in Table 13. Mutrino is an Intel Haswell system, while Stria is an ARM-based system.

Problem	Size	Ranks	LAMMPS	ExaMiniMD	Ratio
SNAP Ta	262144	32	1108.8	1022.9	0.92
SNAP Ta	262144	128	1145.0	1031.8	0.90
SNAP Ta	262144	512	1166.6	1052.4	0.90
SNAP Ta	262144	2048	1189.6	1044.08	0.88
SNAP Ta	512000	32	1121.8	929.6	0.83
SNAP Ta	512000	128	1154.6	936.4	0.81
SNAP Ta	512000	512	1190.0	951.8	0.80
SNAP Ta	512000	2048	1235.6	-	-
SNAP Ta	32768	32	348.4	365.8	1.05
SNAP Ta	32768	128	356.4	366.4	1.03
SNAP Ta	32768	512	367.6	372.0	1.01
SNAP Ta	32768	2048	372.8	371.1	1.00
SNAP W	64000	32	1871.6	2513.0	1.34
SNAP W	64000	128	1914.3	2531.7	1.32
SNAP W	64000	512	1901.7	2545.8	1.34
SNAP W	64000	2048	1916.4	2579.2	1.35
lj	262144	32	1016.6	1142.5	1.12
lj	262144	128	1041.8	1157.1	1.11
lj	262144	512	1063.6	1178.7	1.11
lj	262144	2048	-	-	-

Table 12: LAMMPS and ExaMiniMD average times on Mutrino

For these results, we use three problems. We use the SNAP Ta problem with three sizes, the SNAP W problem, and a Lennard-Jones problem. The size is the numbers of lattice points per rank. For the SNAP problems, LAMMPS has 2 atoms per lattice point, while ExaMiniMD has 1 atom per lattice point. For the Lennard-Jones problem, both codes have 4 atoms per grid point. For the results that are not represented in the tables, those problems were too large to run. For a given problem at a given size, the ratio between the codes is fairly constant, but overall, ExaMiniMD

Problem	Size	Ranks	LAMMPS	ExaMiniMD	Ratio
SNAP Ta	149767	56	1374.8	904.1	0.658
SNAP Ta	149767	224	1377.8	916.0	0.665
SNAP Ta	149767	896	1394.4	910.2	0.653
SNAP Ta	149767	3584	1454.0	917.5	0.631
SNAP Ta	292571	56	1387.8	817.2	0.589
SNAP Ta	292571	224	1389.8	823.9	0.593
SNAP Ta	292571	896	1419.0	829.5	0.585
SNAP Ta	292571	3584	1494.2	-	-
SNAP Ta	18725	56	436.6	331.1	0.758
SNAP Ta	18725	224	436.0	333.4	0.765
SNAP Ta	18725	896	439.4	335.7	0.764
SNAP Ta	18725	3584	446.8	337.5	0.755
SNAP W	36571	56	2259.5	2262.3	1.00
SNAP W	36571	224	2262.4	2354.0	1.04
SNAP W	36571	896	2266.0	2358.3	1.04
SNAP W	36571	3584	2280.2	2296.6	1.01
LJ	524288	774.8	56	1101.7	1.42
LJ	524288	780.2	224	1104.2	1.42
LJ	524288	777.6	896	1108.7	1.43
LJ	524288	780.0	3584	1131.1	1.45

Table 13: LAMMPS and ExaMiniMD average times on Stria

does not seem to be predictable of LAMMPS execution time. What these results do show is that the two codes weak scale similarly.

6.2 Nek5000 and Nekbone

Nek5000 is a Computational Fluid Dynamics (CFD) code that solves problems using GMRES and the spectral element method. Nekbone solves a standard Poisson equation using the conjugate gradient method with a fixed number of iterations with a simple or spectral element multigrid preconditioner. A comparison of the run times for the two on Mutrino is in Table 14 and on Stria is in Table 15.

Problem	Ranks	Nek5000	Nekbone	Ratio
small	32	110.2	1.05	0.0095
small	128	153.1	1.11	0.0072
small	512	120.8	1.17	0.0096
small	2048	176.3	1.22	0.0069
large	32	1040.9	4.07	0.0039
large	128	556.3	4.17	0.0075
large	512	529.2	4.25	0.0080
large	2048	1541.3	4.38	0.0028

Table 14: Nek5000 and Nekbone average times on Mutrino

Problem	Ranks	Nek5000	Nekbone	Ratio
small	56	229.4	1.33	0.0058
small	224	151.9	1.42	0.0094
small	896	257.9	1.46	0.0057
small	3584	314.3	1.56	0.0050
large	56	661.9	4.39	0.0066
large	224	776.2	4.58	0.0059
large	896	788.9	4.66	0.0059
large	3584	1184.5	4.88	0.0041

Table 15: Nek5000 and Nekbone average times on Stria

The problem that is being used for Nek5000 is an eddy simulation while Nekbone is run with the same number of elements per rank. The small problem has 160 elements per rank, while the large problem has 500 elements per rank. The ratio of time is the time of Nekbone divided by the time for Nek5000. We can see that there is no correlation of times between the two codes. Nekbone does show a reasonable weak scaling for both of the problems on both machines, but the Nek5000 timings seem to vary without any pattern. Since GMRES and the conjugate gradient method are both Krylov methods, the underlying operations, such as matrix multiplication and dot products, are similar and Nekbone could be a good proxy for Nek5000.

6.3 PICSAR and PICSARlite

PICSAR is a Particle-In-Cell code that is an acronym for Particle-In-Cell Scalable Application Resource from which the proxy code PICSARlite can be extracted. A comparison of the run times for the two codes on Mutrino is in Table 16 and on Stria is in Table 17. The runs for PICSAR were done with I/O disabled since that added a large and variable overhead.

Problem	Ranks	PICSAR	PICSARlite	Ratio
input_file	32	523.4	238.2	0.455
input_file	128	565.2	258.9	0.458
input_file	512	816.1	343.2	0.421
input_file	2048	963.0	407.4	0.423
homogeneous plasma	32	369.1	378.6	1.026
homogeneous plasma	128	793.4	778.8	0.982
homogeneous plasma	512	1201.8	1226.1	1.020
homogeneous plasma	2048	2309.6	2305.2	0.998
langmuir wave	32	908.0	387.9	0.427
langmuir wave	128	1024.0	491.5	0.480
langmuir wave	512	1057.9	503.8	0.476
langmuir wave	2048	1073.8	521.1	0.485

Table 16: PICSAR and PICSARlite average times on Mutrino

Problem	Ranks	PICSAR	PICSARlite	Ratio
input_file	56	1548.6	427.8	0.276
input_file	224	1657.0	458.9	0.277
input_file	896	1872.5	525.0	0.280
input_file	3584	1895.8	554.3	0.292
homogeneous plasma	56	612.0	559.6	0.914
homogeneous plasma	224	938.0	858.7	0.915
homogeneous plasma	896	1554.4	1420.1	0.914
homogeneous plasma	3584	2471.0	2271.0	0.919

Table 17: PICSAR and PICSARlite average times on Stria

All three of the problems are weak scaled to the number of ranks that they are run on. The Langmuir wave problem would not run on Stria. What these tables show is that PICSARlite is not predictive of PICSAR. However, the two codes have a similar weak scaling. In looking at profiles of the codes, much of the underlying code is the same and has similar timings for a run for a given problem on some number of ranks. However, PICSAR calls a couple of routines that do energy deposition which takes a good amount of time. In that case, PICSARlite could be a good representation of the portion of PICSAR for the code that they have in common.

6.4 SW4 and SW4lite

SW4 is a 3-D seismic modeling code and stands for Seismic Waves, 4th order accuracy. SW4lite is a bare bones version of SW4 that is used to test versions of the important numerical kernels of

SW4 for performance optimization. A comparison of the run times for the two codes on Mutrino is in Table 18 and on Stria is in Table 19.

Problem	Size	Ranks	SW4	SW4lite	Ratio
GaussianHill	h=0.008	32	1334.2	1283.1	0.962
GaussianHill	h=0.005	128	1430.5	1351.1	0.944
GaussianHill	h=0.003	512	1666.7	1584.4	0.951
GaussianHill	h=0.002	2048	1694.7	1644.4	0.970
LOH1	h=50.0	32	1671.9	1572.2	0.940
LOH1	h=31.5	128	2379.9	2265.1	0.952
LOH1	h=19.8	512	3621.5	3423.3	0.945
LOH1	h=12.5	2048	6410.1	6112.9	0.954
point source	sg1	32	17.83	8.09	0.454
point source	sg2	32	250.1	116.6	0.466
point source	sg2	128	71.34	30.97	0.434
point source	sg3	128	1002.9	459.2	0.458
point source	sg2	512	24.05	10.03	0.417
point source	sg3	512	287.3	124.7	0.434
point source	sg3	2048	98.34	45.03	0.458
point source	sg4	2048	1173.3	532.8	0.454

Table 18: SW4 and SW4lite average times on Mutrino

Problem	Size	Ranks	SW4	SW4lite	Ratio
GaussianHill	h=0.008	56	1282.7	1304.4	1.017
GaussianHill	h=0.005	224	1367.7	1412.3	1.033
GaussianHill	h=0.003	896	1547.3	1620.3	1.047
GaussianHill	h=0.002	3584	1627.7	1647.1	1.012
LOH1	h=50.0	56	1475.1	1494.4	1.013
LOH1	h=31.5	224	2306.2	2380.0	1.032
LOH1	h=19.8	896	3621.3	3738.9	1.032
LOH1	h=12.5	3584	6150.7	6288.8	1.022
point source	sg1	56	141.2	8.13	0.057
point source	sg2	56	1853.4	118.5	0.064
point source	sg2	224	560.3	32.80	0.059
point source	sg3	224	7380.2	467.6	0.063
point source	sg2	896	195.5	9.41	0.048
point source	sg3	896	2234.6	131.7	0.059
point source	sg3	3584	783.8	41.78	0.053
point source	sg4	3584	8935.0	533.5	0.060

Table 19: SW4 and SW4lite average times on Stria

The tables show that the runtimes for SW4 and SW4lite correlate fairly well for the GaussianHill and LOH1 problems, but not for the point source problem. If we look at profiles of the code, we find that the point source problem is a special case for both SW4 and SW4lite. For SW4, the code checks the simulation answer at each timestep which takes a significant amount of the run time, while for SW4lite the answer is checked once at the end of the simulation. Therefore, we can say that for some problems, SW4lite is predictive of SW4.

6.5 Vite and miniVite

Vite is an application that does graph clustering and graph community detection using the Louvain method. The Louvain method consists of several phases each of which has a number of iterations. It is implemented in MPI and OpenMP and has several parallel heuristics and approximate computing techniques. MiniVite is a proxy application that implements a single phase of the Louvain method. A comparison of the run times for the two on Mutrino is in Table 20 and on Stria is in Table 21.

Problem	Size	Ranks	Vite	miniVite	Ratio
USA road	23.9M	32	32.3	23.6	0.65
random	25.6M	32	622.1	91.8	0.148
random	25.6M	128	189.4	21.0	0.111
random	102.4M	128	1700.2	101.5	0.060
random	102.4M	512	494.4	23.8	0.048
random	164M	512	777.5	38.8	0.50
random	164M	2048	241.1	13.9	0.057

Table 20: Vite and miniVite average times on Mutrino

Problem	Size	Ranks	Vite	miniVite	Ratio
USA road	23.9M	56	39.5	27.8	0.71
random	25.6M	56	924.8	130.7	0.141
random	25.6M	224	249.9	28.5	0.114
random	102.4M	224	2464.2	138.7	0.056
random	102.4M	896	888.1	85.1	0.096
random	164M	896	1490.1	89.8	0.060
random	164M	3584	2122.1	73.7	0.035

Table 21: Vite and miniVite average times on Stria

The size of the problem refers to the number of verticies in the graph, the times are the average of five runs, and the ratio is the miniVite time divided by the Vite time. As can be seen from the tables, there seems to be no correlation between the times for Vite and miniVite. MiniVite performs one phase of the Louvain method on the problem, while Vite does several phases until the method converges. The number of iterations that are done on the one phase that miniVite performs corresponds to the number of iterations for the first phase of Vite, but the subsequent phases for Vite have more iterations and the number of phases and iterations per phase varies both with the problem and, to a lesser effect, the number of ranks being used to solve the problem. So, if we look at the strong scaling of Vite and miniVite for the problems we have here, they do not scale similarly. However, even with miniVite not being predictive of the runtime of Vite, its behavior should be similar to Vite for a single phase since the underlying routines are the same.

6.6 Conclusion

For the five sets of proxies and parents that we studied, we found that only one proxy was predictive of its parent. For the example problems that we ran, SW4lite was predictive of SW4. In looking deeper at the other parent/proxy pairs, we see various differences. For example, with Vite and miniVite, the underlying routines are the same, but miniVite does one iteration while Vite iterates the problem to solution. PICSARlite is derived from PICSAR and for the problems that we ran, the underlying timings are very similar except that PICSAR calls some routines to do energy deposition that are fairly costly that PICSARlite does not call. Nek5000 and Nekbone both have Krylov based solvers, but the solver for Nek5000 is more complex and solves a real problem while Nekbone solves a Poisson problem. The last pair, LAMMPS and ExaMiniMD were solving the same problems with similar algorithms, but there seems to be no correlation for their timings. These various proxies seem to be representative of their parents in some ways, but not necessarily predictive of run time with the problems we ran them with.

7 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.

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.

A Performance Counter Groups

Here we list all of the component groups that we define and the events that belong to these respective groups. Event groups with a *Primary* indicator mean that this group can be used to compare similarity behavior across different systems. Again, these groupings are not perfect, but are an initial attempt to capture per-component behavior.

Table 22: Instruction and L1D Cache Component Groups

Instruction		L1D	
IBM P9	Intel SKX	IBM P9	Intel SKX
PM_L1_ICACHE_MISS	ICACHE_64B:IFTAG_HIT	PM_LD_MISS_L1	L1D:REPLACEMENT
PM_INST_FROM_L2	ICACHE_64B:IFTAG_MISS	PM_LD_REF_L1	MEM_LOAD_RETIRED:L1_HIT
PM_INST_FROM_L2MISS	L2_RQSTS:ALL_CODE_RD	PM_ST_MISS_L1	MEM_LOAD_RETIRED:L1_MISS
PM_L2_INST_MISS	L2_RQSTS:CODE_RD_HIT	PM_ST_FIN	MEM_LOAD_UOPS_RETIRED:HIT_LFB
PM_INST_FROM_L3	L2_RQSTS:CODE_RD_MISS	PM_L1_DCACHE_RELOAD_VALID	
PM_INST_FROM_L3MISS		PM_DATA_FROM_L2	
PM_INST_FROM_LL4		PM_DATA_FROM_L2MISS	
PM_INST_FROM_LMEM		PM_DATA_FROM_L3	
PM_INST_FROM_RL4		PM_DATA_FROM_L3MISS	
PM_INST_FROM_RMEM		PM_DATA_FROM_LMEM	
PM_INST_FROM_DL4		PM_DATA_FROM_RMEM	
PM_INST_FROM_DMEM		PM_DATA_FROM_DMEM	
PM_IC_PREF_WRITE		PM_L1_PREF	
PM_L2_INST		PM_LD_MISS_L1_FIN	
PM_L2_IC_INV			
PM_ISIDE_DISP			
PM_L2_LD			
PM_L2_ST			
PM_INST_FROM_L1			
PM_IC_MISS_CMPL			

Table 22 shows events for the IBM Power9 and Intel Skylake for the instruction and L1 data caches.

B Application Profiles

Table 23: Execution Profile, miniQMC

% time	cum secs	self secs	calls	self s/call	total s/call	name
27.83	35.09	35.09	419175	0.00	0.00	MultiBspline::evaluate_vgh
12.28	50.57	15.48	1	15.48	15.48	einspline_spo::set
9.67	62.76	12.19	418980	0.00	0.00	DistanceTableAA::evaluate
6.53	82.28	8.23	419153	0.00	0.00	DistanceTableAA::move
5.85	89.66	7.38	330606	0.00	0.00	einspline_sp::evaluate_v
5.09	96.08	6.42	210104	0.00	0.00	TwoBodyJastrow::acceptMove
4.78	102.10	6.02	330693	0.00	0.00	DistanceTableAA::moveOnSphere
3.64	106.69	4.59	419036	0.00	0.00	DiracDeterminant::ratioGrad
2.66	110.04	3.35	419182	0.00	0.00	TwoBodyJastrow::ratioGrad
2.57	113.28	3.25	330772	0.00	0.00	DiracDeterminant::ratio
1.14	116.80	1.44	28	0.05	0.05	DistanceTableAA::evaluate
1.05	118.13	1.33	419070	0.00	0.00	DistanceTableBA::evaluate
0.84	119.19	1.06	56	0.02	0.02	DiracDeterminant::DiracDeterminant
0.68	120.05	0.86	27	0.03	0.03	DistanceTableAA::resize
0.55	120.74	0.69	28	0.02	0.02	TwoBodyJastrow::evaluateLog
0.51	121.38	0.64	419155	0.00	0.00	DistanceTableBA::move
0.44	122.52	0.55	56	0.01	0.01	DiracDeterminant::evaluateLog
0.40	123.03	0.51	330616	0.00	0.00	TwoBodyJastrow::ratio
0.36	123.49	0.46	330681	0.00	0.00	DistanceTableBA::moveOnSphere
0.24	124.19	0.30	504998	0.00	0.00	OneBodyJastrow::computeU3
0.11	124.63	0.14	418943	0.00	0.00	OneBodyJastrow::ratioGrad
0.10	124.75	0.12	210080	0.00	0.00	WaveFunction::acceptMove
0.10	124.87	0.12	28	0.00	0.01	OneBodyJastrow::evaluateLog
0.00	126.07	0.00	1	0.00	15.48	build_SPOSet

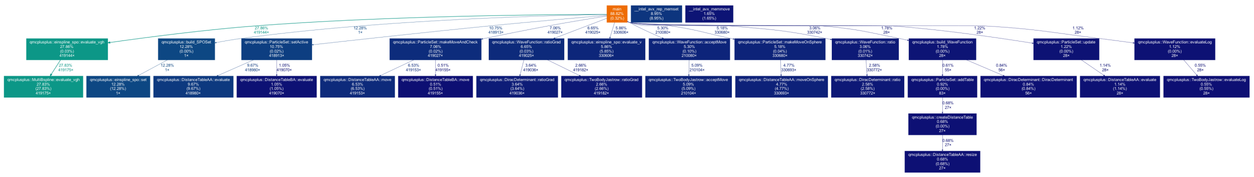

main 88.82% (0.32%)
__intel_avx_rep_memset 8.95% (8.95%)
__intel_avx_memmove 1.65% (1.65%)
27.86% 419144×
12.28% 1×
10.75% 418913×
7.06% 419027×
6.65% 419025×
5.86% 330606×
5.30% 210080×
5.18% 330680×
3.06% 330742×
1.78% 28×
1.22% 28×
1.12% 28×
qmcplusplus::einspline_spo::evaluate_vgh 27.86% (0.03%) 419144×
qmcplusplus::build_SPOSet 12.28% (0.00%) 1×
qmcplusplus::ParticleSet::setActive 10.75% (0.02%) 418913×
qmcplusplus::ParticleSet::makeMoveAndCheck 7.06% (0.02%) 419027×
qmcplusplus::WaveFunction::ratioGrad 6.65% (0.03%) 419025×
qmcplusplus::einspline_spo::evaluate_v 5.86% (5.85%) 330606×
qmcplusplus::WaveFunction::acceptMove 5.30% (0.10%) 210080×
qmcplusplus::ParticleSet::makeMoveOnSphere 5.18% (0.04%) 330680×
qmcplusplus::WaveFunction::ratio 3.06% (0.01%) 330742×
qmcplusplus::build_WaveFunction 1.78% (0.00%) 28×
qmcplusplus::ParticleSet::update 1.22% (0.00%) 28×
qmcplusplus::WaveFunction::evaluateLog 1.12% (0.00%) 28×
27.83% 419175×
12.28% 1×
9.67% 418980×
1.05% 419070×
6.53% 419153×
0.51% 419155×
3.64% 419036×
2.66% 419182×
5.09% 210104×
4.77% 330693×
2.58% 330772×
0.61% 55×
0.84% 56×
1.14% 28×
0.55% 28×
qmcplusplus::MultiBspline::evaluate_vgh 27.83% (27.83%) 419175×
qmcplusplus::einspline_spo::set 12.28% (12.28%) 1×
qmcplusplus::DistanceTableAA::evaluate 9.67% (9.67%) 418980×
qmcplusplus::DistanceTableBA::evaluate 1.05% (1.05%) 419070×
qmcplusplus::DistanceTableAA::move 6.53% (6.53%) 419153×
qmcplusplus::DistanceTableBA::move 0.51% (0.51%) 419155×
qmcplusplus::DiracDeterminant::ratioGrad 3.64% (3.64%) 419036×
qmcplusplus::TwoBodyJastrow::ratioGrad 2.66% (2.66%) 419182×
qmcplusplus::TwoBodyJastrow::acceptMove 5.09% (5.09%) 210104×
qmcplusplus::DistanceTableAA::moveOnSphere 4.77% (4.77%) 330693×
qmcplusplus::DiracDeterminant::ratio 2.58% (2.58%) 330772×
qmcplusplus::ParticleSet::addTable 0.92% (0.00%) 83×
qmcplusplus::DiracDeterminant::DiracDeterminant 0.84% (0.84%) 56×
qmcplusplus::DistanceTableAA::evaluate 1.14% (1.14%) 28×
qmcplusplus::TwoBodyJastrow::evaluateLog 0.55% (0.55%) 28×
0.68% 27×
qmcplusplus::createDistanceTable 0.68% (0.00%) 27×
0.68% 27×
qmcplusplus::DistanceTableAA::resize 0.68% (0.68%) 27×

Figure 41: Execution Profile Call Graph, miniQMC

Table 24: Execution Profile, QMCPACK

% time	cum secs	self secs	calls	self s/call	total s/call	name
22.83	42.18	42.18	294927	0.00	0.00	BsplineSet::evaluate
17.12	73.81	31.63	959235	0.00	0.00	SoaDistanceTableAA::move
13.64	99.00	25.19	664308	0.00	0.00	BsplineSet::evaluate
7.26	126.16	13.41	294927	0.00	0.00	SoaDistanceTableAA::evaluate
4.11	133.75	7.59	137802	0.00	0.00	J2OrbitalSoA::acceptMove
2.97	139.24	5.49	63	0.09	0.11	BsplineSet::evaluate_notranspose
2.64	144.12	4.88	391695	0.00	0.00	SplineC2RSoA::assign_vgl
2.44	148.62	4.50	294912	0.00	0.00	DiracDeterminant::evalGrad
1.90	155.90	3.51	294927	0.00	0.00	J2OrbitalSoA::ratioGrad
1.72	159.08	3.18	959235	0.00	0.00	SoaDistanceTableBA::move
1.15	163.92	2.13	73	0.03	0.03	CoulombPBCAA::evalSR
1.10	165.96	2.04	2103504	0.00	0.00	solve_periodic_interp_1d_d
1.09	167.98	2.02	22	0.09	0.09	SoaDistanceTableAA::evaluate
1.09	169.99	2.01	174	0.01	0.03	DiracDeterminant::updateBuffer
0.97	171.79	1.80	294927	0.00	0.00	SoaDistanceTableBA::evaluate
0.88	175.21	1.62	2	0.81	2.02	SplineAdoptorReader::initialize_spline_pio_gather
0.61	176.33	1.12	664308	0.00	0.00	J2OrbitalSoA::ratio
0.50	177.26	0.93	32	0.03	0.03	DiracMatrix::invert_transpose
0.40	178.00	0.74	294927	0.00	0.00	DiracDeterminant::ratioGrad
0.28	179.07	0.51	294927	0.00	0.00	CoulombPBCAB::evalSR
0.25	180.05	0.47	16	0.03	0.03	J2OrbitalSoA::evaluateLog
0.17	180.37	0.32	2103504	0.00	0.00	find_coefs_1d_d
0.16	180.96	0.29	32	0.01	0.14	DiracDeterminant::evaluateLog
0.14	181.22	0.26	72	0.00	0.71	NonLocalECPotential::evaluate
0.12	181.45	0.23	137802	0.00	0.00	DiracDeterminant::acceptMove
0.12	181.68	0.23	41	0.01	2.58	VMCUpdatePbyP::advanceWalker
0.11	182.09	0.20	19	0.01	0.01	SoaDistanceTableBA::evaluate
0.02	184.25	0.03	17	0.00	1.79	DMCUpdatePbyPWithRejectionFast::advanceWalker
0.00	184.72	0.00	1	0.00	9.85	DMC::resetUpdateEngines()
0.00	184.72	0.00	1	0.00	38.51	DMC::run()
0.00	184.72	0.00	1	0.00	113.13	VMC::run()
0.00	184.72	0.00	1	0.00	30.51	VMC::resetRun()

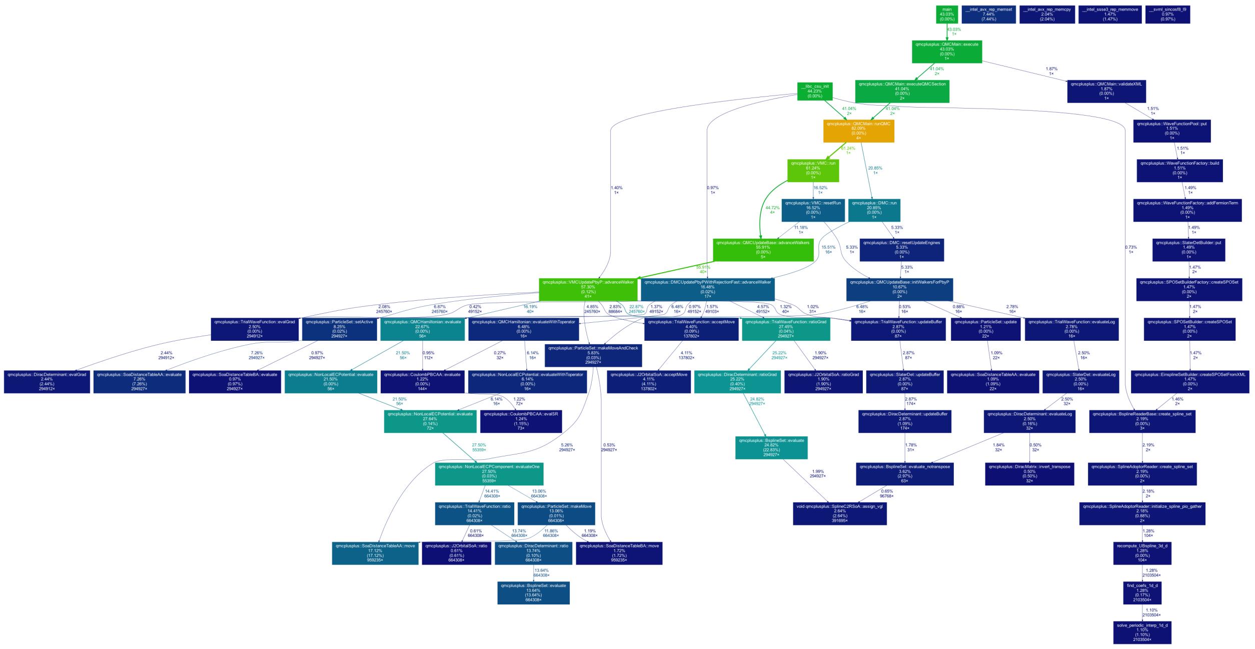


Figure 42: Execution Profile Call Graph, QMCPACK

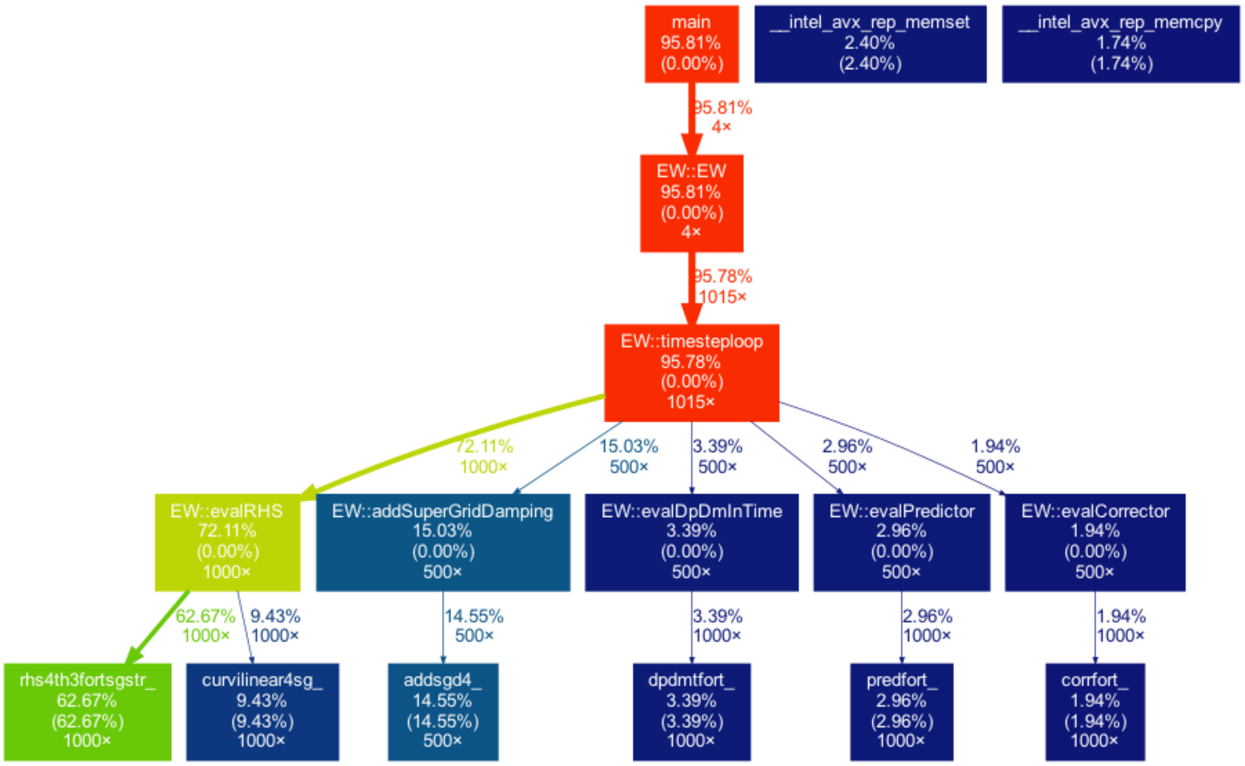


Figure 43: Execution Profile Call Graph, sw4lite

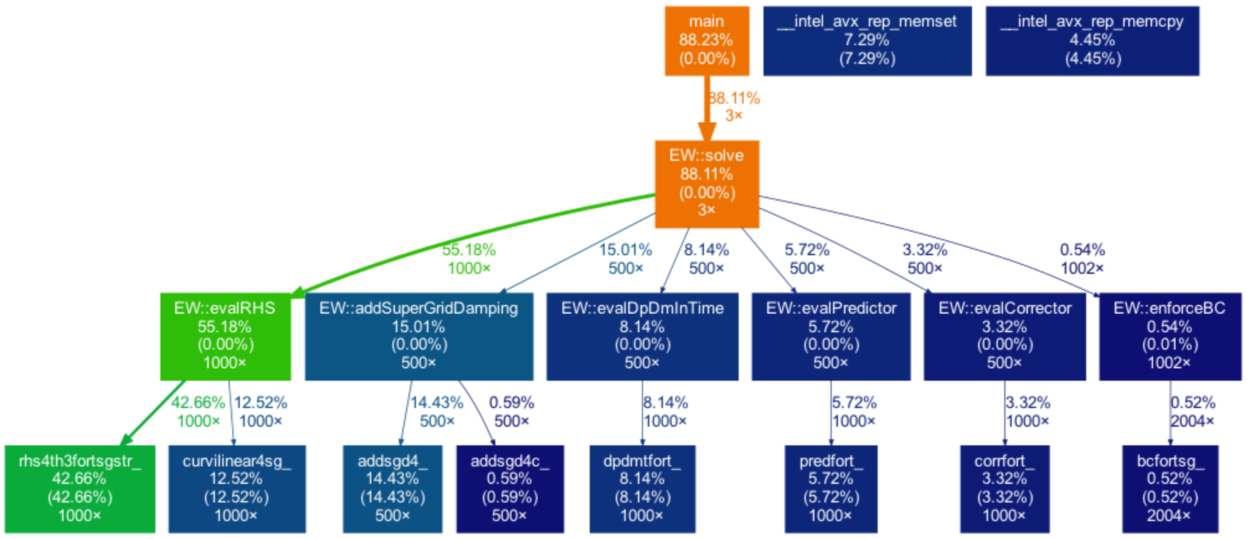


Figure 44: Execution Profile Call Graph, sw4

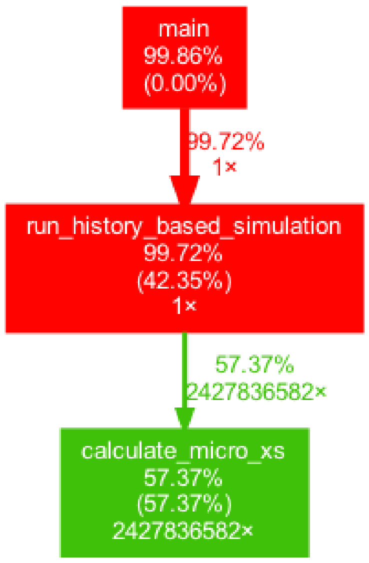


Figure 45: Execution Profile Call Graph, XSBench

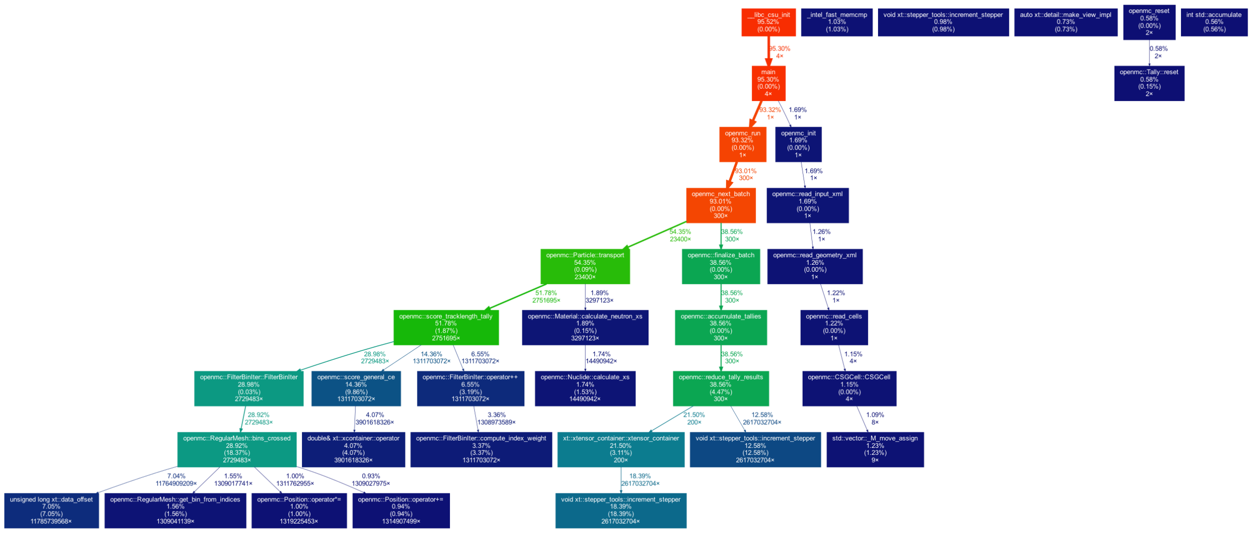

__libc_csu_init 95.52% (0.00%)
_intel_fast_memcmp 1.03% (1.03%)
void xt::stepper_tools::increment_stepper 0.98% (0.98%)
auto xt::detail::make_view_impl 0.73% (0.73%)
openmc_reset 0.58% (0.00%) 2×
int std::accumulate 0.56% (0.56%)
95.30% 4×
0.58% 2×
main 95.30% (0.00%) 4×
openmc::Tally::reset 0.58% (0.15%) 2×
93.32% 1×
1.69% 1×
openmc_run 93.32% (0.00%) 1×
openmc_init 1.69% (0.00%) 1×
93.01% 300×
1.69% 1×
openmc_next_batch 93.01% (0.00%) 300×
openmc::read_input_xml 1.69% (0.00%) 1×
54.35% 23400×
38.56% 300×
1.26% 1×
openmc::Particle::transport 54.35% (0.09%) 23400×
openmc::finalize_batch 38.56% (0.00%) 300×
openmc::read_geometry_xml 1.26% (0.00%) 1×
51.78% 2751695×
1.89% 3297123×
38.56% 300×
1.22% 1×
openmc::score_tracklength_tally 51.78% (1.87%) 2751695×
openmc::Material::calculate_neutron_xs 1.89% (0.15%) 3297123×
openmc::accumulate_tallies 38.56% (0.00%) 300×
openmc::read_cells 1.22% (0.00%) 1×
28.98% 2729483×
14.36% 1311703072×
6.55% 1311703072×
1.74% 14490942×
38.56% 300×
1.15% 4×
openmc::FilterBinIter::FilterBinIter 28.98% (0.03%) 2729483×
openmc::score_general_ce 14.36% (9.86%) 1311703072×
openmc::FilterBinIter::operator++ 6.55% (3.19%) 1311703072×
openmc::Nuclide::calculate_xs 1.74% (1.53%) 14490942×
openmc::reduce_tally_results 38.56% (4.47%) 300×
openmc::CSGCell::CSGCell 1.15% (0.00%) 4×
28.92% 2729483×
4.07% 3901618326×
3.36% 1308973589×
21.50% 200×
12.58% 2617032704×
1.09% 8×
openmc::RegularMesh::bins_crossed 28.92% (18.37%) 2729483×
double& xt::xcontainer::operator 4.07% (4.07%) 3901618326×
openmc::FilterBinIter::compute_index_weight 3.37% (3.37%) 1311703072×
xt::xtensor_container::xtensor_container 21.50% (3.11%) 200×
void xt::stepper_tools::increment_stepper 12.58% (12.58%) 2617032704×
std::vector::_M_move_assign 1.23% (1.23%) 9×
7.04% 11764909209×
1.55% 1309017741×
1.00% 1311762955×
0.93% 1309027975×
18.39% 2617032704×
unsigned long xt::data_offset 7.05% (7.05%) 11785739568×
openmc::RegularMesh::get_bin_from_indices 1.56% (1.56%) 1309041139×
openmc::Position::operator*= 1.00% (1.00%) 1319225453×
openmc::Position::operator+= 0.94% (0.94%) 1314907499×
void xt::stepper_tools::increment_stepper 18.39% (18.39%) 2617032704×

Figure 46: Execution Profile Call Graph, openMC

References

[1] Hpc challenge benchmark. https://icl.utk.edu/hpcc/.

[2] The pennant mini-app. https://github.com/lanl/PENNANT.

[3] Picsar: Particle-in-cell scalable application resource. https://picsar.net/code/.

[4] Scipy wasserstein distance. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wasserstein_distance.html.

[5] Snap: Sn (discrete ordinates) application proxy. https://github.com/lanl/SNAP.

[6] Hpcg benchmark. https://www.hpcg-benchmark.org/, 2020.

[7] 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.

[8] 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.

[9] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating an evolutionary fast path to exascale. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 355–365, 2012. doi:10.1109/SC.Companion.2012.55.

[10] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984. doi:https://doi.org/10.1016/0021-9991(84)90073-1.

[11] James Dickson, Steven Wright, Satheesh Maheswaran, Andy Herdman, Mark C. Miller, and Stephen Jarvis. Replicating HPC I/O Workloads with Proxy Applications. In *2016 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 13–18, 2016. doi:10.1109/PDSW-DISCS.2016.007.

[12] 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.

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

[14] Dominik Maria Endres and Johannes E Schindelin. A new metric for probability distributions. *IEEE Transactions on Information theory*, 49(7):1858–1860, 2003.

[15] 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.

[16] 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.

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

[18] Xiaofei He, Deng Cai, and Partha Niyogi. Laplacian score for feature selection. *Advances in neural information processing systems*, 18, 2005.

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

[20] https://asc.llnl.gov/CORAL benchmarks/Summaries/Nekbone_Summary_v2.3.4.1.pdf. Nekbone. URL: https://asc.llnl.gov/CORAL-benchmarks/Summaries/Nekbone_Summary_v2.3.4.1.pdf.

[21] 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.

[22] D. Kothe, S. Lee, and I. Qualters. Exascale Computing in the United States. *Computing in Science Engineering*, pages 1–1, 2018. doi:10.1109/MCSE.2018.2875366.

[23] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[24] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009. doi:10.1145/1654059.1654100.

[25] Jianwei Li, Wei keng Liao, A. Choudhary, and V. Taylor. I/O analysis and optimization for an AMR cosmology application. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 119–126, 2002. doi:10.1109/CLUSTR.2002.1137736.

[26] Ofir Lindenbaum, Uri Shaham, Erez Peterfreund, Jonathan Svirsky, Nicolas Casey, and Yuval Kluger. Differentiable unsupervised feature selection based on a gated laplacian. *Advances in Neural Information Processing Systems*, 34, 2021.

[27] 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.

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

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

[30] 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.

[31] Selva Prabhakaran. Mahalanobis distance: Understanding the math with examples (python). https://www.machinelearningplus.com/statistics/mahalanobis-distance/, April 2019.

[32] 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.

[33] David Richards, Omar Aaziz, Jeanine Cook, Jeffery Kuehn, Gregory Watson, Peter McCorquodale, William Godoy, Jenna Delozier, Mark Carroll4, Courtenay Vaughan, and The ECP Proxy App Team. Quantitative performance assessment of proxy apps and parents: Report for ecp proxy app project milestone adcd-504-11. https://proxyapps.exascaleproject.org/wp-content/uploads/2021/07/mainAssessment4.pdf.

[34] 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.

[35] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[36] L. I. Sedov. *Similarity and Dimensional Methods in Mechanics.* 1959.

[37] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[38] 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.

[39] 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.

[40] 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.

[41] 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. URL: https://doi.org/10.21105/joss.01370, doi:10.21105/joss.01370.