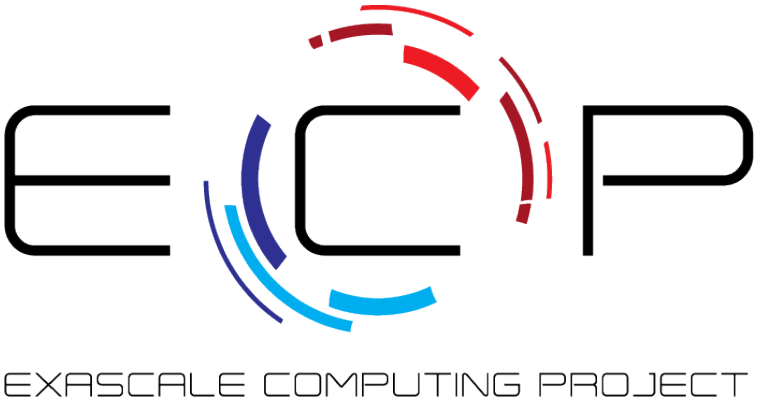
Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone ADCD504-28

Jeanine Cook[1], Omar Aaziz[1], Gregory Watson[2], William Godoy[2], Jenna Delozier[2], Mark Carroll[2], Courtenay Vaughan[1], and The ECP Proxy App Team[6]

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

December 2022



SAND2023-11614R

Contents

1 Executive Summary

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

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

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

- Apply a quantitative similarity comparison between proxy and parent CPU kernels.
- Using cosine similarity and 2-3 input problems per proxy/parent pair, quantify the performance differences between proxy and respective parents when using these different inputs.
- Create a prototype model to accurately model in I/O in AMR-based applications.

This report presents highlights of these efforts.

2 A Proxy Application for Modeling I/O Workloads

Our previous report [3] detailed our efforts to model pre-exascale AMR parallel I/O workloads using proxy applications. The work on modeling AMReX-Castro I/O patterns using MACSio [6] was subsequently published [5]. In this work, we identified limitations in accurately modeling adaptive mesh refinement (AMR) with a granularity at the MPI process and grid refinement levels. While MACSio served as a great proxy to simulate workload at the timestep and MPI process level, there was a need to capture the I/O requests and write operations for each level. In the previous work, we were able to create a relatively simple model to describe the observed I/O behavior, however our plan was to extend this to more complex models that could be used to drive an I/O simulation too. The current effort leverages models developed in this way using a a newly developed proxy application: Reproducible Input Ouput (I/O) Pattern Application (RIOPA)[1]. We have undertaken an extensive data collection and analysis exercise in order to develop models that can be used with this framework. This document reports on the data collection and analysis for model creation as well as providing details on the new RIOPA framework we have developed.

2.1 Methodology

2.1.1 Data Collection & Model Creation

The I/O characteristics of a system can fluctuate based on its working environment. In order to understand the representative file system's behavior (we are continuing to use Summit for this purpose), a temporal stability analysis was performed using the hydrodynamics Sedov test within Castro. The same test was performed for 24 hours on each day of the week four times (28 runs total) to analyze the workloads that the Summit file system undergoes on different times of the day and specific days of the week. Using the Castro log files and the low level I/O logging software Darshan, the a fairly accurate representation of the I/O characteristics of the Sedov runs on Summit were obtained.

During this collection exercise, it was hypothesised that the complexity of the physics model as well as the selected solver could also have an impact on the I/O characteristics. Consequently, an additional canvasing study was performed prior to the scheduled parameter sensitivity study we detail later. During this canvasing, all examples from Castro (version 22.05) were compiled using Summit's GNU compiler with MPI enabled. The job scripts for each example were adapted from the example job scripts used in the prior stability study, with necessary flags adjusted for cases with special requirements. The main goal of this canvasing was to identify other physics simulations that are suitable for the subsequent parameter sensitivity study related to adaptive meshing. The results of this canvasing study are detailed in Table 1. The majority of the examples were deemed not suitable for the parameter sensitivity study due to the fact that the simulations finished without triggering adaptive meshing. Some cases were unable to run due to conflicted internal configurations, which were reported back to Castro's developers as GitHub issues. These affected physical simulation modules were quickly patched by the developers of Castro in the subsequent release.

There are a total number of 21 physics models that are suitable for the parameter sensitivity study. As a result of balancing available human and computing resources, the Kelvin-Heimholtz and the acoustic pulse cases from the hydro category as well as the nova model from the science category became the final subject of parameter sensitivity analysis.

[1] https://github.com/ORNL/RIOPA.jl

Example category	Total cases run	Suitable for this project
gravity	7	2
hydro	18	9
mhd	21	0
radiation	30	0
reacting	19	2
scf	1	0
science	36	8

Table 1: Table of all the Castro examples tested during the additional canvasing.

Using the hydrodynamics tests Kelvin-Heimholtz and acoustic pulse in Castro, the following experiments were designed and performed. Due to the finite number of compute resources, the experiments were focused down to the perceived most important parameters based on how they may affect the I/O during the simulations. Parameter decisions were base on the Castro documentation and the previous research [5] done in this area.

Listing 1: Planned scan table for Kelvin-Heimholtz parameter sensitivity study

```
{
"castro.cfl": [0.3, 0.5, 0.7],
"castro.change_max": [1.1, 1.3, 1.5],
"amr.max_level": [0, 1, 3],
}
```

Base parameters across both hydro cases stay the same but the numerical values in each parameter differed slightly based on the specifics of each case. 27 runs in Listing 1 and 96 runs in Listing 2 were performed.

Listing 2: Planned scan table for acoustic pulse parameter sensitivity study

```
{
"castro.cfl": [0.2, 0.4, 0.6, 0.8],
"castro.change_max": [1.1, 1.3, 1.5],
"amr.max_level": [0, 2, 4, 6],
}
```

The example run-time configuration file for the nova simulation contains 56 adjustable parameters. It is impossible to perform detailed parameter sensitivity analysis via orthogonal experiment for all of the 56 parameters given the limited time and computing resources available. Therefore, the candidates for parameter sensitivity analysis were pruned down to five parameters as shown in Listing 3. This JSON file was consumed by a helper Python script[2] that automatically generate a LMOD-like folder structure as well as corresponding job scripts that can be directly submitted to Summit's job scheduler for execution. The nova's sensitivity resulted in a total number of 495 runs, the outcome of which was used for subsequent modeling attempt for the I/O characteristics of the nova model.

Listing 3: Planned scan table for Nova parameter sensitivity study

```
{
```

[2]This helper script is available on ORNL's Gitlab.

```
    "castro.cfl": [0.2, 0.3, 0.4, 0.5],
    "castro.change_max": [1.1, 1.2, 1.3, 1.4, 1.5],
    "amr.max_level": [2, 4, 8],
    "amr.refine.dengrad.max_level": [2, 4, 8],
    "amr.refine.tempgrad.max_level": [2, 4, 8]
    }
```

2.1.2 I/O Workload Generation Framework

RIOPA.jl is written using the novel Julia programming language and ecosystem [1] and its potential application in HPC systems [2]. RIOPA.jl goal is to provide a proxy framework that can generate independent "stream" levels from a single MPI job that can be fine-tuned using a hierarchical production approach. User interaction is solely through a configuration YAML file to setup the intended independent streams. Listing 4 shows and example configuration used to generate several data streams, similar to AMReX Castro I/O workloads for the default output data backend using POSIX. The configuration file inputs allow for generating and splitting data in a distributed manner across producers in a MPI run using the `proc_payload_groups` entry. In addition, time evolution of the I/O workloads can be modeled using a functional form through an exponential `GrowthFactor` or using a polynomial approach. Thus, the end goal is to give users the degrees of freedom that are closer to how the application produces data with a yet still simple proxy application and user interface.

Listing 4: Configuration file example as input to RIOPA.jl

```
datasets:
  - type: "output"
    name: "plot"
    basename: "plt"
    nsteps: 10
    step_conversion_factor: 10
    compute_seconds: 1.0
    data_streams:
      - name: "Level_0"
        initial_size_range:
          - 3000
          - 3600
        evolution:
          function: "GrowthFactor"
          params: [ 1.15 ]
        proc_payload_groups:
          - size_ratio: "1/3"
            proc_ratio: 0.5
          - size_ratio: "2/3"
            proc_ratio: 0.5
      - name: "Level_1"
        initial_size_range:
          - 6000
          - 7200
        evolution:
```

```
    function: "Polynomial"
    params: [ 0.0 1.0 ]
  proc_payload_groups:
    - size_ratio: "1/3"
      proc_ratio: "1/4"
    - size_ratio: "2/3"
      proc_ratio: "3/4"
```

As shown in Listing 4, the YAML configuration file is a multi-level hierarchy, since we want to flexibly model I/O patterns in terms of <dataset, step, [level, ...], rank>. The following briefly describes the usage of this file in terms of its keywords.

Datasets The `datasets` keyword introduces a list of dataset configurations (with list elements denoted by the '-'). Each is a separate series of I/O operations that may have completely different rules for when and how data is written (or read).

- `type:` Must be either `"input"` or `"output"`
- `name:` Full name for what the dataset represents
- `basename:` Literal string base for sequences of files or directories
- `io_backend:` Choose which file format to use:
  - `"IOStream"`
  - `"HDF5"`
- `nsteps:` How many times this I/O operation will take place in sequence
- `step_conversion_factor:` For some models if the number of I/O steps is not the same as the number of steps (*e.g.*, time steps) that meaningful to the application, you can use this to convert. This allows for the `evolution function` to be expressed in terms of the application step. So, for example, if the application you are modeling writes output every 10 time steps, you would have `step_conversion_factor: 10` and specify your `evolution function` with respect to time steps.
- `compute_seconds:` Time (in seconds) for simulating application compute time between I/O steps for this dataset. RIOPA will use a "sleep" operation to simulate compute time between I/O steps.
- `data_streams:` List of data stream configurations.

Data Streams All the data streams for a given output dataset are written in the same operation, but each stream represents a distinct aspect of the output and may have different rules for how data grows and is distributed among process ranks.

- `name:` Literal string used for naming files or directories. You may arbitrarily nest stream directories using a `/`, like `name: "a/b/c"`
- `initial_size_range:` List of two (integer) values specifying the total size of the output at the first I/O step. This is specified as a range to allow for variability in output. Note that the evolution function below is applied to both limits in the range so that for each stream (and then for each rank executing the stream) there is a range of sizes that can be used. If you want to avoid this variability, just use the same number twice.
- `evolution:` Group specifying how the data size grows from step to step
  - `function:` Name of function type to use. Must be one of the following:

    - * "GrowthFactor"
    - * "Polynomial"
  - params: List of coefficients that fully specify the function
    - * For "GrowthFactor", there must only be one parameter, which is the growth factor
    - * For "Polynomial", there can be one or more parameters, which act as $[a_1, a_2, ...]$. That is, the constant term a_0 is left out. This term is already provided, derived from the initial_size_range

Payload Groups Process payload groups are specified with size ratio and process ratio. These ratios may be given as fractions (or Julia Rationals) in quotes or as floating point values.

- proc_payload_groups: List of payload group configurations for distributing the payload across groups of processes (ranks). This allows for more fine-grained control of how I/O operations behave. If this is not needed, simply specify one group with both the size_ratio and the proc_ratio set to 1.0
  - size_ratio: Portion of stream output size executed by this group of processes
  - proc_ratio: Portion of processes belonging to this group

  Note that the size_ratio entries must sum to 1.0. Likewise for the proc_ratio entries.

2.2 Results

2.2.1 I/O Workloads

The file output from Castro simulation can be categorized into three main types: checkpoint file, plot file and metadata file. The checkpoint file is used to store the simulation state resolved at the mesh resolution and is a binary format designed to store multi-dimensional data separately so as to improve its multi-processing performance under MPI. When adaptive meshing is triggered, the corresponding generated mesh and its associated state is stored in a new file. Consequently, the number of checkpoint files for each increment varies with respect to the adaptive meshing process. The checkpoint file is intended as a cache that can be used for diagnostic purposes during simulated physics evolution as well as to resume interrupted runs. Similarly, the plot file is a snapshot of the simulation state, but stored as a view instead of resolved along each dimension. Each level of the adaptive mesh has its own plot file, and the visualization tools can aggregate them into one single state for user to inspect. The plot file is usually output synchronously with the checkpoint file, however it is also possible to write these two types of output asynchronously. In this study, the synchronous writing was selected as it is the most commonly used configuration among Castro users. The metadata file, as indicated by its name, is a collection of auxiliary files, including a simulation log, mesh definition, and other debugging output specified by the user.

The main objective of the temporal stability study was to evaluate whether the I/O characteristics has a time dependency. The result of this study, depicted as a pair-plot (Figure 1), shows that the I/O characteristics remain stable over the period of the data collection. More specifically, the distribution of each feature of interest (file size, write operation time, and number of write operations), has a very narrow distribution despite the data collection continuing for a long period of time. This lack of variation is an indicator that the I/O characteristics of Castro are close to time independent. In other words, the execution time should not be considered as a variable when building a model to approximate the I/O characteristics. In addition we also discovered that the metadata has almost no impact on the overall I/O characteristics. This is because the I/O of the

metadata file occurs mainly at the beginning and the end of the simulation, consisting of relatively small but frequent write operations. Considering the limited contribution from the metadata, it was decided to excluded this from the subsequent analysis, focusing only on the checkpoint and plot file I/O instead.

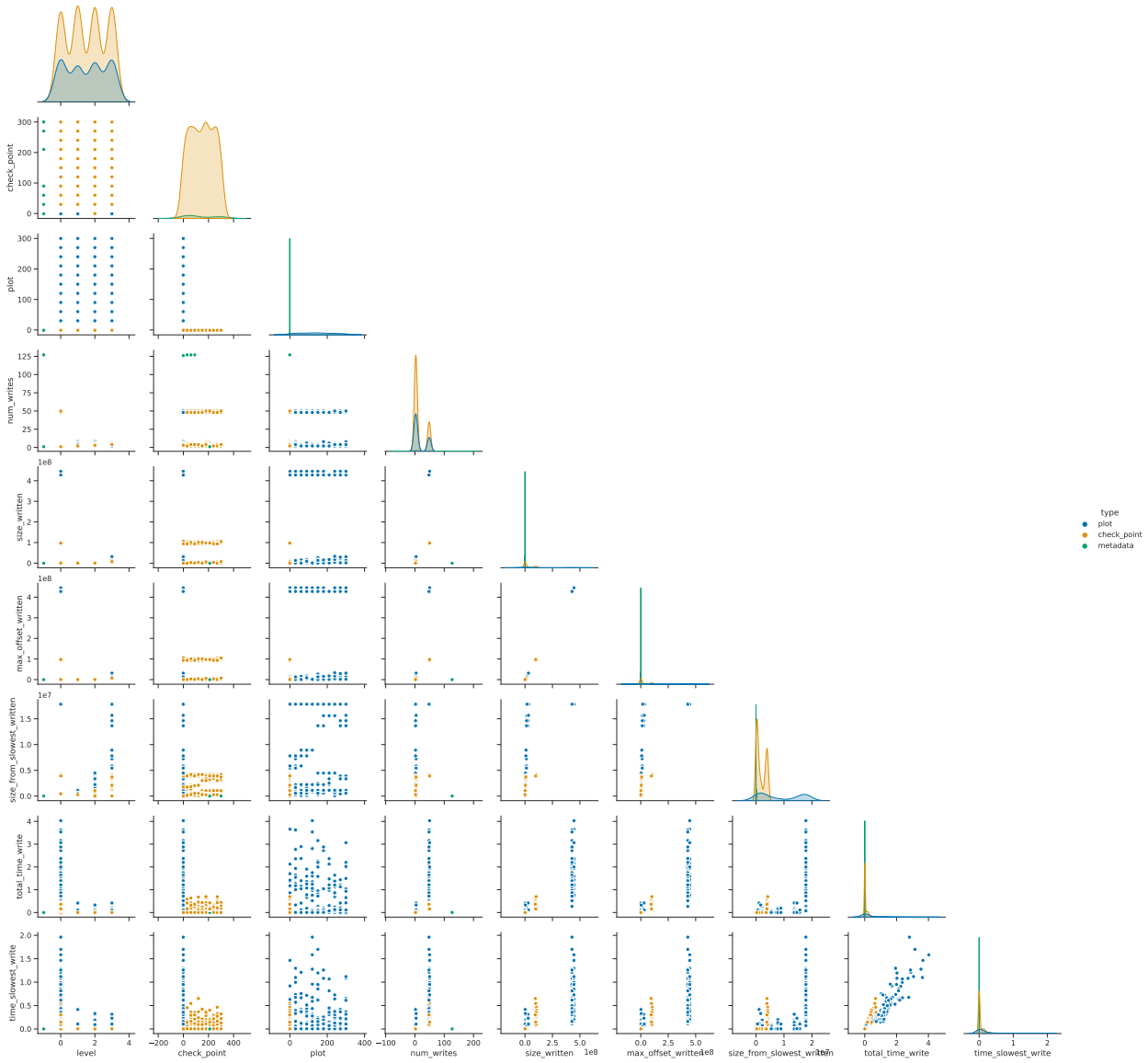


Figure 1: The pair-plot of the initial stage, stability study with the following feature vectors (left to right; top to bottom): AMReX leve (level), checkpoint file increment (check_point), plot file increment (plot), number of writes performed (num_writes), total size written in 100 MB (size_written), maximum offset during writing in 100 MB (max_offset_written), size from slowest write operation in 10 MB (size_from_slowest_written), total time in seconds (total_time_write), time used in slowest write in seconds (time_slowest_write). The color is used to distinguish the file type, including plot file (blue), checkpoint file (yellow), and metadata file (green).

During examination of the data from the stability study, some additional interesting outcomes were observed. For instance, when comparing the number of write operations and file size of check-

point file with those of plot file (Figure 2), it is evident that checkpoint file has larger variations in both number of operations and the file size, especially for those of the base mesh (level 0). This variation suggests that when modeling the checkpoint file and plot file, they should be treated separately as they are fundamentally different processes. Another interesting finding is the difference between I/O from the base mesh (level 0) and adaptive mesh (level 1 to 3). More specifically, the file size and write frequency at level 0 is often orders of magnitude larger than the higher levels (see Figure 3 and Figure 4). Furthermore, the file size and write frequency of adaptive meshing seems to be monotonically increasing with respect to the adaptive level. Therefore, the I/O modeling of the base mesh should be separated from the adaptive mesh, and the I/O modeling of the adaptive mesh can focus on the first adaptive level since the I/O of the rest of the levels can be interpreted for simplicity.

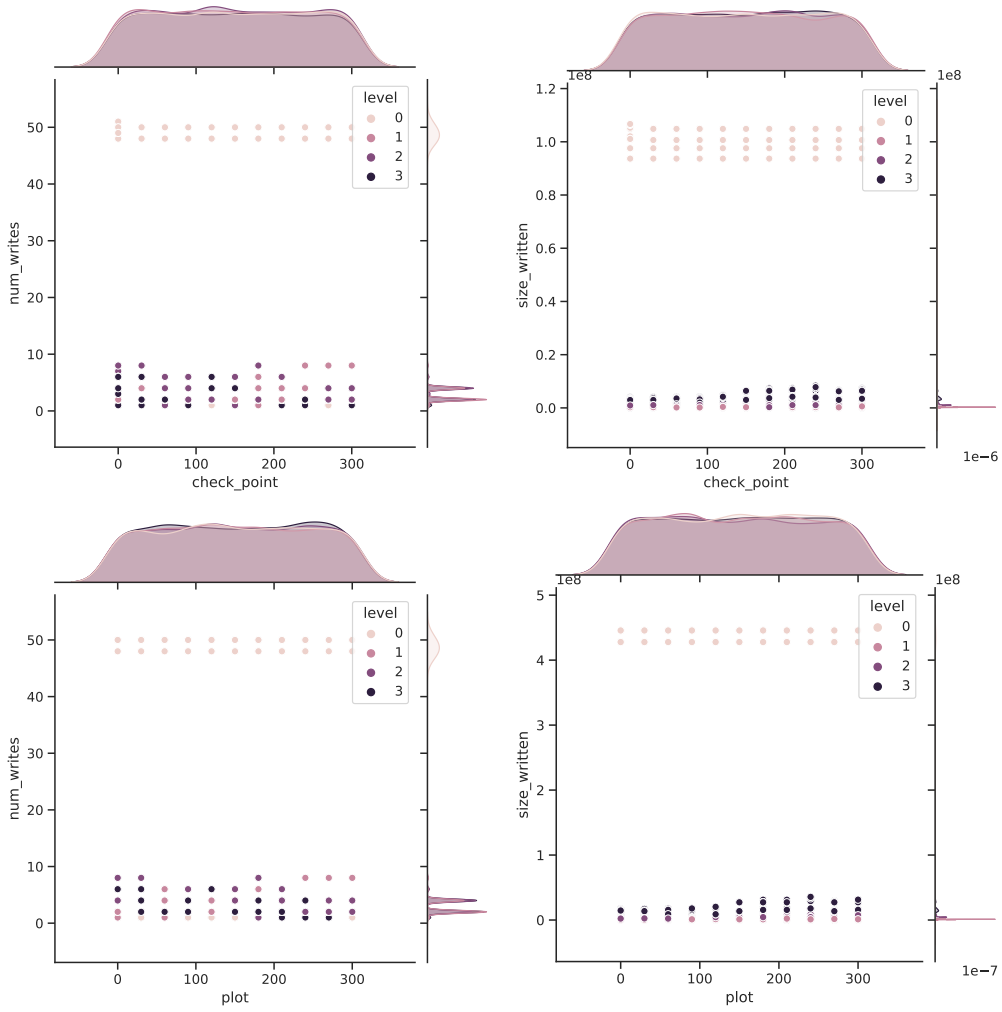


Figure 2: The joint-plot that demonstrates the relationship between file size, number of writing operations with respect to the checkpoint file (upper) and plot file (lower) increment, categorized by corresponding meshing level.

Using the data collected from the parameter sensitivity study, the Kelvin-Heimholtz I/O data was successfully modeled via a series of linear regressions. Given all runs were run at the standard grid size provided in Castro for this specific case, the linear models assume this grid size. The data

was first separated into plot file and checkpoint file sections. Then, each type of file was modeled to all four refinement levels (0,1,2,3) along with a total output size model. This resulted in 10 different linear functions to model the Kelvin-Heimholtz data - with an example of this found in Table 2. Upon further examination of the data during modeling, the most important factors in determining the output size at each level were the time steps, level which each I/O operation occurred, and the max level parameter of the run. Other parameters used, "cfl" and "change max", had a negligible impact on the I/O of this case. Examining all functions across both the plot file and checkpoint sections, it's worth noting the checkpoint file output grow at approximately double the rate of the plot file output across all levels.

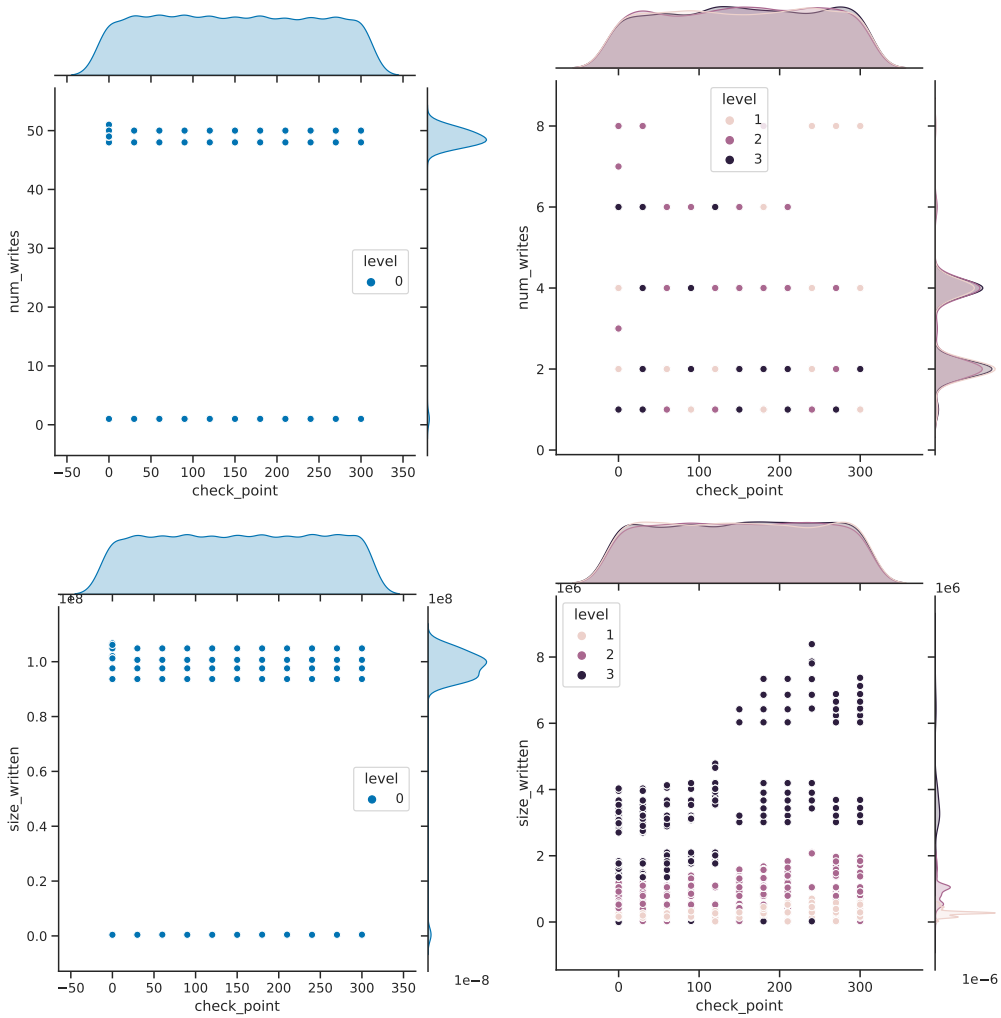


Figure 3: The joint-plot separated to level 0 (left) and the adaptive meshes (right) demonstrates that the I/O characteristics of the base mesh in the checkpoint file (level 0) is distinctively different from the corresponding adaptive meshes.

The analysis of the parameter study of the nova case started with a correlation analysis between the feature of interests, namely the parameters in Listing 3 as well as the mesh level and file sizes, the result of which is shown in Figure 5. Other than "arm.maxlv", the rest of the parameters do not seem to have any direct correlation with the file size of either type. To further confirm this observation, independent correlation analysis was performed for check point file as well as the

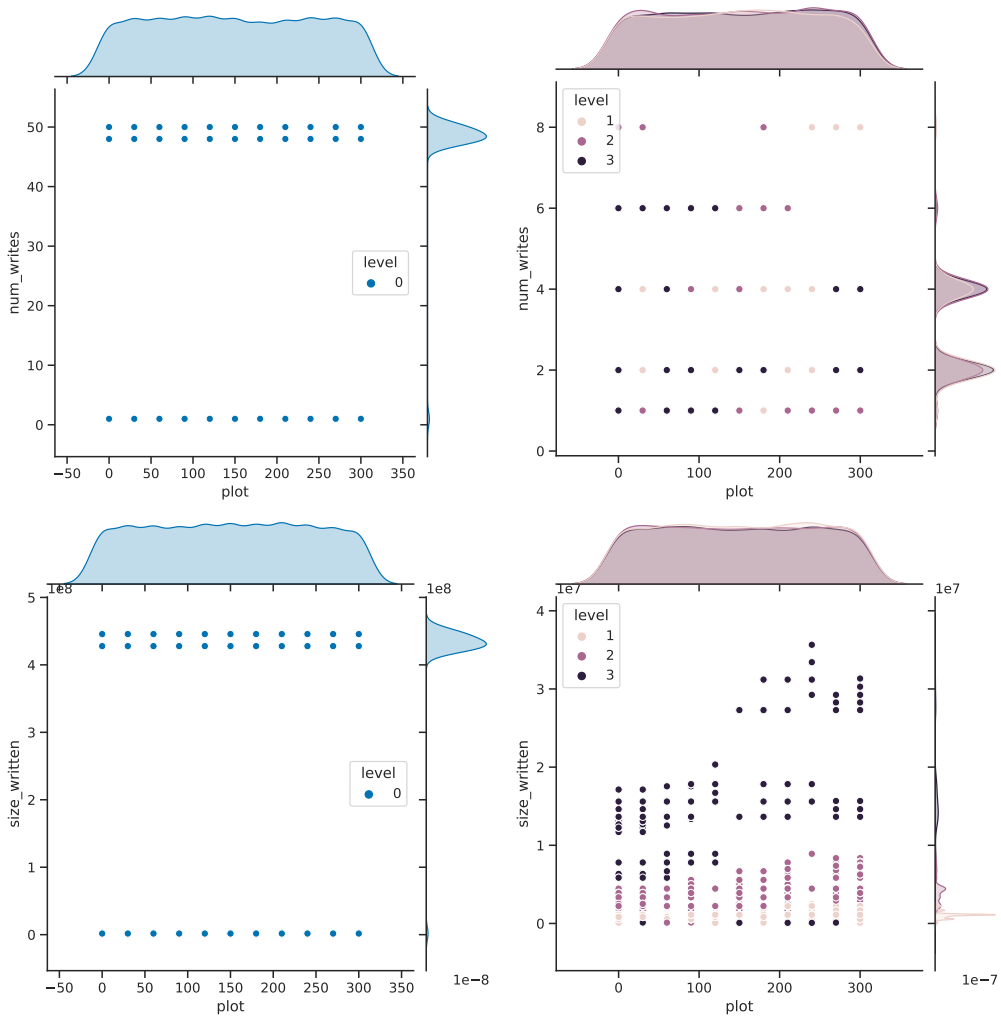


Figure 4: The joint-plot separated to level 0 (left) and the adaptive meshes (right) demonstrates that the I/O characteristics of the base mesh in the plot file (level 0) is distinctively different from the corresponding adaptive meshes.

plot file, resulting in the two additional correlation matrices in Figure 6. Although the actual value differs slightly, the two correlation matrices share a similar pattern, which is also consistent with the overall one in Figure 5. To check if the changes in these parameters have any impact on the file size distribution, the base mesh file size as well as the first adaptive mesh are plotted against increment number in Figure 7 and Figure 8 respectively. The overlapping curves in Figure 7 and Figure 8 further prove that the file size is not heavily impacted by these run-time configuration parameters. The file size insensitivity of these input parameters could be the results of small parameter mapping space, or the deterministic nature of the problem where the changes in computing path does not alter the convergence rate. Nevertheless, this analysis suggested that a statistical approach is more appropriate for modeling the file size of nova cases as it is not possible to derive a formula based on input configurations to compute the file size with the collected data. Consequently, the I/O characteristic modeling of nova case was implemented by generating lookup tables for the checkpoint file and plot file (Figure 9). More specifically, the number of counts (n), mean of file size (μ, in MB) and the associated standard deviation (σ, in MB) are converted to 2D histograms with respect to the associated mesh level and increment number. When computing a supposed file size with mesh level of i at increment j using a mesh of $M \times N$, the proxy app should use the following formula:

$$\Phi(i, j, M, N) = \frac{M \cdot N}{32768} \mathbf{H}(c_{i,j}) \mathbf{G}(\mu_{i,j}, \sigma_{i,j}) \tag{1}$$

where $\mathbf{G}$ is a Gaussian distribution sample function that generates a random size from a Gaussian distribution defined by $\mu_{i,j}$ and $\sigma_{i,j}$, $\mathbf{H}(c_{i,j})$ is the Heaviside step function that zero out query with zero count, and the computed file size is re-scaled by $\frac{M \cdot N}{32768}$ to take into the account that the lookup table is generated with a mesh consists of 32768 elements.

Plot Files
Total Output: $y = 34.153x + 9670$
Level 0: $y = 5153$
Level 1: $y = 3.61x + 542.89$
Level 2: $y = 9.4023x + 1085.6$
Level 3: $y = 21.141x + 2691.3$

Checkpoint Files
Total Output: $y = 67.953x + 19804$
Level 0: $y = 10305$
Level 1: $y = 6.8044x + 1813.2$
Level 2: $y = 18.593x + 2284.6$
Level 3: $y = 42.354x + 5362.7$

Table 2: List of equations used to model a Kelvin-Heimholtz example of a set grid size. Input x is the time step of the simulation and output y is the output file size in bytes. The "max level" of this example is 3.

2.3 Future Work

Although we were able to generate some initial models that represent Castro I/O behavior, the number of these models were were able generate was limited by the core hours that were available for data collection. As discussed, we had to significantly reduce the parameter space we were

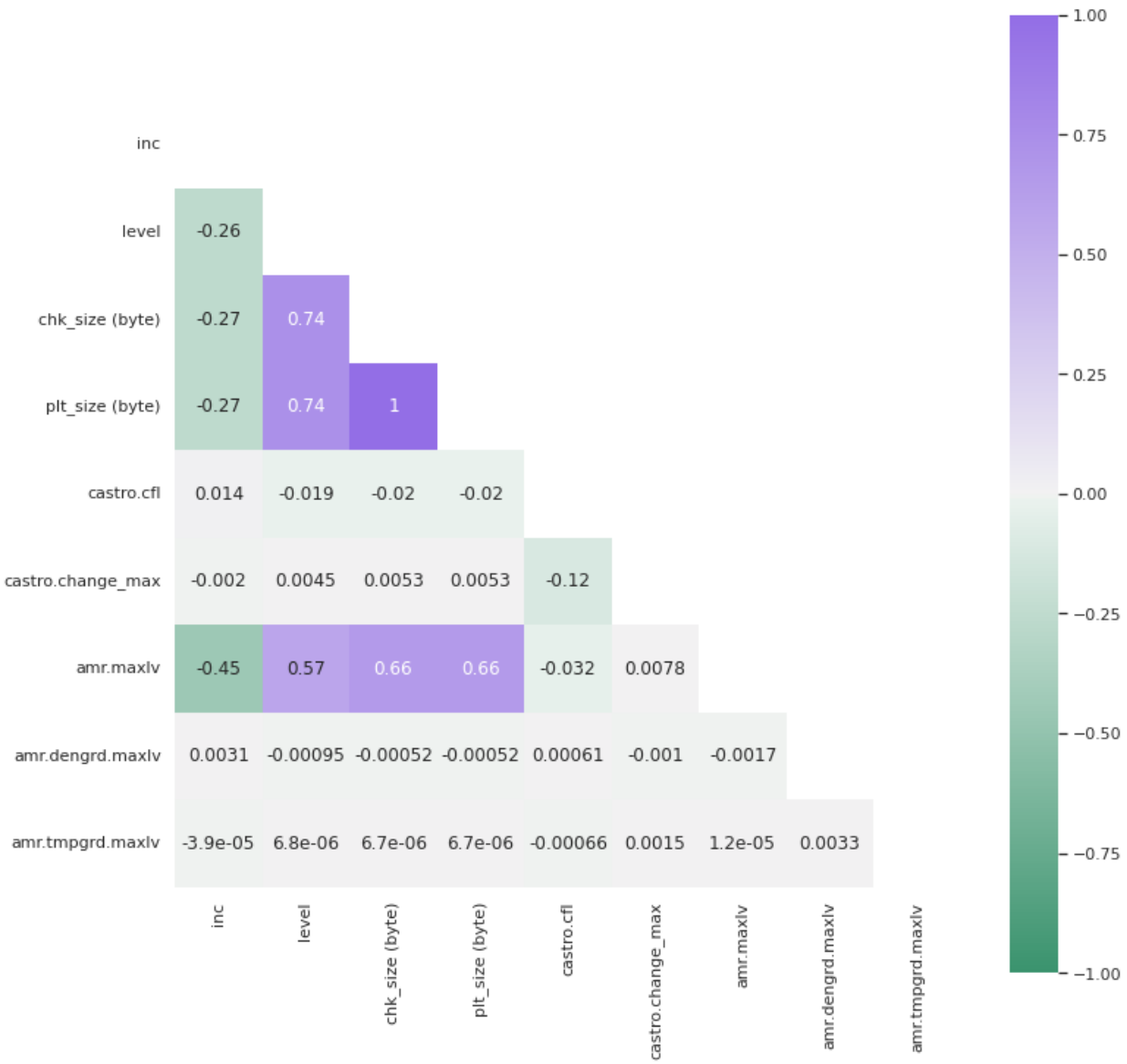


Figure 5: Correlation matrix from the parameter study of nova case with file size data.

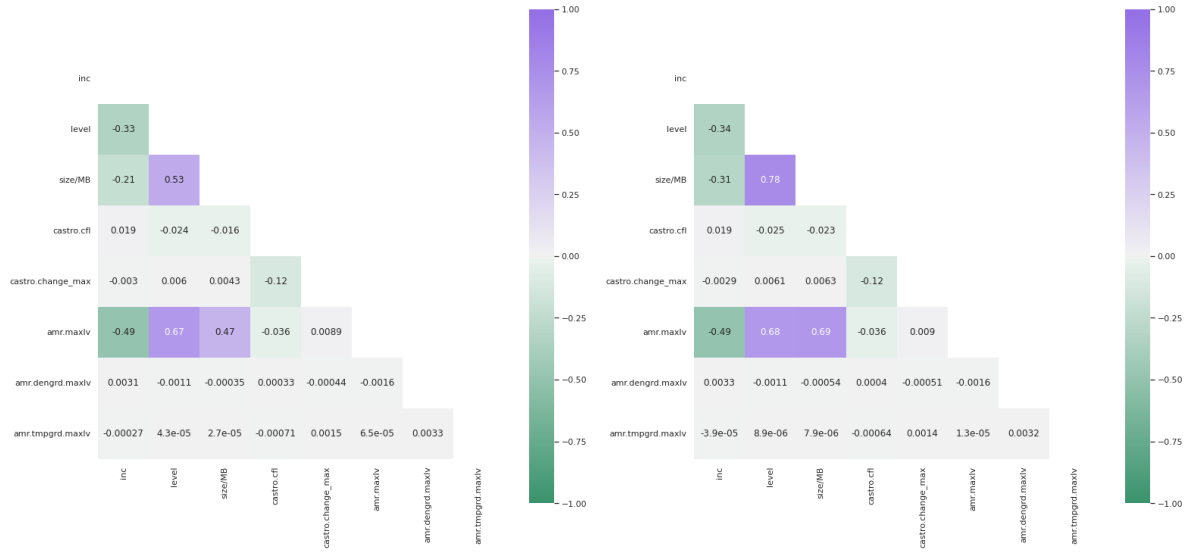


Figure 6: Correlation matrix from the parameter study of nova case (left: checkpoint file; right: plot file).

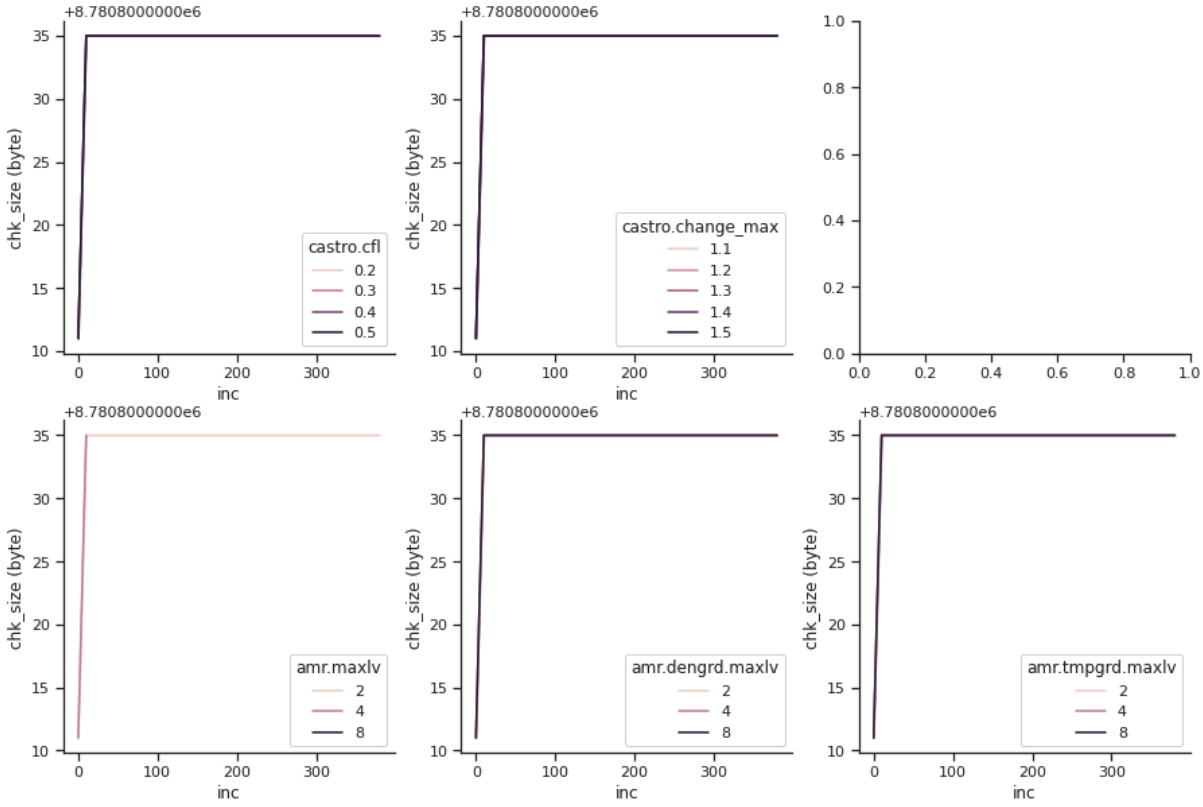


Figure 7: Checkpoint file size aggregated at level 0 plotted with respect to the increment. The color in each sub-plot is used to represent a parameter of interest.

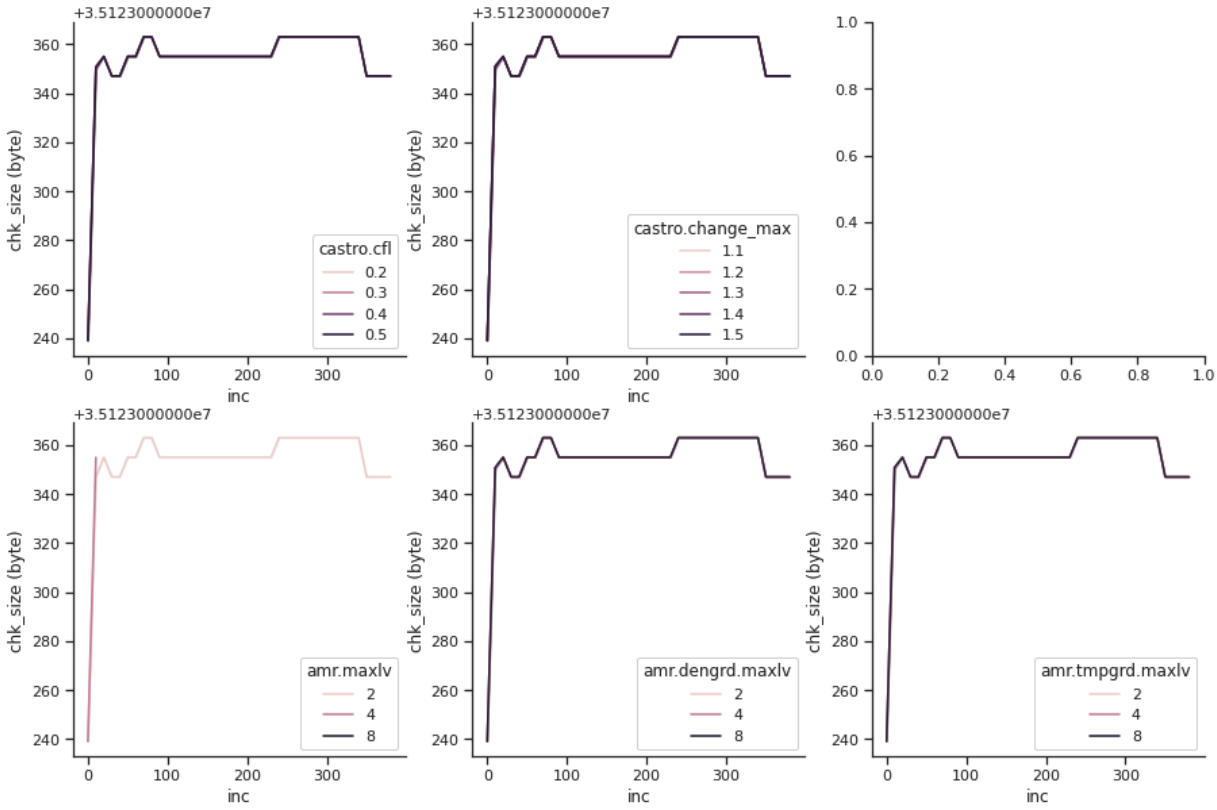


Figure 8: Checkpoint file size aggregated at level 1 plotted with respect to the increment. The color in each sub-plot is used to represent a parameter of interest.

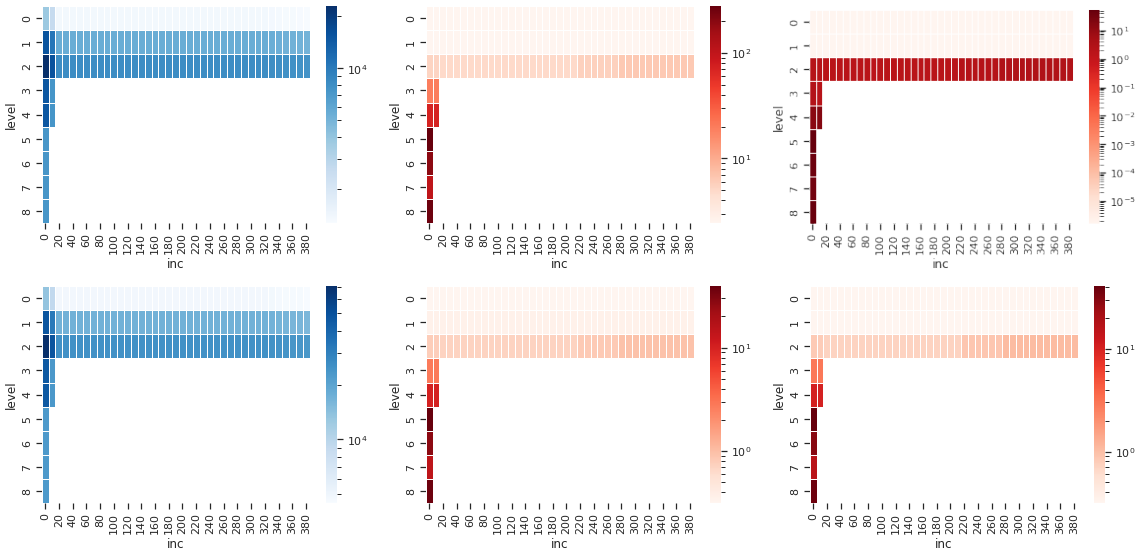


Figure 9: Visual representation of the lookup table for checkpoint file size (upper) and plot file size (lower) at difference level and increment for nova case (left: counts; middle: mean in MB; right: one standard deviation in MB).

able to examine for this reason. Future work in this area would allow a great number of models to be developed and tested, and this would ultimately improve the accuracy and efficacy of this approach. Additional work on the RIOPA.jl framework would also provide useful improvements in functionality. As the project was also limited to a single platform, replicating the experiments on other platforms, such as Frontier, would help improve the usefulness of the models.

2.4 Conclusion

In this study, we undertook a temporal stability analysis of the Sedov test from Castro, and an additional canvasing study to identify any other Castro physics simulations that were suitable for a parameter sensitivity study. From this we identified 21 suitable physics models, and selected the Kelvin-Heimholtz and the acoustic pulse cases from the hydro category, as well as the nova model from the science category. We then undertook a limited (by resources) sensitivity study of these cases, and an analysis of the results used to construct two representative models of I/O behavior.

In addition, we developed a new I/O workload generation framework called RIOPA.jl. This framework provides a configurable interface that allows I/O behaviour to be modeled and tested against real-world behavior. Although RIOPA.jl is already available as an open source project, we also plan to release it as part of the Proxy Application Suite.

3 Skel-IO

I/O Proxy Apps are important for exascale and post-exascale computing. As the increase of I/O performance lags behind that of computational performance, it is becoming increasingly essential to identify and test real-world application scenarios on new hardware being delivered. Additionally, middleware packages aimed at improving I/O performance must be tested with realistic situations that provide evidence that claimed improvements will translate to improved application performance.

At the same time, application I/O patterns are expanding beyond simply moving data to and from disk. Libraries such as ADIOS allow I/O to be done flexibly and controllably, providing options for aggregation and in-memory staging. Understanding the performance implications of all of these options requires more sophisticated benchmarking capabilities.

Many proxy applications are built by hand and thus incorporate a great deal of technical debt that becomes due should they need to adopt to situations beyond those envisioned by their original developers.

Skel-IO has been developed as an alternative to the individual, bespoke I/O Proxy. Skel-IO uses model-driven code generation techniques to produce a working proxy application from a set of high-level I/O model without the need for a user to write any code. To use Skel-IO, a user provides a model that describes the I/O pattern of an application of interest, and Skel-IO uses that model to instantiate a set of code templates that together produce a full application that mimics the described I/O behavior. Models can be hand-written, though that can be quite tedious, or may be extracted from other sources, such as file metadata (ADIOS) or application traces (TAU).

3.1 Requirements and Challenges

The various Skel-IO capabilities have thus far been developed as separate mechanisms and without a particularly friendly user interface, making them useful for just the handful of users most closely connected to their development. So, our current work has been focused on designing a better user interface for Skel-IO capable of collecting all the generative capabilities into one tool and guiding users through the process of using these tools.

This work is challenging for several reasons. The first is due to the current and anticipated variety of these model-driven capabilities. A primary design goal is that the user interface can gather the information that makes up the models in question, freeing the user from explicitly dealing with underlying json files, and from a need to directly interpret the expected/allowed schema of a particular model. With Skel-IO, we anticipate the need for many related but different models to address specific questions that researchers need to answer. Thus it is not sufficient to create a static UI that addresses current models. Instead, we need to be able to produce a customized UI component for any given set of models.

A second challenge stems from our aim to produce I/O benchmarks specifically for HPC systems. Such systems are often limited in terms of graphical user interface options. Users typically connect via ssh connections with limited windowing capabilities. X forwarding can generally be used, but is often slow, and is sometimes not available. Providing a remote UI capability is therefore critical for connecting the user to benchmarking activities happening on the HPC system.

3.2 Browser Based Prototype

Given the challenges, we set out to build a prototype system capable of supporting existing and future generators. Conceptually, this system leverages the model-driven code generation technique

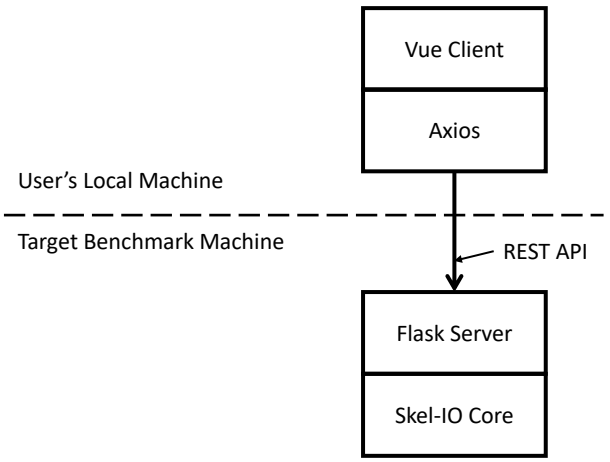


Figure 10: Current Skel-IO software stack

that has underpinned Skel-IO from the start. With this technique, a generator can be viewed as a pair of objects (S, T) where S is a model schema which defines a set of acceptable I/O models M that the generator will accept, and T is a collection of templates, each of which accepts m, where $m \in M$, and produces a code artifact. Together, the code artifacts produced by $t(m), \forall t \in T$ form a complete benchmark application that embodies the behavior described by m. By requiring generators to follow these structural conventions, we can generalize the interactions that our framework (and user interface) will need to have with current and future generators.

The needs of I/O benchmarking tools for HPC have led to a natural split between front-end (browser-based) components, and back-end services that interact via restful interfaces. Our software stack, illustrated in Figure 10 includes specific libraries that support this split. Front end, browser-based components are typically implemented in JavaScript, and we quickly settled on Node.js as the right tool to provide necessary libraries for building the front end. It quickly became apparent that a JavaScript framework, such as Angular, React, or Vue is an essential element of modern browser-based development. Having no particular experience with these, we selected Vue for its somewhat more gradual learning curve.

For the back end server we selected Flask, a python-based server-side microframework. The use of Flask to manage the restful interfaces was a natural first choice, as the existing Skel-IO components have also been developed using python, making connection to Skel-IO libraries fast and easy.

Overall, this effort has been focused on developing a prototype that exercises the selected libraries and provides proof-of-concept that this technology stack can address the challenges identified above. To address the present and future variety of I/O models, several capabilities have proven useful. First, Vue performs dynamic binding in a way that allows UI components to be created based on input data coming from Flask. This allows a new model, specified in the Skel-IO core, to be immediately supported with the right set of UI input components without the need to modify the UI code itself. Instead, a representation of the UI schema can be sent to the Vue client and "interpreted" to produce the needed text inputs, buttons, etc., to gather an instance of the model.

Our architecture, which has been inspired by Jupyter Notebook, addresses the second challenge directly, as the back-end components can be started on the target machine, and a URL generated that allows a connection to be started from a browser, in which the Vue client runs. The client uses Axios to pass data (mostly json) between the Vue client and the flask server. Thus our User Interface can support machines with no local User Interface capabilities.

3.3 Future Work

Our next steps will be to flesh out a complete implementation that fully supports the various Skel-IO generators that have been created so far. For this step, the generators will not need major changes, but for each generator we will need to provide a schema document that describes model requirements. Our investigations point to json schema as a reasonably mature technology that will suit this purpose. Providing a concrete schema will allow us to harden the UI generation and build a flexible and capable system. Some refactoring of templates will also be needed to make the result reporting from individual proxies more consistent. Our plan is to support building an application proxy that combines individual proxies that each focus on a particular use or aspect of an application. Ultimately it should be simple to put together a useful collection of models that provide a more complete view of application I/O than most existing proxies are able to provide.

4 Examining Proxy/Parent Application Kernel Similarity

Scientific applications consist of one or more kernels that collectively solve a certain scientific problem. By definition, a kernel should account for a large percentage of an application's total execution time. Kernels may be computational or communication bound or both. It is unusual for an application to have a single kernel. Rather scientific applications usually comprise multiple kernels (3–5) that accounts for varying percentages of execution time.

In prior work [8], we quantified the fidelity of proxy apps by collecting CPU hardware performance counters from full execution runs of proxies and their respective parents. We found that some proxies diverge from their parents and we wanted to identify from where that divergence was originating. Therefore, in this work we first identify kernels in proxy/parent applications, then we quantify behavioral similarity between these kernels.

4.1 Methodology

In this section, we present our methodology to determine the similarity between the kernels of proxy and parent applications regarding computational node performance to understand if proxies are fundamentally a good representation of their parent applications. We do not focus on communication in this work, but we will extend this study in the future to understand the kernel similarity of proxy/parent pairs regarding communication.

We look at cosine similarity between the application pairs using a smaller set (compared to [8]) of available and reliable performance events as the application signature. This set was discovered from our previous work that identifies the most important events for quantifying similarity of applications based on each primary system component (e.g., memory, branch predictor, cache). Performance event/component categories/groups include:

- Dispatch pipeline
- Execution pipeline
- Instruction cache
- L1 data cache
- L3 cache
- Memory off core requests
- Memory pipeline
- Retirement pipeline

The similarity technique we use in this work is cosine similarity, which is explained in [8] that is available on our project website. All applications are compiled with Intel 20.0 and they are executed with MPI only on 128 ranks. This scale is chosen because it is large enough to observe important communication patterns but not so large that jobs are forced to wait for days in a scheduling queue to execute.

We run each application with each input five times to account for performance variation and any variation in the performance counters. Data is collected throughout execution from each MPI rank, then 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 (3) normalize the event counts by cycles executed. This results in a vector of approximate length 500 for each application, which we then use as input to the cosine similarity algorithm (described in FY21 Quantitative Assessment Report on our project website).

We use HPCToolkit to collect the data on an Intel Skylake-based platform with an Omnipath

interconnect. In our previous work, we used the Lightweight Distributed Metric Service (LDMS) monitoring tool to collect the hardware performance data using the PAPI sampler for the entire application execution. LDMS samples the performance counters from each MPI rank every second and passes the data for analysis. It is a handy tool because it can also link the data to other performance data collected from the application or the system. In this work, we used HPCToolKit to gather the performance counters from the application kernels instead of LDMS. We needed a granularity sampling of less than 1 second to capture kernels that are called millions of times and last for a short period (<10th of a second). We understand that LDMS is a monitoring tool rather than a debugging tool like HPCToolKit so the capabilities of collecting such small granularity is not possible.

4.2 Applications and Key Kernels

Based on our prior work in quantifying similarity between proxy/parent applications, for this study we chose proxy/parent pairs that covered a range of similarity - some are highly similar (i.e., SW4lite/SW4) and some not very similar (miniQMC/QMCPACK). The applications we chose are ExaMiniMD/LAMMPS, MiniVite/Vite, SW4lite/SW4, and miniQMC/QMCPack.

In the following sub-sections, we provide the kernel execution times for each proxy/parent pair in a table. The reader may see communication time at the end of the timing table. Note that this is not a communication kernel but is the time spent in the MPI communication library. HPCToolKit can not link the time spent in the communication to actual kernels. In the future, we will further investigate this issue with the HPCToolKit team.

4.2.1 ExaMiniMD/LAMMPS

LAMMPS is a classical molecular dynamics code, with particles ranging from a single atom to a large composition of material. It implements mostly short-range solvers, but does include some methods for long-range particle interactions. ExaMiniMD, which is a proxy for LAMMPS, implements limited types of interactions, and only short-range ones.The key kernesl for LAMMPS and ExaMiniMD are illustrated in table 3.

Kernel	ExaMiniMD	LAMMPS
compute_zi	39.8	11.2
compute_deidrj	17.7	11.6
compute_duidrj	21.9	12.2
compute_ui	21.2	11.6
compute_yi	N/A	32.7
Communication	N/A	6.1
Total %	95.1	95.2

Table 3: Kernel Function Timing, ExaMiniMD and LAMMPS

LAMMPS can run different models, but we chose the SNAP force model calculations to match the ExaMiniMD workload. Table 3 shows the main kernel functions for both pairs, where the functions *compute_ui* and *compute_zi* are two kernels that are responsible for bispectrum components that characterize the local neighborhood of each atom in a general way. *compute_duidrj* and *compute_deidrj* are the derivative calculators for the bispectrum. The last kernel is *compute_yi* that exists in LAMMPS only, and it is another kernel that participates in the bispectrum calculation.

We profiled both pair applications, and we found that the timing of *compute_yi* dominates the runtime execution in LAMMPS, where *compute_zi* is dominant in ExaMiniMD. The rest of the functions show closer execution runtimes than the later two functions in both application pairs.

4.2.2 MiniVite/Vite

Vite is an implementation of the Louvain method for (undirected) graph clustering or community detection. MiniVite is a proxy application for Vite that implements a single phase of the Louvain method in distributed memory for community detection. The key kernels for Vite and MiniVite are illustrated in table 4.

Kernel	MiniVite	Vite
fillRemoteCommunities	39.2	39.1
updateRemoteCommunities	10.3	21.1
distGetMaxIndex	18.7	15.9
distBuildLocalMapCounter	10.4	13.6
distbuildNextLevelGraph	N/A	4.2
Communication	19.4	5.8
Total %	98.1	98.6

Table 4: Kernel Function Timing, MiniVite and Vite

MiniVite and Vite use the Louvain algorithm to optimize modularity by partitioning the graph vertices in communities (clusters) wisely. In our profiling, we found that the largest kernel execution time reported was *fillRemoteCommunities*. The rest of the functions are divided by the time over the rest of the execution. Vite has slightly larger kernel times in the short-time functions compared to MiniVite. Also, Vite has an extra function that does not exist in MiniVite, *distbuildNextLevelGraph* that we suspect is doing the moving to the next level graph as Vite is a complete simulation application that traverses multiple graph levels.

4.2.3 SW4lite/SW4

SW4 is a geodynamics code that solves 3D seismic wave equations with local mesh refinement. SW4lite is a scaled-down version of SW4 that has limited seismic modeling capabilities, but does solve the elastic wave equation and uses some of the same numerical kernels as those implemented in SW4. The key kernels for SW4 and SW4lite are illustrated in table 5.

Kernel	SW4lite	SW4
evalRHS	59.1	48.5
addSuperGridDamping	11.7	12.1
evalDpDmInTime	3.4	5.5
evalPredictor	3.4	5.5
evalCorrector	2.2	3.6
Communication	17.9	20.8
Total %	97.7	95.9

Table 5: Kernel Function Timing, SW4lite and SW4

SW4lite is developed using the same code base as SW4, so we found that most functions report close execution time percentages, especially the most significant function, *evalRHS*.

4.2.4 MiniQMC/QMCPACK

QMCPACK is a quantum Monte Carlo package for computing the electronic structure of atoms. MiniQMC covers QMCPACK's essential computational kernels. The computational themes of miniQMC and QMCPACK are particle methods, dense and sparse linear algebra, and Monte Carlo methods. The key kernels for QMCPACK and MiniQMC are illustrated in table 6.

Kernel	miniQMC	QMCPACK
Determinant	73.9	71.6
Single-Particle Orbital (SPO)	11.5	11.0
Distance	12.8	12.2
Two Body Jastrow	1.4	4.5
Total %	99.6	99.3

Table 6: Kernel Function Timing, MiniQMC and QMCPACK

Kernel	miniQMC	% Time	QMCPACK	%Time
Determinant	DiracDeterminant::acceptMove	57.8	DiracDeterminantBase::acceptMove	49.3
	DiracDeterminant::ratioGrad	6.2	DiracDeterminantBase::ratioGrad	7.7
	MKL	5.2	DiracDeterminantBase::ratio	2.3
	DiracDeterminant::ratio	4.7	MKL	10.0
			DiracDeterminantBase::evaluateLog	2.3
Single-Particle Orbital (SPO)	einspline_spo::MultiBspline::evaluate_vgh	9.3	SPOSetBuilderFactory::createSPOSet	11.0
	einspline_spo::MultiBspline::evaluate_v	1.2		
	einspline_spo::MultiBspline::set	1.0		
Distance	ParticleSet::makeMoveAndCheck	4.8	ParticleSet::makeMoveOnSphere	11.2
	ParticleSet::setActive	4.8	ParticleSet::makeMoveAndCheck	1.0
	DistanceTableAA::makeMoveOnSphere	3.2		
Two Body	TwoBodyJastrowOrbital::BsplineFunctor::acceptMove	1.4	TwoBodyJastrowOrbital::BsplineFunctor::ratio	4.0
			OneBodyJastrowOrbital::BsplineFunctor::ratioGrad	0.5

Table 7: Kernel Function Profiles

MiniQMC and QMCPACK have several function kernels. Table 7 shows the timing of each function and to which kernel they belong. The four key kernels in both miniQMC and QMCPACK are the following:

1. Determinant update (inverse update): This kernel uses the Sherman-Morrison algorithm to compute the Slater determinant. The Slater determinant provides an accurate approximation of the wave functions being solved. This kernel relies on BLAS2 functions and is the source of the N^3 scaling in the application.
2. Splines: This kernel is invoked for every potential electron move. It computes the 3D spline value, the gradient (4x4x4xN stencil), and the Laplacian of electron orbitals. This kernel is memory bandwidth limited. Its large memory footprint makes data layout and memory hierarchy considerations critical to performance.
3. Jastrow factors (1, 2, and 3-body): The Jastrow factor represents the electronic correlation beyond the mean-field level in QMC simulations. Correlations are decomposed into 1, 2, and 3-body terms (electron-nucleus, electron-electron, and electron-electron-ion, respectively). This is a computationally intensive kernel.
4. Distance tables: These tables hold distances between electrons and electrons and atoms as matrices of all pairs of particle distances. Two tables are maintained - one for electron-electron pairs and one for electron-ion pairs. Minimum image and periodic boundary conditions are applied. Tables are updated after every successful MC electron move. Algorithms implementing this kernel have a strong sensitivity to data layout.

For more details about the kernels please refer to our report [7].

4.3 Results and Analysis

We use two tools to profile the applications and gather the function timings and HW counter performance data. We use GProf to collect the timing profile of the functions, generate the call tree, and identify the main kernels shared between the parent and the proxy application. We then use HPCToolKit to collect the hardware counter performance metric groups per function. Below we will show the result of cosine similarity applied to the application pair kernels.

4.3.1 ExaMiniMD/LAMMPS

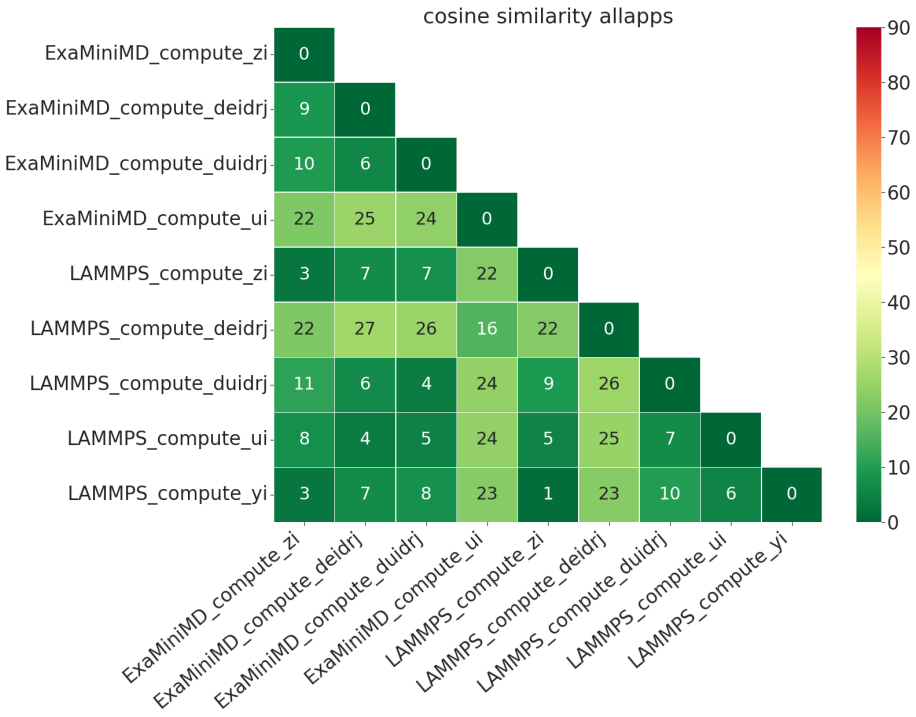
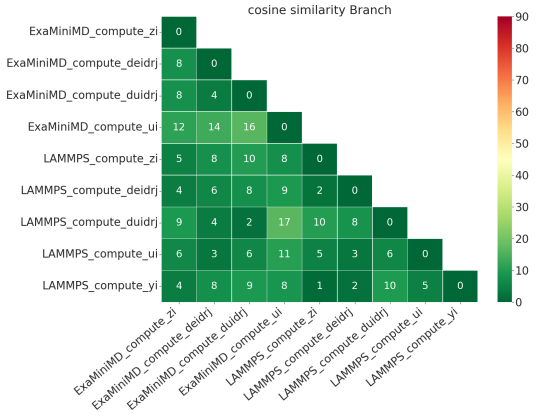


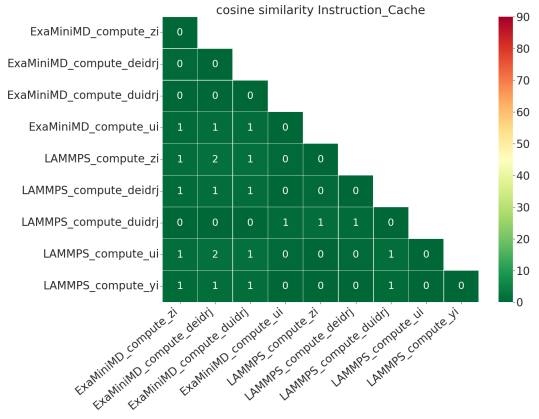
Figure 11: Cosine Similarity Matrix for ExaMiniMD/LAMMPS Kernels for All Counter Groups

Figure 11 shows the cosine similarity matrix for ExaMiniMD and LAMMPS. Here we see that the textitcompute_zi kernel behavior is very similar in both applications, although it dominates the time for ExaMiniMD, unlike LAMMPS. *compute_deidrj* and *compute_ui* kernel behavior are a bit divergent in both applications, similar to the difference in the timings. As ExaMiniMD has no *compute_yi* kernel that is timing dominant in LAMMPS, we think this kernel behavior causes the most divergence in the overall application performance.

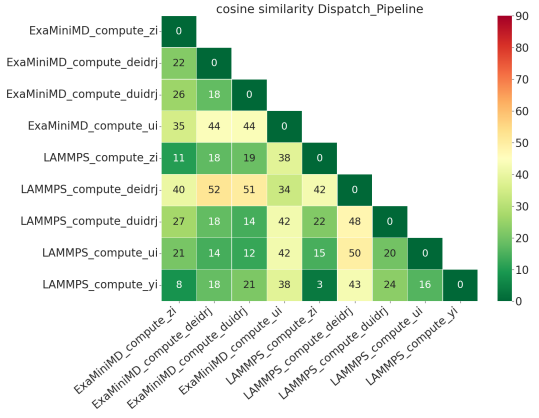
Figures 12 and 13 show the kernel similarity for the 9 performance subgroups. The similarity matrix of each group aids in determining which system behavior most influenced the observed divergence. From the subgroups matrices, we see there is similarity in cache and instruction-mix related performance counters. However, the memory pipeline presents the most divergence. Cosine similarity divergence within these subgroups is some evidence as to what causes the divergence, but examining these codes in addition to further experimentation can help identify the root cause.



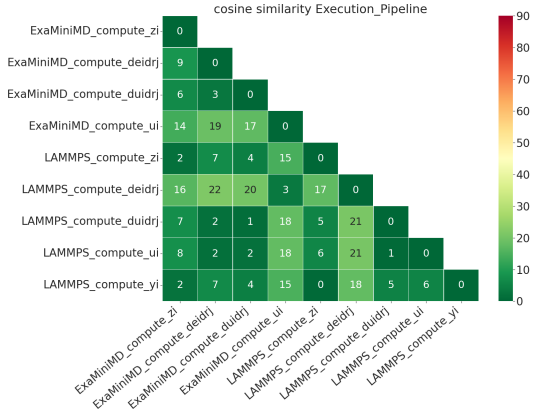
(A) Branch



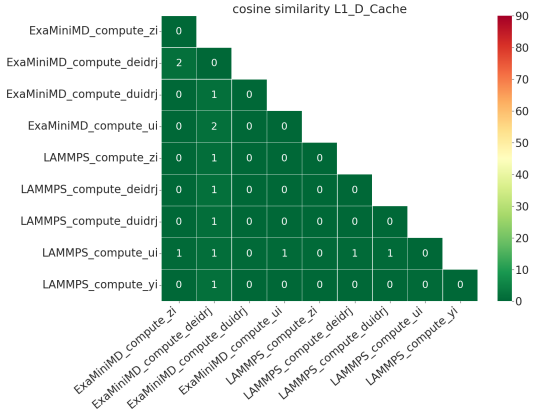
(B) Instruction Cache



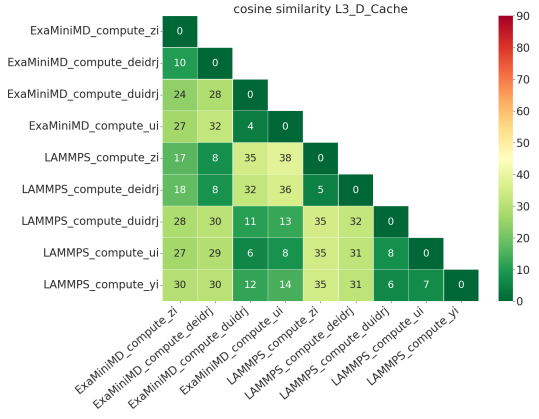
(C) Dispatch Pipeline



(D) Execution Pipeline

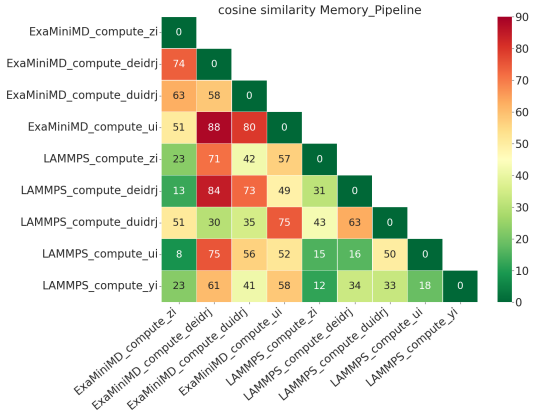


(E) L1 Data Cache

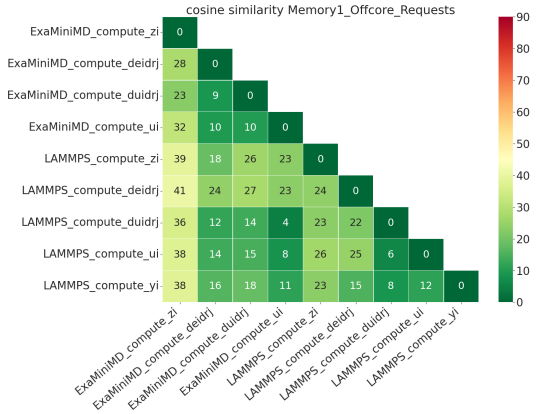


(F) L3 Cache

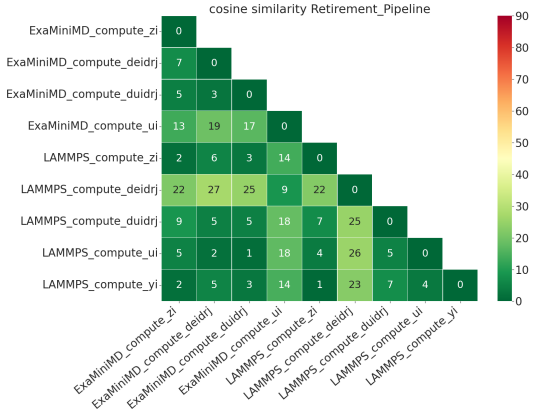
Figure 12: Cosine Similarity Subgroups for ExaMiniMD/LAMMPS kernel



(G) Memory Pipeline



(H) Memory Offcore Requests



(G) Retirement Pipeline

Figure 13: More Cosine Similarity Subgroups for ExaMiniMD/LAMMPS kernel

4.3.2 MiniVite/Vite

MiniVite and Vite show some divergence in our previous work, where we compare the performance counters that were collected from the whole run. However, the similarity between the kernels is actually quite good. This leads us to believe that the divergence that we see for the whole execution is probably because miniVite only does a single phase of the Louvain algorithm.

Figure 14 shows the largest divergence (which is actually small) in the *fillRemoteCommunities* kernel. However, the timing data shows these two kernels have almost identical execution time percentages. The *updateRemoteCommunities* kernel has the largest timing difference between miniVite/Vite, but it shows very similar behavior in the cosine matrix.

Although the kernel divergence between miniVite/Vite is quite small, we still want to see where that divergence may originate, particularly for *fillRemoteCommunities*. Figures 15 and 16 show that the execution and retirement pipelines, respectively, have the most variation for the *fillRemoteCommunities* kernel (although it is small).

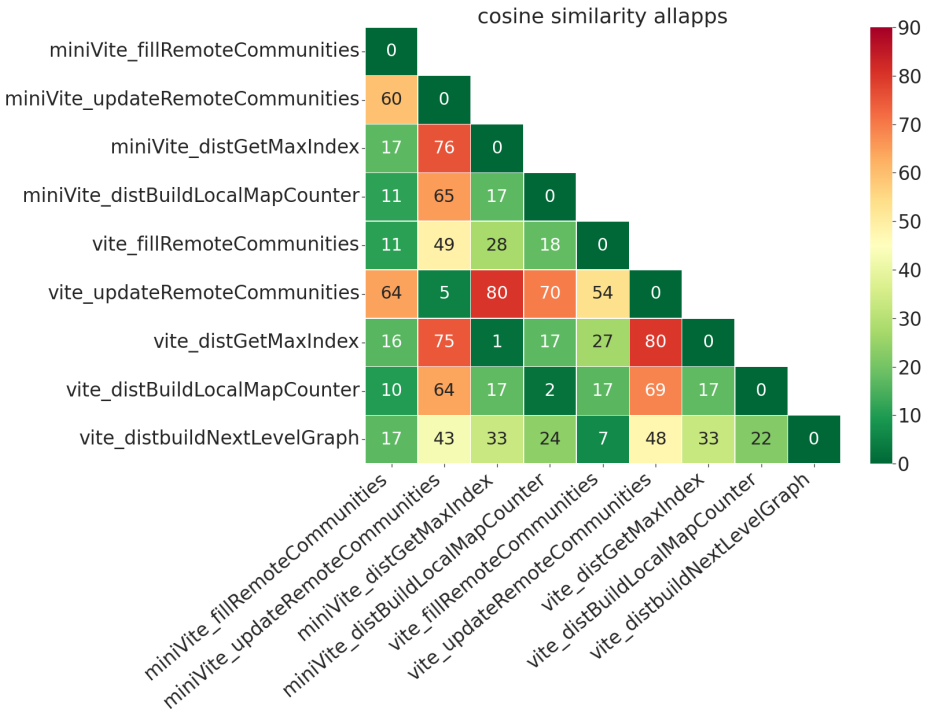
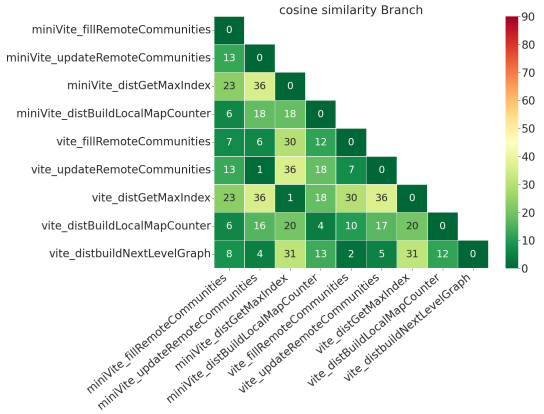
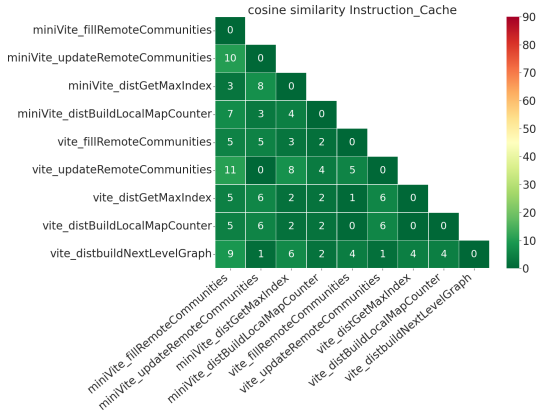


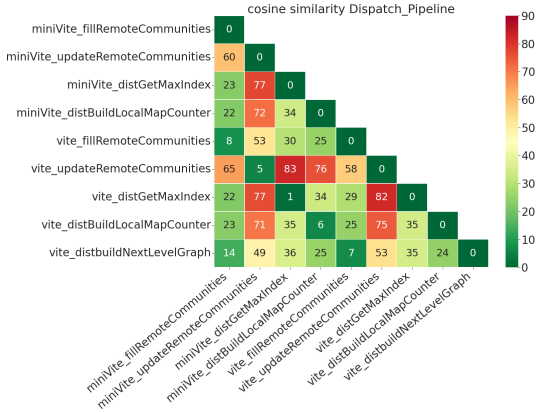
Figure 14: Cosine Similarity Matrix for MiniVite/Vite Kernels for All Counter Groups



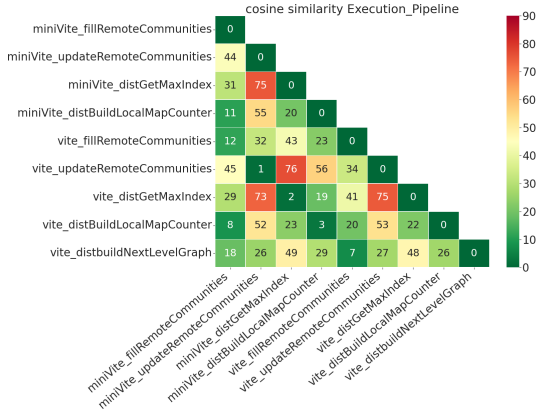
(A) Branch



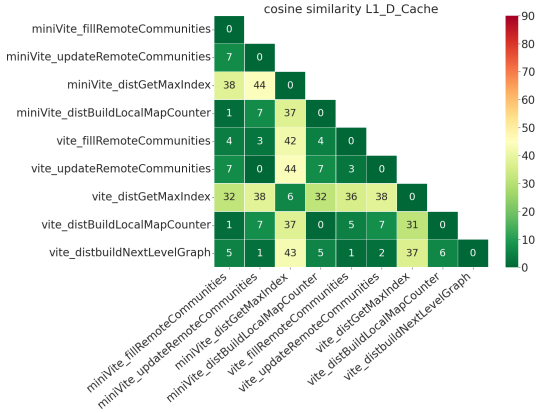
(B) Instruction Cache



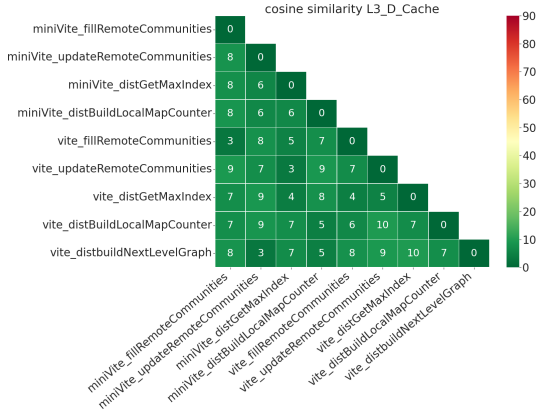
(C) Dispatch Pipeline



(D) Execution Pipeline

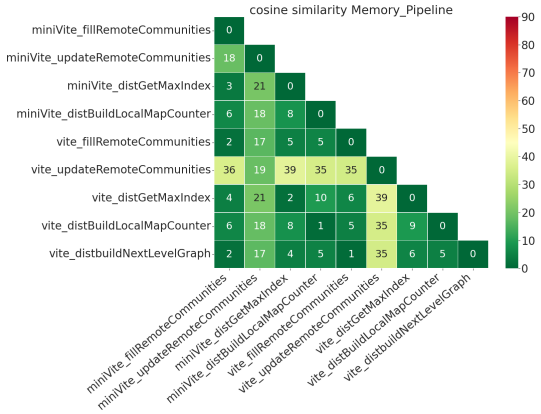


(E) L1 Data Cache

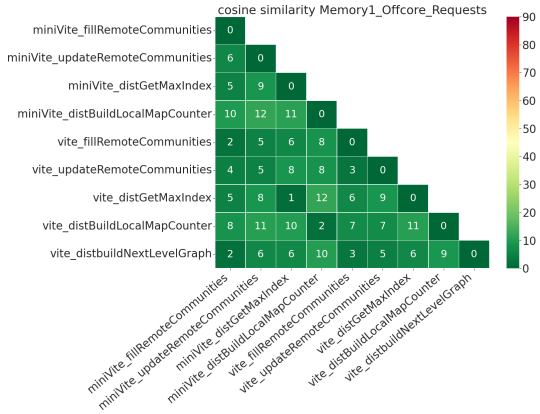


(F) L3 Cache

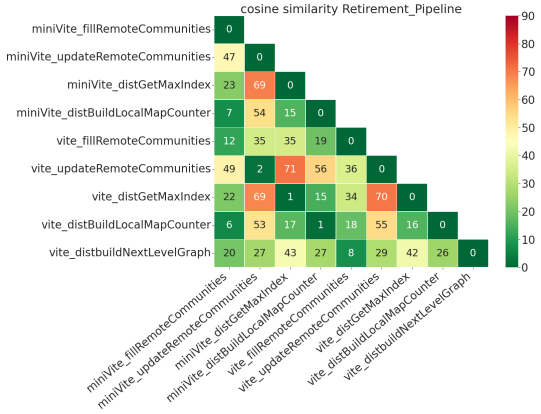
Figure 15: Subgroups Cosine Similarity for MiniVite/Vite kernel



(G) Memory Pipeline



(H) Memory Offcore Requests



(G) Retirement Pipeline

Figure 16: Subgroups Cosine Similarity for MiniVite/Vite kernel

4.3.3 SW4lite/SW4

SW4 and SW4lite show the highest similarity over the whole execution because the proxy shares a large portion of the same code base with its parent. SW4lite is lite because it does not support some of the complex seismic/plate interactions that SW4 supports. Interestingly, Figure 17 shows that the pair kernels *evalDpDmInTime*, *evalPredictor*, and *evalCorrector* slightly diverge, although they are still green. We notice that the kernel *evalRHS* in both applications accounts for the largest percentage of execution time, which is around 50%. This kernel shows very similar behavior in SW4 and SW4lite, which is likely why the overall execution of this proxy/parent pair is very similar and the slight divergence in the less important kernels does not largely affect this.

Figures 18 and 19 shows the similarity matrices for the performance counter subgroups. Notice that SW and SWlite branch similarity matrix shows some divergence which could indicate that the branching for both codes executes differently. The figure also shows that a relatively large divergence appears in the L1 data cache behavior for the *evalPredictor* and *evalCorrector* kernels, but is largest for the *evalDpDmInTime* kernel. Just looking at all of these subgroup similarity matrices, it seems that there is something different in the *evalDpDmInTime* kernel of SW4lite and SW4. Further code review is needed in the future to understand how the divergent behavior in branching and L1 data cache relates to the code implementation.

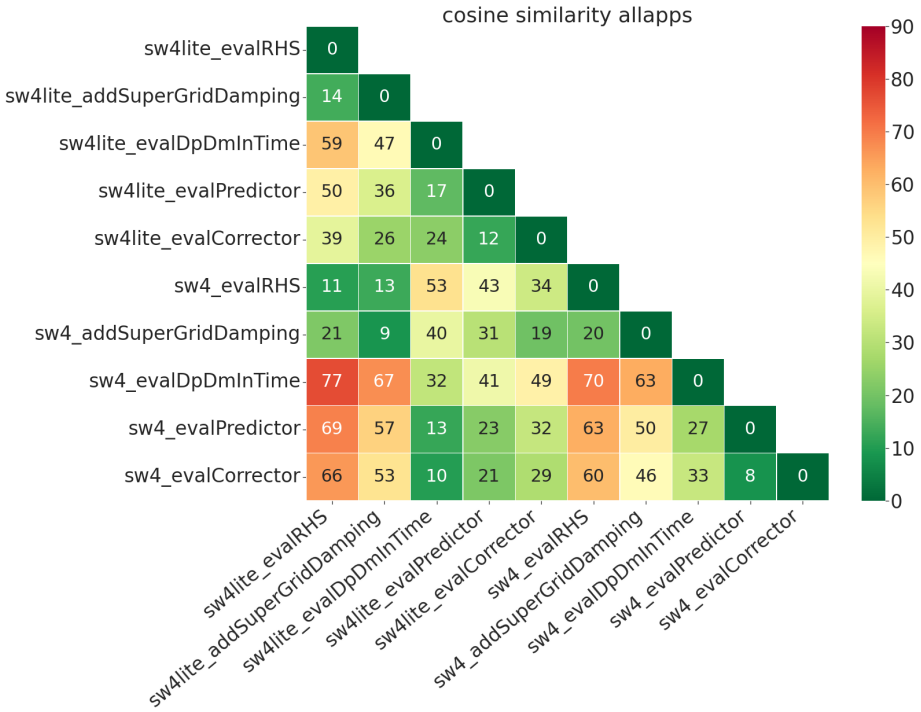
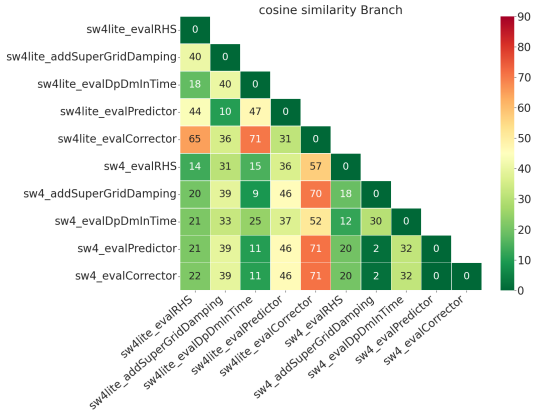
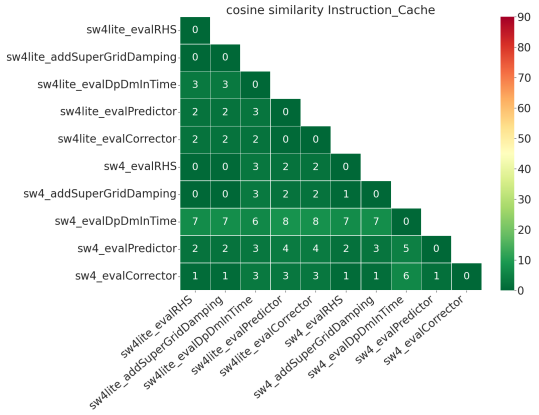


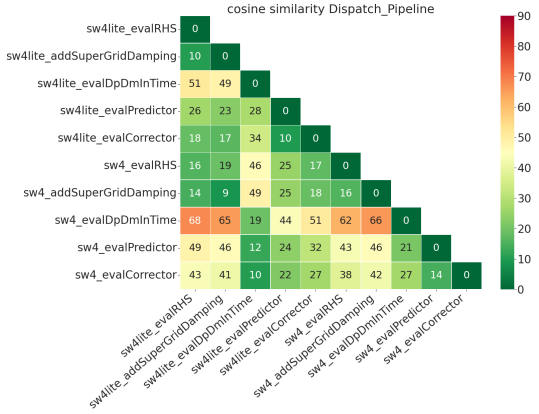
Figure 17: Cosine Similarity Matrix for SW4lite/SW4 Kernels for All Counter Groups



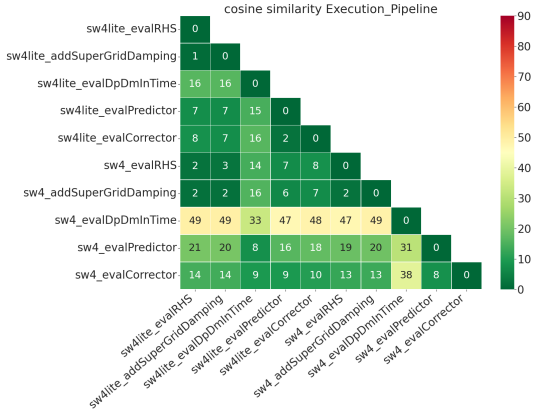
(A) Branch



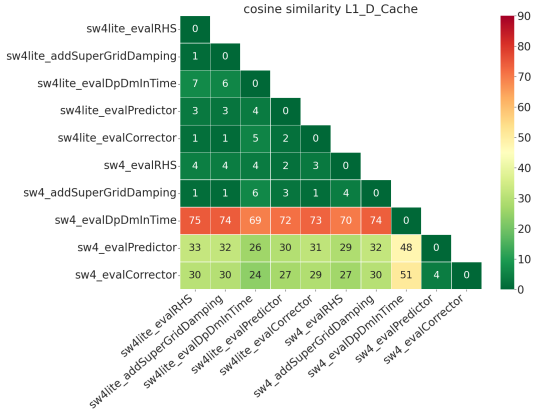
(B) Instruction Cache



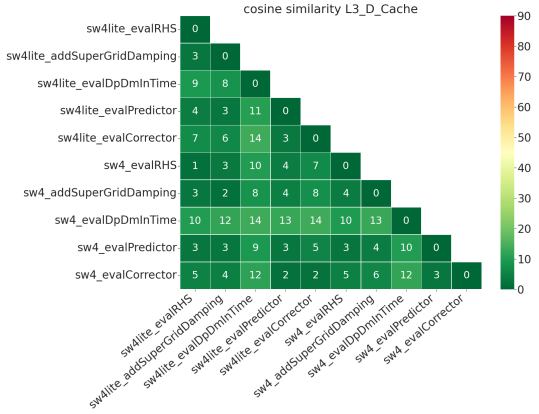
(C) Dispatch Pipeline



(D) Execution Pipeline

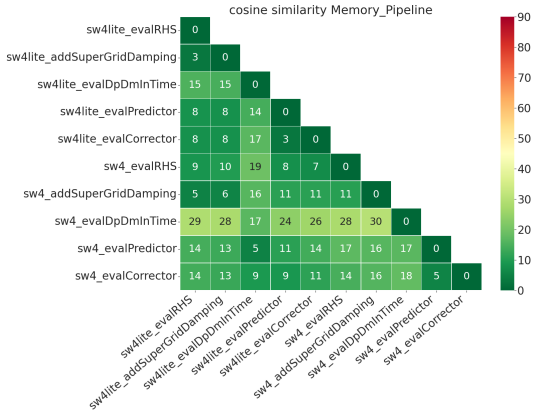


(E) L1 Data Cache

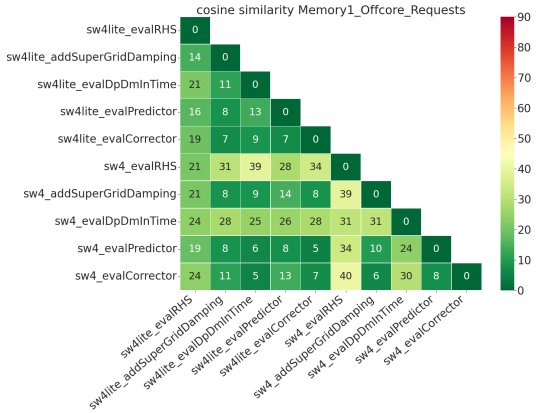


(F) L3 Cache

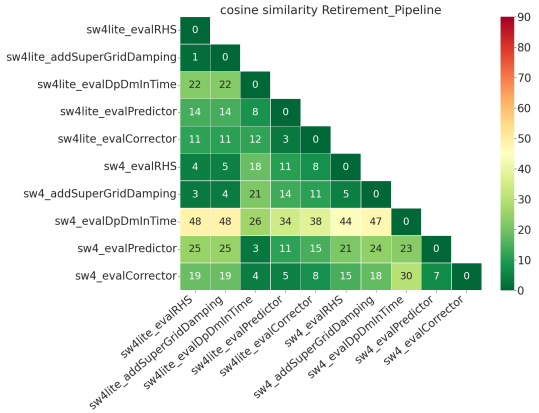
Figure 18: Subgroups Cosine Similarity for SW4lite/SW4 kernel



(G) Memory Pipeline



(H) Memory Offcore Requests



(G) Retirement Pipeline

Figure 19: Subgroups Cosine Similarity for SW4lite/SW4 kernel

4.3.4 MiniQMC/QMCPACK

We studied the similarity of miniQMC/QMCPACK kernels in prior work [4], including the behavior of the kernels. Here we have collected additional data and done more detailed analysis.

Figure 20 shows the cosine similarity matrices for all performance event groups for all functions. Note that the functions in the Determinant kernel (Table 7) are very similar in name; in the other kernels, names of functions between the proxy and parent are different. Therefore, we can compare similarity of the functions in the Determinant kernel, but then must compare the proxy/parent pair by kernel, averaging the similarity of the functions that make up that kernel.

The result demonstrates that the *DiracDeterminant::acceptMove* function in the Determinant kernel has similar behavior for both MiniQMC and QMCPACK, and it shows that it accounts for most of the overall execution time in both applications. The *DiracDeterminant::ratio* and *DiracDeterminant::ratioGrad*, also part of the Determinant kernel, are the most divergent in our similarity matrix. However, these account for a small portion of the kernel and application execution runtime.

The average cosine similarity of all of the functions in the Determinant kernel is about 36 degrees. The SPO kernel accounts for around 11% of the execution time of both applications, and their functions have an average cosine similarity of 53.33 degrees. The Distance kernel accounts for around 13% of the execution time of each application and has an average cosine similarity of 37.5 degrees.

Figures 21 and 22 show the cosine similarity matrices for the performance counter subgroups. Considering only the functions that make up the Determinant kernel, we see the highest divergence in the L1 data cache and in portions of the execution pipeline. Instruction cache behavior (and instruction mix, which is not shown because for all proxy/pairs and all inputs, it is all dark green, meaning highly similar) seems to be highly similar for all functions. This is the case for all of the applications that we examine in this work.

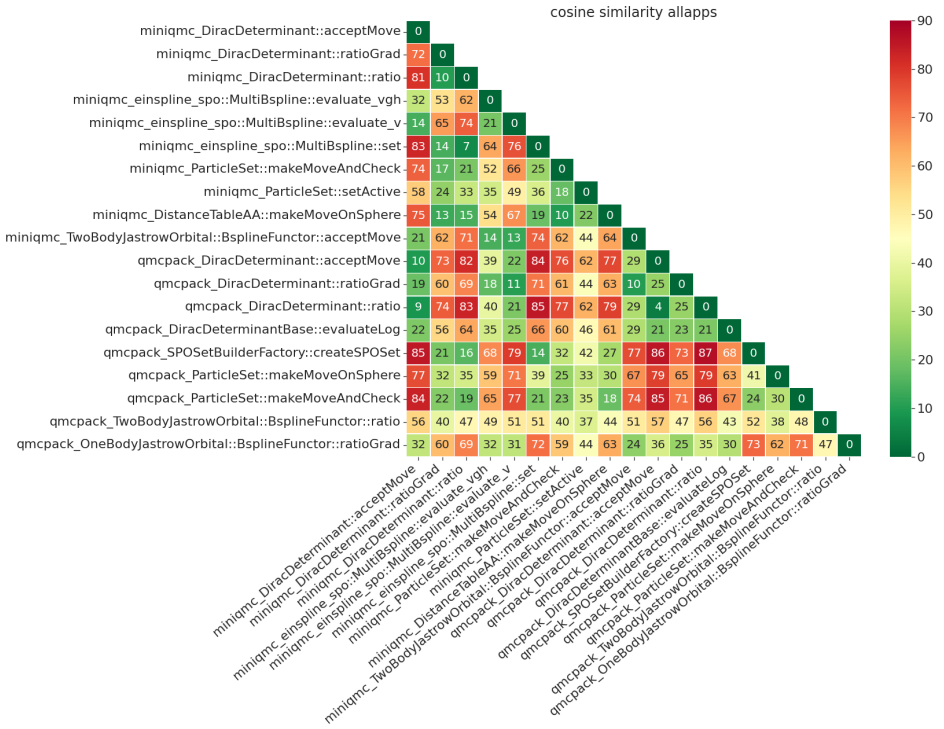
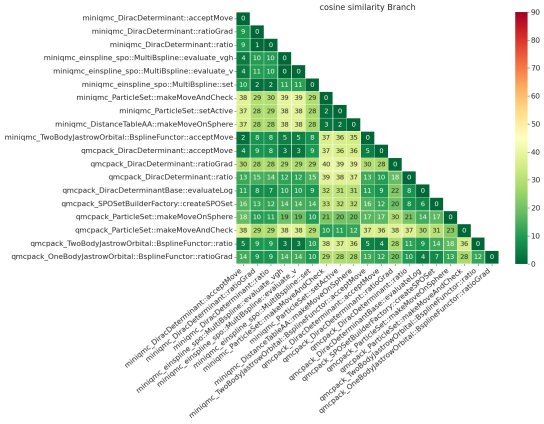
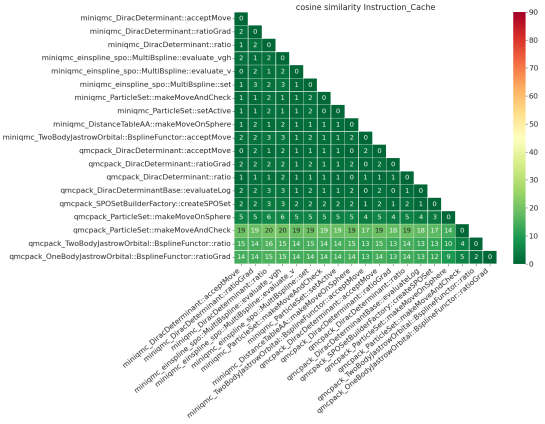


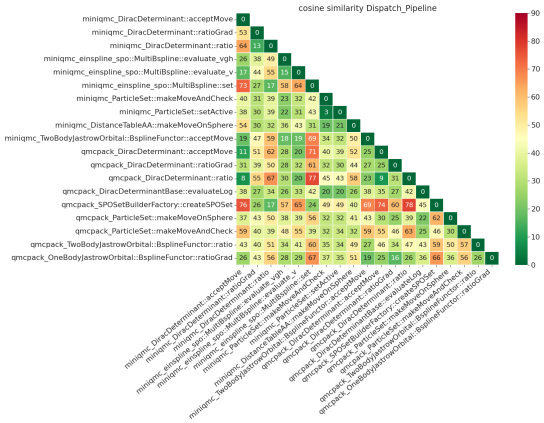
Figure 20: Cosine Similarity Matrix for MiniQMC/QMCPACK Kernels



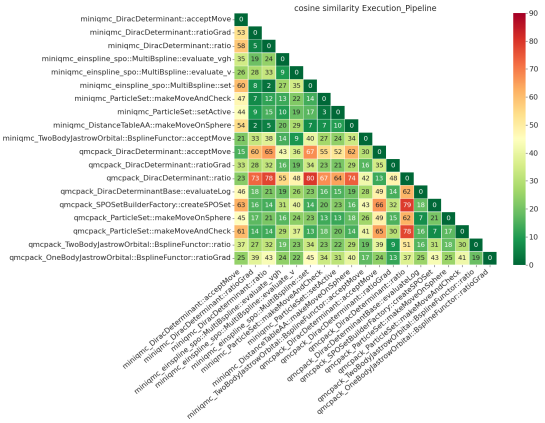
(A) Branch



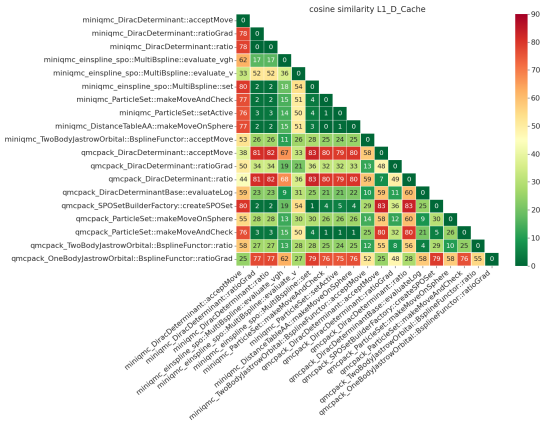
(B) Instruction Cache



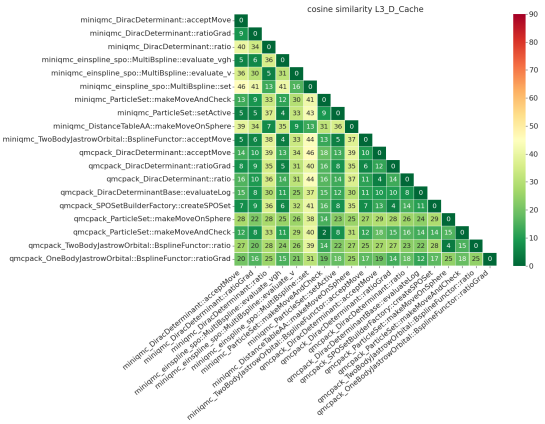
(C) Dispatch Pipeline



(D) Execution Pipeline

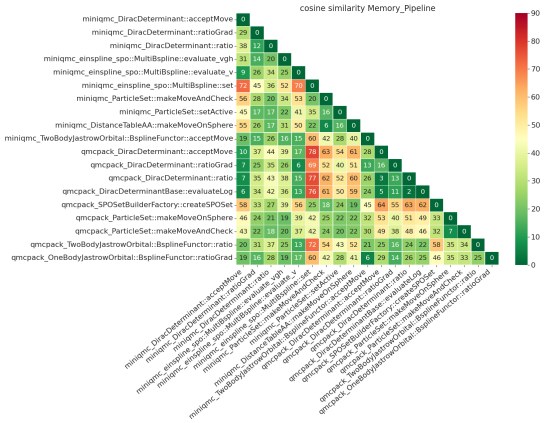


(E) L1 Data Cache

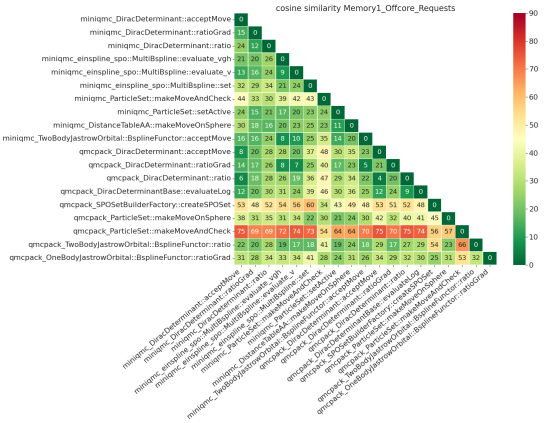


(F) L3 Cache

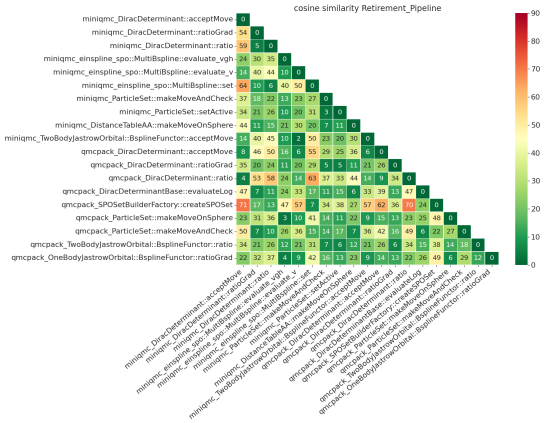
Figure 21: Subgroups Cosine Similarity for MiniQMC/QMCPACK kernel



(G) Memory Pipeline



(H) Memory Offcore Requests



(G) Retirement Pipeline

Figure 22: Subgroups Cosine Similarity for MiniQMC/QMCPACK kernel

4.4 Conclusion

We investigated four proxy/parent pairs to determine their kernel similarity using a smaller set of performance counters that was chosen, in previous work, to be the top features we can use to determine similarity. We apply cosine similarity to determine the fidelity of the proxy compared to its respective parent. We looked at the similarity in kernels for these proxy/parent pairs: ExaMiniMD/LAMMPS, MiniVite/Vite, SW4lite/SW4, and MiniQMC/QMCPACK. For the most part, our results show that primary kernels of the proxy/parent pairs demonstrate good similarity. However, we do see some divergence in less important kernels (i.e., kernels that account for a small percentage of execution time compared to the primary kernel).

Our results show that the kernels of ExaMiniMD and LAMMPS show high similarity in general. Their primary kernel diverges in the memory pipeline, but is generally similar for the other performance groups.

MiniVite and Vite show good similarity in all kernels. The largest divergence in kernels, although small, is observed in the execution and retirement pipeline behavior.

The primary kernel, which accounts for about 50% of the total execution time, shows high similarity between SW4lite and SW4. Slight divergence is observed in less important kernels that do not affect the overall high similarity because of their relatively small execution time.

The last pair of our study, MiniQMC, and QMCPACK, were already studied in previous work in detail, but did not apply cosine similarity to all of the kernels. For miniQMC and QMCPACK, the primary kernel, Determinant, which accounts for about 70% of the execution time, has an average (over all Determinant functions) cosine similarity of about 36 degrees. The SPO kernel accounts for around 11% of the execution time of both applications, and their functions have an average cosine similarity of 53.33 degrees. The Distance kernel accounts for around 13% of the execution time of each application and has an average cosine similarity of 37.5 degrees. The result was not surprising as the early study indicates that QMCPACK needed a more robust proxy, and modifications must be done to MiniQMC better capture the behavior of QMCPACK.

5 Using Cosine Similarity to Understand Performance Changes for Application Problem Change

Over the duration of this project, we have developed techniques based on similarity algorithms to quantify performance differences between proxy apps and their respective parents. In this work, we apply these same techniques to understand if changing the input problem for a proxy or parent changes the performance/behavior of these applications and their similarity. Below we describe the methodology we use to collect the data to make these comparisons and the we present the results from these experiments.

5.1 Experimental Methodology

We perform these experiments on an Intel Skylake-based system with an Omnipath interconnect. All applications are compiled with the Intel 20.0 compiler. Applications were run with MPI only on 4 nodes, with a total of 144 ranks.

5.1.1 Application Suite

The proxy/parent applications that we use in this work are the following:

- ExaMiniMD/LAMMPS: LAMMPS is a classical molecular dynamics code, with particles ranging from a single atom to a large composition of material. It implements mostly short-range solvers, but does include some methods for long-range particle interactions. ExaMiniMD, which is a proxy for LAMMPS, implements limited types of interactions, and only short-range ones. We use the Snap potential for in this work.
- PICSARlite/PICSAR: PICSAR is Particle-In-Cell solver, while its proxy application, PICSARlite, is a subset of the actual codebase.
- SW4lite/SW4: SW4 is a geodynamics code that solves 3D seismc wave equations with local mesh refinement. SW4lite is a scaled-down version of SW4 that has limited seismic modeling capabilities, but does solve the elastic wave equation and uses some of the same numerical kernels as those implemented in SW4.

Table 8: Applications and Input Problems

Proxy/Parent	Domain	Input Files
ExaMiniMD/LAMMPS	Molecular dynamics	snapA, snapB, snapC
PICSARlite/PICSAR	Particle-in-cell	homogeneousPlasma, langmuirWave, plasmaBeam
SW4lite/SW4	Seismology	Ghz, LOH25, LOH31

For each of the proxy/parent pairs, we generated two–three different input problems, varying the type of problem solved where possible and the size. Table 8 shows the inputs that we use, but some explanation to understand the differences between these is required:

- ExaMiniMD/LAMMPS Snap inputs: We use the snap potential here, which is computationally complex, and we vary the grid size. The goal was to determine if the change in memory size used significantly altered the performance. SnapA is the smallest grid size.
- PICSARlite/PICSAR plasma inputs: The homogeneousPlasma is a homogeneous plasma mirror. The plasmaBeam input focuses high order harmonic beams by a relativistic plasma

mirror. The langmuir wave input introduces rapid oscillations of the electron density in the plasma.

- SW4lite/SW4: The Ghz input uses a Gaussian hill model where the parameters so that we create a mesh where we vary the number of cells. The LOH inputs known use a point moment tensor source in a simple 3-D layered material model and use a purely elastic material. Here we varied the grid spacing so that the LOH25 problem is twice as large as the LOH31 problem.

5.1.2 Data Collection

We use the Light Weight Distributed Metric Service (LDMS) monitoring infrastructure developed at Sandia to collect performance counter data that includes over 500 events. We use the PAPI sampler within LDMS to interface and collect data from the performance counters. We collect from every event available on the processor, with the exception of events that return no data or unstable data (i.e. vastly varying across runs).

We run each application with each input five times to account for performance variation and any variation in the performance counters. Data is collected every second throughout execution from each MPI rank, then 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 (3) normalize the event counts by cycles executed. This results in a vector of approximate length 500 for each application, which we then use as input to the cosine similarity algorithm (described in FY21 Quantitative Assessment Report on our project website).

5.2 Results

Results that compare the similarity of all proxy/parent pairs for all inputs used for all performance counter groups is shown in Figure 23. This result is a bit surprising in that it shows that the different inputs for all of the proxy/parent pairs show high similarity. However, it is good to see that there is some divergence between every proxy/parent pair and every input of these pairs, so none of the application runs that we did are highly similar (they range from (20-38 degrees).

The inputs used for ExaMiniMD/LAMMPS are all similar problems, but each is a larger grid size. If that grid size does not overwhelm the cache/memory subsystem, then we would expect this similarity in behavior. If we look at the similarity matrix for cache in Figure 24, we see that ExaMiniMD and LAMMPS start to diverge a bit (but not significantly) in L2 and L3 cache. Perhaps the long-range interactions that are not implemented in ExaMiniMD are changing the access pattern in some way that causes this behavior. We need to investigate the behavior in conjunction with the code to know if this is the case.

The surprise in the SW4lite/SW4 results is that the Gaussian-Hill problem is very similar to the LOH problems, but these use very different material models. The Gaussian-Hill is a very simple 2-D topography, where the LOH problem uses a 3D-layered elastic material with different properties per layer. Looking at the similarity matrices for performance counter subgroups, we see in Figures 24 and 25 that there is more divergence between these problems in L3 cache and the pipeline dispatch stage than in any of the other subgroups.

For PISCARlite/PICSAR, we use only two, but very different problems. However, the similarity matrix in Figure 23 shows that the behavior using these different problems is highly similar. Looking at the similarity matrices for the performance counter subgroups, PICSARlite/PICSAR remain highly similar for all subgroups for the two different input problems.

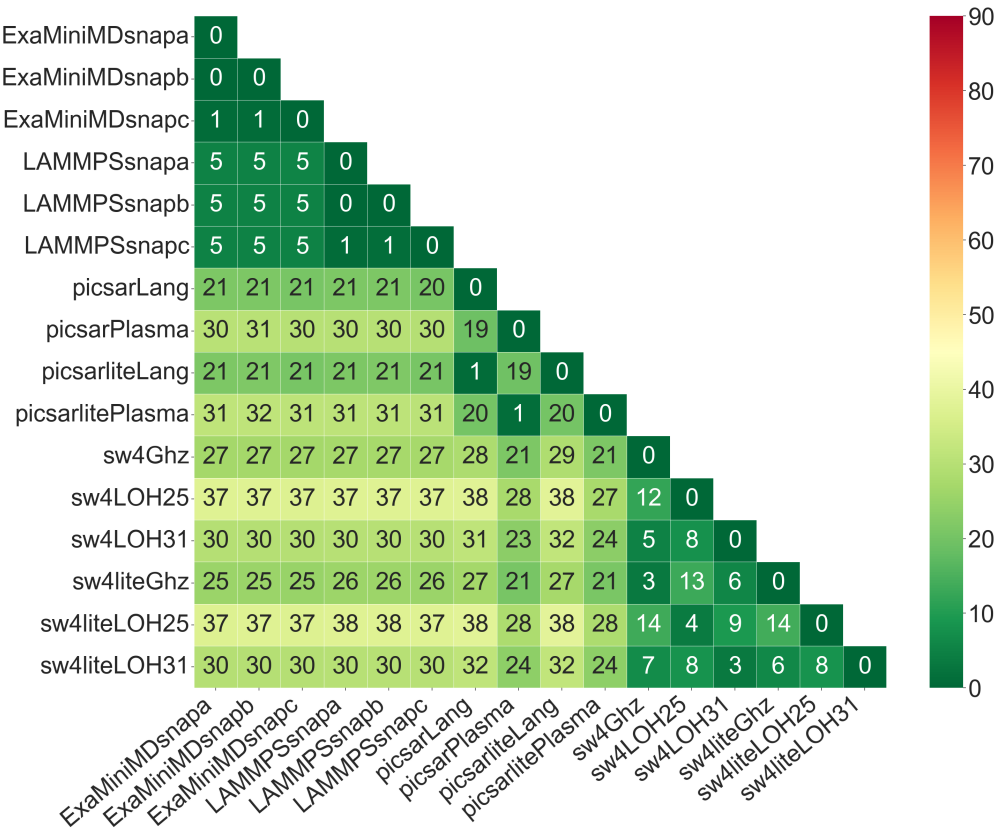
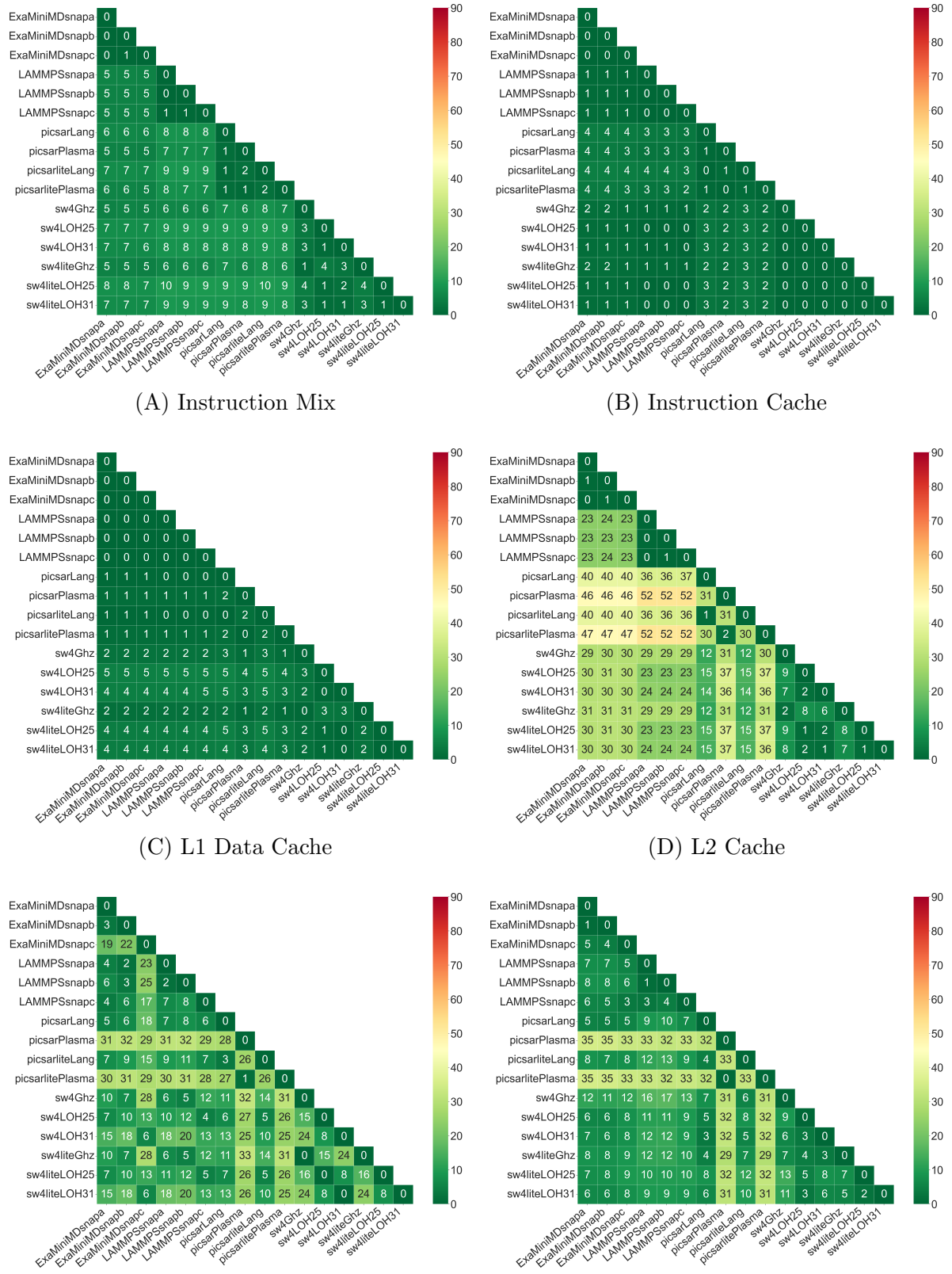


Figure 23: Cosine Similarity Matrix for All Proxy/Parent Pairs and All Inputs



(A) Instruction Mix

(B) Instruction Cache

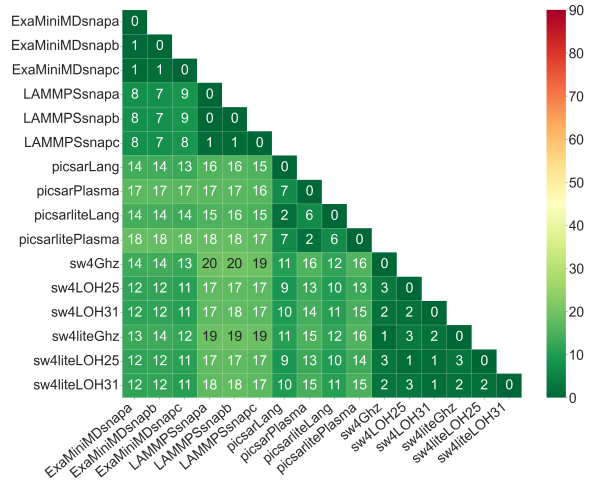
(C) L1 Data Cache

(D) L2 Cache

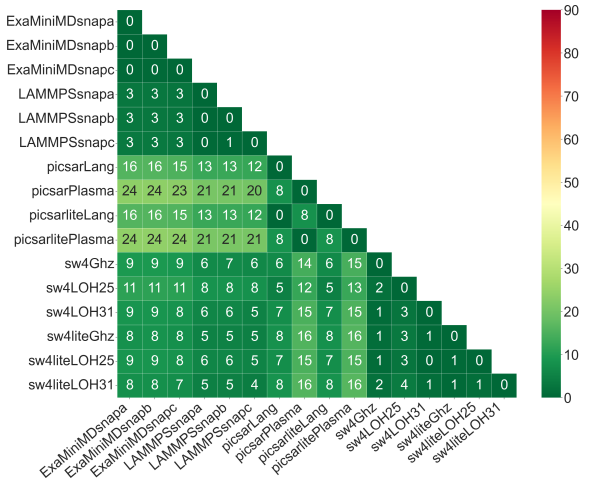
(E) L3 Cache

(F) Memory

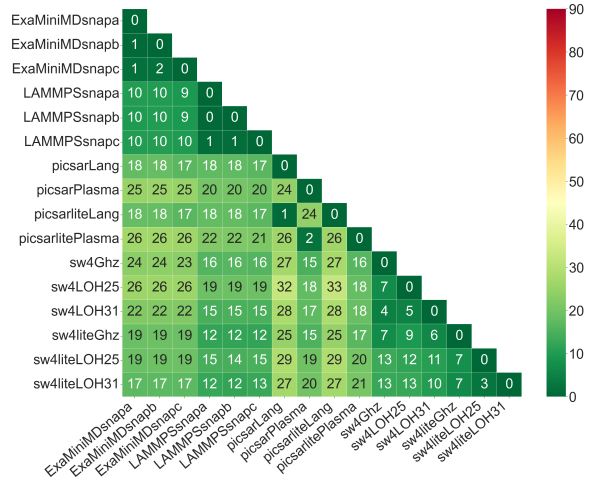
Figure 24: Cosine Similarity Subgroups1 for All Proxy/Parent Pairs and All Inputs



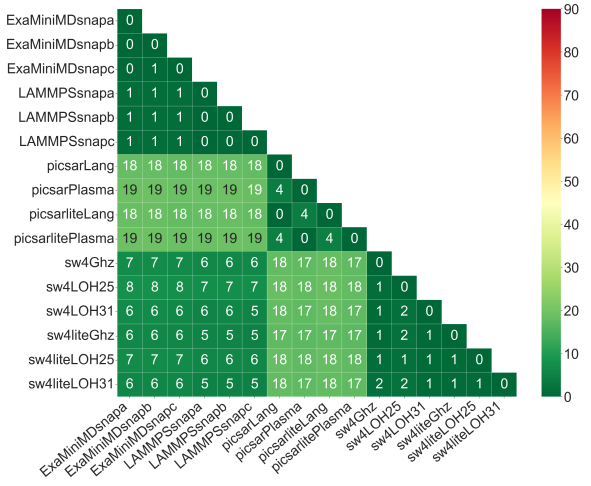
(G) Pipeline Frontend Stage (Fetch)



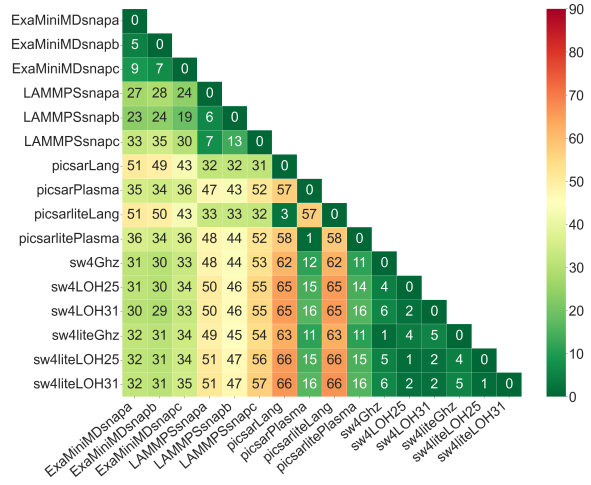
(H) Pipeline Decode/Issue Stage



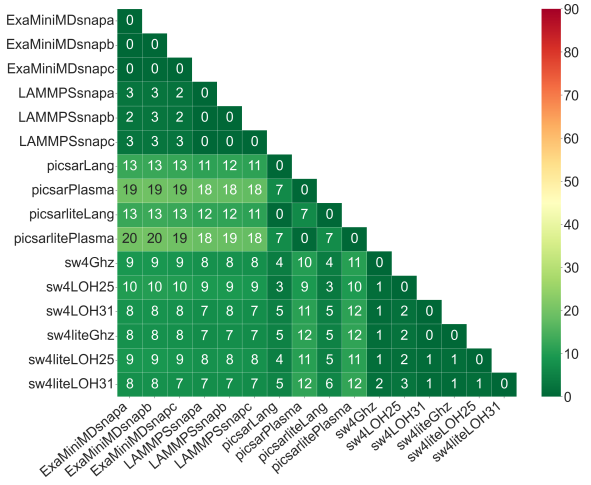
(I) Pipeline Dispatch Stage



(J) Pipeline Execution Stage

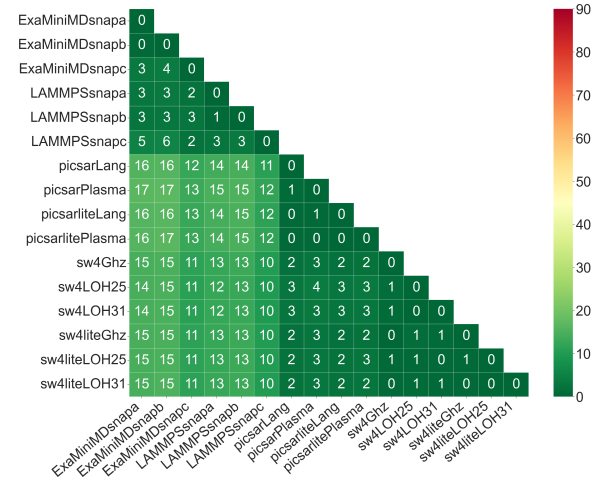


(K) Pipeline Memory Stage

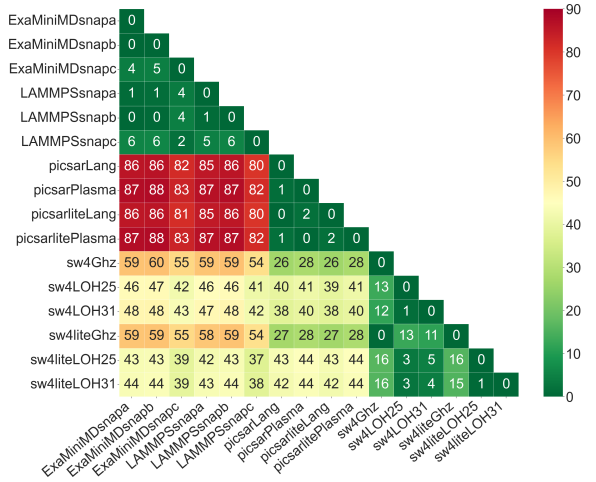


(L) Pipeline Retirement Stage

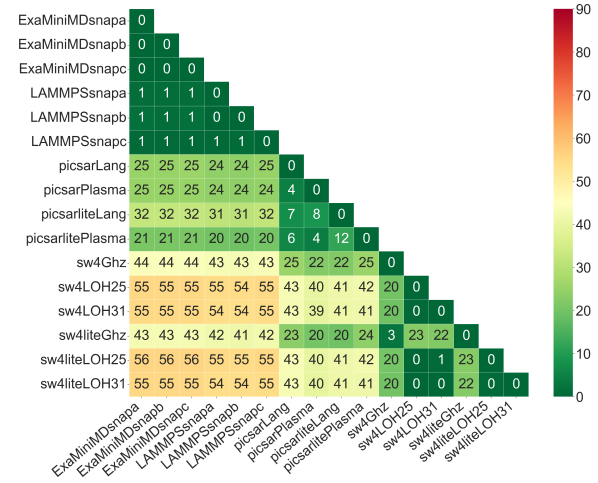
Figure 25: Cosine Similarity Subgroups2 for All Proxy/Parent Pairs and All inputs



(M) Branching/Branch Predictor



(N) Power (Throttling & Turbo)



(O) Miscellaneous (Machine Clears/Interrupts)

Figure 26: Cosine Similarity Subgroups3 for All Proxy/Parent Pairs and All inputs

5.3 Conclusion

Although it has long been believed that a different problem used as input to the same application is a different application with respect to the underlying affect on the hardware, our results show the opposite. For all of the proxy/parent pairs and the 2–3 different problems we use for each, we see highly similar behavior from the cosine similarity algorithm using all of the performance counters or subgroups of them.

This result could be an artifact of the problems that we chose. Therefore, we plan to go back to the application developers with these results to ensure that the problems we used are indeed quite different. In the past, we did a small study of varying inputs for an AMR-based code and we saw divergence in behavior when using the same method we applied in this work for similarity. All of our proxy/parent pairs and varying inputs being similar could also be affected by the type of applications we chose. All of the proxy/parent pairs are from essentially typical scientific domains – we did not choose any graph-processing, AMR-based, or something like circuit simulation. In the future, we will look at including applications from more widely varying domains to see if the result remains consistent.

Problem	Size	Ranks	LAMMPS	ExaMiniMD	Ratio
SNAP Ta	8388608	4	511.1	2357.6	4.61
SNAP Ta	8388608	16	512.5	2359.7	4.60
SNAP Ta	8388608	64	516.6	2362.1	4.57
SNAP W	8192000	4	848.6	7614.6	8.97
SNAP W	8192000	16	851.8	7621.2	8.95
SNAP W	8192000	64	853.4	7624.0	8.93

Table 9: LAMMPS and ExaMiniMD average times on Vortex

6 Proxy Applications as Performance Predictors of Applications on GPUs

In order to access whether proxy applications are predictive of parent applications when run on GPUs, we ran two pairs of proxy/parent applications in a scaled study on a local machine called Vortex. Vortex is a small scale version of the Sierra system at LANL in which each node has a dual socket Power 9 CPU and four Nvidia Tesla V100 GPUs and the nodes are connected by a fat-tree Mellanox 100 Gbit/s EDR Infiniband interconnect. Since each node on Vortex has 4 GPUs, we ran each node as 4 MPI ranks, each with a CPU and a GPU.

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. Both codes use Kokkos to provide an interface to run on the GPUs. A comparison of the average run times over five runs for the two codes on Vortex is in Table 9.
For these results, we used two problems. We used the SNAP Ta problem and the SNAP W problem. The size is the numbers of lattice points per rank. For the SNAP problems, LAMMPS has 2 atoms per lattice point. For a given problem at a given size, the ratio between the codes is fairly constant, but overall, ExaMiniMD does not seem to be predictable of LAMMPS. What these results show is that the two codes weak scale similarly.

6.2 SW4 and SW4lite

SW4 is a 3-D seismic modeling code and stands for Seismic Waves, 4th order accuracy. SW4lite is a bare bone version of SW4 that is used to test versions of the important numerical kernels of SW4 for performance optimization. RAJA is used with SW4 to allow it to run on the GPUs, while SW4lite has the options of running on the GPUs with either RAJA or CUDA. The size is the grid spacing in each of the three directions and is chosen so that the problem scales with the number of GPUs. A comparison of the average run times over five runs for the two codes on Vortex is in Table 10 for the RAJA version of SW4lite and is in Table 11 for the CUDA version of SW4lite.
The tables show that the run times for SW4 and SW4lite do not correlate for the GaussianHill and LOH1 problems either for the runs with SW4lite compiled with RAJA or CUDA. Therefore, we can say that SW4lite does not seem to be predictive of SW4. They also do not seem to scale similarly.

Problem	Size	Ranks	SW4	SW4lite	Ratio
GaussianHill	h=0.008	4	1409.1	4077.7	2.84
GaussianHill	h=0.005	16	1358.4	4327.8	3.19
GaussianHill	h=0.003	64	1644.1	4554.4	2.77
LOH1	h=39.7	4	972.3	534.8	0.55
LOH1	h=25.0	16	1561.6	879.2	0.56
LOH1	h=15.75	64	2523.1	1161.3.1	0.46

Table 10: SW4 and SW4lite (both RAJA) average times on Vortex

Problem	Size	Ranks	SW4	SW4lite	Ratio
GaussianHill	h=0.008	4	1409.1	197.2	0.140
GaussianHill	h=0.005	16	1358.4	299.3	0.220
GaussianHill	h=0.003	64	1644.1	448.2	0.273
LOH1	h=39.7	4	972.3	91.2	0.094
LOH1	h=25.0	16	1561.6	156.6	0.100
LOH1	h=15.75	64	2523.1	268.3	0.106

Table 11: SW4 (RAJA) and SW4lite (CUDA) average times on Vortex

6.3 Predictability Conclusion

For the two sets of proxy and parent applications that we studied, we found that neither proxy was predictive of its parent's run time. Even though SW4lite is a bare bones version of SW4 and has similar performance on CPUs, neither the RAJA nor the CUDA GPU implementation is similar to the RAJA GPU implementation of SW4. There was also quite a difference with the two different problems. LAMMPS and ExaMiniMD were solving the same problems with similar algorithms, but there seems to be no correlation for their timings, but the two codes seem to scale similarly. 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. This is similar conclusion to the case in our previous report where we compared five sets of proxy and parent applications on CPUs and found that only one proxy, SW4lite, was predictive of execution time for its parent.

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.

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

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

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

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

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

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

References

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

[2] Valentin Churavy, William F Godoy, Carsten Bauer, Hendrik Ranocha, Michael Schlottke-Lakemper, Ludovic Räss, Johannes Blaschke, Mosè Giordano, Erik Schnetter, Samuel Omlin, Jeffrey S. Vetter, and Alan Edelman. Bridging hpc communities through the julia programming language, 2022. URL: https://arxiv.org/abs/2211.02740, doi:10.48550/ARXIV.2211.02740.

[3] J. Cook, O. Aaziz, S. Chen, W. Godoy, A. J. Powell, G. Watson, C. Vaughan, A. Wildani, and The ECP Proxy App Team. Quantitative Performance Assessment of Proxy Apps and Parents. Technical Report ADCD-504-28, 2022.

[4] J. Cook et al. FY22-1 Quantitative Performance Assessment of Proxy Apps and Parents: Report for ECP Proxy App Project Milestone ADCD-504-28. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, 2022.

[5] William F Godoy, Jenna Delozier, and Gregory R Watson. Modeling pre-exascale amr parallel i/o workloads via proxy applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 952–961, 2022. doi:10.1109/IPDPSW55747.2022.00153.

[6] 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.

[7] D. Richards et al. FY19 Quantitative Performance Assessment of Proxy Apps and Parents: Report for ECP Proxy App Project Milestone AD-CD-PA-504-5. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, 2019.

[8] D. Richards et al. FY21 Assessment Report. Report for ECP Proxy App Project Milestone ADCD-504-11. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA, 2021.