Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone AD-CD-PA-504-5

David Richards[1], Omar Aaziz[2], Jeanine Cook[2], Hal Finkel[3], Brian Homerding[3], Peter McCorquodale[4], Tiffany Mintz[5], Shirley Moore[5], Vinay Ramakrishnaiah[6], Courtenay Vaughan[2], and Greg Watson[5]

[1]Lawrence Livermore National Laboratory, Livermore, CA
[2]Sandia National Laboratories, Albuquerque, NM
[3]Argonne National Laboratory, Chicago, IL
[4]Lawrence Berkeley National Laboratory, Berkeley, CA
[5]Oak Ridge National Laboratory, Oak Ridge, TN
[6]Los Alamos National Laboratory, Los Alamos, NM

July 2019

EXASCALE COMPUTING PROJECT

LLNL-TR-772413

Contents

1 Executive Summary

Proxy applications are small, simplified codes that allow application developers to share important features of large applications without forcing collaborators to assimilate large and complex code bases. Proxies can also be though of as models for larger codes. Because proxies omit features and capabilities of their parent applications (otherwise they wouldn't be small), the question naturally arises whether any given proxy is actually a faithful model for its parent application for some particular modeling purpose. This report sets out to answer that question for some of the proxies in the ECP proxy app collection[1], with particular attention to performance. It also completes the AD-CD-PA-504 Milestone:

> We will assess the fidelity of 8 proxy applications with respect to their respective parent applications using the methodology previously developed.

Note that we use the term "parent" somewhat losely here. In some cases there is a clear lineage relationship where the proxy was created directly from the large application we are comparing it to. In others, the proxy may have been created independently, or even from an application other than the one we are comparing. Hence, in this report, parent merely refers to the application for which the proxy is being evaluated for model fidelity.

The complete set of proxy-parent pairs evaluated in this report is shown in Table 1. Because application performance can vary significantly depending on the problem being solved, for each proxy and parent we took special care to identify input parameters to specify problems that were as similar as possible from proxy to parent and also representative of exascale-class problems. These problem specifications are one of the key artifacts of this report. Summary descriptions of the problems are given in this report and full specifications are published on the ECP Proxy App web site.

We found that some of the proxies are very good models for the performance of their parents. In particular, miniQMC, miniVite, and XSBench have performance characteristics that are very similar to their corresponding parents. In contrast, PICSARlite, ProfugusMC, MACSIO, SW4lite, and NEKbone showed significant performance differences from their parents. The reasons for these differences vary and are more fully explored in the respective sections of the document. Although we highlight the performance differences between proxies and parents we do not want to imply that any of these proxies are poor or should not be used. We do however warn users that blindly

Proxy	Parent
miniQMC	QMCPack
miniVite	Vite
PICSARlite	WarpX
ProfugusMC, XSBench	Shift
thornado-mini	Castro/FLASH
MACSIO	Castro
SW4lite	SW4
NEKbone	NEK 5000

Table 1: Proxies and parents analyzed in this report.

[1]The current version of the ECP Proxy App Suite can be found at http://proxyapps.exascaleproject.org/ecp-proxy-apps-suite

assuming that all proxies are faithful representations of all aspects of the parent performance will lead to incorrect conclusions.

Finally, we note that the majority of the work in this report was performed on CPU platforms. This is somewhat unfortunate as GPU hardware will provide the majority of the computing power on exascale-class systems. We will continue to publish GPU-based assessments as more proxies and parents are ported to GPUs and assessment tools improve.

2 Methodology

This section describes our methodology for both CPU- and GPU-based assessments. We give specifications for the compter hardware systems used and the tools we used for profiling and measurement. Compiler details are presented for each proxy/parent in their respective sections on assessment. Finally, during the course of this work we uncovered some significant problems with tools and systems. We have provided a list of these issues to inform the broader community.

2.1 CPU Performance Assessment

For CPU-based assessment of proxy apps and parents, the goal is to understand performance similarity between the proxy and parent at both a high and low level. We aim to understand high-level similarity through profiling and models such as roofline. At a lower level, the desire is to understand similarities in how the applications execute on the underlying hardware by examining their performance using key metrics. A stretch goal is to understand the similarity of performance bottlenecks at the hardware level. The outcome of this assessment is to fundamentally know how well the proxy represents the parent, and understand the limitations to this representativeness.

To this end, our CPU assessment for each proxy/parent pair minimally includes the following:

1. a clear qualitative comparison and description of the proxy versus the parent with respect to primary functionality,
2. dynamic profiling to understand if key kernels and functions implementing these kernels are as expected and consistent across the two applications,
3. roofline modeling to understand at a high-level if the two applications are characterized by the same bottlenecks (e.g., compute-bound versus memory bound, cache bound, floating-point operations bound),
4. hotspot analysis that indicates memory or compute boundedness for each function and/or kernel.

For some proxy/parent pairs, further analysis was done. Note that in this assessment, we examine the performance of MACSio and how well it can generate I/O behavior that matches an ECP application of interest. MACSio is very different than the other proxies analyzed in this work. It is a parameterizable I/O proxy that generates I/O behavior based on the input parameters that are extracted from the application of interest. The assessment methodology is, therefore, quite different for this proxy as is noted in Section 8.

We also did some assessment of GPU-enabled proxy applications. The assessment methodology for this is described in the next section.

2.2 GPU Performance Assessment

Most applications use GPUs by offloading computationlly intensive kernels to these accelerators. Programming models used include CUDA, OpenACC, OpenMP 4.5, and the portability frameworks Raja and Kokkos. The GPU performance assessment is carried out by identifying the most important GPU kernels and measuring their performance characteristics. We attempt to compare characteristics of corresponding kernels between the proxy and the parent application.

The most important GPU kernels cannot necessarily be identified by running the GPU version of the code on a default input set and collecting a timing profile. The input set used for the profiling should ideally be representative of the challenge exascale problem, scaled down to represent performance on a single node or small number of nodes. Determination of the appropriate input set is carried out by consulting milestone reports for the relevant ECP parent application project and/or by consulting with the proxy and/or full application developers. In some cases, the proxy app may use an input for a synthetic or analytical problem to avoid complexities and dependencies of the full application. We attempt to use recent versions of the proxy and parent applications that are implemented using the programming model(s) being used by the parent application development team moving forward toward scaling to exascale.

Once the most important GPU kernels have been determined, we carry out more detailed performance data collection and analysis for these kernels. We collect data to calculate instruction mix, arithmetic intensity, achieved memory bandwidth, and percentage of peak attained for each kernel. We use the arithmetic intensity to place the kernel on the relevant GPU roofline model. We compare the values of the performance data and derived metrics between corresponding kernels for the proxy and full application to try to determine how well the proxy represents the full application. We use the NVIDIA nvprof command-line profiler to collect performance data.

2.3 Computational Platforms

Between the various groups working on this project at the different labs, three primary platforms were used for measurement of CPU characteristics (for all proxy/parent pairs except MACSio). These are the Cori system at NERSC, the Blake (Intel Skylake) testbed at Sandia National Labs, Albuquerque, and an Intel Skylake platform at Argonne National Lab. In each section on proxy/parent analysis, the platform used is called out.

Cori is an Intel machine with two partitions, one being Haswell, the other Knight's Landing. For CPU measurement, only the Haswell partition was used. Hardware characteristics of Cori are shown in Table 2.

The Blake testbed at SNL is an Intel Skylake Platinum 8160 with characteristics as shown in Table 3. This architecture has six memory channels per socket, with a total of eight execution units that includes three vector units (two support 512-bit ops) that all do general vector ops and FMA. Skylake supports the AVX512 ISA, and has a new core-to-core memory fabric that is a two-dimensional mesh that potentially reduces latency and increases bandwidth between cores and memory.

An Intel Skylake Platinum 8180M, very similar to Blake, was also used for CPU assessment. Its characteristics are also shown in Table 3. The only difference between this machine and Blake is that it has a few more cores and a larger L3 cache.

For characterizing MACSio as a proxy, we used the Titan system at ORNL. This is an AMD

μop cache	1536 μops, 8 way, 6 μop line size, per core
L1 data cache	32 KB, 8 way, 64 sets, 64 B line size per core
L1 instruction cache	32 KB, 8 way, 64 sets, 64 B line size, per core
L2 cache	256 kB, 8 way, 512 sets, 64 B line size, per core
L3 cache	2–45 MB, 12–16 way, 64 B line size, shared
Memory (per node)	128 GB DDR4-2133 MHz (64GB per socket)
Cores/threads	16/32
Sockets/node	2
Total nodes	32
Interconnect	Mellanox FDR Infiniband
Max Memory BW	68 GB/sec

Table 2: Hardware characteristics of Haswell platform.

Component	8160	8180M
L1 data cache	32 KB, 8 way, 64 B line size per core, private	Same
L1 instruction cache	32 KB, 8 way, 64 B line size, per core, private	Same
L2 cache	1 MB, 16 way, 64 B line size, per core, private	Same
L3 cache	33 MB, 12–16 way, 64 B line size, shared, non-inclusive	Same except 38.5MB
Memory (per node)	192GB DDR4-2666 MHz	Same
Cores/threads	24/48	28/56
Sockets/node	2	Same
Total nodes	40	13
Interconnect	Intel Omnipath	Same

Table 3: Hardware characteristics of Skylake platforms.

Component	Opteron 6274	Kepler K20X
L1 data cache	16x16 KB, 4 way	64KB per SM
L1 instruction cache	8x64 KB, 2 way	N/A
L2 cache	8x2 MB, 16 way	1.5MB
L3 cache	2x8 MB, 64 way	2x8 MB, 64 way
Memory (per node)	32 GB UDDR3-1600 MHz	6 GB DDR5
Cores/threads	16/16	14 SM/2688 SP/896 DP/2048
Sockets/node	1	1
Total nodes	18,688	18,688
Interconnect	Gemini (6.4 GB/s) 3D Torus	PCIe 2.0

Table 4: Hardware (node) characteristics of Titan platform.

Filesystem	Lustre 2.5
Bandwidth	1.4 TB/s read, 1.2 TB/s write
Capacity	32 PB
Raid Controller	DDN SFA12KX
Number of Disks	20,160
Connectivity	IB FDR
Number of Object Storage Targets (OSTs)	2,016
Number of Object Storage Services (OSSs)	288

Table 5: Titan I/O subsytem.

Cores	80SM, 5120 CUDA, 640 Tensor
FP64/FP32	7.5/15 TFLOPS
HBM2 Bandwidth	900 GB/sec
NVLink Bandwidth	300 GB/sec
HBM2	16GB
L2 Cache	6 MB
L1 Caches	10 MB

Table 6: Tesla V100 GPU

FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256KB
Unified L1/Shared memory	128 KB
Active Threads	2048

Table 7: Tesla V100 SM

Opteron 6274 Interlagos system with NVIDIA Kepler K20X GPUs, with node characteristics as shown in Table 4. Table 5 shows the configuration of the Spider2 I/O subsystem.

We carried out our GPU assessments on the Summit supercomputer at ORNL. Summit has 4,608 nodes, each of which contains two IBM POWER9 CPUs and six NVIDIA Volta (V100) GPUs, all connected with NVIDIA's high-speed NVLink. V100 characteristics are shown in Table 6 with details of its SMs (streaming multiprocessors) shown in Table 7. GPU programming models supported by compilers installed on Summit include CUDA, OpenACC, and OpenMP 4.5. Kokkos and Raja are in use by some applications but must be installed by the user.

2.4 Profiling and Measurement Tools

As a team, we use a fairly large suite of tools to perform assessment. For profiling, Intel VTune Amplifier [22] (2018 and 2109), CrayPat [25], or HPCToolkit [21] were used. Gprof [23] is not used for profiling due to issues noted in Section 2.5. Intel Advisor (2018 and 2019) is used to generate the roofline models for all of the CPU assessments. We noted some issues pertaining to roofline models, which are also outlined in Section 2.5. Some teams use Intel VTune to produce per-function hotspot analysis. Application samplers implemented in LDMS [17] (Light-Weight Distributed Monitoring System) are also used to collect various per-function and whole execution hardware performance counter data as is HPCToolkit.

2.5 Systems and Tool Problems

In performing the work presented in this report, several tool and system issues were encountered. We report the issues we encountered in this section to inform the community. We will be doing more work in the future to understand the tool problems that we encountered so that we can work with tool developers and vendors to fix these issues in future releases.

Testbed systems at SNL have proven to be very rigid. The Blake system that was chosen as a measurement platform has a very old kernel (several years old) that constrains the hardware performance counter events that can be collected. We brought this to the attention of the administration group, and the kernel was eventually upgraded (after a significant delay). Performance tools need to have current OS kernels. The underlying mapping of events to actual performance counters changes rapidly. If kernels are not routinely updated, a performance analyst is constrained not only by what can be measured, but also by the tools through which this measurement is done. For example, we could not install HPCToolkit on Blake because the kernel and all its components were too old.

The old kernel on the Blake (Skylake) system meant that we could not collect events that enable us to compute memory bandwidth (hyperthreading was also enabled, see below). Therefore, we attempted to use the one testbed that has a very updated kernel for memory bandwidth measurement. Unfortunately, we ran into a problem. It seems that for some reason, the events associated with memory bandwidth do not collect any information on this system. We are currently working with the PAPI development team to try to resolve this issue.

Hyperthreading also affects the accuracy of measurement by performance tools on Intel-based systems. If hyperthreading is turned on, even if the application does not use hyperthreading, the performance counters are affected. Performance counters are shared on a core between threads. If hyperthreading is used, the performance counter set is divided by two. If multiplexing is not being used, that means at least twice as many runs must be done to collect the same number of performance counter events. However, the more concerning issue is that even if hyperthreading is not used by the application, performance counts of the application can be corrupted and inaccurate because the other thread, even if it's not an application or user thread, could potentially be incrementing the counter (an OS job may be running on the other thread without the user's knowledge). Experts continually emphasize that the only way to get truly accurate performance counter data is to turn hyperthreading off. Unfortunately, most systems have it turned on. We are working with the testbed administrators to arrange dedicated time with hyperthreading turned off to better understand its impact on accuracy.

In using Intel's VTune Amplifier and Advisor, several problems were encountered. The most concerning here is the inconsistency between results generated by VTune and those generated by HPCToolkit and LDMS. Several tests were done in measuring hardware performance counter events using these three tools on the same application and input. We measured several different events including those pertaining to cache misses, branch prediction, and instruction execution. HPCToolkit and LDMS application samplers were always in agreement. VTune generated very different event counts. This was observed during analysis of miniQMC and QMCPACK, which are heavily templated C++ codes. Anecdotally, this is not the first time that VTune has been noted as generating performance results that did not match other tools. In our experience, VTune is often unreliable for complex codes that rely on extensive use of templates. We need to cross-validate many of the commonly-used performance tools to check for inconsistencies. This also calls into question the accuracy of the roofline models generated by the Intel Advisor tool. Examining the roofline in conjunction with the hardware performance counter data collected from both HPCToolkit and LDMS seems to show some agreement and an indication that the rooflines of at least miniQMC and QMCPACK are accurate as far as we can tell. However, we did not do a thorough comparison of the execution time profile data that is also generated with the roofline model. In contrast, in the WarpX/PICSARlite study, execution time profiles output with the roofline model were compared to those generated by HPCToolkit. The execution time profiles generated by these two tools were highly inconsistent (see Section 5.4.3). Further investigation should be done to make a final call on the accuracy of rooflines generated by Intel Advisor.

We also discovered several shortcomings in the tools that we used for measurement. We attempt to list these below.

The following issues and discrepancies were encountered while using Intel VTune Amplifier and Intel Advisor for profiling Vite/miniVite:

- The option to list the hotspots in only the user defined functions captured library function calls.
- The core times in VTune (even though an approximation) are not proportional to the actual execution times. For example, Vite takes more time for execution than miniVite, but the core times indicate otherwise.
- Intel Advisor did not handle more than two processes for roofline analysis of Vite/miniVite. The limitations of the tool were unclear.

In using the HPCToolkit PAPI interface for collecting hardware performance counter data on miniQMC and QMCPACK, we noted the following:

- The tool outputs profile information and function names that are fully specified (unlike other tools). However, the actual dynamic profile has to be manually extracted. Since the tool generates most of the information needed for a dynamic profile, it would be great if it could generate full information and automatically generate a dynamic profile rather than forcing the user to sift through a function call stack and compute total execution times.
- In using HPCToolkit to collect performance counter data, the tool seems to have a limit to the number of events that can be simultaneously counted. If the tool truly multiplexes, this shouldn't be an issue. But the documentation was unclear and not specific as to whether multiplexing is actually implemented. The tool needs to multiplex and be able to handle large event lists.
- HPCToolkit seems to lack the capability to output data to a csv without using the Viewer. A command-line interface for data analysis would be very helpful.
- When looking at function profile data, the *Expand All* option does not actually show more detailed data for all functions. In order to see this data for all functions, they have to be expanded individually.
- When exporting function profile data, the csv file loses the call hierarchy that is displayed in the Viewer, making it impossible to see which function calls which function.
- A search by function name in the Viewer would be really helpful when trying to manually extract a dynamic profile. Otherwise manual searching of the functions must be done to extract the profile.

We plan to interface with the HPCToolkit ECP project developers to resolve these issues. We've also created a milestone for the next fiscal year that will enable us to more thoroughly investigate performance tool issues across platforms in the future.

3 miniQMC and QMCPACK Performance Assessment

MiniQMC is a quantum Monte Carlo code that comprises the important computational kernels of its parent, QMCPACK. Quantum Monte Carlo methods are often applied in material science to understand the electronic structure of molecular and solid state systems. The computational motifs of miniQMC and QMCPACK are particle methods, dense and sparse linear algebra, and Monte Carlo. MiniQMC and QMCPACK are being developed in the ECP Application Development project *QMCPACK: Predictive and Improvable Quantum-mechanics Based Simulations.* The major goal of this project is to re-factor the current petascale version of QMCPACK for performance portability and better exploitation of concurrency that will be required for exascale architectures.

3.1 Algorithms and Key Kernels

Quantum Monte Carlo is used in quantum mechanics to understand the electronic structure of molecular and solid state systems. MiniQMC, like QMCPACK, implements a direct solve of the Schrodinger wave equation, which provides accuracy at the expense of computational intensity. From the wave equation, the probability of particle position and particle energy are computed. MiniQMC and QMCPACK both implement Variational and Diffusion Monte Carlo (DMC). DMC samples the exact wave function, where Variational Monte Carlo (VMC) uses an approximate wave function. In this work, we focus on DMC.

The pseudocode for DMC in miniQMC is shown in Figure 1. Each walker represents a 3D particle position, R. An ensemble of walkers is generationally and stochastically propagated through a defined electronic structure. Each propagation step moves the particle through the structure using a drift-diffusion process. The particle's local energy is computed at each step to determine if the particle dies, continues propagation, or reproduces. This changing particle population potentially creates imbalance that is addressed by periodic load balancing.

The lines shown in bold in Figure 1 are the most computationally and/or memory intensive steps in the algorithm. These are associated with key kernels. The four key kernels in both miniQMC and QMCPACK are the following:

```
1. for MC generation = 1..M do
2.    for walker = 1..Nw do
3.       let R = {r1…rN}
4.       for particle I = 1…N do
5.          set r'_i = r_i + del
6.          let R' = {r_1…r'_i…r_N}
7.          ratio p = Psi_T(R')/Psi_T(R)    (Jastrow factors, 3D B-spline, Determinants)
8.          derivatives \nabla_i log(Psi_T), \nabla^2_i log(Psi_T)    (Jastrow factors, 3D B-spline, Determinants)
9.          if r—> r' is accepted then
10.            update state of a walker (inverse update)
11.         end if
12.      end for {particle}
13.      local energy E_L = H Psi_T(R)/Psi_T(R).   (Jastrow factors, 3D B-spline)
14.      reweight and branch walkers
15.   end for {walker}
16.   update E_T and load balance
17. End for {MC generation}
```

Figure 1: miniQMC pseudocode.

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. This is clearly seen in the dynamic profile, presented in Section 3.4.1.
2. Splines: This kernel is invoked for every potential electron move. It computes the 3D spline value, the gradient ($4{\times}4{\times}4{\times}N$ 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.

3.2 Problem Selection and Validation

MiniQMC comprises all of the key kernels that are in QMCPACK, although their relative importance in terms of percentage of total execution time is slightly different. MiniQMC implements one walker per MPI rank, but has no inter-rank communication—it is meant for single-node explorations only. Conversely, QMCPACK is fully MPI parallelized and is designed to take advantage of large-scale systems. Both miniQMC and QMCPACK support OpenMP threads, where the number of threads for a DMC calculation should be chosen to be only slightly larger than the number of walkers according to QMCPACK documentation [16]. However, from our own trials and from the data in [18] (see Figure 2), performance is not largely sensitive to an increase in the number of threads. Therefore, to compare the proxy to the parent, we chose a single-node configuration, with one OpenMP thread per MPI rank; we chose the number MPI ranks based on the available socket memory as explained below.

The exascale challenge problem for QMCPACK is to simulate transition metal oxide systems of approximately 1000 atoms to 10 meV statistical accuracy with performance portability. The transition oxide of choice is nickel oxide (NiO), and the target number of atoms is 1024. The 1024 atom problem is extremely memory intensive and cannot practically be executed on any contemporary systems without running out of memory. We chose a fairly large, contemporary testbed system at Sandia for performance measurement and the largest problem that we can execute on this system (192GB/node memory) is 256 atoms (3072 electrons), which uses about 12GB per core. The system we ran on has 24 cores per socket, two sockets per node, but we use only 4 cores per socket (48GB) in order to force a reasonable run time and to ensure we execute within memory limits.

For QMCPACK, we use an input file that was obtained from the development team. To match this input for miniQMC, we use the "-g 2 2 2" flag, which according to the table listed in [19], is 256 atoms and 3072 electrons. We also use the "-r 0.999" to more accurately reproduce a DMC run. MiniQMC is designed to execute one walker per rank on a single node. For comparison of miniQMC and QMCPACK, we use 8 ranks on a single node.

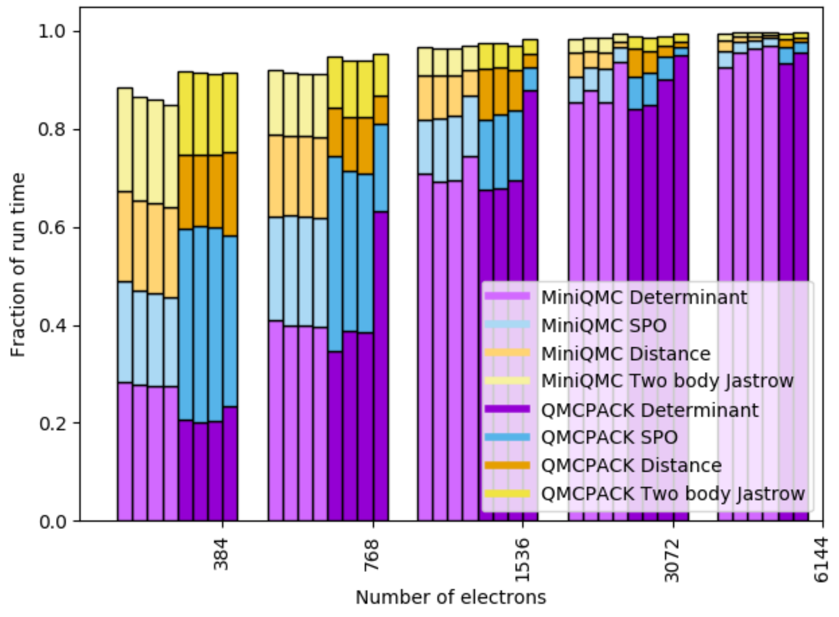


Figure 2: Comparison of kernel execution times for various thread counts (reproduced from [18]).

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 8: Measured kernel function timing for miniQMC and QMCPACK.

	miniQMC	QMCPACK
Execution time (secs)	1551.45	1612.32
Executed instructions	4732011524545	2185741090029

Table 9: Execution time and instructions executed.

We validate that our QMCPACK and miniQMC runs perform as expected by reproducing a portion of the data from [18], reproduced here as Figure 2. This figure shows key kernel execution times as a fraction of total runtime for both miniQMC and QMCPACK using a varying number of threads (1, 14, 28, 56). The data was generated on a Skylake 8160, which is the same platform we use for our runs (although L3 cache and memory size may vary across the two systems). In the figure, there are four bars for each application, with one bar for each of the four different thread count configurations. We validate our data against only the single-thread case using 3072 atoms. Table 8 shows results of our validation, which are the percentages of total execution time for each of the kernels in miniQMC and QMCPACK. Comparing these percentages to those in Figure 2, we see they're quite close. The plot shows the *Determinant* kernel to be about 80% of the total execution time, and the rest to be roughly 15%, with *SPO* and *Distance* accounting for about the same percentage of execution time, and *Two Body Jastrow* being about half of both *SPO* and *Determinant*. About 5% of the execution time, according to the plot, is outside kernel execution. Our results show slightly less *Determinant* kernel execution time, and little more time in the other three kernels. But overall, our results are quite similar to those in Figure 2. We also confirmed our run configurations with an application expert. Therefore, we feel confident that our run configuration and input parameters are sufficiently correct.

We also examined execution time and the total number of instructions executed to determine if miniQMC is representative of QMCPACK in terms of execution time. Table 9 shows the execution time and the total number of instructions executed for each of the applications. The execution time differs by about 4%. The number of instructions executed differs by about 74%, with miniQMC executing a significantly larger number of instructions. These are actual instructions retired, so do not count instructions that are executed speculatively. Data presented in Section 3.4.3 indicates a higher percentage of load instructions and a much lower percentage of vectorization for miniQMC. This could be why we see such a difference in number of instructions executed. QMCPACK is doing fewer loads and fewer FP insructions (but both loads and FP instructions work on larger data since they are vector instructions).

3.3 Comparison Methodology

The goal of this work is to understand if miniQMC is fundamentally a good proxy of QMCPACK. miniQMC is designed to represent the key kernels that are in QMCPACK. Therefore, computationally and with respect to memory behavior, miniQMC should be a good proxy of its parent. Since miniQMC is a single-node implementation, its communication is not meant to fully represent that in QMCPACK, although the communication in QMCPACK is minimal due to the algorithm (limited to gathering walkers and load balancing at the end of each particle generation step). To further understand the similarities and differences between miniQMC and QMCPACK, we perform the following quantitative analysis in addition to that outlined in Section 2.1:

1. measurement of important metrics (e.g., cache bandwidths, FLOPs, instruction mix) for the entire application execution,
2. identification of microarchitecture bottlenecks for the whole application execution.

We report the results of this analysis in the following sections.

3.4 Results and Analysis

Here we present our data and analysis for comparing miniQMC and QMCPACK. We examine the results from each analysis, then make final observations with respect to similarity and representa-

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 10: Kernel function profiles.

tiveness in Section 3.6.

We compiled both proxy and parent using the Intel 18.1.163 compiler (-xCORE-AVX512). We use the Intel Advisor 2018.2.0 to generate the cache-aware roofline models, HPCToolkit for dynamic profiling and some hardware performance counter collection, and the hardware counter and appInfo samplers within LDMS (Light-Weight Distributed Monitoring System) to collect hardware performance counter data. Within the hardware counter sampler, we have implemented Intel's Top-Down Microarchitecture Analysis (TMA) [36] to identify hardware bottlenecks for whole application execution. We are currently working on completing tools that will enable us to collect TMA data per function.

3.4.1 Dynamic Profiling

We generated dynamic profiles using HPCToolkit. We initially used *gnuprof*. However, this tool did not report the full namespace of all function names, so we could not accurately identify and compare if the kernels were similar in the proxy and parent apps. From experimentation, it seems that *gnuprof* exhibits this issue for C++ codes, but works as expected for codes written in C. This is why we use HPCTooklit in this case.

Table 10 shows the dynamic profiles of miniQMC and QMCPACK, respectively, and also shows which functions implement each of the key kernels. The profiles appear to be fairly similar. In terms of percentages of execution time, the developers of miniQMC do state that miniQMC implements the key kernels of QMCPACK, but the relative percentages of execution time may vary, and looking at the table, this is absolutely true.

- The *Determinant* kernel comprises the same functions, except that QMCPACK has an additional function, *evaluateLog*.
- *SPO* has no functions in common between miniQMC and QMCPACK.
- The *Distance* kernel has two of three functions in common, with the third function only appearing in the profile of miniQMC (namely, *ParticleSet::setActive*).
- In the *Two Body Jastrow* kernel, there is some similarity, but there are more functions in QMCPACK.

Adding up the contributions of each of the kernels for both miniQMC and QMCPACK as shown in Table 8, we see that execution time contributions of the key kernels are fairly similar with the exception of *Two Body Jastrow*, which shows the largest different in total execution time. Given this, we have to ask ourselves, "what are we after in a proxy"? Must functions be the same and contribute similar behavior to the composite execution, or is the composite execution the important characteristic in maintaining similarity? And do we look at composite execution as the whole application or do we look at the composite behavior of each key kernel? This likely depends on the proxy consumer. We will address this further in Section 3.6. The data in Table 10 minimally indicates that we need to examine per-kernel data rather than per-function data.

3.4.2 Roofline Model

We use a cache-aware roofline model to get a general understanding of the behavior of both the proxy and parent and to determine at a high level if that behavior is similar. A roofline model is a useful tool for a very high-level understanding of how close an application is to peak performance and whether the application is bound by cache and memory bandwidths or by computation (FLOPs). The roofline curves show the maximum cache and memory bandwidths and FLOPs attained by the system and the points in the roofline show where each application function lies with respect to that curve.

Figures 3 and 4 show the roofline models for miniQMC and QMCPACK, respectively, for a single core and single thread execution. The values on each of the bandwidth lines and FLOPs lines are somewhere close to what we would expect. To get the bandwidths and FLOPs for the whole socket, multiply by the number of cores (24 in this case). Each dot on these curves represents a function or a loop within a function. Functions that lie to the left of the knees of the cache and memory curves are cache/memory bound; those that are to the right of the FLOPs knees are computationally bound. Functions that fall somewhere in between these two knees are both memory and computation-bound. From these rooflines, we see that both applications are characterized by some memory/cache-bound, some computationally-bound performance, and some functions that are both cache/memory and computationally bound. To determine how each function or function loop (dot) is bound, draw a straight line up from the particular function and the first line it hits determines what its bound by.

Starting with miniQMC in Figure 3, the first thing to note is that there is a relatively large number of functions bound by DRAM bandwidth. There is a smaller number of functions bound by L3 bandwidth and even fewer bound by L2 bandwidth. No functions are bound by L1 bandwidth. Also note the large red and yellow dots. These denote functions that have the most opportunity for optimization; if one was optimizing this code, he/she would begin by trying to optimize the function for whatever the function is bound by (e.g., cache/memory BW or FLOPs). The larger the dot, the more opportunity for optimization. In this plot, the two red dots are loops in the MKL BLAS library. The dot to the left is bound by scalar add GFLOPs; the rightmost red dot is bound by L3 bandwidth and scalar add GFLOPs. Assuming that these BLAS operations are matrix-matrix, they should be compute bound. MKL is a top function in terms of total execution time and it's implemented in the determinant kernel, which is known to be computationally bound.

There are actually four yellow dots, but two of these lie directly under a green function dot. Unfortunately, the tool does not seem to output a text version of the information on the roofline, but only provides an interactive html file that shows function/loop names when a dot is moused over. In this case, the interactive html roofline plot only shows the function/loop associated with the green dots, so we can't tell which functions are represented by these yellow dots. The other two yellow dots that are visible are the *einspline_spo::MultiBspline::evaluate_vgh* (lower) and *einspline_spo::MultiBspline::evaluate_v* (upper) functions from the SPO kernel. These are both DP Vector Add GFLOPs bound. These functions are both implemented in the SPO kernel, which is stated to be memory-bound. This team has implemented several data optimizations to attempt to alleviate these bounds, and it could be that they've shifted the bound to be computational rather than memory. It is on the border for being bound by L3 bandwidth, so good layout optimizations could be a reasonable explanation of its compute-boundedness.

The functions represented by the right-most dots are mostly related to Jastrow-factors, which

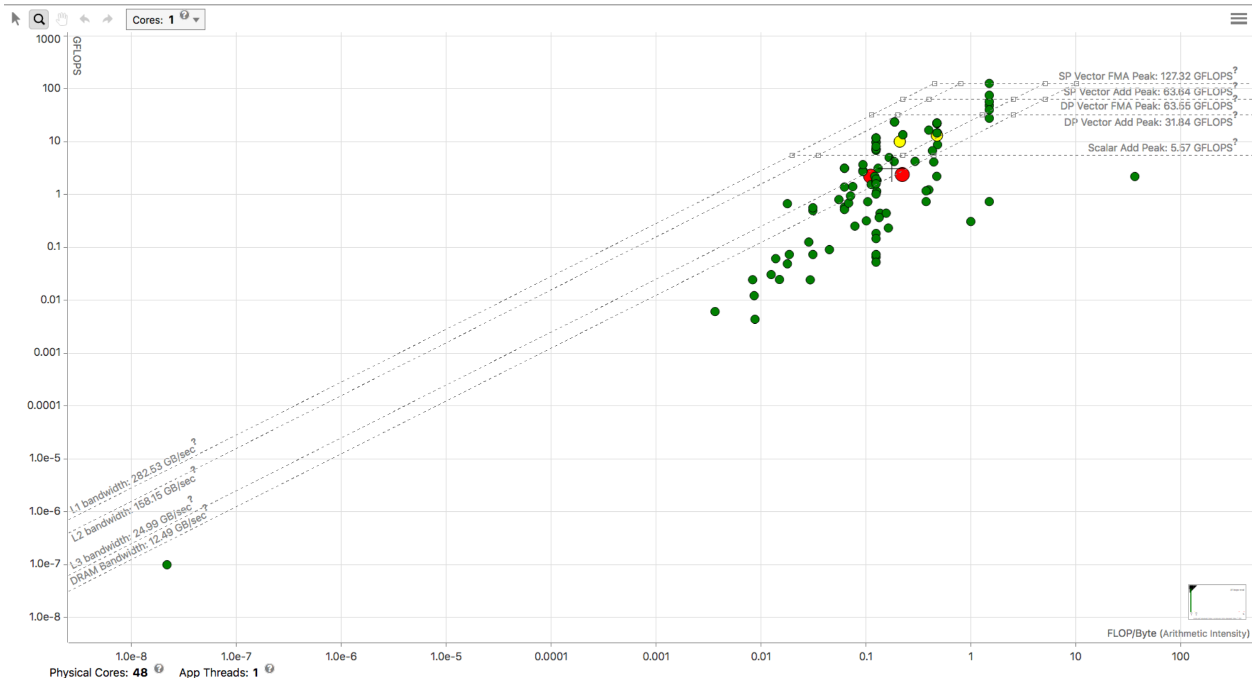


Figure 3: miniQMC roofline.

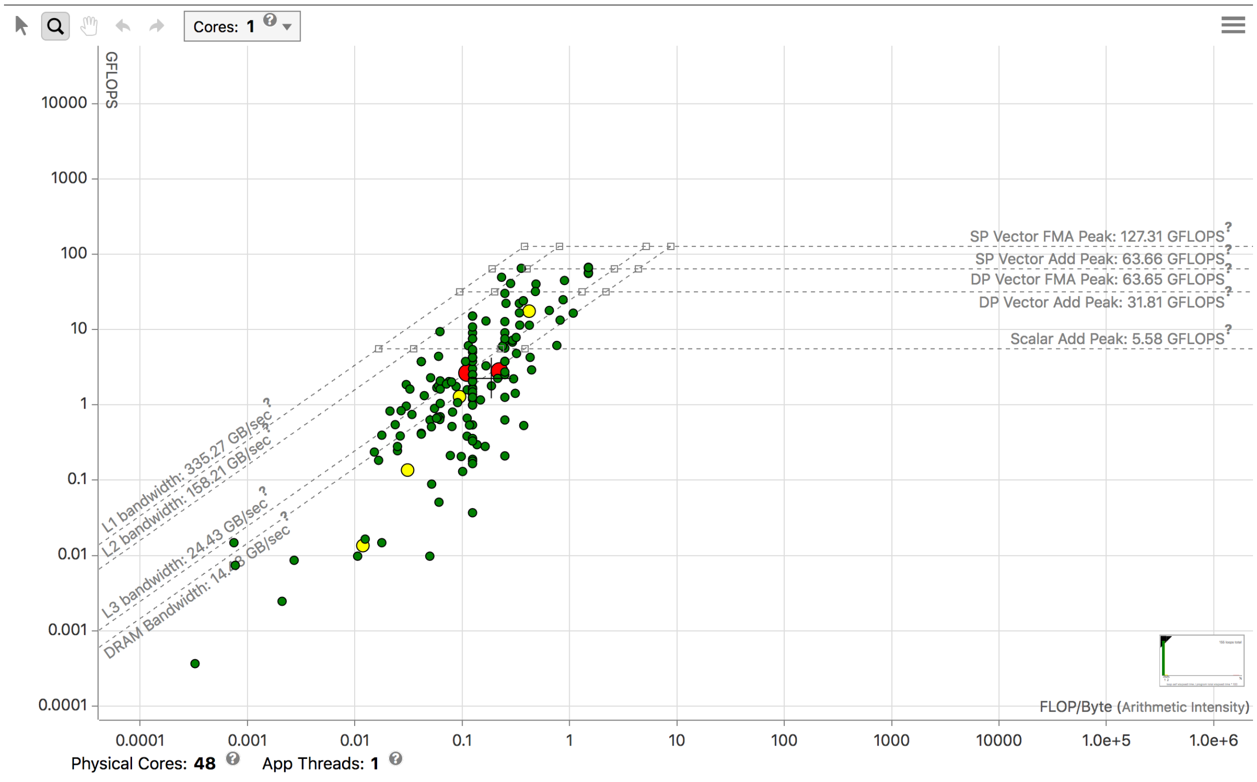


Figure 4: QMCPACK roofline.

are known to be computationally intensive. The functions that are in the left-most lower corner (memory-bound) of the plot are other spline functions and some distance functions, both potentially expected to be memory-bound. The functions laying in the middle region are both memory and compute-bound and include a mix of spline, determinant, and Jastrow functions.

Figure 4 is the roofline plot for QMCPACK. The first thing to note is that there are more functions represented in this graph than in miniQMC. There are still similar relative numbers of functions bound by DRAM, L3, and L2 bandwidths compared to miniQMC's roofline. In this plot, there are four yellow dots and two red dots. The red dots are again the MKL BLAS loops. The left red dot is bound by scalar add GFLOPs; the right red dot is bound by L3 bandwidth. Again, this seems plausible given the same explanation for MKL in miniQMC.

Looking at the functions represented by the yellow dots, the left-most lower yellow dot is a distance table function that is bound by DRAM bandwidth, which is not surprising. The next higher yellow dot is a two-body Jastrow orbital function called *evaluateLogandStore*. Looking at the code, this seems plausible that this particular function, which is called in the computation of a two-body Jastrow, is also bound by DRAM bandwidth. Two-body Jastrow computations are expected to be computationally bound, but they still operate on a large data set. The yellow dot that practically sits on the DRAM bandwidth line looks like it's a spline function in a solve loop. It is barely bound by L3 bandwidth (this kernel should be memory/cache bound). The highest and right-most yellow dot is a spline function that is part of the SPO kernel, and probably expected to be computationally bound.

In summary, the roofline plots of miniQMC and QMCPACK show boundedness of functions that seem to mostly match the expectations of how the four key kernels are bound. The rooflines of miniQMC and QMCPACK are fairly similar, showing similar relative numbers of functions that are memory, compute-and-memory, and compute bound.

3.4.3 Using Hardware Performance Counters to Understand Behavior Similarity

Prior to examining kernel-level behavior in more detail, we exposed several metics in our LDMS hardware performance counter sampler that help in understanding differences and similarities in proxy/parent performance and apply these to the entire execution of both applications. These metric definitions are taken largely from the Likwid [20] performance tool, with some metrics based on our own analysis. We include here only the metrics that show distinctive behavior between miniQMC and QMCPACK. All metrics are derived from events measured during the whole execution of each application; metrics presented are averaged over all the processes of the execution. Each process executes on a single core and the deviation in results between processes is within acceptable statistical limits (they are effectively equivalent, meaning no outliers).

We start by looking at throughput shown in Figure 5. MiniQMC has lower cycles per instruction (CPI) and cycles per micro-op (CPU), meaning a higher throughput than QMCPACK. Because their dynamic profiles are not very similar on a per-function basis, this may not be surprising. We need to look at these metrics for each of the kernels to really understand this. Ideal CPI for this architecture is 0.17 (i.e., can issue 6 instructions per cycle), so both applications are significantly higher than ideal (lower is better), indicating some stall behavior somewhere in the pipeline (see Section 3.5).

Figure 6 shows the branch behavior of miniQMC and QMCPACK. QMCPACK does more frequent branching and has a much lower branch misprediction rate than miniQMC. MiniQMC implements only one walker, so does not have the inner loop shown in Figure 1, line 2, which may account for some of the differing branch behavior since this is the main computational loop. To fully understand the difference in behavior here, we need to look more deeply into each key kernel

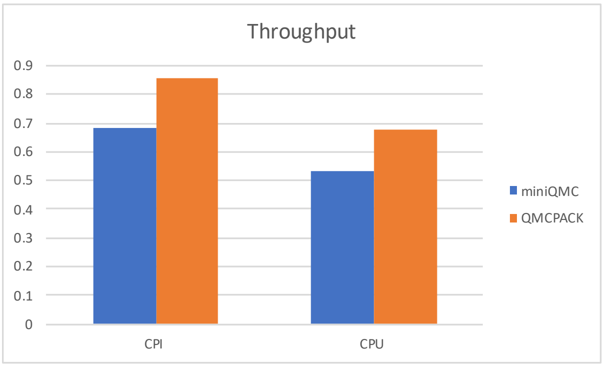


Figure 5: Cycles per instruction and per UOP.

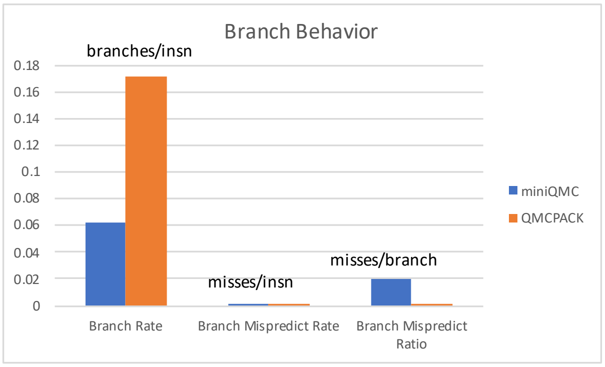


Figure 6: Branch behavior.

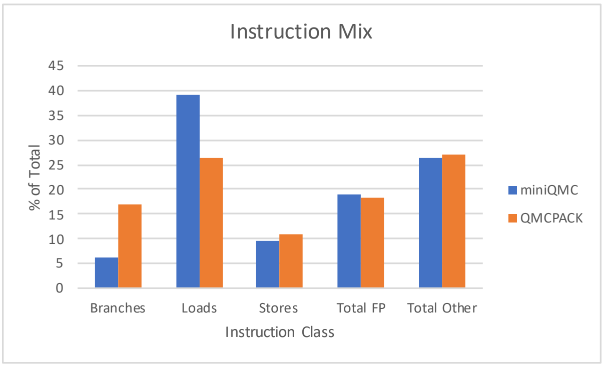


Figure 7: Instruction mix.

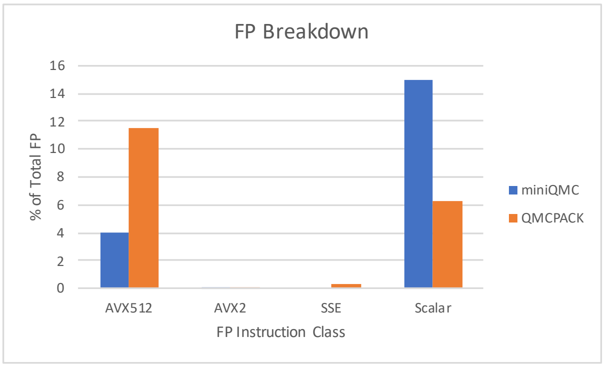


Figure 8: FP instruction mix.

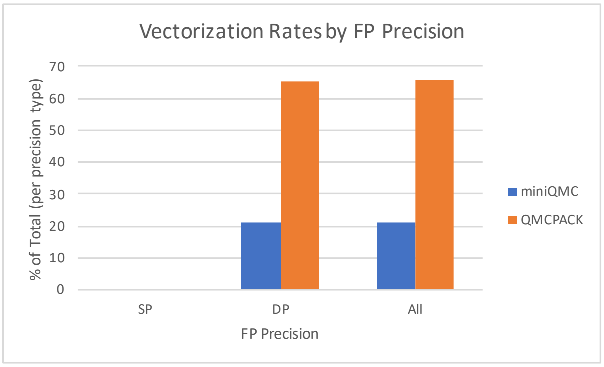


Figure 9: Vectorization rates.

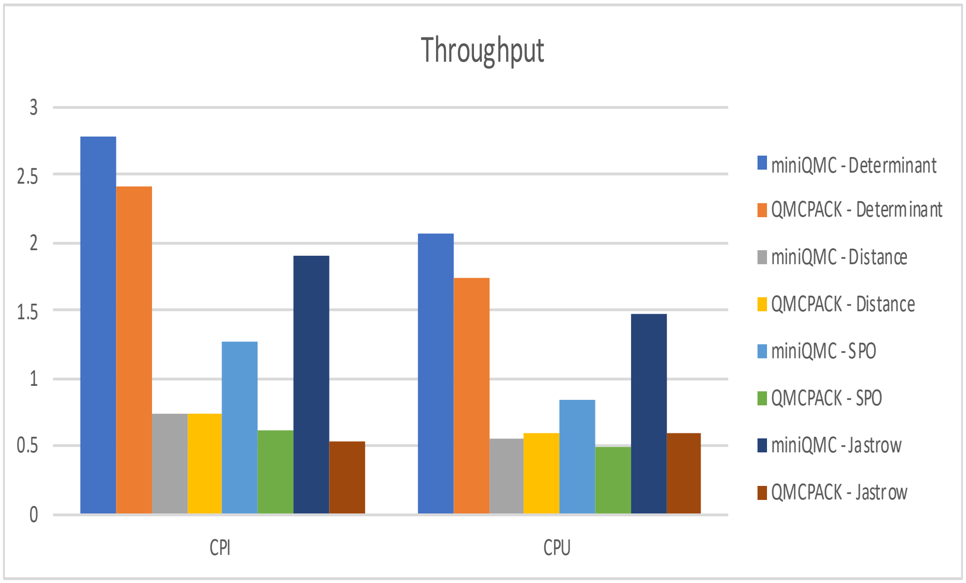


Figure 10: Kernel throughput.

and its associated functions.

Looking at the instruction mix in Figure 7 we see that the two applications are similar except in the percentage of loads. MiniQMC executes significantly more load instructions than does QMCPACK. This could be because QMCPACK does much more AVX512 than miniQMC as shown in Figure 8. AVX512 operates on wide data words, so the actual load instructions issued to memory will fetch wider words, meaning fewer overall memory instructions issued. Because miniQMC does relatively smaller number of AVX512 instructions, its scalar FP instruction percentage is large.

Figure 9 shows the vectorization rates for single-precision and double-precision FP instructions. All of the vectorization for both applications is done on the double-precision FP instructions. This is likely due to the data in the main kernels of both apps being double- rather than single-precision FP.

The underlying reasons for all of the above differences in the instruction mix are essentially unknown at this point. Detailed code inspection and comparison of the key kernels in miniQMC and QMCPACK has not been done. We examined these kernels at a high-level, but did not do a detailed comparison of data structures and all methods within these kernels. The per-kernel data (rather than per-function, since the correlation between functions in miniQMC and QMCPACK is not high) gives us a clue as to where and why these differences are realized.

3.4.4 Per-Kernel and Per-Function Behavior using Hardware Performance Counters

Because of the low correspondence between functions in the dynamic profiles of miniQMC and QMCPACK, we chose to first look at some performance metrics at the kernel level. We show results for the metrics that exhibit differences in kernel behavior that are helpful in understanding the fundamental differences between miniQMC and QMCPACK. In general, the largest differences across all of the performance metrics we track are generated by the *Jastrow* and *SPO* kernels. We see large differences in the *Determinant* kernel only for cache and memory-related metrics.

Figure 10 shows the throughput in terms of CPI (cycles per instruction) and CPU (cycles per micro-op). The largest difference between miniQMC and QMCPACK is in the *Jastrow* kernel, which also exhibits no function similarity at all. The next largest difference is seen in the *SPO*

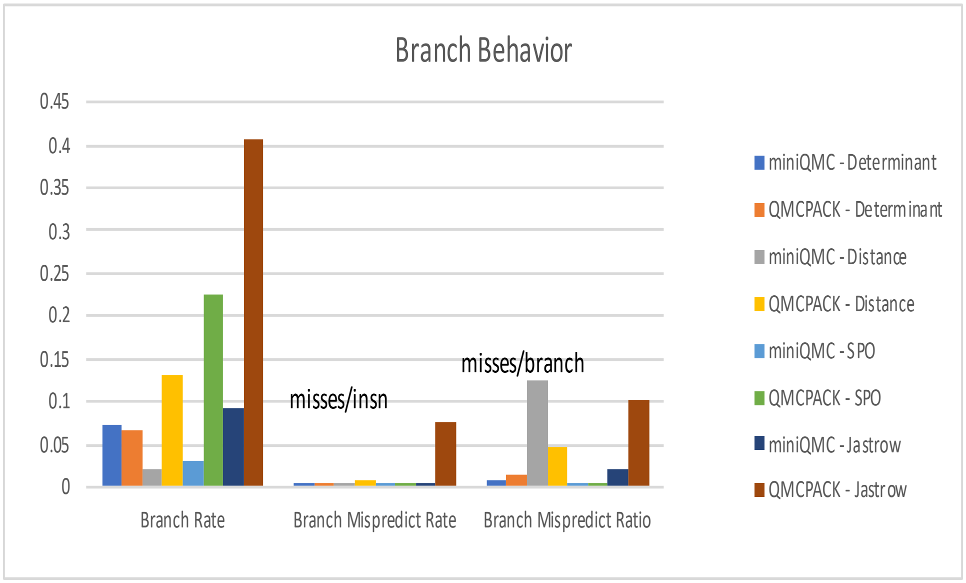


Figure 11: Kernel branch behavior.

kernel, again, where no function similarity is seen in the profile.

Figure 11 indicates that the largest differences in branches per instruction between miniQMC and QMCPACK are again coming from the *Jastrow* and *SPO* kernels. The misses per branch metric is showing the largest differences in the *Distance* and *Jastrow* kernels.

The final metrics that show significant and interesting differences between the miniQMC and QMCPACK kernels are related to cache behavior–cache misses and bandwidths. This is interesting because the largest differences are seen in the *Determinant*, *Distance*, and *Jastrow* kernels. The *Determinant* and *Distance* kernels have the most function and profile similarity of the four kernels. Because we see these differences only in cache (and memory, although we don't show these results here) metrics, a question arises pertaining to the data structures and layout in these functions comprising these kernels across the two applications. Maybe the data structures and layouts differ, and/or perhaps the working set size is different across these functions. Further investigation is needed here to tease this out. We do have per-function data for all of the kernel functions and need to analyze that to more precisely understand these kernel differences.

In Figure 12, we show cache miss ratios (although the cache miss rates show the same relative differences). The *Determinant* kernel shows the largest percent difference between miniQMC and QMCPACK for L1D (data cache) and L3. For the L2 cache, the largest difference is seen in the *SPO* kernel.

Figure 13 shows cache misses per thousand instructions for the four key kernels of miniQMC and QMCPACK. MPKI is a very reliable metric because both the numerator and denominator comprise reliably accurate hardware performance counter events. This shows very large percent differences in the *Determinant* kernel between the two applications. Again, this may be due to a data layout or working set size discrepancy.

Cache bandwidth data again shows the *Determinant* kernel being an outlier. All of the other kernels show very low cache bandwidth utilization, but this corresponds to both the cache miss rate and cache MPKI data. The *Determinant* kernel particularly for QMCPACK has relatively high bandwidth usage primarily between the L1 and L2 cache. Note that this is per-process (per-core) bandwidth data. According to [32], sustainable read/write cache bandwidth between L1 and L2

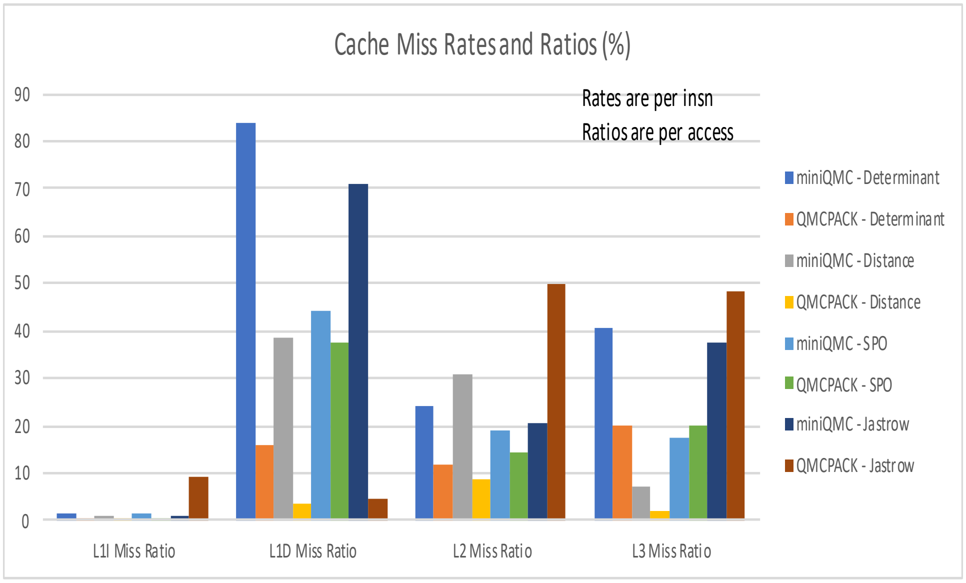


Figure 12: Kernel cache miss ratios.

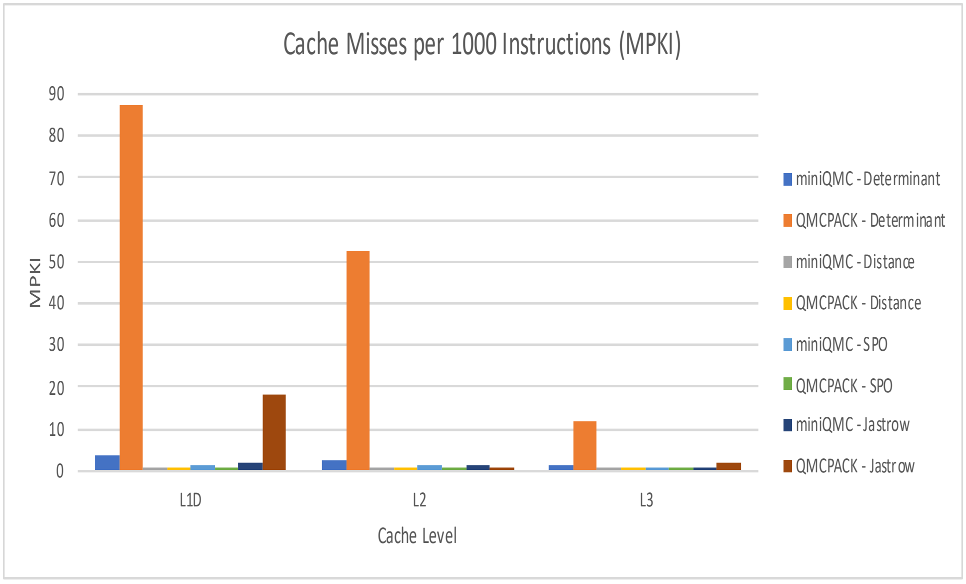


Figure 13: Kernel cache MPKI.

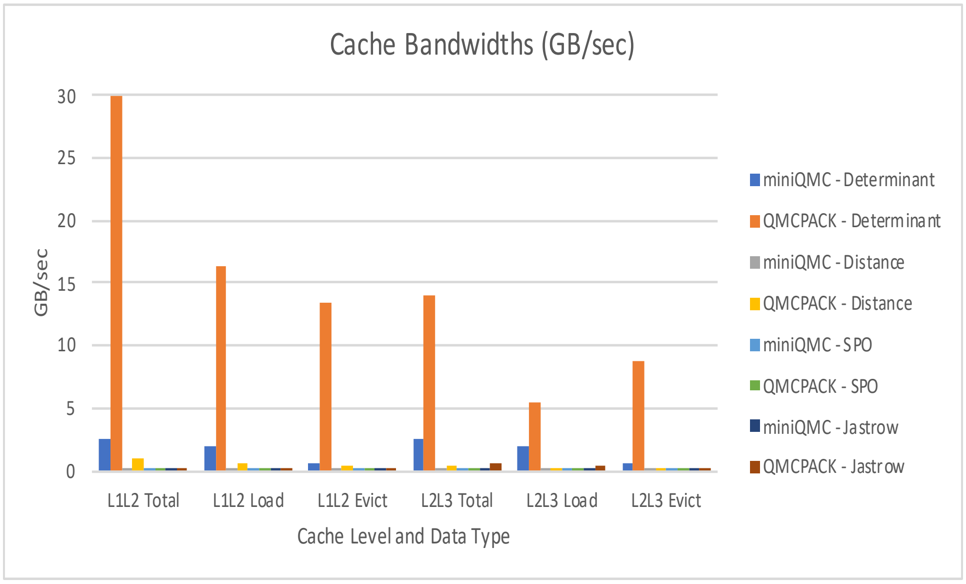


Figure 14: Kernel cache bandwidths.

is around 60–80/20–40 GB/sec per-core for our platform. The range accounts for the data size, ranging from small to large. Sustained per-core read/write bandwidth between L2 and L3 for our platform is approximately 30–70/18-22 GB/sec. So the *Determinant* kernel does not exceed bandwidth limits, but pushes the bandwidth the most out of all the kernels.

The takeaway from this kernel data is that we clearly see the function profile differences in the *Jastrow* and *SPO* kernels for various non-cache-related metrics. This is likely where the overall execution differences are generated. However, we also see large differences in cache and memory-related metrics largely in the *Determinant* kernel, which may indicate a mis-match in data layout, data structures, and/or working set size between the two applications, in spite of executing the same problem.

3.5 Identifying Hardware-Level Bottlenecks in Applications using TMA

Top-Down Microarchitecture Analysis (TMA) [36] is a hierarchical methodology that uses cycle counts to identify which components in the architecture (if any) are causing performance bottlenecks. This hierarchy is shown in Figure 15. Each category in the hierarchy represents an area in the microarchitecture that demonstrates a hardware bottleneck in that the method contributes to it a larger than expected number of total execution cycles. The expected number of execution cycles is assigned to each component by a defined threshold. This threshold is set for definitions pertaining to HPC applications.

The method is hierarchical in that starting at the top, if a bottleneck is identified, the user can choose to drill-down to the next level set of metrics through the LDMS hardware counter sampler. A user can continue to drill down the hierarchy until no bottlenecks are identified. At that point, the user can start by addressing the lowest-level bottleneck first.

Figure 16 shows TMA for miniQMC and QMCPACK, starting at the top level. At Level 1, we see that both applications are backend bound, meaning there's a performance issue in the microarchitecture backend, which indicates that uops are not being delivered to the issue pipe due to lack of resources for accepting them in the backend. This could be due to either execution stalls due to the memory subsystem (memory bound) or stalls due to poor execution port utilization

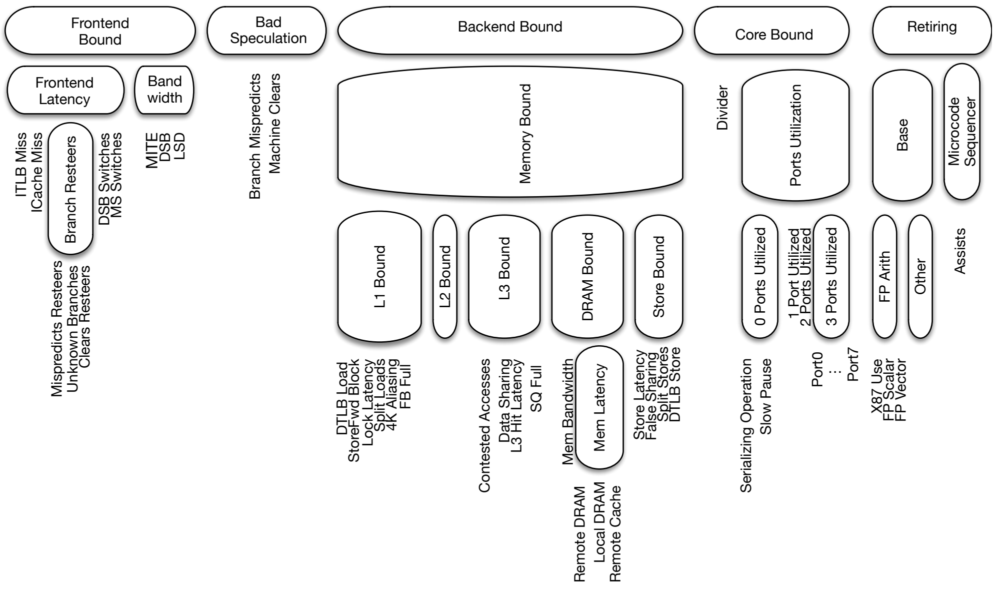


Figure 15: Top-Down microarchitecture analysis.

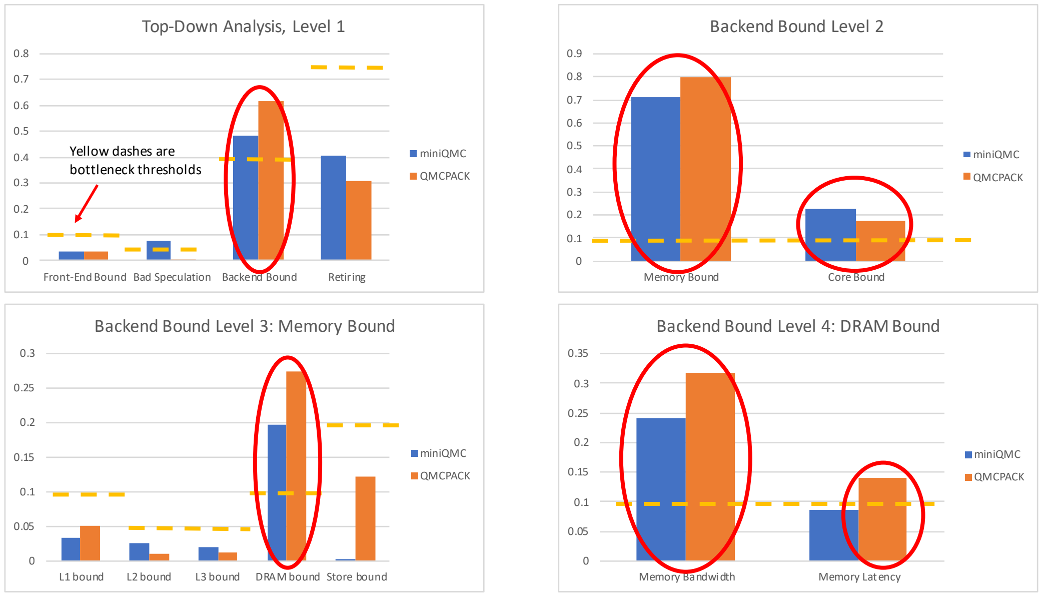


Figure 16: TMA backend bound, miniQMC and QMCPACK.

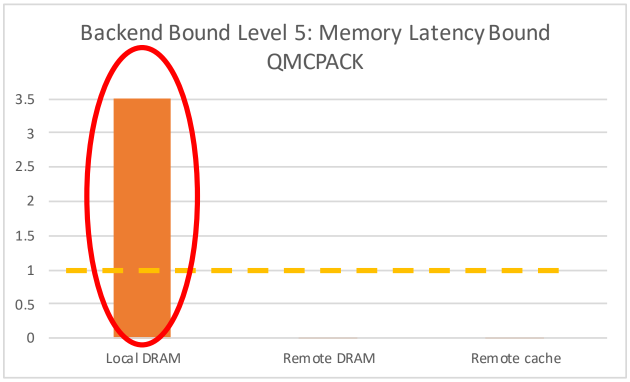


Figure 17: Memory latency causes, QMCPACK.

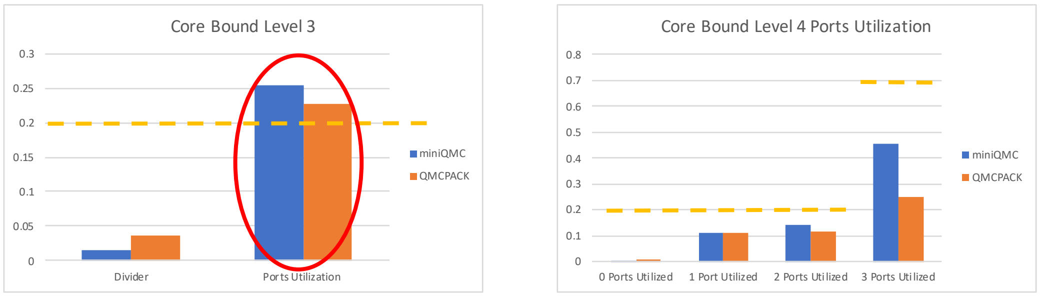


Figure 18: TMA core bound, miniQMC and QMCPACK.

(core bound). Note that miniQMC is also bound on bad speculation, which we'll address later.

Drilling further down the backend hierarchy, we see that at level 2, both applications again exhibit the same behavior in that they're both memory bound and slightly core bound. At the third level of the backend bound hierarchy, we see that execution stalls are occurring due to DRAM issues for both applications. The fourth level shows that the DRAM causes execution stalls due to both its bandwidth (for miniQMC and QMCPACK) and its latency (QMCPACK). QMCPACK has more complexity in function than does miniQMC. So it being also latency bound where miniQMC is not is not necessarily surprising. The takeaway here is that both the proxy and the parent appear to be identifying performance degradation due to the same hardware resources.

If we drill down on the memory latency bottleneck, we see in Figure 17 that the latency of local DRAM accesses is the root cause of the execution stalls due to memory latency in the backend pipeline of this architecture.

If we drill down on the core bound bottleneck identified for both apps in Figure 18, we see that both applications indicate a bottleneck in terms of execution port utilization. This indicates that maybe execution port over-utilization could be causing excessive execution stalls in the backend. However, drilling down further and looking at how often multiple ports are utilized, it doesn't look like this is a problem. We wonder if there could still exist a problem, but the HPC threshold is actually too low to flag it in the methodology. We are currently exploring these sorts of issues in the methodology.

Pertaining to miniQMC's bad speculation identified bottleneck, if we look at the lower level metrics in Figure 19, we see that branch mispredicts could potentially be causing a performance degradation. Cycles are lost fetching the mispredicted execution path and from the subsequent pipeline flush. Why this happens in miniQMC and not in QMCPACK is not completely clear.

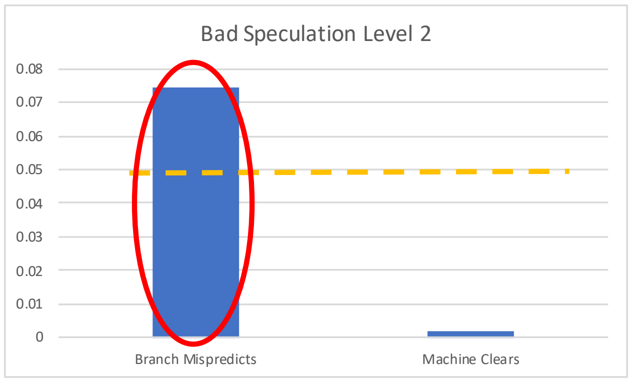


Figure 19: Bad speculation, miniQMC.

miniQMC has much lower number of branches per instruction compared to QMCPACK and also has comparatively much larger basic blocks. This seems like these characteristics would lead to better branch prediction rates in miniQMC rather than worse. It could be that since miniQMC only encapsulates the key kernels in QMCPACK that we see only that behavior and there is enough additional code in QMCPACK with very different behavior from the kernels that dominates the overall branching behavior in QMCPACK. Further investigation is needed to more accurately identify this observed difference in branching behavior. We are presently analyzing per kernel TMA data to help determine this.

3.6 Is miniQMC a Good Proxy for QMCPACK?

From all of the analysis presented above, our conclusion is that miniQMC is a good proxy for QMCPACK when evaluated at the kernel or whole-application level. At the function level, these applications are quite different. At the kernel level, we see similar behavior as presented in Section 3.4.4. Finally, looking at the whole execution and the hardware bottlenecks realized, we see fairly good similarity. In the near future, we will apply our similarity method outlined in [6] to see if this algorithm shows statistical similarity between miniQMC and QMCPACK.

4 miniVite and Vite Performance Assessment

A graph is an ordered pair comprising of a set of vertices and edges. Graph theory is the study of graphs and its origin dates back to Euler's solution of the puzzle of Konigsberg's bridges[2]. The study of networks has brought significant advances to our understanding of complex systems. Graphs are most relevant in representation of these complex systems, whether they are biological, social, technological, or information networks, as they can be used to analyze the pairwise interaction between objects.

4.1 Graph communities

Clusters or communities in graphs are groups of nodes that are densely connected to each other but separeately connected to other nodes outside the cluster. They can be considered as fairly independent compartments of a graph, playing a similar role. The word community itself refers to a social context, i.e., people naturally tend to form groups; proteins interact with only certain other types of proteins [31], etc. Detecting communities in graphs is of great importance and it finds many applications. For example, detecting modules[3] in protein-protein interaction networks is associated with cancer and metastasis [37] [24], web graphs can be used to improve search results by eliminating artificial clusters created by click farms and is widely used by search engines [26], community detection in social media can be used to understand behavioral similarities [29], graph models are used to analyze vulnerabilities in electric power networks [15]. Community detection is also used in machine learning, especially for unsupervised learning in clustering and classification of data sets [30].

The examples above indicate that graph community detection is an important problem. Because of the massive amounts of data generated by real world networks, graph community detection is also computationally expensive. Searching protein-protein interaction networks to look for effective drugs for cancer or analyzing technological networks like the internet will require exascale computing power.

Although a human can intuitively detect communities by looking at a graph, the concepts of community and partition are mathematically not well defined. Various measures have been proposed to evaluate the goodness of partitioning produced by a community detection method. One such measure is the *modularity* (Q), which is given by

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) \delta(c_i, c_j),$$

where A is the adjacency matrix (A_{ij} is one if there is an edge between vertex i and j, zero otherwise), m is the sum of all the edge weights, k_i is the weighted degree of vertex i, c_i is the community that contains vertex i, $\delta(c_i, c_j) = 1$ if $c_i = c_j, 0$ otherwise. Modularity is a statistical way to quantify the goodness of a given community-wise partitioning on the basis of the fraction of edges that lie within the community. Although an effective metric, modularity suffers from a resolution limit and modularity optimization is an NP complete problem.

Despite the limitations, modularity is widely used as a measure of goodness in practice and many efficient heuristics have been developed. One such heuristic is the Louvain method. It is a multi-phase, multi-iteration heuristic that starts from an initial state of communities and iteratively

[2]The seven bridges of Konigsberg is a historically notable problem in mathematics. Its negative resolution by Leonhard Euler is 1736 laid the foundations of graph theory and prefigured the idea of topology [7].

[3]A 'module' refers to the conserved part of a protein that can function independently, repeatedly found in diverse proteins.

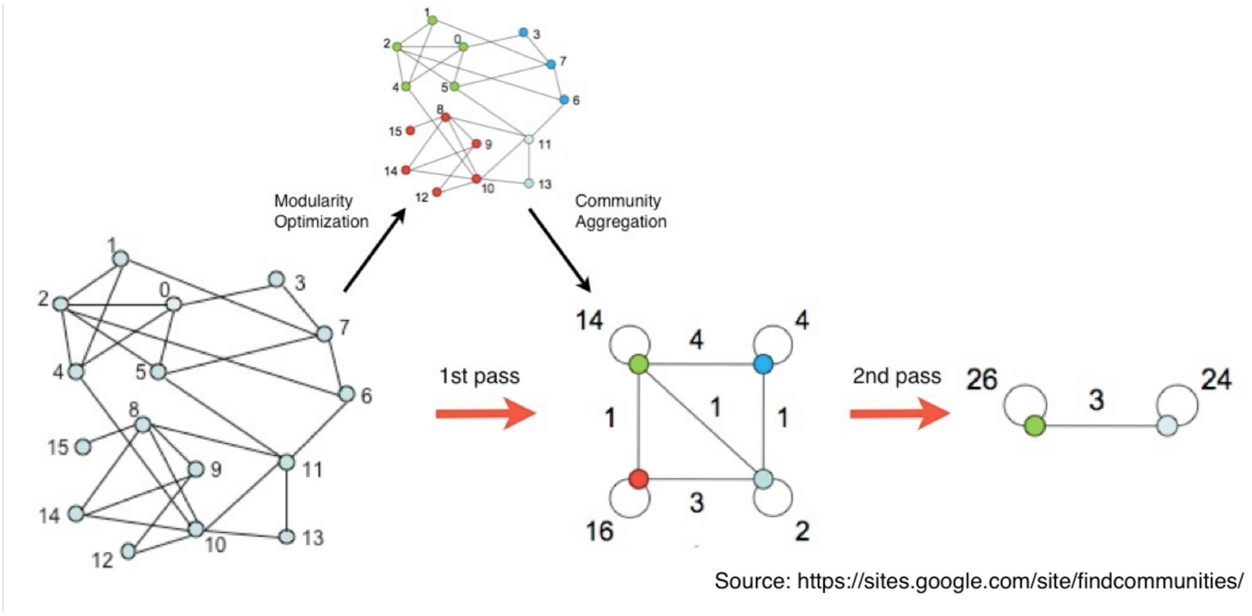


Figure 20: A graphical representation of the Louvain method.

improves the quality of community assignment until the gain in quality becomes negligible. From a computation standpoint, this translates into performing multiple sweeps of the graph and graph coarsening. The Louvain method is fast and produces high quality output.

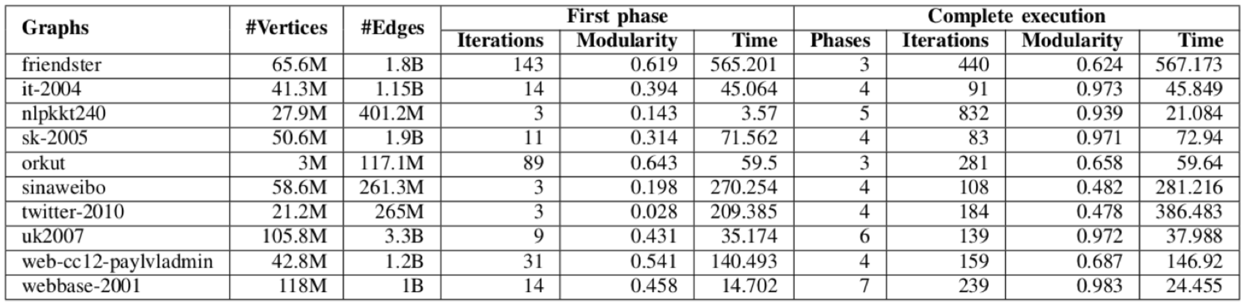
4.2 Louvain Method

The Louvain method works by initially assigning all vertices to their own community. In each phase, the change in modularity (ΔQ) that would result from moving each vertex to each of its neighbor communities is calculated. The vertex is then assigned to the community that yields the maximum ΔQ (as long as ΔQ is positive). This assignment process continues iteratively until the gain in modularity between two successive iterations is smaller than a threshold value. At the end of the phase, each community is collapsed to a single vertex and the graph is reconstructed to begin a new phase. This process is repeated until the gain in modularity is nominal. The algorithm is shown graphically in Figure 20. Grappolo [14] is a shared memory multi-threaded implementation and Vite [11] is the distributed memory implementation of the Louvain algorithm.

4.2.1 Vite

Vite implements a parallel Louvain algorithm. In the parallel version of Louvain, the vertices and their edge lists are evenly distributed across multiple processes. Each process also keeps track of "ghost" vertices that represent those remote vertices that have edges connected to vertices in the process. Each process performs its own Louvain iteration, updates the community assignment, and communicates updates to the ghost vertices. Once the modularity converges, the graph is reconstructed and the resulting set of vertices is once again distributed across processes to start a new phase. This process is repeated until a certain threshold is met.

Figure 21 shows the computation times for different real world graphs tested on the Haswell nodes of the NERSC Cori system. We can see that the first phase of the Louvain method takes the vast majority of the total run time. The authors in [11] also indicate that Vite shows good strong



Graphs	#Vertices	#Edges	First phase			Complete execution			
			Iterations	Modularity	Time	Phases	Iterations	Modularity	Time
friendster	65.6M	1.8B	143	0.619	565.201	3	440	0.624	567.173
it-2004	41.3M	1.15B	14	0.394	45.064	4	91	0.973	45.849
nlpkkt240	27.9M	401.2M	3	0.143	3.57	5	832	0.939	21.084
sk-2005	50.6M	1.9B	11	0.314	71.562	4	83	0.971	72.94
orkut	3M	117.1M	89	0.643	59.5	3	281	0.658	59.64
sinaweibo	58.6M	261.3M	3	0.198	270.254	4	108	0.482	281.216
twitter-2010	21.2M	265M	3	0.028	209.385	4	184	0.478	386.483
uk2007	105.8M	3.3B	9	0.431	35.174	6	139	0.972	37.988
web-cc12-paylvladmin	42.8M	1.2B	31	0.541	140.493	4	159	0.687	146.92
webbase-2001	118M	1B	14	0.458	14.702	7	239	0.983	24.455

Figure 21: Comparison of the first phase of the Louvain method with complete exectution for real world inputs on 1K processes of NERSC Cori. Source: [12]

and weak scaling. They show that the method is communication intensive and about 60% of the time is spent in communication. These observations strongly influenced the design of the proxy app for Vite.

4.2.2 miniVite

miniVite [12] is a proxy app intended to capture most of the computation and communication operations in Vite. miniVite implements a single phase of the Louvain method (without rebuilding the graph). Since the first phase of the Louvain method takes most of the execution time, omitting graph reconstruction and subsequent phases significantly simplifies the code but preserves the most time consuming portion of the algorithm. miniVite is capable of generating synthetic random geometric graphs in parallel and can add random edges across processes. However, real world graphs can also be used as inputs similar to Vite. In addition to capturing the communication aspects of Vite, parts of the code in miniVite have multiple communication options that can be selected during compile time. By processing the same input graphs as the parent application and having a largely intact Louvain method, miniVite is designed be representative of and a testing ground for the most intensive portions of Vite.

4.3 Methodology

Vite and miniVite were tested using real world graphs as inputs, and the following subsections provide detailed description of the testing process. The authors in [12] report that non-blocking iSend/iRecv and collective communications are about 5% faster than blocking Send/Recv and RMA. Therefore, the non-blocking iSend/iRecv was used in miniVite for the assessment.

4.3.1 Obtaining, building and running the applications

miniVite version 1.0:
Source location: https://github.com/Exa-Graph/miniVite
Vite version 1.0:
Source location: http://hpc.pnl.gov/people/hala/grappolo.html

Building and running the code is straight forward. Example run scripts can be found in the respective README files that accompany the source codes.

4.3.2 Problem sizes and scaling information

The 'Road_USA' graph was used for assessment, which consists of 23,947,347 rows and 23,947,347 columns with 57,708,624 non-zero nodes. This graph represents the network of roads in the United States and was used in the DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science) challenge [1] for benchmarking applications and is available for use in the public domain.

The authors in [11] and [12] demonstrate that both Vite and miniVite have similar strong and weak scaling performance. The scaling analysis was performed using different real world graphs as inputs. The synthetic graph generator in miniVite can also be employed in scalability analysis.

The largest real-world graph that has been tested on both miniVite and Vite is uk2007 (3.3 billion edges). However, other larger graphs could be used. The largest synthetic file generated using miniVite was about 2.14 B vertices and 27.8 B edges. To give an idea of the file I/O overhead for reading a real-world graph file, Vite/miniVite takes about 2–4 sec to read a 55 GB binary file on NERSC Cori with Burst buffer or Lustre file striping (about 25 OSTs, default 1M blocks). Subpar file I/O can negatively affect performance for large real-world graphs.

4.3.3 Tools used

Vite and miniVite were profiled using Intel VTune Amplifier with the 'Road_USA' graph as input. 32 MPI processes were used with 8 OpenMP threads per process. Advanced hotspot analysis and Memory access analysis were conducted and performance metrics were recorded. Intel Advisor was used to generate the roofline and to measure the FLOPS of different functions. The data analysis and collection with Advisor has a significant overhead and we were only able to work with one MPI process using a single thread.

Compiler: Intel 18.0.1.163 with cray-mpich 7.7.3.

Performance Analysis Tools:

Intel VTune 2018: Intel VTune Amplifier XE is a performance analysis tool that helps in identification of performance bottlenecks of applications. To compile codes to work with VTune, use the "-g" flag during compilation to generate source-level debugging/symbol information in the executable. If using compiler wrappers, use the "-dynamic" flag during linking for visibility of symbols. Other performance analysis modules such as Darshan are unloaded as they conflict with Intel performance tools.

Intel Advisor 2018: Intel Advisor analyzes the vectorization and threading, and identifies factors that are blocking effective vectorization and threading design. Use the "-g" and "-dynamic" flags as before.

4.4 Results and Analysis

As we saw from Figure 21, the first phase of the Louvain method is the most expensive in terms of overall execution time. miniVite implements only the first phase of the Louvain method, therefore, capturing the major computational and communication aspects of Vite. This feature is reflected in the execution times of the proxy and parent application, and can be used for quick validation of results and the build system.

Source Function Stack	CPU Time (in s)	
	Vite	miniVite
Total	7213.531	7327.760
[Outside any known module]	3574.084	3695.082
MPIDI_CH3I_Progress	822.347	807.122
poll_active_fboxes	489.708	515.646
_kmp_yield	364.727	375.626
distExecuteLouvainIteration	158.690	161.645

Table 11: Advanced hotspots: Expensive functions (both Vite and miniVite) and their CPU core times. (Times are the sum over all active threads.)

4.4.1 Execution Time

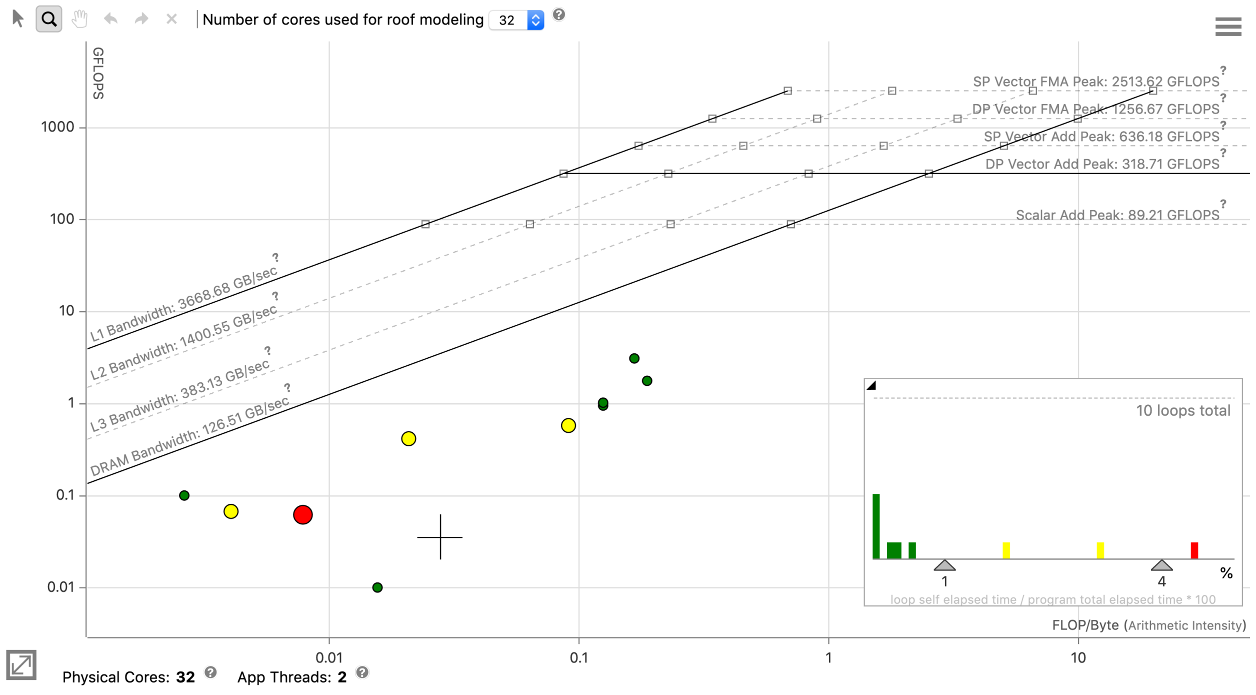
We first compared the execution time for the first phase of the Louvain method using Vite to the execution time of miniVite. On the Haswell nodes of NERSC Cori we found miniVite converged after 16 iterations in a run time of 98.3 seconds. The first phase of Vite also converged in 16 iterations and ran for 103.5 seconds. However, when we exclude the graph reconstruction time (recall, miniVite does not implement reconstruction), the Louvain iteration portion of Vite's first phase ran for 91.5 seconds. In both cases the modularity converged to 0.62. Clearly the agreement in run time is excellent.

4.4.2 Dynamic Profiling

Dynamic profiling of Vite and miniVite was performed using Intel VTune Amplifier and the results are shown in Table 11. The table lists the 5 most expensive functions and their CPU times for both Vite and miniVite. Note that the CPU time is the sum of the individual core times over all threads during which the application is actively executed, and is thus greater than the wall-clock execution time. We can see from Table 11 that the same 5 functions are consuming most of the time in both the proxy and the parent application and their cores times are quite similar. This indicates that miniVite effectively captures the major characteristics of Vite. The table entry for [Outside any known module] represents the fact that VTune is unable to resolve some library symbols. The command line output of VTune indicates the symbols that were not resolved belong mostly to the Cray MPICH library. This shows that both Vite and miniVite are communication intensive. We can also see that the MPI function `MPIDI_CH3I_Progress` constitutes a fair chunk of the execution time. The `MPIDI_CH3I_Progress` is the MPI Process Engine, which is a series of lists for MPI internal tasks such as: send, receive, collectives, and so on, designed specifically for asynchronous communications. The `poll_active_fboxes` function polls the active fast-boxes. Each pair of MPI processes on the same computing node has two shared memory fast-boxes, for sending and receiving eager messages. The function `_kmp_yield()` is an OpenMP library function used to direct threads to yield to other threads, which indicates some OpenMP overhead in the applications. The Louvain iteration, implemented in the function `distExecuteLouvainIteration`, by itself, is not very expensive.

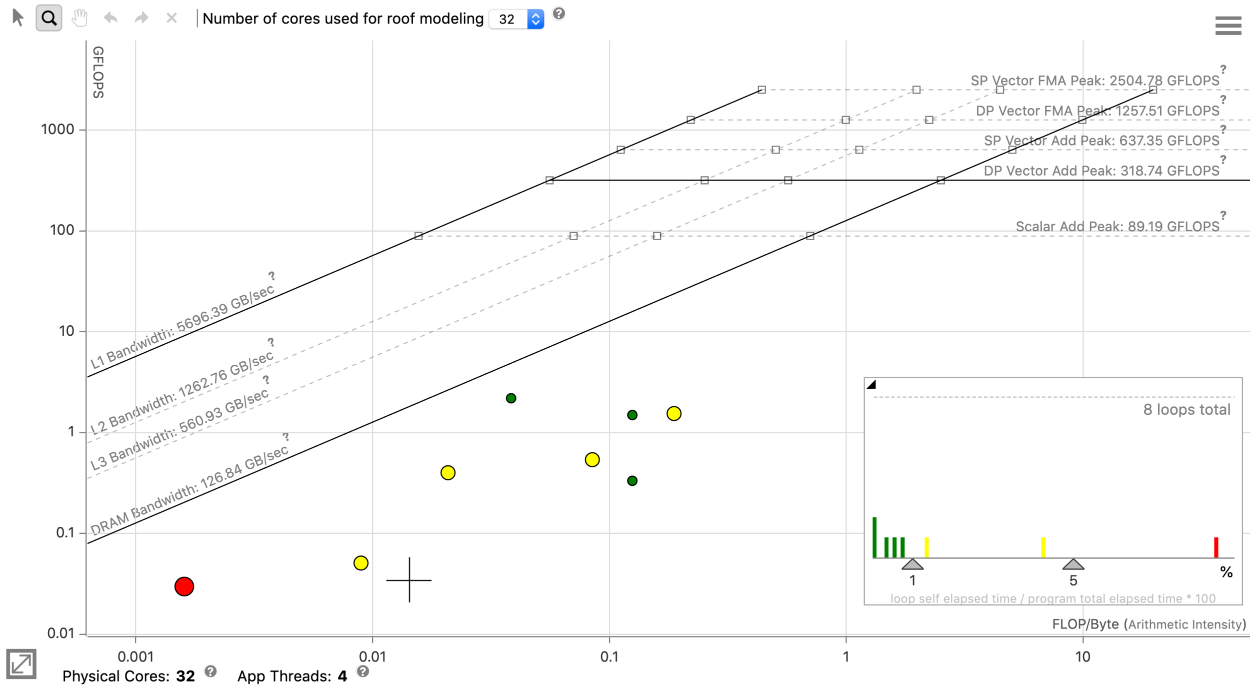
4.4.3 Roofline Model

The roofline chart for Vite and miniVite are shown in Figure 22 and Figure 23, respectively. In general, there is more room for improvement if the dot (functions and loops) is farther away from


Number of cores used for roof modeling 32

Physical Cores: 32 App Threads: 2

Figure 22: Vite roofline.


Number of cores used for roof modeling 32

Physical Cores: 32 App Threads: 4

Figure 23: miniVite roofline.

its limiting roofline, which is the nearest line above the dot. Intel Advisor recognizes the same 4 loops (red and yellow dots on the chart) in Vite and miniVite that are the best candidates for optimization and are shown below:

Loop in distExecuteLouvainIteration: Scalar loops present
Loop in distBuildLocalMapCounter: Outer loop not vectorized
Loop in distUpdateLocalCinfo: Non-vectorized loop due to data dependencies
Loop in distGetMaxIndex: Outer loop not vectorized

We can see from Figure 22 and Figure 23 that these loops and functions have comparable arithmetic intensities and performance. Some of the non-vectorized loops can be vectorized using explicit SIMD directives and the data dependencies have to be analyzed for possible resolution. Even though there is some potential for optimizing computations in Vite and miniVite, the overall performance gains will be nominal ($< 5\%$) as the communication dominates the overall time as indicated by the dynamic profile of the applications in Section 4.4.2.

4.4.4 Per-Function Characteristics and Memory Access Analysis

The distributed Louvain method is communication intensive. However, we can still quantitatively analyze the performance of Vite and miniVite in terms of arithmetic intensity. Per-function metrics were collected along with the roofline chart using Intel Advisor. The most computationally intensive function in both Vite and miniVite is `distGetMaxIndex`. The measured FLOP rates of 2.174 GFLOPS and 2.117 GFLOPS in Vite and miniVite, respectively, agree well. Moreover, since a single node of a Haswell system is capable of achieving upwards of half a TeraFLOP, these numbers also verify that the applications are emphatically not compute bound.

Memory access analysis was performed using the Intel VTune Amplifier. The memory access analysis [2] uses hardware event-based sampling to collect data. Metrics including number of loads and stores, last level cache misses, the fraction of cycles spent waiting due to demand load and store instructions (Memory bound), how often the CPU is stalled on main memory (DRAM bound), and average latency are collected. Table 12 shows the system bandwidth utilization by Vite and miniVite. We can see that the maximum and average bandwidth utilization by Vite and miniVite are almost identical. Even though the maximum observed bandwidth is close to the maximum bandwidth of the platform, the average sustained bandwidth is very low. Therefore, the applications are not significantly limited by the memory bandwidth. VTune indicates that Vite is memory bound 22.9% and miniVite is memory bound 28.3% of pipeline slots[4]. Further drilling down into the cache bandwidth utilization, VTune identifies 3 dominant hotspots that were stalled due to missing L1 data cache and they are shown in Table 13. Again, we can see that the L1 boundedness of Vite and miniVite are similar. The analysis also showed that none of the L1 bound hotspots are limited by the DRAM bandwidth, indicating a very small potential performance gain by memory optimization.

4.5 Summary

miniVite implements the first phase of the distributed Louvain method as it takes most of the total time for execution. Profiling and assessment of the proxy-parent pair indicated that miniVite and Vite behave similarly with identical hotspots. Both the proxy and the parent application are communication intensive and are well optimized. The assessment suggests that oportunities for performance gain by further improving computations and memory accesses are very limited.

[4]A pipeline slot represents hardware resources needed to process one uOp

Bandwidth Domain	Platform Maximum	Observed Maximum		Average Bandwidth	
		Vite	miniVite	Vite	miniVite
DRAM, GB/sec	128	97.6	101.3	4.926	3.736
DRAM Single-Package, GB/sec	64	48.8	50.7	2.586	2.044
QPI Outgoing, GB/sec	68	15	13.8	0.265	0.148

Table 12: System bandwidth utilization of Vite and miniVite.

Function	L1 Bound	
	Vite	miniVite
distExecuteLouvainIteration	20.2%	21.3%
_int_malloc	22.3%	19.7%
distBuildLocalMapCounter	14.4%	13.3%

Table 13: Data cache (L1) boundedness for Vite and miniVite.

Through this methodology we are able to confirm that miniVite not only captures the behavior of Vite but also manages to capture the same performance and quantitative characteristics of Vite.

5 PICSARlite and WarpX Performance Assessment

WarpX [35] is a software package for exascale modeling of plasma-based particle accelerators. Its goal is to assist the development of compact and affordable high-energy physics colliders.

The components of WarpX are:

- **PICSAR** [4], a library of Fortran 90 subroutines for Particle-In-Cell (PIC) operations;
- **AMReX**, a software framework written in C++ and Fortran 90 for massively parallel, block-structured adaptive mesh refinement (AMR) applications, including ExaSky, Combustion-Pele, ExaStar, and MFIX-Exa as well as other ECP applications;
- **WarpX-Source**, a C++ driver and set of subroutines for interfacing with PICSAR and AMReX.

The PICSAR library includes a stand-alone driver that can be used instead of WarpX-Source to call PICSAR subroutines without AMReX, if mesh refinement is not needed.

PICSARlite is a trimmed down version of PICSAR (including its stand-alone driver) that is generated from the PICSAR code by running a script that selects only a subset of the options available in PICSAR for particle pusher, deposition, Maxwell solver, etc. The purpose of PICSARlite is to have a smaller code to work with (19,702 lines, vs. 70,181 lines for the full PICSAR) while maintaining essential functionality. The command to invoke the script with the particular settings we use to generate PICSARlite is:

```
$ python utils/generate_miniapp.py --geom 3d --solver fdtd --pusher boris
  --depos direct --charge off --order 2 --laser off --optimization off
  --diags off --errchk off
```

These options are for 3D geometry, finite-difference time-domain solver, Boris particle pusher, direct current deposition, no charge deposition, second-order current deposition, no laser injection, no optimization, no diagnostics, and no flagging of warning messages when code is trying to use unavailable routines.

5.1 Algorithms and Key Kernels

The Particle-In-Cell (PIC) method is widely used to simulate the motion of charged particles in a plasma environment. PIC codes use both particle and grid data structures to represent the physics of the problem. The electric and magnetic fields are solved on a grid, and particle trajectories are computed according to those fields. Figure 24 shows the steps that a PIC code cycles through at each time step:

1. Push particles: Using the electric and magnetic fields at each particle position, update the position and velocity of the particle using the Newton-Lorentz equations.
2. Deposit charge and/or current densities through interpolation from the particle distributions to points on the grid.
3. Evolve Maxwell's equations (for electromagnetic) or Poisson's equation (for electrostatic) on the grid.
4. Gather forces: Interpolate the fields from the grid onto the particles for the next particle push.

PICSARlite calls the same subroutines as the full application WarpX for the elementary PIC operations, when applied to the same problem with the same options selected. Where WarpX runs different code from PICSARlite is in communication, load balancing, and data iterators.

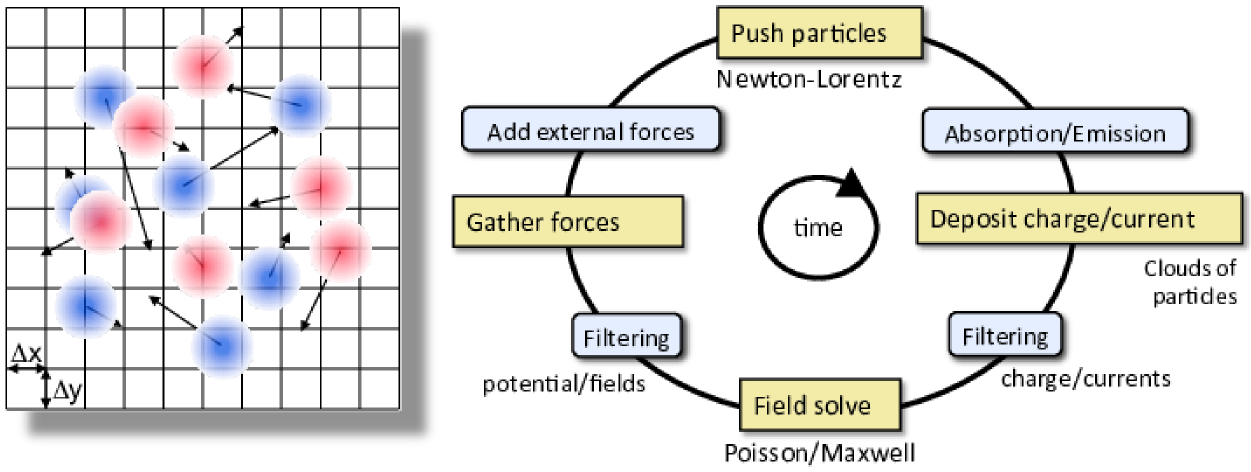


Figure 24: The Particle-In-Cell (PIC) method [34] follows the evolution of a collection of charged macro-particles (left plot: positively charged in blue, negatively charged in red) that evolve self-consistently with their electromagnetic (or electrostatic) fields.

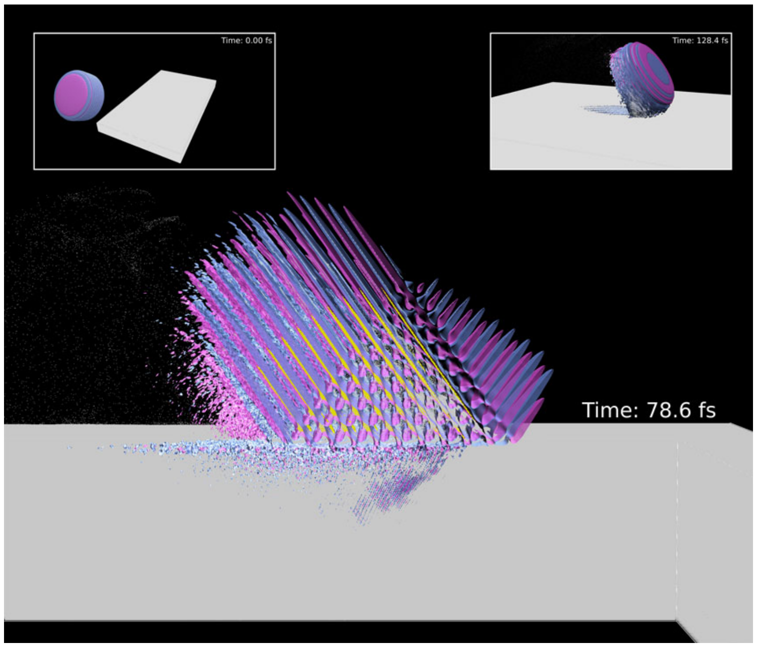


Figure 25: Plasma mirror problem. Laser pulse (blue and magenta) from empty space (black) in upper part, into plasma (gray) in lower part. *Thanks to WarpX team for the figure.*

5.2 Problem Selection and Validation

The problem for evaluation was given to us by the WarpX team. Physically, as depicted in Figure 25, the problem is the focusing of high-order harmonic beams by a relativistic plasma mirror. Such plasma mirrors can be formed by focusing a femtosecond laser pulse of intensity $I > 10^{18}$ W/cm^2 with high temporal contrast on a solid target such as plastic or glass. Upon reflection on the relativistically oscillating mirror surface, high-order harmonic beams are generated by the Doppler effect in the reflected field propagating along the specular direction. For Gaussian laser profiles at focus, the inhomogeneous laser radiation pressure can curve the plasma mirror surface, forming a parabolic plasma mirror, which can in turn tightly focus the harmonic beams. This process can lead to extreme intensities $I > 10^{25}$ W/cm^2 at plasma mirror focus and is currently a very promising candidate to access yet unobserved quantum electrodynamic processes such as electron-positron cascades or vacuum breakdown. For our performance testing, *the laser is omitted* in order to simplify the problem, because it does not take much computational time.

The domain of the problem we ran on WarpX and PICSARlite is the rectangle from points (-1.42e-5, -8.5e-6, 0) to (2.8e-6, 8.5e-6, 3.4e-5). The plasma consists of two species of particles, electrons and protons, both distributed in the rectangular subregion from (0, -8.4e-6, 1e-6) to (2.5e-6, 8.4e-6, 3.3e-5), with 2 particles of each species per grid cell. The particle-containing region has dimensions 2.5e-6 $\times$ 1.68e-5 $\times$ 3.2e-5, within the full domain of dimensions 1.7e-5 $\times$ 1.7e-5 $\times$ 3.4e-5. This is 14.7%, 98.9%, and 94.1% of the length of the domain in the three dimensions.

We did not do any validation to check that WarpX and PICSARlite give the same answers. We asked the WarpX team for ways to validate results, but they told us that the different output formats of the two programs as well as the nature of the output itself make a comparison of results difficult.

5.3 Methodology

We ran WarpX and PICSARlite on the Cori system at NERSC, on Haswell nodes, which have the hardware characteristics listed in Table 2.

We build WarpX with

```
$ make -j 8 COMP=intel DEBUG=FALSE TINY_PROFILE=TRUE
  CXXFLAGS="-g -O3 -qopenmp" F90FLAGS="-g -O3 -qopenmp"
```

and PICSARlite with

```
$ make SYS=cori1 ARCH=hsw FC=ftn MODE=prod FARGS="-g -O3 -qopenmp"
```

and then, for each, rerun the link line preceded by `hpclink`.

The domain is split up so that each Haswell node has a grid of 192×192×384 cells. We ran with 4 MPI ranks per node, and 8 OpenMP threads per MPI rank. We did a weak scaling test with 1 node and with 8 nodes. Each MPI rank has a subdomain of 192×192×96 cells. In the 1-node example, all 4 MPI ranks include part of the region with particles. But in the 8-node example, only 16 of the 32 total MPI ranks include part of the region with particles.

The number of time steps for this problem was chosen to be large enough to provide a good enough sample, but not so large as to take too much execution time. About 40 time steps was adequate.

	1 node 4 MPI ranks, 32 threads WarpX	 PICSARlite	8 nodes 32 MPI ranks, 256 threads WarpX	 PICSARlite
Push	10.2 (3.75%)	6.2 (4.4%)	9.7 (2.8%)	11.3 (5.2%)
Deposit	136.1 (50.1%)	26.6 (18.8%)	83.6 (23.6%)	26.9 (12.5%)
Field solve	24.8 (9.1%)	40.6 (28.7%)	27.6 (7.8%)	42.4 (19.7%)
Gather	27.8 (10.2%)	16.0 (11.3%)	23.2 (6.5%)	14.2 (6.6%)
OMP Barrier	52.4 (19.3%)	44.4 (31.4%)	141.7 (40.0%)	106.5 (49.6%)
HPCToolkit	0.0 (0.0%)	7.2 (5.1%)	0.0 (0.0%)	6.4 (3.0%)
Other	20.7 (7.6%)	0.5 (0.4%)	68.1 (19.2%)	7.1 (3.3%)
Total	271.9	141.6	353.9	214.8
Wall-clock time	286	150	361	218

Table 14: Dynamic profile of WarpX and PICSARlite on 1 node and on 8 nodes. The table shows the average time over all threads and ranks (in seconds) spent in each component of the PIC code, as recorded by `hpcviewer` for code segments classified according to Tables 15 and 16. The wall-clock time is the measured time from start to finish on the command line.

5.4 Results and Analysis

5.4.1 Dynamic Profiling

HPCToolkit's "Top-down view" in `hpcviewer` displays a tree of non-overlapping code segments and the inclusive CPUTIME for each segment, averaged over all threads and all MPI ranks on that code segment. At the highest level, the Top-down view splits the total time into three parts, termed "fake" procedures by the HPCToolkit User's Manual, which are <`program root`>, <`partial call paths`>, and <`thread root`> —the last of which includes all OpenMP-threaded code segments. Using our best understanding, from conversations with the WarpX team, and from looking at the code itself, we divided the total time into the four main components of the PIC algorithm as given in Section 5.1. We also included a separate category for time waiting at OpenMP barriers, as reported by `hpcviewer`, and time for HPCToolkit overhead, which was reported by `hpcviewer` for PICSARlite but not for WarpX. The assignments of code segments to PIC components are shown in Table 15 for WarpX, and in Table 16 for PICSARlite. There were other code segments that we did not know how to classify and code segments that took less than 0.1% of the total time were also not included in the classification.

The dynamic profile for our runs is shown in Table 14. The "Other" category is for code segments that we did not classify; for WarpX, these were mostly AMReX functions for operations such as copying and arithmetic on arrays, when it was not clear which PIC code component was calling them.

Based on the intended similarity of the problem specification for WarpX and PICSARlite, we expected total run times to be similar for the two codes. However, we found that PICSARlite took 40%–50% less time than WarpX. Of the four components of the PIC algorithm, the largest amount of time in WarpX is spent in current deposition, which is what the WarpX team told us should be expected. However, PICSARlite actually spends more time in the field solve. The field solve in PICSARlite takes longer even in absolute time than that in WarpX. The particle push and field gather components of the PIC algorithm take similar fractions of time in both WarpX and PICSARlite for both sizes of problem.

Push particles:

```
<thread root> PhysicalParticleContainer::PushPX
<thread root> AMReX_ParticleContainerI.H: 1807
<partial call paths> PhysicalParticleContainer::PushPX
<thread root> amrex::ParIterBase<false, 0, 0, 10, 0>::GetPosition
```

Deposit from particles to grid:

```
<thread root> WarpXParticleContainer::DepositCurrent
<thread root> amrex_particle_set_position
<partial call paths> WarpXParticleContainer::DepositCurrent
<program root> WarpX::SyncCurrent
```

Field solve on grid:

```
<thread root> WarpXEvolve.cpp: 501
<thread root> WarpX.cpp: 938
<thread root> WarpXEvolve.cpp: 630
<partial call paths> warpx_geteb_energy_conserving
<program root> WarpX::EvolveE
<program root> WarpX::EvolveB
<program root> WarpX::FillBoundaryB
<program root> WarpX::FillBoundaryE
```

Gather from grid to particles:

```
<thread root> warpx_geteb_energy_conserving
```

Barrier:

```
<thread root> __kmp_fork_barrier
<thread root> __kmp_join_barrier
<partial call paths> __kmp_join_barrier
```

Table 15: Assignment of code segments of WarpX listed in `hpcviewer` to the four components of the PIC algorithm (Section 5.1), as well as separate barrier time.

Push particles:

```
<partial call paths> pxr_boris_push_u_3d_
<partial call paths> pxr_set_gamma_
<partial call paths> pxr_pushxyz_
<partial call paths> outline particle_boundaries.F90: 461
<partial call paths> particle_boundary_mp_particle_bcs_mpi_non_blocking_
<program root> particle_boundary_mp_particle_bcs_
```

Deposit from particles to grid:

```
<thread root> outline current_deposition_manager_3d.F90: 353
<partial call paths> pxrdepose_currents_on_grid_jxjyjz_
<partial call paths> field_boundary_mp_current_bcs
<program root> pxrdepose_currents_on_grid_jxjyjz_
```

Field solve on grid:

```
<thread root> outline yee.F90: 376
<thread root> outline yee.F90: 185
<partial call paths> push_bfield_
<partial call paths> push_efield_
<partial call paths> field_boundary_mp_bfield_bcs
<partial call paths> field_boundary_mp_efield_bcs
```

Gather from grid to particles:

```
<partial call paths> loop at particle_pusher_manager_3d.F90: 1520
<partial call paths> loop at particle_pusher_manager_3d.F90: 537
<partial call paths> loop at particle_pusher_manager_3d.F90: 538
<partial call paths> loop at particle_pusher_manager_3d.F90: 536
<partial call paths> loop at particle_pusher_manager_3d.F90: 535
<partial call paths> loop at particle_pusher_manager_3d.F90: 534
<partial call paths> loop at particle_pusher_manager_3d.F90: 539
<partial call paths> loop at particle_pusher_manager_3d.F90: 521
<partial call paths> loop at particle_pusher_manager_3d.F90: 520
<partial call paths> loop at particle_pusher_manager_3d.F90: 523
<partial call paths> loop at particle_pusher_manager_3d.F90: 522
<partial call paths> loop at particle_pusher_manager_3d.F90: 524
<partial call paths> loop at particle_pusher_manager_3d.F90: 519
<partial call paths> particle_pusher_manager_3d.F90: 467
```

Barrier:

```
<thread root> __kmp_fork_barrier
<thread root> __kmp_join_barrier
<partial call paths> __kmpc_barrier
```

HPCToolkit overhead:

```
<partial call paths> hpcrun_restart_timer.isra.3.part.4
```

Table 16: Assignment of code segments of PICSARlite listed in `hpcviewer` to the four components of the PIC algorithm (Section 5.1), as well as separate barrier time and HPCToolkit overhead.

We have not determined the root cause of the timing differences between WarpX and PICSARlite. Most likely the differences are caused by some combination of:

- Ability/inability of the compiler to perform certain optimizations in one code but not the other.
- Overhead imposed by the AMReX framework.
- Tiling and threading are done in different ways in the two codes.

Based on the significant mismatch in timings, we conclude that PICSARlite may not accurately represent the algorithms and coding structures of WarpX. More effort to determine how PICSARlite could be made more representative (or to determine how WarpX could be improved to match PICSARlite performance) is probably warranted.

5.4.2 Load Balance

As reported in Table 14, a large fraction of the time is spent waiting at OpenMP barriers in both codes. The barrier time is significantly larger on 8 nodes than on 1 node, with PICSARlite spending nearly 50% of the run time in barriers on 8 nodes. With `hpcviewer`, it is possible to obtain graphs of the amount of time that each thread spends in a particular code segment. Such graphs can be used to visualize load imbalance.

Figure 26 shows the amount of time spent in each thread on the `__kmp_fork_barrier` code segment under <`thread root`>. This one code segment accounts for approximately 99% of the time that is assigned to the Barrier component in Table 14. It corresponds to time waiting at barriers in OpenMP threaded code.

As shown by Figure 26, the overall load is well balanced on the 1-node WarpX and PICSARlite runs but not on the 8-node runs. We expect to see results like this because, as mentioned in Section 5.3, in the 8-node runs, half of the grids have no particles, and so the MPI ranks assigned to those grids do not spend time on pushing particles, depositing, and gathering. WarpX does, however, have a dynamic load-balancing capability that can alleviate this problem.

5.4.3 Roofline Model

For roofline model information, we profile with **Intel Advisor**. In order to use it, for PICSARlite, we build with

```
$ make SYS=cori1 ARCH=hsw FC=ftn MODE=advisor
```

and for WarpX, we build with

```
$ make COMP=intel DEBUG=TRUE TINY_PROFILE=FALSE CXXFLAGS="-g -O3 -xCORE-AVX2
  -qopenmp -dynamic -debug inline-debug-info" F90FLAGS="-g -O3 -xCORE-AVX2
  -qopenmp -dynamic -debug inline-debug-info -align array64byte"
```

Overall, the roofline summary tells us that the WarpX run averages 1.67 GFLOPS and 1.69 GINTOPS, with 20 vectorized loops taking 16% of the total time. The PICSARlite run averages 5.14 GFLOPS and 4.28 GINTOPS, with 35 vectorized loops taking 14% of the total time.

The roofline charts for one MPI rank of the 1-node runs of WarpX and PICSARlite are shown in Figure 27. Red dots correspond to procedures that take more than 5% of the total time, yellow dots to procedures that take between 1% and 5% of the total time, and green dots to procedures that take less than 1% of the total time.

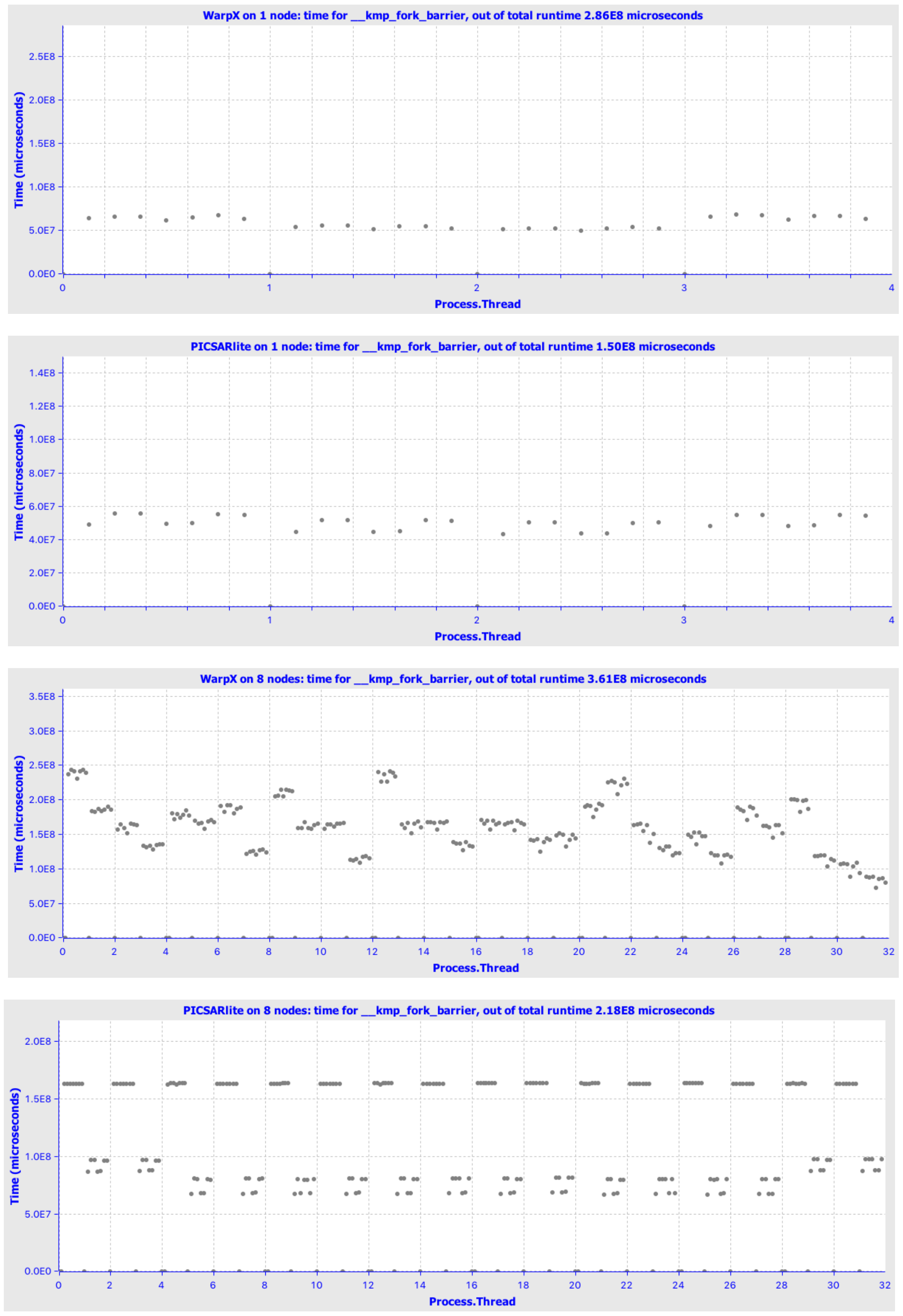
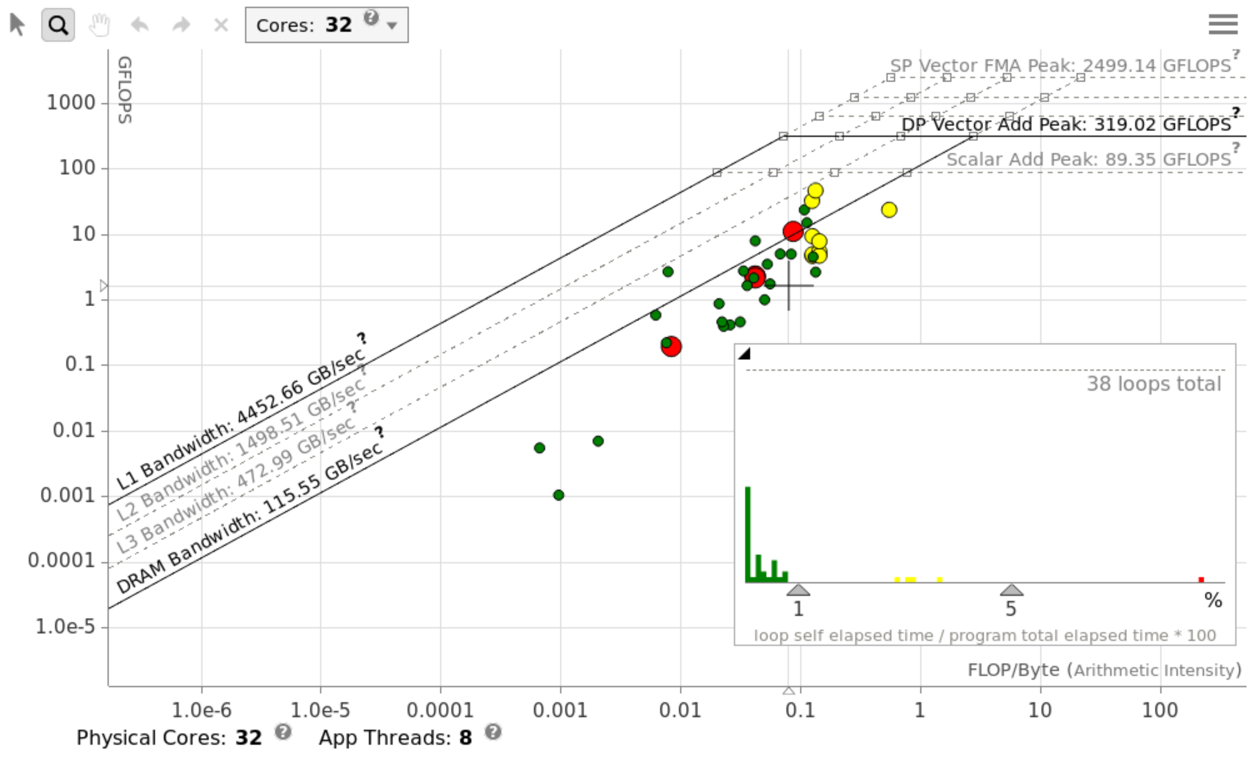


Figure 26: Amount of time spent in `__kmp_fork_barrier` as reported by `hpcviewer`, for (top to bottom) WarpX on 1 node, PICSARlite on 1 node, WarpX on 8 nodes, and PICSARlite on 8 nodes. This is a measure of the time that each thread spends waiting at OpenMP barriers. In each graph, the upper limit of the vertical axis is the total wall-clock time for that run.



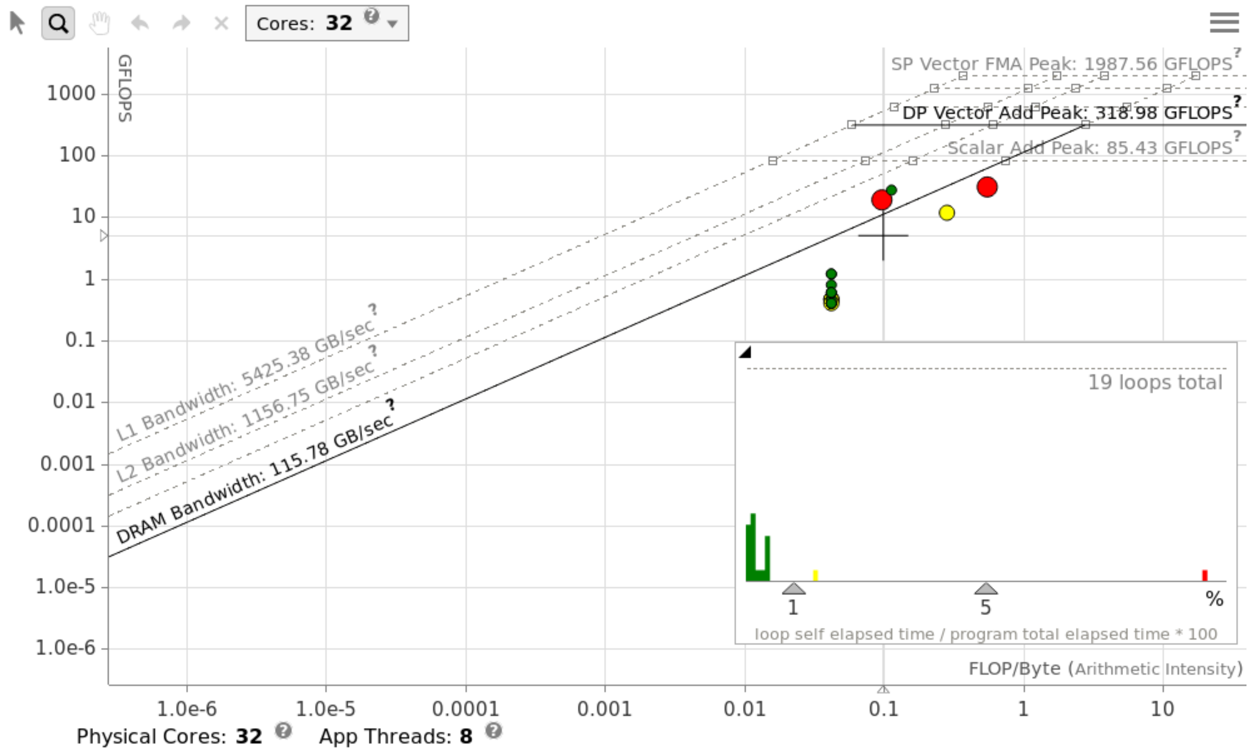


Figure 27: Roofline model on WarpX (top) and PICSARlite (bottom), for one MPI rank of the 1-node runs.

Most of the dots are located below the diagonal lines, indicating that their corresponding procedures are memory-bound. The highest arithmetic intensity is shown by the rightmost dot in both graphs (yellow for WarpX, red for PICSARlite) for the same loop in `pxr_boris_push_u_3d` that is part of the particle pushing component of the PIC code. The other red dot on the PICSARlite graph corresponds to a procedure in field gathering, as does the top red dot on the WarpX graph. In the WarpX graph, the lowest red dot is for a parallel copy routine in AMReX. The middle red dot (actually two overlapping red dots) corresponds to current deposition.

Note that the PICSARlite roofline plot in Figure 27 shows a total of only 19 loops and that 15 of these all have the same arithmetic intensity of 0.042 flop/byte. Although this roofline was run multiple times, the results remained consistent. We believe this data is anomalous. Further, when we examined execution times of various loops in both of the roofline plots using the interactive HTML format, we observed many discrepancies between the execution time profiles of the roofline versus those generated by HPCToolkit. For example, for the field solve phase in PICSARlite, Intel Advisor does not identify a single loop that accounts for more than 1% of the execution time while HPCToolkit reports that this execution phase takes 28.7% of the total time, which again better matches intuitive expectations. Similar discrepancies were noted in the profile data for WarpX. We conclude that further investigation into these potential anomalies in the Intel tool should and will be investigated. We realize these anomalies may not be the tool but could be an issue with how we are using it due to its complexity (many flags and analyses within this single tool) or could be due to an old or faulty installation. A deeper investigation should enable identification of the root cause.

5.5 Is PICSARlite a Good Proxy for WarpX?

From the analysis presented above, we conclude that PICSARlite may not be an adequate proxy for WarpX primarily because of the large difference in execution times ($\sim 2\times$) and the differences in their execution profiles (Table 14). We plan to work with the development team to further validate the results presented here and to re-factor PICSARlite if necessary to improve its representativeness of WarpX for the chosen challenge problem.

6 Profugus, XSBench, and Shift Performance Assessment

ProfugusMC and XSBench are proxy applications that strive to represent aspects of Monte Carlo transport solvers. ProfugusMC represents the complex, random branching execution path typical of Monte Carlo transport, XSBench focuses on lookups of nuclear cross section (probability) data. Shift is a Monte Carlo solver used in the ECP ExaSMR project. The ExaSMR project's goal is to bring operational, commercial Small Modular Reactors (SMRs) online. Current reactor simulations rely on an experimental database and constrained design margins. This becomes limiting when experimenting with innovative designs, such as Small Modular Reactors. In order to address this issue ExaSMR aims to simulate the reactor core beyond the beginning of life. In order to accomplish this simulation they will couple the Monte Carlo solver (used for Monte Carlo neutron transport simulation) with a Computational Fluid Dynamics code and utilize the computational power of Exascale systems.

Because Shift is and export controlled code, distribution is limited. XSBench and ProfugusMC were created to enable collaborations that otherwise would be hindered by security concerns. This highlights an important use case for proxy applications. Note that ProfugusMC is even smaller version of a larger proxy, Profugus. Profugus was originally used for algorithmic exploration for both deterministic and Monte Carlo transport. Because Profugus had functional solvers as well as dependencies on Trilinos and HDF5, it wasn't very "mini". ProfugusMC is a stripped-down version that focuses on the Monte Carlo solver and has no external dependencies. The Monte Carlo solver is identical between the two.

6.1 Science and Algorithm

Monte Carlo neutron transport simulations track neutrons from their origin at a source to their absorption or leakage from the system. Particles travel through materials that are determined by the simulation geometry. The simulation of a neutron from birth to death is called a history. The behavior of the neutrons is estimated by simulating many histories. Throughout this history various events can happen to the particles, such as scattering or fission.

The decisions about how a neutron will react within a specific material are determined by randomly sampling probability distributions known as cross sections. These cross sections are determined experimentally and are stored in large tables. Table lookups require memory accesses which are close to random. These lookups are a known performance bottleneck of Monte Carlo neutron transport, making the code suffer from memory latency while waiting for memory accesses.

Two kinds of cross section tables are commonly used. Continuous energy (CE) cross section tables attempt to fully resolve physical effects such as nuclear resonances that are very sensitive to small differences in particle energy. To capture such differences, CE tables require a very fine table spacing resulting in a large memory footprint for the cross section tables. 1-5 Gbytes is not uncommon. Pulling data from such large tables can be a significant performance bottleneck. MC transport codes can spend 80–90% of run time doing CE cross section lookups. In contrast, multigroup (MG) cross section tables divide the energy spectrum into a set of groups and tabulate an average value of the cross section for all energies in the group. Multigroup tables usually have 200–300 groups greatly reducing the size of the tables as well as the fraction of run time spent accessing them. Although Shift typically uses CE tables, ProfugusMC makes the decision to use multigroup energy lookups in order to expose the branching and divergent program flow aspects of code performance instead of the CE cross section bottleneck.

The performance of continous energy lookups is well represented in the by the XSBench proxy application. XSBench is specifically focused on the study the continuous energy cross section lookups and does not include the full neutron transport logic.

The transport algorithm has historically been history-based, wherein a unit of work is the simulation of a particle from birth to death. Transport can also be implemented using an event-based algorithm where a unit of work is a single event in a particle's history. ProfugusMC/Shift use an event-based algorithm to try to exploit the greater data-level parallelism in recent HPC products.

Throughout the simulation information (tallies) are collected about the events taking place within the simulation. These tallies are the output data that can be used in coupling the Monte Carlo simulation with a CFD code. The exact nature of the tallies to be collected is uncertain at this point, as the MC/CFD coupling is new and the amount of information needed is unclear.

The ProfugusMC proxy application models the parent application's neutron transport algorithm along with the tallying logic. The parent application makes use of the Multiple-Set Overlapping Domain (MSOD) algorithm for its domain decomposition. This algorithm divides the geometry into structured Cartesian "blocks" based on a user input along with a second user input to define the "overlap" fraction which controls how much of each block is duplicated between neighbors. The overlap helps to reduce the number of particles continuously scattering repeatedly across blocks. This domain is decomposition is not represented in the proxy application. Since the proxy application problem geometry is small it is replicated on every node.

The Monte Carlo neutron transport simulation requires simulating a large number of particles to obtain a statistically accurate result. Because the simulation of these particles is major component of the simulation workload the ExaSMR project measures its performance in terms of the number of particles transported per second.

6.2 Problem Selection and Validation

The ProfugusMC proxy application uses a builtin problem geometry that represents athree dimensional reactor core and a multigroup cross section library with 252 groups. The number of particles to simulate is controlled on the command line. Vertically the core is broken down into a 3×3 grid with 2×2 fuel assembly in a corner. The fuel assembly is heterogenous with two UO2 assemblies and two MOX assemblies. The MOX assemblies use MOX 4.3%, MOX 7.0% and MOX 8.7%. The assemblies are described in a 17×17 grid. The full problem description is available at https://www.oecd-nea.org/science/docs/2005/nsc-doc2005-16.pdf, with the difference that ProfugusMC uses a 252 group cross-section library while the benchmark description uses 7 groups.

The number of particles to be simulated depends on the precision required. Early information from the developers suggests that for a full Exascale simulation a number of particles across the entire simulation around 1 billion should be common. While the common case is expected to require 1 billion particles it is possible that more will be needed for some simulations. That being said, using a number of particles above 1 trillion will very likely not be useful. In order to derive a working problem size we consider the pre-Exascale system, Summit. Therefore we analyze a problem using 36,168 particles per utilized node (1,000,000,000 particles / 27648 gpus).

ProfugusMC preforms two built-in validation checks on results of the simulation. This validation is done by comparing to a reference with a range defined by standard deviations. The first check is against the computed eigenvalue, with a value within three standard deviations considered passing. The second check is against the computed tally results. These tally results are compared to the reference and considered passed if all cells are within five standard deviations of the reference.

XSBench's run configuration is controlled by the '-p <particles>'. Each cross section lookup includes two randomly sampled inputs, one for the neutron energy and one for the material. The number of lookups for each particle is control by the command line parameter '-l <lookups>'. The

Kernel	% Time
distance_to_boundary	21.6%
process_collision	24.7%
process_boundary	17.9%
path_length	13.8%
MPI_Barrier	13.9%
Other	8.1%

Table 17: ProfugusMC Dynamic Profile

default number of lookups is set to 34, which is the average for a light water reactor problem. This should only be changed if trying to simulate a reactor with different behaivor. Additionally XSBench can be run using the event based algorithm by setting the simulation method to event ('-m event').

For the parent application, Shift, we run a 1/4 core depleted core model. This depleted problem represents a common use case and offers additional complexity due to the presence of fuel with fission products.

6.3 Results and Analysis

For our performance analysis of ProfugusMC we used four skylake nodes with the problem size of 144,672 particles. We found that the event based algorithm which ProfugusMC models is represented in the 'transport' function of ProfugusMC. Within this function the time is spent across five kernels as seen in Table 17. In our investigation of the Shift application, we found the transport function's profile is significantly different. Spending around 85.6% of the time in routines that calculate the macroscopic cross sections, which involves the costly continuous energy lookups. The difference in performance between ProfugusMC and Shift additionally manifests itself with a slowdown of over 150× in the active cycle timing in our experiments.

Within ProfugusMC, the 'distance_to_boundary' function determines what event will happen in the current iteration. This function has a very low Arithmetic Intensity of 0.043 while not pressuring the memory bandwidth, making use of less than one percent of the L2, L3 and DRAM bandwidth. It is bound by memory access latency in addition to being slowed by preforming some atomic operations. 'path_length' is the function which carries out the tallying operations and make use of atomic operations. The 'process_boundary' function handles the event of the particles crossing into a new boundary. This additionally spends some time preforming atomic operations. The 'process_collision' preforms the Multigroup lookup and determines what action to take (Scattering, Fission, etc) This is the part most expensive of the code which is even more dominate to the performance when using full continuous energy cross section lookups. Processing the collisions is the most floating point expensive part of the code, having an Arithmetic Intensity of 0.26 and pulling around 10% of the L2 bandwith. However it is still bound by the memory latency of accessing the energy lookup table. Finally the remainder of the time is spend in an 'MPI_BARRIER' to synchronize. Across the various functions which make use of atomic operations the total percentage of time spent preforming these operations is 17.3% of the runtime. As seen in figure 28, the memory access limitation manifests itself in the roofline plot. ProfugusMC's performance falls below both the memory bandwidth and floating point peaks of the system, including the 'process_collision' function seen in red.

XSBench is built with openmp for thread level parallelism. We investigated the performance of

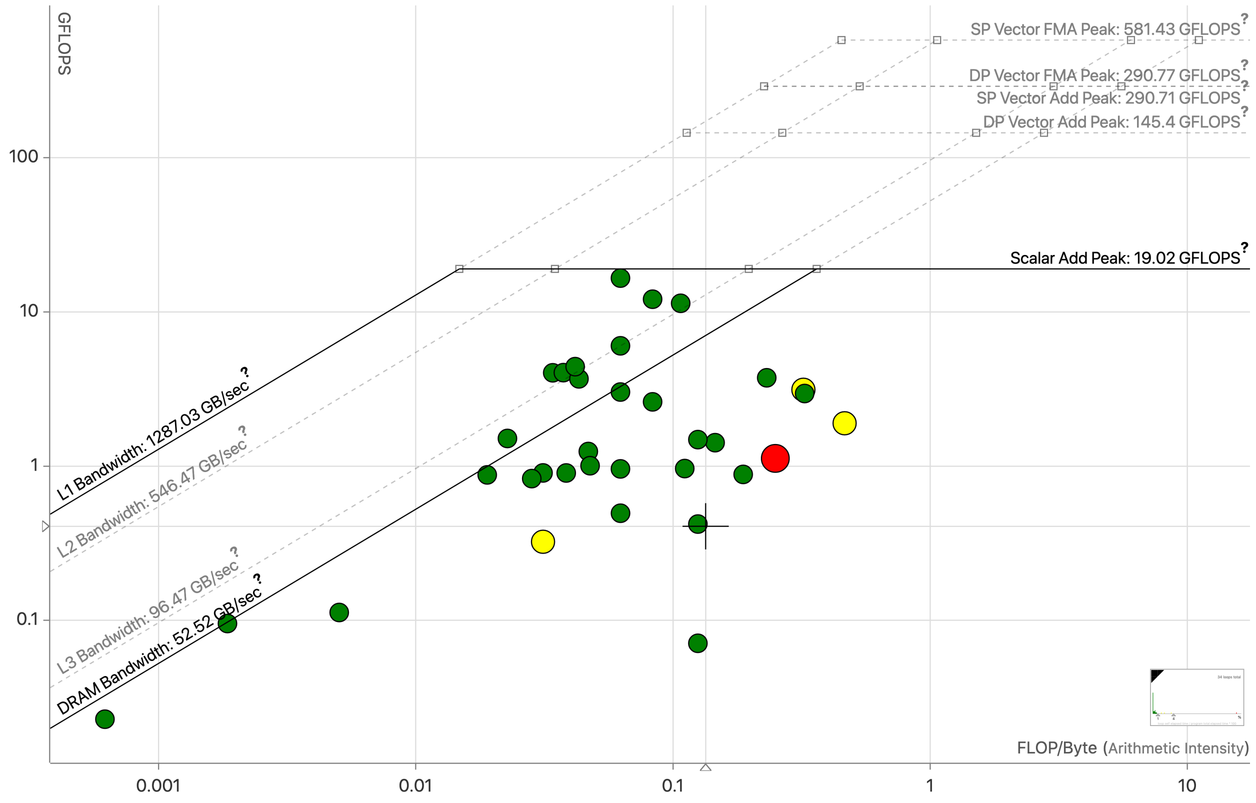


Figure 28: ProfugusMC Roofline

XSBench by running on a single Skylake running a single thread per core. We found that 83.4% of the runtime is spent in calculating the macroscopic neutron cross sections. The performance for the continuous energy lookups is heavily bound by memory access to distant data. This manifests itself with large cache miss ratios, the L1 being 7%, the L2 being 45%, and the L3 being 67%. The large amount of accesses to DRAM start to utilize a significant amount of the DRAM bandwidth, pulling over 64% in our study.

6.3.1 GPU Implementation

The implementation of ProfugusMC is built to run either on a CPU or GPU based on the command line argument '-arch <cpu/gpu>'. We collected performance data for the GPU version on an Intel Skylake with four Volta 100 NVidia GPUs in addition to the performance data collected for the CPU version on four Intel Skylakes. Looking at the performance metrics of the GPU run we see similar performance bottlenecks with nvprof showing that 77% of the stalls are from memory dependencies.

6.4 Conclusion

ProfugusMC is used to enable collaboration on an export controlled code, allowing study of the event based algorithm, the performance of tallying and the investigation of the Cuda based GPU implementation. The proxy application makes the decision to not model the parent application's key performance bottleneck fully as it is well studied and elsewhere well represented. It instead focuses on the study of the event-based algorithm, the tally performance and the GPU implementation.

The code stresses the memory system by being heavily dominated by memory accesses.

Looking at the proxy application with respect to Shift, as expected ProfugusMC is a poor model for the overall performance of Shift. However, XSBench is a good model for Shift's performance. ProfugusMC should only be utilized to investigate performance for the parts of the code not involved with the CE cross section lookups ('process_collision'), while understanding that those sections map to a small part of the parent application's performance profile.

7 Thornado-mini Performance Assessment

Thornado-mini is a proxy app developed in the ExaStar ECP AD project. The ExaStar project targets multi-physics simulations of astrophysical explosions. Their target application, Clash, will be a component-based multi-physics toolkit, built on capabilities of current astrophysics simulation codes Flash and Castro and on the massively parallel adaptive mesh refinement framework AMReX. The Clash code suite will have modules for hydrodynamics, advanced radiation-transport, thermonuclear kinetics, and nuclear microphysics. In this code suite, the radiation transport kernel is expected to consume 80% of the full application runtime.

Thornado-mini is a proxy app for finite element, moment-based radiation transport. The proxy application code is a serial code written in Fortran 90 that solves the equation of radiative transfer using a semi-implicit, discontinuous Galerkin (DG) method for the two-moment model of radiation transport. Thornado-mini currently implements deleptonization from electron capture on nucleons and nuclei using tabulated neutrino opacities provided by the WeakLib library. The deleptonization problem is representative of both the flops and memory access patterns that are expected in the exascale problem.

7.1 Radiation Transport Kernel

The ExaStar project has been working to develop advanced transport techniques as the treatment of radiation transport is typically the most computational expensive component of current stellar explosion simulations [3]. This is due to the need for high spatial resolution and unconstrained spatial dimensionality to follow important fluid instabilities as the explosion develops, as well as a sensitivity of neutrino heating rates to the neutrino energy distribution, which requires retention of the energy dimension of momentum space. To balance computational cost with the required physical fidelity, the number of unknowns can be reduced by truncating the moment expansion of the angular dimensions of momentum space; i.e., by solving for a finite number of angular moments. Thornado-mini was developed in year one of the ExaStar project to solve the transport equation in the two-moment formalism. The two-moment model is obtained by truncating the moment expansion at the level of the so-called first order moments (akin to an expansion in spherical harmonics up to degree 1), so that the unknown moments are the spectral particle density, and particle flux—a two-moment model. To solve the neutrino transport equations the moment equations are discretized in energy-position space using high-order Discontinuous Galerkin (DG) methods. DG methods combine elements from both spectral and finite volume methods, and are considered a desirable option for solving hyperbolic partial differential equations (PDEs). Without modification, the DG methods recover the correct asymptotic behavior in the diffusion limit, characterized by frequent collisions.

Thornado-mini's implementation of the radiation transport kernel was ported into both the FLASH and Castro source codes in the second year of the ExaStar project. The use of thornado-mini as a prototype or design exploration for the Clash toolkit demonstrates another increasingly common use case for proxy apps.

7.2 Representative Exascale Problem: Deleptonization

Stars more massive than about 8 solar masses end their lives as core-collapse supernovae. They are triggered from the collapse of the stellar core due to electron captures and the photodissociation of heavy nuclei. The collapse continues until normal nuclear matter density is reached. The nucleon pressure from the short-range repulsive nuclear interaction halts the collapse and the supersonically collapsing core bounces back. A sound wave turns into a shock wave which then propagates out of the core. The object which forms at core bounce is the protoneutron star (PNS), being hot and lepton rich in which sense it differs from the final supernova remnant, the neutron star. Deleptonization is the loss of leptons from the PNS.

Thornado-mini implements a simulation of deleptonization. The simulation is constructed by adopting analytic profiles for initial mass density, temperature, and electron fraction. The initial mass density is set such that the maximum density is consistent with nuclear matter, and neutrinos will be trapped. As the density decreases with radius, the neutrino mean free path increases, and the neutrinos created by electron capture will be able to escape the computational domain (deleptonization). The values set for the initial mass density, temperature and electron fraction profiles can be found in [5]. The radiation field is initialized by setting the distribution function equal to the local Fermi-Dirac distribution. The computational domain covers a radius of 0 to 100 km and energy of 0 to 300 MeV, and evolves until 100 ms.

7.3 Thornado-mini Performance Assessment

This performance assessment will include results from thornado-mini only. The assessment of thornado-mini was planned with the understanding that the application development team had a milestone to implement the Deleptonization problem or a problem comparable to Deleptonization in Castro in FY 2018. This milestone has been rescheduled, and to date, Castro or Flash lack the capabilities that would be required to allow a meaningful comparison between the parent application and proxy. We will return to this comparison when the relevant capabilities are available in Castro.

7.4 Methodology

Performance profiles were gathered on the Haswell nodes of the Cori system at the National Energy Research Scientific Computing Center (NERSC). Hardware characteristics of Cori are given in Section 2.3. For profiling, we used Craypat [25] performance measurement and analysis tools. Craypat is a toolkit offered by Cray for the XC platform. To profile the thornado-mini proxy app, we used the "pat_build" utility to instrument the application, and after running the instrumented application, we used the "pat_report" utility to generate a summary and analysis of the profile.

As was stated previous sections, thornado-mini is a serial application and there is no domain decomposition within the thornado-mini kernel. Because of this, we focus this analysis on function and loop level profiles of the execution time and memory. We performed two stages of profile and analysis. The first stage was an overall profile of the entire proxy app, and the second stage was a more specific profile and trace of functions and loops of interest. Although the kernel is serial, we execute the kernel across all 32 cores on a node to simulate resource sharing within a node, and the results are averaged across the cores.

7.5 Results and Analysis

In the overall profiling and analysis, we first analyze the execution time. In this analysis, craypat bins function calls into three categories: user defined functions, system calls (e.g. _pgo_clock, _cray2_EXP_01, _cray_ALOG10, etc.), and BLAS library calls. What craypat refers to as system calls would generally fall into the category of standard library calls. Figure 29 shows that the majority of the execution time is nearly evenly split between user defined function calls and system function calls with a small amount of time spent in BLAS library calls. A memory analysis of the overall kernel execution shows a peak memory usage of 410 MiB and average memory traffic of 22.379 GB.

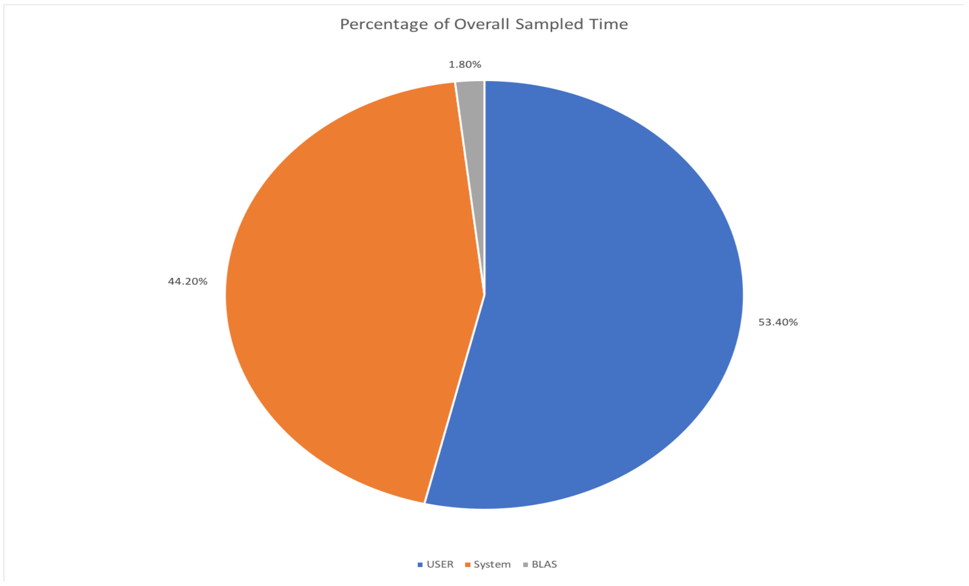


Figure 29: Profile of Thornado-mini kernel execution time

From this overall analysis, a more detailed analysis of the user defined functions was performed. In this analysis we trace the functions list in the overall analysis and do additional loop level analysis on the execution time. From the execution time trace of the user-defined functions, Table 18 provides a percentage of time spent in each function relative to the time spent in user defined function calls. From this analysis we see that there are three functions that dominate execution time: detectdiscontinuities, evolvefields and computetempfromintenergy_lookup. The function detectdiscontinuities checks and flags all cells in the domain that need processing by a limiter; evolvefields is the main time stepping routine, evolving the initial condition to the requested final time. It calls all the other functions in the kernel; computetempfromintenergy_lookup computes the temperature from internal energy by interrogating the equation of state table.

A further loop analysis of the proxy app is shown in Tables 19 and 20. Table 19 shows the percentage of inclusive time spent in a loop. The loop that represents the most inclusive execution time spent in a loop found in the evolvefields function which is one the three functions that dominates the runtime and is also the main function in the code. Loops 2–6 in detectdiscontinuities have the next largest inclusive loop execution time and are nested loops where loop 2 is the outermost and loop 6 is the inner most loop in the loop nest. Table 20 shows the exclusive time spent in a loop in seconds. When considering exclusive time, the previous mentioned loop is fourth among the longest running loop, with the longest running loop being found in the couplefluidradiation function and the second longest running loop being found in the detectdiscontinuities function. This loop found in couplefluidradiation solves a nonlinear system of equations for the coupling of neutrinos to matter due to emission and absorption. This loop iterates until all cells in the spatial domain have converged. The loop found in detectdiscontinuities runs over all radiation variables and maps them from one representation to a different representation that is more convenient for detecting discontinuities.

Function Name	% Time
detectdiscontinuities	31.80
evolvefields	15.80
computetempfromintenergy_lookup	15.60
loginterpolatesinglevariable_1d3d	7.90
applyslopelimiter_m1_dg	6.10
computerhs_m1_dg_x1	5.70
applypositivitylimiter_m1_dg	4.00
couplefluidradiation_emabscatt	3.80
setequilibrium	3.00
mapnodaltomodal_radiation	1.40
loginterpolatesinglevariable_3d	1.20

Table 18: Thornado-mini user defined function trace

File Name	Function Name	Loop Number	Line Number	% Time
timesteppingmodule	evolvefields	1	217	99.80
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	2	116	32.00
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	3	117	32.00
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	4	121	32.00
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	5	122	32.00
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	6	126	31.80
wlinterpolationmodule	computetempfromintenergy_lookup	1	565	27.60
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation	1	333	25.80
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation_emabscatt	1	122	7.80
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation_emabscatt	2	123	7.80
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation_emabscatt	3	124	7.80

Table 19: Trace of inclusive time spent in Thornado-mini loops

File Name	Function Name	Loop Number	Line Number	Time (sec)
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation	1	333	1,614.81
momentequationslimiterutilitiesmodule_dg	detectdiscontinuities	6	126	1,066.84
wlinterpolationmodule	computetempfromintenergy_lookup	1	565	713.749597
timesteppingmodule	evolvefields	1	217	532.474847
momentequationsslopelimitermodule_dg	applyslopelimiter_m1_dg	07	100	182.406208
fluidradiationcouplingsolutionmodule_emabscatt	setequilibrium	1	606	100.018552
fluidradiationcouplingsolutionmodule_emabscatt	couplefluidradiation_emabscatt	6	164	96.530805
momentequationssolutionmodule_m1_dg	computerhs_m1_dg_x1	12	158	95.868396
wlinterpolationmodule	loginterpolatesinglevariable_1d3d	5	418	79.103723
momentequationspositivitylimitermodule_dg	applypositivitylimiter_m1_dg	10	124	67.635828
wlinterpolationmodule	loginterpolatesinglevariable_1d3d	2	410	65.819025

Table 20: Trace of exclusive time spent in Thornado-mini loops

The results of this analysis are consistent with the expected performance of the thornado-mini radiation transport kernel. The kernel is essentially a compute intensive kernel with a primary loop that encompasses nearly all of the computation in the proxy app. In this analysis, the defined problem size has relatively low memory usage and a memory access pattern that indicates very little memory intensity.

8 MACSio Performance Assessment

Characterizing the I/O workload and performance of computational science applications has not been as well studied as other aspects of application performance. I/O studies are difficult because applications can use I/O for a variety of different purposes, such as initializing data structures, temporary storage of intermediate results that exceed memory capacity, recording or resuming the state of the computation, and saving the results of the computation, and these activities can produce significantly different workloads. Most studies to date rely on either recording I/O traces (e.g., Recorder [27]), or characterizing I/O using statistical methods (e.g., Darshan [9]), and then reproducing these with an I/O workload generator (e.g., IOR [33]). Both I/O trace and I/O characterization are limited to recording a specific run of a single application, and the I/O characterization technique throws away a lot of information, which makes it harder to regenerate an accurate workload. Additionally, generation of parallel I/O workloads typically requires more than just simulating read/write operations for each process, since there is usually some kind of synchronization involved between the processes as well. Unless these synchronization activities can also be simulated, the resulting I/O workload is unlikely to match the real workload with any accuracy. Benchmarking tools like IOR also focus mainly on sequential read and write performance using simple one-dimensional arrays of data. Although a large proportion of applications only use sequential I/O, not all do, and most employ complex multi-dimensional data structures rather than one-dimensional arrays.

MACSio is an effort to move up to higher layers of the I/O stack in order to better simulate a variety of different application I/O patterns for the purpose of assessing I/O performance. Rather than starting with low-level I/O primitives and adding additional delay and synchronization steps, MACSio begins by providing a framework that emulates a mesh-based computation using complex "real world" data structures such as those used for multi-physics applications like finite-element methods or computational fluid dynamics simulations. Various parameters are provided to control the structure of the mesh, how many variables are being simulated, and the volume and frequency of I/O operations, with the goal that this will approximate with some degree of accuracy the I/O behavior of real applications.

MACSio's funcationality can be extended using plugins that enable the tool to utilize a particular I/O library, such as HDF, netCDF, etc. These plugins allow finer grained control of performance and event logging, as well as providing additional capabilities for simulating more complex application models. MACSio enables accurate comparison with existing codes only when it has a plugin that utilizes the same data management and I/O middleware as the code it is being compared with. For example, without a LibMesh plugin for MACSio, it is rather difficult to compare other codes utilizing LibMesh for I/O. This is true even if LibMesh is utilizing a lower-level I/O library like Exodus or HDF5 (for which MACSio does have plugins) because of the rather large flexibility of using Exodus or HDF5 to perform I/O.

MACSio supports two parallel I/O paradigms, multiple independent file (MIF) where each process or group of processes accesses a separate file, and single shared file (SSF) where all processes access a single file. It should be noted that the current version of MACSio only supports I/O write operations. There are no options available to specify read operations. The key configuration parameters for MACSio are shown in Table 21.

For MACSio to be useful as an I/O workload proxy for a range of ECP applications, it needs to be able to accurately simulate a wide variety of different workload patterns, and the selection of the MACSio input parameters is crucial to how closely the target application I/O workload will be simulated. However, this can be a difficult task, since it requires detailed knowledge of the application data model, data distribution, and I/O behavior, so it would be advantageous if there

Parameter	Description
parallel_file_mode	This is used to choose between Multiple Input File (MIF) and Single Shared File (SIF) modes. This parameter takes an argument that specifies the number of files (MIF) or by grouping processes to produce the specified number of files (SIF).
part_type	Specifies the mesh type. Default 'rectilinear'.
part_dim	The dimension of the mesh. Default 2, but this does not seem to affect the I/O behavior.
part_size	The nominal I/O request size used by each task.
avg_num_parts	The average number of mesh elements (parts) per task, which also configures the total number of elements in the mesh as $avg\_num\_parts \times tasks$.
vars_per_part	The number of mesh variables per element, which specifies the number of I/O requests each task must make to complete a "dump".
num_dumps	The number of "dumps" to perform. A "dump" roughly correlates to a checkpoint, so this can be used to specify the number of checkpoints (usually based on checkpoint frequency and the number of timesteps executed.)
dataset_growth	Allows simulation of increasing dataset size by specifying a multiplier factor.
meta_size	Used to simulate the creation of additional "metadata" that is included in the output. This specifies the size of metadata objects that are included in the dump.
meta_type	Specify the type of the metadata objects (either tabular or amorphous.)
compute_work_intensity	Adds compute workload between "dumps", which seems to be the main mechanism for specifying a time delay between each dump, but can also simulate some level of computation.

Table 21: Main MACSio Configuration Parameters

was some means of obtaining at least some of this information in an automated manner.

Dickson, et al. [10] discussed one approach to simplifying the determination of correct MACSio input parameters by post processing Darshan characterization logs generated from application runs. We decided to adopt this approach to ascertain if it was a feasible way of reducing the complexity of generating parameters. However, in addition to the characterization logs, we also decided to collect Darshan trace information to provide further details about the I/O behavior of the application.

Our approach to configuring MACSio and assessing its I/O behavior was as follows:

1. Instrument the application to be proxied and collect Darshan characterization and trace logs.
2. Post-processes these logs to extract useful information, such as the number of processes/ranks, the total number of files generated, the number of files per process, the average I/O request size, and the number of dumps to marshal.
3. Obtain application specific information that can't be determined from the logs, such as mesh size and dimension, mesh levels, and parallel file mode.
4. Run a Darshan instrumented version of MACSio at the same scale and collect the logs.
5. Compare the application and MACSio logs to determine how closely they match.

After undertaking this exercise for a sample application, we assessed the feasibility of this approach, determined the limitations that currently exist in MACSio, and proposed a path forward for using MACSio as an I/O proxy application.

8.1 Performance Assessment

The purpose of this experiment was to assess the suitability of using MACSio as a proxy for the I/O workloads of a number of ECP applications. Our initial study was to see if MACSio was capable of emulating the I/O behavior of the Castro application. Castro [8] is a compressible hydrodynamics code that is used to simulate astrophysical flows. Castro utilizes adaptive mesh refinement (AMR) in order to reduce computational overheads. Castro is a good choice for this experiment because it has a reasonably complex AMR mesh (provided by the AMReX framework), and well defined I/O behavior for generating plot files, and reading and writing checkpoint files. The behavior of the plot and checkpoint files is controlled by configuration parameters in the Castro input file. When generating output files, Castro creates a directory structure that associates a subdirectory with each level of the AMR hierarchy, and writes one or more files to each directory. Castro uses AMReX's FArrayBox (FAB) data type for distributing data and hence computation across the mesh. These FABs are collected into MultiFABs which are shared across computational units. A MultiFAB is the fundamental data structure that is read/written to storage. Additionally, one of the processes writes a special file that describes how the MultiFABs are laid out on disk. Castro provides a configurable limit on the number of files that can be created.

8.2 Methodology

In order for MACSio to produce an accurate reproduction of an application I/O workload, it must be configured using the parameters described previously. A valid approach to this would be to inspect the application documentation to determine the configuration options associated with I/O. In the case of Castro, the AMReX options that control the mesh configuration, plot file generation, checkpoint generation, and checkpoint restart operations are defined in the configuration input file. By inspecting this configuration file, it may be possible to find a mapping between Castro (and AMReX) options and MACSio parameters.

As suggested by Dickson, et al. [10], we attempted to determine MACSio parameters by instrumenting the application using Darshan and post-processing the characterization logs. By default, Darshan characterization logs produce counts of the common I/O operations, such as file open, read/write, etc. This type of information only provides a limited ability to reconstruct the I/O behavior, since there is no way to correlate I/O activity to I/O request size, among other things. Recently, Darshan has added an extended trace format called DXT that provides provides additional details on individual I/O operations. Using Dickson's code as a reference, we wrote a small python program that read the Darshan logs and tries to summarize the I/O behavior in a manner that can be used to determine a variety of MACSio parameters.

For the experiment, we chose to use Castro's carbon detonation problem[5]. The documentation describes this problem as follows:

> This sets up a domain with a uniform density (dens). A large temperature (T_l) is placed in the left side of the domain, and an ambient temperature (T_r) is in the right. The parameter center_T specifies where, as a fraction of the domain length, to put the interface, while width_T determines how wide the transition region is, as a fraction of the domain length. Finally, the parameter cfrac is the initial C12 fraction (the remaining material is O16).
>
> The left side and right side can optionally approach each other with a non-zero infall velocity, vel.
>
> When run, the large temperature in the left side instantly flashes the C fuel, generating a lot of energy and a large overpressure. The signature of a propagating detonation is that a narrow energy generation zone will keep pace with the rightward propagating shock wave (as seen in the pressure field).

We increased the problem geometry to 32 x 32 x 32 and the number of AMR levels to 6 in order to increase the size of the resulting checkpoints. We also disabled plot file generation and checkpoint restart in order to concentrate exclusively on the checkpoint write behavior (MACSio only supports write operations.) We fixed the number of steps at 100, and configured checkpoints to be generated every 10 steps (10 checkpoints in total.) A number of application runs were then undertaken, with varying task counts up to 2048. Following each run, the generated Darshan logs were post-processed, and the computed parameters for each run were recorded.

MACSio produces a log file that details the I/O operations performed by each task, along with a timings file that records timing information. However it is not practical to compare this information with the information generated by Darshan as the format and information is considerably different. Instead, we also instrumented MACSio using Darshan in order to be able to directly compare the MACSio-generated Darshan logs with the Darshan logs from the application runs. Unfortunately, MACSio does not provide a parameter for turning off its own log file generation, but the generation of this information would likely interfere with the Darshan-generated logs. As a result, we had to modify the source code of MACSio in order to disable logging.

The tools and applications that were used for the experiment are listed in Table 22. The experiment was conducted on Titan, a Cray XK7.

8.3 Results

Figure 30 shows the aggregate I/O behavior of a 2,048 task run of the Castro application as captured by the Darshan logs. This run transferred 80.9 MB of data at 138.05 MB/s, generated

[5]`https://github.com/AMReX-Astro/Castro/tree/master/Exec/science/Detonation`

Tool/Application	Details
Darshan	Version 3.1.7 required to support DXT with std::ofstream
CrayPE	Version 2.x
PrgEnv-Intel	icc (ICC) 17.0.0 20160721
json-cwx	Required for MACSio (version 0.12-20140410)
Silo	Required for MACSio (version 4.10.2)
AMReX	Required for Castro (version 19.01-14-g8cc1f62d9c3a)
Castro	Version 19.01.1

Table 22: Tools & applications used to assess MACSio.

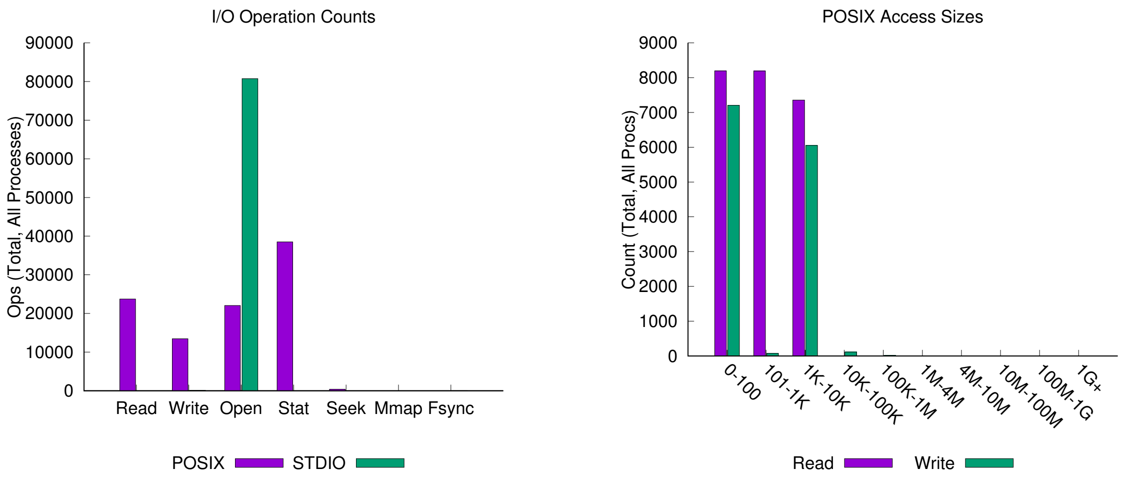


Figure 30: Castro Aggregate I/O Behavior (Read + Write)

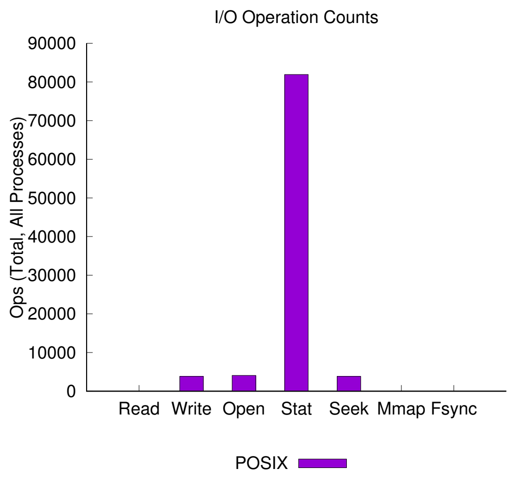
3,823 files, and performed 13,462 write operations and 23,740 read operations. The majority of write operations were in the ranges 0–100 bytes and 1K–10K bytes. Read operations were evenly split across the 0–10K byte range. The large number of STDIO open operations were thought to be generated by Castro log initializations even though STDIO logging was disabled in the Castro input file.

We decided that a good starting point for the MACSio analysis would be to see if MACSio could replicate this basic aggregated I/O behavior. However, it was immediately apparent that even replicating this behavior with MACSio would be problematic for two reasons. First, both read and write operations were being issued, but MACSio currently only supports write operations. Even if we focused only on the write behavior, the second issue was that the I/O access sizes vary from less than 100 bytes to over 100K bytes in a small number of cases. MACSio does not provide any support for varying write operation sizes. In order to proceed with the baseline case, we decided to use the average write size which was reported as 3.9K bytes by Darshan.

At best, only a few Castro input options could be mapped directly to MACSio parameters. The parameters *part_size* and *vars_per_part* could not be determined via a mapping, so they were instead derived from the Darshan log information. The *part_size* parameter sets the nominal I/O request size for each task, and the *vars_per_part* determines the number of I/O requests each task makes to complete a given dump, so these were chosen to the approximate the values from the Castro run. The final MACSio parameters that were used for the simulation are shown in Table 23.

MACSio Parameter	Source	Discussion
avg_num_parts	amr.n_cell x y z	Castro specifies the mesh dimensions, so the total number of mesh cells is $cells = x \times y \times z$ (for a 3-dimensional mesh). MACSio specifies the average number of mesh elements per task, we assume this is $\frac{cells}{tasks}$.
num_dumps	max_step amr.check_int	Castro controls the simulation time through either the number of time steps or a final stop time. Since MACSio requires an absolute number of dumps, we must use the number of time steps. Also, the number of checkpoints is determined by the amr.check_int option which specifies how often the checkpoint is written. Therefore $num\_dumps = \frac{max\_step}{amr.check\_int}$.
vars_per_part part_size	Darshan	We assume the checkpoint operation writes out the main state variables, but not the derived variables. According to the Castro documentation there are at least 22 variables. If we assume each variable is a Real (double) type, so is 8 bytes, this means a checkpoint should be at least $22{\times}8 \times mt$ bytes where mt is the number of mesh elements per task. Since the Darshan logs reported 20461770 bytes were written, we determined mt to be 56 for a 2048 task run. We approximated this by setting *vars_per_part* to 22 and using a per task I/O request size of 56.
parallel_file_model	none	We left this as the default, MIF
part_type	none	We left this as the default, 'rectilinear'
dataset_growth	castro.grown_factor	We assumed the domain is not growing, so this was set to 0
meta_size	none	We left this as the default (10K for each rank and 50K for rank 0), however it may need to be modified to more accurately represent the data
meta_type	none	This was left as the default, 'tabular'
compute_work_intensity	none	0

Table 23: Source for MACSio Parameters



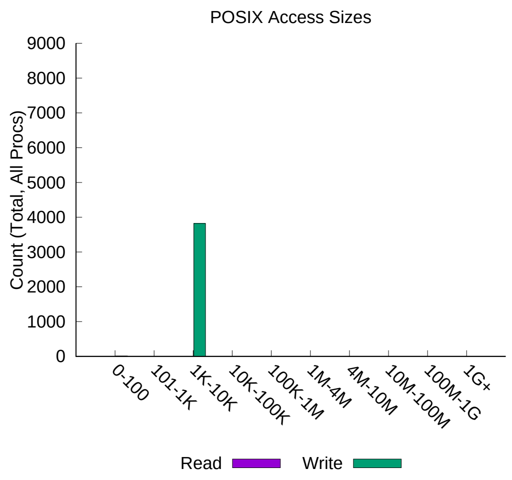


Figure 31: MACSio Aggregate I/O Behavior (Write Only)

	Castro	MACSio
files written	3,823	3,831
files read	3	0
total written (MB)	19.5	16.8
total read (MB)	61.4	0
total bandwith (MB/s)	138.05	22.10
read ops	23,740	0
write ops	13,462	3,830
open ops	22,055	4,068
stat ops	38,510	81,920
seek ops	410	3,820

Table 24: Comparison of I/O Behavior Between Castro and MACSio

Figure 31 shows the aggregate I/O behavior of the corresponding MACSio run using the derived parameters. For this run, Darshan reported 16.8 MB of data transferred at 22.1 MB/s, 3,831 files were generated, and 3,830 write operations were performed. As expected, virtually all the I/O operations were in the range 1K–10K bytes. Table 24 summarizes the differences in I/O behavior between the Castro and MACSio runs.

Although we ultimately did not use the tracing capability of Darshan, this is clearly a feature that would be beneficial for future work. However, we encountered two issues when enabling DXT trace logs. The first was that Castro utilizes the C++ std::ofstream library for checkpoint generation and only the latest version of Darshan (3.1.7 or later) captured I/O generated via this mechanism. Neither of our test systems had this version of Darshan available, so we were forced to build our own version. The second issue was that MACSio uses the C stdio library for generating its output, and this is not currently supported by the Darshan DXT module. Our solution to this was to modify MACSio to use file I/O rather than the stdio library, something that is clearly not an optimal solution. The Darshan team has indicated that they intend to rectify this in the next release[6].

[6]https://xgitlab.cels.anl.gov/darshan/darshan/issues/248#note_28920

8.4 Analysis

The results show that other than the total file count (3,823 vs. 3,831), the amount of data transferred (19.5 MB vs. 16.8 MB), and the write bandwidth (33.3 MB/s (estimated) vs. 22.1 MB/s) we were unable to get close correlation between the I/O behavior of Castro and MACSio. It's possible that by extensive experimentation with the *vars_per_part* and *part_size* parameters we might have achieved a closer approximation, but this would still be unlikely to give good correlation across the whole application run. To be fair however, MACSio is not designed to replicate low-level I/O behavior in this way, and tools such as IOR would be better suited to this.

The main issue is the use of AMR by Castro produces multiple files per mesh level, and as the Castro computation proceeds, the number of files and volume of data generated varies depending on the computation. MACSio, on the other hand assumes, a fixed mesh topology, and while there is the ability to increase the amount of data using the *dataset_growth* parameter, there is no capacity to vary the number of files generated at each checkpoint. The best that is achievable is to match the final file count and data volume, which is unlikely to provide a useful I/O comparison.

Additional factors that may also affect the ability to accurately replicate the I/O behavior, but were untested, include the fact that Castro generates multiple files in multiple levels of directory hierarchy, each level representing a level of the AMR mesh, as well as the number of read operations issued. MACSio uses a flat directory structure, so it is conceivable that directory-related operations may impact I/O behavior, particularly for vast numbers of files and directories on a parallel filesystem. As Castro is issuing both read and write operations, it is also possible that the combined behavior may also impact on overall I/O performance.

8.5 Future Work

It is clear from the results that without support for AMR, MACSio is not going to be able to accurately replicate the workload of a large class of ECP applications. However, it's possible that by limiting the AMR used by the applications to a single level rather than multiple levels, more accurate replication might be achieved in some limited cases. It would be worth studying this to see if the impact on the application performance was too significant to make it worthwhile.

MACSio is designed to be extensible via a plugin interface, and there is currently support for I/O emulation of HDF5, TyhonIO, PDB, and Exodus. It seems reasonable to assume that support for AMR could also be added via this plugin architecture. One approach to this might be to utilize an AMR library such as AMReX to handle the mesh generation. Discussions with the MACSio project lead have confirmed that this is a viable option.

It is perhaps unfortunate that our inital assessment used an AMR-based application rather than one that would be more compatible with MACSio, particularly as there are many other ECP applications that MACSio is likely to be able to reproduce more accurate I/O behavior. However, this experiment still serves are useful purpose in evaluating the capabilities of MACSio and the overall methodology for assessing MACSio as an I/O proxy application. Our experience with MACSio showed that it has promising capabilities for applications that utilize static meshes, so it would be well worth considering this for a future study.

9 GPU Performance Assessment of SW4 and SW4lite

The SW4 siesmic simulation code is used by the EQSIM ECP project, the objective of which is to provide simulation tools that can execute earthquake hazard (ground motions) and risk (structural response) simulations at frequencies pertinent to the evaluation of critical engineering infrastructure (0–10 Hz). The EQSIM team has developed the stand-alone SW4lite proxy application that solves the elastic wave equation with limited seismic modeling capabilities. SW4lite has been used for experiments with memory layouts, hybrid MPI/OpenMP threading, and GPU programming models to identify improvements to be incorporated into SW4. To achieve performance portability, the EQSIM team is focusing current development on using the RAJA portability layer. The team has used the SW4 RAJA version, together with the regional-scale model of the San Francisco Bay Area (SFBA), to reproduce the Hayward Fault science demonstration simulation with a model frequency resolution of 6.9 Hz, using 1024 nodes and 4096 GPUs on Sierra.

9.1 Algorithms and Key Kernels

SW4 stands for Seismic Waves, 4th order. It is a seismic wave propagation code that solves the seismic wave equations in displacement formulation using a node-based finite difference method. SW4 is fourth order accurate in space and time. Compared to a second order method, the error diminishes at a faster rate as the grid size is reduced. A fourth order method is also more efficient when the solution needs to remain accurate for longer times because the phase error grows at a slower rate. The fourth order method is also more accurate for calculating surface waves when the ratio between the compressional and shear wave speeds is large. Capabilities implemented in SW4 include a free surface boundary condition on the top boundary, absorbing super-grid conditions on the far-field boundaries, and an arbitrary number of point force and/or point moment tensor source terms.

SW4 supports a 3-D heterogeneous material model that can be specified in several input formats. The material model uses a curvilinear mesh near the free surface to represent the free surface boundary condition on a realistic topography. The curvilinear mesh is automatically generated from the topography description. To make SW4 computationally efficient, the seismic wave equations are discretized on a Cartesian mesh below the curvilinear grid. SW4 solves the equations in Cartesian coordinates. SW4 can be built to use the Proj.4 library for calculating the mapping between geographic and Cartesian coordinates, or it can use an approximate spheroidal mapping. SW4lite supports only the approximate spheroidal mapping. Cartesian local mesh refinement can be used to make the computational mesh finer near the free surface, where finer resolution may be needed to resolve short wave lengths in the solution. The mesh refinement is performed in the vertical direction using user specified refinement levels.

SW4 implements a super-grid modeling technique to reduce artificial reflections from the far-field boundaries. Layers are added around the domain of interest, and a stretching function is used in each of these layers to mimic a much larger physical domain. Super-grid layers are added by default to all sides of the computational domain except along the free surface, with a default thickness of 30 grid points.

The SW4 code consists of the following double-precision computational kernels for simulating high-fidelity seismic waves in realistic geological settings:

- a forcing function that simulates the earthquake source in space and time
- a stress calculation (RHS) that applies compact finite-difference stencils to displacements and material properties to calculate the divergence of the stress tensor.

nvprof metric	description
flop_count_dp	Number of double-precision floating-point operations executed
dram_read_bytes	Total bytes read from device memory to L2 cache
dram_write_bytes	Total bytes written from L2 cache to device memory
flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations
inst_integer	Number of integer instructions executed

Table 25: Metrics collected using Nvprof profiler

- supergrid damping near far-field domain boundaries
- boundary conditions on the free surface and far-field boundaries
- a predictor-corrector procedure for updating displacements in time

9.2 Code Versions Used

For this assessment, we ran used the most recent RAJA branch versions of SW4 (commit eafd30a) and SW4lite (commit 1349ab9) from their github repositories. We built the RAJA versions of the codes for the Summit supercomputer at ORNL using the instructions provided in the README files and the provided Makefile. Prior to building the codes, we downloaded and installed RAJA 0.7.0, as instructed in the READMEs.

9.3 Problem Selection and Validation

For this assessment we used two different test problems: gaussianHill and Berkeley. The gaussianHill-rev.in input was recommended to us by the SW4 developers as a scaled-down problem that is representative of the exascale challenge problem and exercises the relevant parts of the code with similar node performance. Both SW4 and SW4lite can run gaussianHill on a system with NVIDIA V100 GPUs; however,the code should be run on a single GPU because there is not sufficient work to scale to multiple GPUs without having the boundary conditions portion of the execution time grow and become dominant. The input specifies an isotropic source (explosion) with a Gaussian time function. An analytical Gaussian topography is specified with a given location, amplitude, and spread.

The Berkeley problem was also recommended by the SW4 developers as a representative input for work with the ECP Application Assessment Project. The berkeley-h20.in inputs for SW4 represent a heterogeneous material model with realistic topography, with the topology read from an input file. This problem exercises both the curvilinear and the Cartesian inner loops in the SW4 code. Note that SW4lite cannot run the Berkeley inputs because it does not have the capability of reading the topology from an input file.

Using these two problems, we can make two separate evaluations: First, whether SW4 and SW4lite have similar properties when run with the same input (gaussianHill), and second, whether the behavior of SW4 changes significantly depending on the problem (gaussianHill vs. Berkeley). We ran both inputs for 100 time steps. We ran the gaussianHill-rev.in input on one V100 GPU on a Summit node, and we used three V100 GPUs to run the berkeley-h20.in with a similar runtime. We measured the execution time overall and for each kernel, and we used nvprof to count number of calls per kernel and collect the metrics listed in Table 25 on a per-kernel granularity.

	berkeley-h20 problem			gaussianHill problem		
Kernel	time (%)	time (s)	calls	time (%)	time (s)	calls
Supergrid Damping						
W10addsgd4_c	20.64	1.7418	100	36.61	2.853	100
W11addsgd4_c	6.98	0.5891	100	0.69	0.0540	100
Cartesian Stress						
rhs4th3fort	18.87	1.5923	200	27.16	2.116	200
Curvilinear						
curvilinear4sg_c E0	8.45	0.7136	200	1.52	0.1184	200
curvilinear4sg_c E1	8.41	0.7096	200	1.44	0.1121	200
curvilinear4sg_c E2	7.36	0.6209	200	1.27	0.0992	200
curvilinear4sg_c E	2.59	0.2182	200	0.46	0.0361	200
curvilinear total	26.81			4.69		
Predictor-Corrector						
predfort_c	6.34	0.5355	200	7.11	0.5542	200
dpdmtfort_c	5.39	0.4549	200	5.99	0.4672	200
corrfort_c	5.09	0.4295	200	5.74	0.4470	200

Table 26: GPU activity profiles for SW4 running 100 time steps on berkeley-h20.in and gaussian-Hill_rev.in inputs.

9.4 Analysis

The SW4 developers told us that the curvilinear kernels, which do the stress calculation on the curvilinear grid, take most of the time on large science problems. They have split the curvilinear kernel into four smaller kernels in order to avoid register spilling. For SW4 with the berkeley-h20.in input, we see from the profiling data in Table 26 that 28% of the run time is spent in the supergrid damping kernels followed by 26% in the curvilinear kernels, and by 19% in the stress calculation on the Cartesian grid. In contrast, with the gaussianHill-rev.in input we see a very different distribution of timings. Table 26 shows that SW4 with gaussianHill spends 37% of the time in the supergrid damping kernel, followed by 27% in the stress calculations on the Cartesian grid, and the curvilinear kernels taking only 4.7% of the runtime. Although the number of calls for each of the curvilinear kernels is the same for SW4 on the two different inputs, the runtime breakdown by routine is much different, going from 27% for the curvilinear kenels on the Berkeley problem to 4.7% on the gaussianHill problem. In this respect, the gaussianHill input is not representative of the science problem, at least not at its current scale. The predfort, dpdmtfort, and corrfort kernels, which are all part of the predictor-correct step, have similar runtimes on the two inputs.

Turning to the comparison of SW4 and SW4lite, both running gaussianHill, we also see some significant differences between the two codes. The self-reported solver time is 7.8 seconds for SW4 and 10.2 seconds for SW4lite. Thus, the SW4lite solver times is about 30% higher on this input than the SW4 solver time. However, the SW4lite solver time includes both CPU and GPU activities. (Some routines that are ported to the GPU in SW4 run on the CPU in SW4lite.) The GPU activity times reported by nvprof for SW4 and SW4lite are shown in Table 27. Two of the curvilinear kernels (E and E0) take substantially more time in SW4lite. There is also a major difference is the GPU times for the addsgd kernels, with this kernel taking 2.9 seconds with SW4 and 1.24 seconds for SW4lite. However, NVTX Range instrumentation shows that SW4 spends 2.9 seconds in the supergrid damping step, including both CPU and GPU time, while SW4lite spends 4.9 seconds in this step. Looking at the source code, it appears that while most of the

	SW4lite gaussianHill			SW4 gaussianHill		
Kernel	time (%)	time (s)	calls	time (%)	time (s)	calls
Supergrid Damping						
addsgd4	19.64	1.239	100			
W10addsgd4_c				36.61	2.853	100
W11addsgd4_c				0.69	0.0540	100
Cartesian Stress						
rhs4sg_rev_c	30.88	1.948	200			
rhs4th3fort				27.16	2.116	200
Curvilinear						
rhs4sgcurv_rev E	4.49	0.2834	200			
rhs4sgcurv_rev E0	4.2	0.2652	200			
rhs4sgcurv_rev E1	1.77	0.1115	200			
rhs4sgcurv_rev E2	1.59	0.1001	200			
curvilinear4sg_c E0				1.52	0.1184	200
curvilinear4sg_c E1				1.44	0.1121	200
curvilinear4sg_c E2				1.27	0.0992	200
curvilinear4sg_c E				0.46	0.0361	200
curvilinear total	12.05			4.69		
Predictor-Corrector						
predfort	8.92	0.5626	200			
dpdmtfort	11.4	0.7191	200			
corrfort	7.21	0.4550	200			
predfort_c				7.11	0.5542	200
dpdmtfort_c				5.99	0.4672	200
corrfort_c				5.74	0.4470	200

Table 27: GPU activity profile for SW4lite and SW4 running 100 time steps on gaussianHill input

addsgd routine in SW4 has been ported to the GPU, only part of the corresponding routine in SW4lite has been ported. This difference explains most of the discrepancy in the solver runtimes.

To focus on comparing the performance of the curvilinear kernels, we show and analyze the metrics for floating point and memory performance. The relevant data for both SW4 and SW4 lite are shown in Table 28. The operation and byte counts shown are the average per kernel call, and variance between calls for these metrics was small. While the counts for flop_count_dp and dram_bytes for the SW4 curvilinear kernels on the Berkeley input are about six times higher, respectively, than for the same metrics for SW4 on the gaussianHill input, the ratio of these two metrics, flops per byte, also known as the aritmetic intensity, or A.I., is about the same. SW4 and SW4lite curvilinear kernels have similar counts for these metrics on the gaussianHill input and thus also have similar arithmetic intensity. We calculated the %peak metric shown in the tables since the flop_dp_efficiency metric reported by Nvprof was sometimes unrealistic, especially for short-running kernels. %peak for the curvilinear kernels is similar for SW4 on both inputs, but %peak for two of the SW4lite curvilinear kernels (E and E0) is significantly lower.

We next looked at cache hit rates and achieved memory bandwidth for the curvilinear kernels across the two applications and the two inputs. Data is shown in Table 29. Cache hit rates are similar except that the L2 cache hit rates are slightly higher for SW4lite on the gaussianHill input. Achieved memory bandwidths are similar except for the drastically lower bandwidths for two of

SW4 berkeley input

kernel	time/call(ms)	flop_count_dp	dram_bytes	A.I.	FLOPS/s	%peak
curvilinear4sg_c E0	3.568	7741841600	1714171264	4.52	2.17E+12	27.8
curvilinear4sg_c E1	3.548	7821654400	1723211040	4.54	2.20E+12	28.2
curvilinear4sg_c E2	3.104	6315187800	1713961312	3.68	2.03E12	26.1
curvilinear4sg_c E	1.091	3591576000	125507872	28.6	3.29E12	42.2

SW4 gaussianHill input

kernel	time/call(ms)	flop_count_dp	dram_bytes	A.I.	FLOPS/s	%peak
curvilinear4sg_c E0	0.5922	1271398400	288208800	4.41	2.15E+12	27.5
curvilinear4sg_c E1	0.5605	1284505600	288971360	4.45	2.29E+12	29.3
curvilinear4sg_c E2	0.4958	1037107200	287216384	3.61	2.09E12	26.8
curvilinear4sg_c E	0.1806	489553920	18175456	26.9	2.71E12	34.7

SW4lite gaussianHill input

kernel	time/call(ms)	flop_count_dp	dram_bytes	A.I.	FLOPS/s	%peak
rhs4sgcurv_rev E0	1.311	1269760000	288091456	4.41	9.58E+11	12.3
rhs4sgcurv_rev E1	.5572	1282867200	288953760	4.44	2.29E+12	29.5
rhs4sgcurv_rev E2	.4964	1035468800	287065920	3.61	2.09E12	26.5
rhs4sgcurv_rev E	1.401	470384640	18175456	25.8	3.32E+11	4.26

Table 28: Floating point and memory access data for SW and SW4lite

SW4 berkeley input

kernel	L1 hit rate	L2 hit rate	Achieved memBW (GB/s)
curvilinear4sg_c E0	83.02	55.18	480
curvilinear4sg_c E1	81.02	59.45	486
curvilinear4sg_c E2	79.43	53.02	552
curvilinear4sg_c E	95.5	68.32	115

SW4 gaussianHill input

kernel	L1 hit rate	L2 hit rate	Achieved memBW (GB/s)
curvilinear4sg_c E0	83.78	53.6	486
curvilinear4sg_c E1	81.39	59.1	516
curvilinear4sg_c E2	79.89	52.9	579
curvilinear4sg_c E	95.81	69.5	101

SW4lite gaussianHill input

kernel	L1 hit rate	L2 hit rate	Achieved memBW (GB/s)
rhs4sgcurv_rev E0	83.88	59.70	217
rhs4sgcurv_rev E1	81.70	63.75	518
rhs4sgcurv_rev E2	80.07	58.90	573
rhs4sgcurv_rev E	95.71	74.61	12.9

Table 29: Cache hit rates and memory bandwidths for SW4 and SW4lite curvilinear kernels

the SW4lite curvilinear kernels (E and E0).

9.5 Is SW4lite a Good Proxy for SW4 for Representing GPU Performance?

Although SW4 and SW4lite GPU performance on the gaussianHill input is similar in some respects, it seems there may be performance bugs in two of the SW4lite curvilinear kernels that cause them to run more slowly. Although the Berkeley and gaussianHill inputs result in similar arithmetic intensities, they produce different runtime profiles when run with SW4. This is somewhat problematic since users of proxy applications often profile them to determine where most of the time is being spent so as to focus performance evaluation and optimization efforts. Having such a short time spent in the SW4lite curvilinear kernels on the gaussianHill input also means that performance evaluation may be less accurate, since metrics collected for such short runtimes may be unreliable. Thus, we recommend scaling up the gaussianHill input to a problem size comparable to the Berkeley problem if possible. If the suspected performance bugs in two of the SW4lite curvilinear kernels could be fixed, and if the gaussianHill input could be scaled up to be representative of the Berkeley problem, then SW4lite with the gaussianHill input could potentially be a good proxy-input pair for representing SW4 GPU performance on the challenge problem. If supergrid damping is an important kernel for the exascale problem, then it would also be good to have similar GPU activity and performance for this kernel between SW4 and SW4lite.

10 GPU Performance Assessment of Nek5000 and Nekbone

Nek5000 is a computational fluid dynamics (CFD) code with spatial discretization based on the spectral element method (SEM). Nek5000 is designed specifically for transitional and turbulent flows in domains and is used by the ExaSMR and Urban ECP application development projects. Nek5000 solves the unsteady incompressible two-dimensional, axisymmetric, or three-dimensional Stokes or Navier-Stokes equations with forced or natural convection heat transfer in both stationary or time-dependent geometry. It also solves the compressible Navier-Stokes in the low Mach regime. Nekbone is a mini-app derived from Nek5000 that exposes the principal computational kernels of Nek5000. Nekbone solves a standard Poisson equation using the spectral element method with an iterative conjugate gradient solver with a simple preconditioner. The Center for Efficient Exascale Discretizations (CEED) ECP Co-design project is also involved with Nek5000 and Nekbone development.

The objective of the ExaSMR project is to carry out extreme-fidelity simulations of small modular reactors (SMRs). The analysis of SMRs is complex because of the multiphysics nature of the problem. The ExaSMR project is developing a coupled Monte Carlo (MC) transport multiphase computational fluid dynamics (CFD) code with exascale capability. The focus on the CFD side is on demonstrating scaling up to a full reactor core for high-fidelity simulations of turbulence. A zonal hybrid approach is being used in which large eddy simulation (LES) is used to simulate a portion of the core and unsteady Reynolds-averaged Navier-Stokes (URANS) handles the rest.

10.1 Algorithms and Key Kernels

The SEM is a high-order weighted residual technique that combines the geometric flexibility of finite elements with the rapid convergence and tensor-product efficiencies of global spectral methods. The SEM is based on a decomposition of the domain into smaller subdomains, called elements, that are curvilinear hexahedra, called bricks. Functions within each element are expanded as Nth-order polynomials case in tensor-product form. Typical discretizations involve 100–10,000 elements of order 8–16 (corresponding to 512–4096 points per element). Temporal discretization is based on a high-order splitting that is third-order accurate in time and reduces the coupled velocity-pressure Stokes problem to four independent elliptic solves per timestep [28]. The velocity problems are diagonally dominant and solved using Jacobi-preconditioned conjugate gradient iteration. The pressure substep requires a Poisson solve at each step, which is carried out using multigrid-preconditioned GMRES iteration coupled with temporal projection.

Nekbone is intended to represent a Poisson problem similar to the pressure solve in Nek5000. Nekbone implements the conjugate gradient (CG) solver and includes domain decomposition and MPI communication for simple bricklike and linear arrangements of the elements. Newer CPU versions of Nekbone also contain a multigrid preconditioner (MG), but the MG portion is not implemented in the GPU versions of Nekbone.

Recent work on Nek5000 has focused on GPU performance improvements. Short-term efforts on improving key kernels (e.g., matrix-vector multiplication) in the OpenACC implementation are limited in performance improvement. Medium-term efforts use a libParanumal interface to leverage optimal GPU kernels in libParanumal and have achieved more than a 4 times speedup on a single GPU. Long-term efforts will revise Nek5000 kernels using a libCEED interface.

10.2 Problem Selection and Validation

The ExaSMR team has developed two performance benchmark problems for Nek5000: 1) a subchannel (single-rod) problem to assess internode performance, and 2) a larger full-assembly problem

kernel	time(s)	calls	time/call(s)	dram_bytes	dp_flops	A.I.	FLOPS/s	%peak
hsmg_do_fast_acc	0.06836	186	3.68E-04	6.843E09	3.256E10	4.76	4.763E11	6.11
sumab_acc	0.06626	15	4.417E-03	4.719E08	9.83E07	0.208	1.48E09	0.019
axhelm_acc	0.03638	218	1.669E-04	6.858E09	5.71E08	0.083	1.57E10	0.201
hsmg_extrude	0.0491	939	5.226E-05	1.891E10	1.727E09	0.091	3.52E10	0.451
global_div3_acc	0.0267	218	1.225E-04	6.859E09	1.37E10	2.00	5.14E11	6.58
global_grad3_acc	0.01600	218	7.320E-05	2.287E09	1.37E10	6.00	8.60E11	11.0

Table 30: Timing and hardware counter profiles for Nek5000 running 5 steps on single-rod input with order 7

kernel	time(s)	calls	time/call(s)	dram_bytes	dp_flops	A.I.	FLOPS/s	%peak
hsmg_do_fast_acc	0.23443	218	1.08E-03	5.098E10	1.338E11	2.62	5.71E11	7.32
sumab_acc	0.20758	15	1.38E-02	1.593E9	3.318E8	0.208	1.60E09	0.020
axhelm_acc	0.16281	293	5.56E-04	4.148E10	2.59E10	0.062	1.59E10	0.204
hsmg_extrude	0.17374	1099	1.58E-04	6.395E10	4.05E10	0.063	2.33E10	0.299
global_div3_acc	0.13765	293	4.70E-04	3.111E10	9.33E10	3.00	6.78E11	8.69
global_grad3_acc	0.09668	293	3.30E-04	1.0374E10	9.33E10	9.00	9.65E11	12.4

Table 31: Timing and hardware counter profiles for Nek5000 running 5 steps on single-rod input with order 11

representative of an SMR. Since we are primarily interested in assessing GPU performance, we chose to use the single-rod problem. The single-rod problem can be run with Nek5000 but not with Nekbone. Nekbone takes as input a data file that specifies the number of elements per processor and the polynomial order. Thus, the closest we can come with Nekbone is to set these parameters to the same values as for the Nek5000 single-rod problem.

As in [28], we ran the single-rod problem with polynomial orders $N = 7$ and $N = 11$, using the case files in short_tests/singlerod in the openacc branch in the Nek5000 github repository. We built and ran the Nek5000 code on a V100 GPU on ORNL Summit using the build and run scripts specifically provided for Summit for this test (makenek.acc.summit and nekgpu.nek5000.summit, respectively). This input set has 2600 elements per processor, and since we are using a single process, this puts 2600 elements on the GPU.

The computational core of the multigrid preconditioner in Nek5000 is a sequence of tensor products over local elements. The Nek5000 kernel "hsmg_do_fast" contains the local tensor products in the multigrid solver for the Poisson equations. We profiled the runs and collected floating point and memory transaction data using the NVIDIA nvprof tool. Results are shown in Tables 30 and 31. Similar to the results reported in [28], we see that the hsmg_do_fast routine takes the most time, and that the arithmetic intensity of this routine decreases with the increase in polynomial order, even though increasing the polynomial order increases the number of floating point operations. The number of steps for Summit is set to 5, resulting in very short runtimes, and there is a comment in the input file that this was done for profiling purposes. We modified the input file to run both cases for 50 steps and found that the number of calls and the timings and hardware counter values for all the routines increased consistently by close to 10 times, with times and counts per call remaining about the same.

We configured the Nekbone input file to have the same number of elements and polynomial orders as the single-rod input case for Nek5000—i.e., 2600 elements and orders 7 and 11. The

order	time(s)	calls	time/call(s)	dram_bytes	dp_flops	A.I.	FLOPS/s	%peak
7	0.0841	200	4.204E-04	1.49E10	2.95E10	1.98	3.51E11	4.51
11	0.3041	200	6.202E-04	5.06E10	1.41E11	2.78	4.62E11	5.93

Table 32: Timing and hardware counter results for CUDA Nekbone CG solver with 2600 elements

N	nelt	dp_flops	CUDA time(s)	CUDA GFLOPS	OpenACC time(s)
7	131072	1.02E+12	3.108	327	
7	65536	5.08E+11	1.55	328	
7	32768	2.54E+11	0.8295	306	
15	16384	1.02E12	3.136	324	3.848
15	8192	5.08E+11	1.604	317	1.976
15	4096	2.54E+11	0.8483	300	1.045

Table 33: Timing results for solver portions of CUDA and OpenACC versions of Nekbone with maximum numbers of elements for orders $N = 7$ and $N = 15$

profiling results for the Nekbone CUDA version CG solver are shown in Table 32. The Nekbone OpenACC version did not appear to run properly on this input, since it produced NaN results.

Results have previously been reported for Nekbone running at close to 300 GFlops/s on the P100 GPU [13], but these results were for a much larger problem. We configured Nekbone to run with the maximum number of elements possible, given memory constraints, for orders 7 and 15. The timing results for these larger problems are shown on Table 33 for both the CUDA and OpenACC versions of Nekbone. We can see that the OpenACC version runs around 20% more slowly than the CUDA version and that the CUDA version achieves over 300 GFLOPS/s.

10.3 Is Nekbone a Good Proxy for Nek5000 for Representing GPU Performance?

Nekbone is intended to represent the principal computational kernels of Nek5000. More specifically, Nekbone is intended to represent a Poisson problem similar to the pressure solve in Nek5000. There are two GPU implementations of Nekbone available—a CUDA version and an OpenACC version. These GPU versions implement the conjugate gradient (CG) solver and not the multigrid preconditioner. The main kernel in the GPU versions of Nekbone is the matrix-vector multiplication (the ax_cuf kernel in the CUDA version and the two ax_acc_xxx kernels in the OpenACC version).

While there are a large number of input configuration options for defining the geometry and physics of the problem with Nek5000, configuration options for Nekbone are limited to specifying the number of elements and the polynomial order. Specifying these inputs appears to determine the exact number of floating point operations to be performed. We were able to configure Nekbone to have the same number of elements and polynomial order as the single-rod input case for Nek5000, but Nekbone is not running the same physics problem. Since the only kernel implemented in the GPU versions of Nekbone is the CG kernel, we looked for a corresponding kernel in the profiling results for Nek5000. Our best guess is that the axhelm_acc kernel is somewhat similar, since it also does matrix-vector multiplication. However, even with the same number of elements and polynomial order, the arithmetic intensity of the CG kernel in Nekbone is substantially different from that of the axhelm_acc kernel in Nek5000. This is also not the kernel that takes most of the time for the single-rod problem in Nek5000.

For Nekbone to adequately represent Nek5000 with respect to the exascale challenge problem for ExaSMR, it should be able to run the most critical kernels for the same physics problem. The relationship between corresponding kernels in Nek5000 and Nekbone should be either obvious or well-documented so that interested parties (e.g., vendors, architecture researchers) can know how to use the proxy app to evaluate different architectures, programming models, performance optimizations, etc., in order to benefit full application performance on exascale architectures. If a library-based approach will be used for GPU implementations of Nek5000 going forward (e.g., libParanumal or libCEED), then the proxy app should also use this approach if it can be implemented without introducing a large number of software dependencies.

The input cases provided are very small problems for the GPU.

Our conclusion at this point is that Nekbone does not adequately represent the GPU performance of Nek5000 with respect to the exascale challenge problems and should not be used as proxy to evaluate proposed GPU architectures for exascale machines.

11 Summary and Conclusion

We have performed a comparison of various ECP proxy application to their respective parent applications with an emphasis on application performance. In each case we have been careful to use problem specifications that are as similar as possible between proxy and parent and to use problems that are representative of Exascale-class problems.

We find that some proxies have performance characteristics that are highly representative of their parents. miniQMC, miniVite, and XSBench are particularly good in this regard. Other proxies demonstrate considerable performance difference. In some cases these differences can be traced to differences in the algorithms or problem specification, in others the root cause of the difference is unclear.

In light of these findings, we caution users of proxy apps that it is unwise to use proxy apps without some understanding of their characteristics. It is especially unwise to use the default problem of proxy applications as canned benchmarks unless they have been specifically configured by the authors for that purpose. Proxy applications are excellent tools, but like any tool can be harmful when misused.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] 9th dimacs implementation challenge. http://users.diag.uniroma1.it/challenge9/competition.shtml.

[2] Developer zone. https://software.intel.com/en-us/home.

[3] Exastar: Multi-physics stellar astrophysics simulations at the exascale. URL: https://sites.google.com/lbl.gov/exastar/home.

[4] P-I-C-S-A-R: Particle-In-Cell Scalable Application Resource. https://picsar.net/.

[5] Towards a discontinuous galerkin method for the multi-group two-moment model of neutrino transport. URL: https://github.com/ECP-Astro/thornado_mini/blob/master/Documents/endeve_etal_2017_ORNL_TM_501.pdf.

[6] O. Aaziz, J. Cook, J. Cook, and C. Vaughan. Exploring and quantifying how communication behaviors in proxies relate to real applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 12–22, Nov 2018. doi:10.1109/PMBS.2018.8641569.

[7] Gerald Alexanderson. About the cover: Euler and königsberg's bridges: A historical view. *Bulletin of the american mathematical society*, 43(4):567–573, 2006.

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

[9] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. Storage*, 7(3):8:1–8:26, October 2011. URL: http://doi.acm.org/10.1145/2027066.2027068, doi:10.1145/2027066.2027068.

[10] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. 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, Nov 2016. doi:10.1109/PDSW-DISCS.2016.007.

[11] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895. IEEE, 2018.

[12] Sayan Ghosh, Mahantesh Halappanavar, Antonio Tumeo, Ananth Kalyanaraman, and Assefaw H Gebremedhin. minivite: A graph analytics benchmarking tool for massively parallel systems. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 51–56. IEEE, 2018.

[13] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul F. Fischer, and Misun Min. Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. *The Journal of Supercomputing*, 72:4160–4180, 2016.

[14] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable static and dynamic community detection using grappolo. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.

[15] Åke J Holmgren. Using graph models to analyze the vulnerability of electric power networks. *Risk analysis*, 26(4):955–969, 2006.

[16] https://docs.qmcpack.org/qmcpack_manual.pdf. Qmcpack: User's guide and developer's manual. URL: https://docs.qmcpack.org/qmcpack_manual.pdf.

[17] https://github.com/ovis-hpc/ovis/wiki. The ovis high performance computing monitoring, analysis, and visualization project. URL: https://github.com/ovis-hpc/ovis/wiki.

[18] https://github.com/QMCPACK/miniqmc/wiki/Comparing-performance-with-QMCPACK. Comparing performance with qmcpack. URL: https://github.com/QMCPACK/miniqmc/wiki/Comparing-performance-with-QMCPACK.

[19] https://github.com/QMCPACK/miniqmc/wiki/How-to-build-and-run-miniQMC. How to build and run miniqmc. URL: https://github.com/QMCPACK/miniqmc/wiki/How-to-build-and-run-miniQMC.

[20] https://github.com/RRZE-HPC/likwid. Likwid performance monitoring and benchmarking suite. URL: https://github.com/RRZE-HPC/likwid.

[21] https://hpctoolkit.org. Hpctoolkit. URL: https://hpctoolkit.org.

[22] https://software.intel.com/en-us/vtune-amplifier-help. Intel® vtune^{TM} amplifier 2019 user guide. URL: https://software.intel.com/en-us/vtune-amplifier-help.

[23] https://sourceware.org/binutils/docs/gprof/. Gnu gprof. URL: https://sourceware.org/binutils/docs/gprof/.

[24] Md Jamiul Jahid and Jianhua Ruan. Identification of biomarkers in breast cancer metastasis by integrating protein-protein interaction network and gene expression data. In *2011 IEEE International Workshop on Genomic Signal Processing and Statistics (GENSIPS)*, pages 60–63. IEEE, 2011.

[25] Steve Kaufmann and Bill Homer. Craypat-cray x1 performance analysis tool. In *Proceedings of Cray User Group Meeting*, 2003.

[26] Sergii V Kavun, Irina V Mykhalchuk, Nataliya I Kalashnykova, and Oleksandr G Zyma. A method of internet-analysis by the tools of graph theory. In *Intelligent Decision Technologies*, pages 35–44. Springer, 2012.

[27] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A multi-level approach for understanding i/o activity in hpc applications. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 1–5, 09 2013. doi:10.1109/CLUSTER.2013.6702690.

[28] E. Merzari, R. Rahaman, Min. M., and P. Fischer. Performance aqnalysis of Nek5000 for single-assembly calculations. In *Fluids Engineering Division Summer Meeting*, volume 2, 2018. doi:10.1115/FEDSM2018-83517.

[29] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[31] Jean-François Rual, Kavitha Venkatesan, Tong Hao, Tomoko Hirozane-Kishikawa, Amélie Dricot, Ning Li, Gabriel F Berriz, Francis D Gibbons, Matija Dreze, Nono Ayivi-Guedehoussou, et al. Towards a proteome-scale map of the human protein–protein interaction network. *Nature*, 437(7062):1173, 2005.

[32] S.D.Hammond, C.T.Vaughan, and C.Hughes. Evaluating the intel skylake xeon processor for hpc workloads. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 342–349, July 2018. doi:10.1109/HPCS.2018.00064.

[33] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=1413370.1413413.

[34] WarpX collaboration. The electromagnetic Particle-In-Cell method. https://ecp-warpx.github.io/doc_versions/dev/theory/picsar_theory.html.

[35] WarpX collaboration. WarpX documentation. https://ecp-warpx.github.io/.

[36] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014. doi:10.1109/ISPASS.2014.6844459.

[37] Jitao David Zhang and Stefan Wiemann. Kegggraph: a graph approach to kegg pathway in r and bioconductor. *Bioinformatics*, 25(11):1470–1471, 2009.