A Methodology for Characterizing the Correspondence Between Real and Proxy Applications

Omar Aaziz
Sandia National Laboratories
Albuquerque, NM 87123
Email: oaaziz@sandia.gov

Jeanine Cook
Sandia National Laboratories
Albuquerque, NM 87123
Email: jeacook@sandia.gov

Jonathan Cook
New Mexico State University
Las Cruces, NM 88003
Email: joncook@nmsu.edu

Tanner Juedeman
Sandia National Laboratories
Albuquerque, NM 87123
Email: tjuedem@sandia.gov

David Richards
Lawrence Livermore National Laboratory
Livermore, CA 87123
Email: richards12@llnl.gov

Courtenay Vaughan
Sandia National Laboratories
Albuquerque, NM 87123
Email: ctvaugh@sandia.gov

***Abstract*—Proxy applications are a simplified means for stakeholders to evaluate how both hardware and software stacks might perform on the class of real applications that they are meant to model. However, characterizing the relationship between them and their behavior is not an easy task. We present a data-driven methodology for characterizing the relationship between real and proxy applications based on collecting runtime data from both and then using data analytics to find their correspondence and divergence. We use new capabilities for application-level monitoring within LDMS (Lightweight Distributed Monitoring System) to capture hardware performance counter and MPI-related data. To demonstrate the utility of this methodology, we present experimental evidence from two system platforms, using four proxy applications from the current ECP Proxy Application Suite and their corresponding parent applications (in the ECP application portfolio). Results show that each proxy analyzed is representative of its parent with respect to computation and memory behavior. We also analyze communication patterns separately using mpiP data and show that communication for these four proxy/parent pairs is also similar.**

***Index Terms*—Workload characterization; Proxy applications; Performance evaluation; Big data**

I. Introduction

Proxy applications, sometimes called mini-apps or representative applications, are relatively small programs that attempt to capture some fundamental aspects of a real, and much larger, application or class of applications. Being much smaller, proxy apps are designed to be easily built, installed, and executed. They typically have few build constraints and dependencies, simple input specifications, and are thus usable with little human overhead or time commitment. Their purpose is to provide an easy-to-use-mechanism to evaluate system performance, find hardware bottlenecks, perform algorithmic, parallel, or system design exploration, and in general gain an understanding of how a particular class of applications might perform on an HPC system, and how to best design and configure both the system and the application to maximize performance.

Underlying all this is the assumption that the proxy app does indeed capture the essence(s) of the real application. Answering the question of the validity of this assumption has long been a relevant issue, not just for proxy apps but for benchmark suites and other forms of indirect assessment of system performance. As applications continue to be created and evolved, proxy apps do as well, and the systems they run on continue to change; thus, a usable, continuous methodology for assessing the correspondence of proxy apps to real applications would and will be beneficial to the community.

Without a methodology or support framework, a one-off attempt at assessing the relationship between a proxy app and a real application might involve by-hand instrumentation of the real application in order to gather some statistics that are similar to what the proxy app gathers (proxies typically are built to output data regarding their own performance), and then use some appropriate data analytics to compare the two data sets. This would obviously be an intensive effort that many would not have the expertise to perform—only someone very familiar with the large and complex codebase of the real application would be able to do this. Fairly recent work appears to have done just this [1], though the same group also generalizes their work [2].

Moreover, it is not clear that this kind of comparison is necessarily appropriate. For example, the proxy app CoMD, which models molecular dynamics computation, outputs among many other things an “average atom update rate” and an “average atom rate.” But should this rate be compared to what a real MD application achieves? Should the more complete and complex codebase of the real application, which may have conditions and constraints that were ignored in the simplified proxy app, be expected to achieve a similar atom update rate on the same system? If not, how different should they be?

How would we know if our real MD application is achieving comparable performance to what CoMD achieved?

Given these difficulties, it is not surprising then that very recent work has somewhat sidestepped the issue; in one example, extracting kernels from the real apps instead of comparing to proxy apps [3], and in another first comparing proxy apps to benchmarks as an intermediate step towards comparing to real applications [4].

In this paper we propose a methodology that evaluates the performance of both the real and proxy applications at a level below that of their domain-specific performance: that of how they exercise the hardware. This avoids the issue of domain-specific metrics and targets the real question of performance and correspondence, which is: do they utilize and exercise the HPC system in similar ways? While this cannot be made too simplistic, such as trying to boil it down to a single metric like instructions per cycle, a holistic view of how the system is utilized should provide a solid foundation for comparing the proxy app to the real application.

II. Methodology

Overall the basis of our methodology is to characterize a run of an application, either real or proxy, by a vector of k values, and then perform clustering and other similarity characterizations of the vectors. We hypothesize several outcomes of this:

- runs of an application will have similar vectors;
- runs of proxy applications should have vectors similar to the vectors of the real applications they represent; and
- runs of different applications should have vectors that are different.

In the above, we purposely do not rigorously define *similar* and *different*, as the paper explores and answers these below. We also define k below.

For a scientific application that runs on HPC platforms, its performance will depend on how it utilizes the following *resource domains*:

- Basic node resources (processors and memory);
- Accelerator resources (e.g., GPUs);
- Communication resources (node interconnect); and
- Storage I/O resources (file systems).

Not only will its performance depend on using these resources, it can also be *characterized* by how it uses these resources. For the purposes of this paper, we only consider the basic node resources and the communication resources. Most proxy applications ignore the issue of filesystem usage, even though this can be critical, and the initial set of proxy applications we are investigating do not use accelerators. Both of these areas are important areas of future work, and we believe they can be addressed similarly to how we approach the two resource domains that we use. We also ignore the issue of shared resource contention (e.g., interconnect contention), and instead assume that for data-collection runs this issue is controlled and alleviated.

For each resource domain r, a set of metrics M_r can be defined that characterize its usage. These metrics should be generic and not change from application to application, or even platform to platform, if a general methodology is to be created. The metrics also have to be relatively *easily collectable*, as the instantiation of our methodology should not be burdensome or depend on many hard-to-use tools or custom configurations. Collecting the data should also not involve high-overhead, high-variance instrumentation costs. For example, tools such as Pin [5] can provide extremely detailed data without internally instrumenting applications, but the overhead can be extremely high and vary by orders of magnitude, making time-based data comparison hard. These constraints lead us to currently avoid anything that might involve internal application instrumentation or high-overhead invasive instrumentation, although in future work the use of both will be addressed.

For the basic node resource domain bn (processors and memory), we take advantage of the ubiquitous hardware counters that are available. While each hardware platform can have some set of unique counters, the most basic counters, such as instructions, CPU, and memory activity, are available on essentially every platform.

The metric set M_{bn} we choose for the basic node domain is:

1) IPC, Instructions per cycle
2) UIPC, Micro-ops per cycle (Intel/AMD platforms only)
3) L1, L2, L3 miss rate: uses the total number of load instructions as the denominator; number of misses to a particular cache level as the numerator.
4) L1, L2, L3 miss ratio: uses the number of accesses to the particular cache level as the denominator; number of misses to a particular cache level as the numerator.
5) L1 to/from L2 bandwidth
6) L2 to/from L3 bandwidth
7) Instruction mix: Floating point, load, store, branch, and other (mostly integer) instructions. We compute each instruction category as a percentage of the total instructions committed.
8) Floating-point operations per second (FLOPS): uses subcategories of the FP instruction mix computed above to compute FLOPS for various categories of instruction types. This is not available on all platforms.

Note that although L3 to/from DRAM bandwidth is typically a measurable quantity, our systems are such that we can not currently measure this. This will be addressed and hopefully changed in the future.

For the communication resource domain cm, network interface cards often also have internal counters that can be accessed, though these are much less standard and less easily accessed than the CPU hardware counters. Operating systems often have metrics available, such as the /proc/net data available under Linux, but these can also be different for different network types. As these get easier to use this domain could potentially move towards using metrics based on these counters and data. For this paper, since we are using real and proxy applications that all use MPI, we use the *mpiP* lightweight profiling library [6] to collect basic communication metrics

TABLE I: Most Significant mpiP Per-Routine Raw Metrics

Metric type	Description	Routines for
Apptime_%	% of total application time	send, isend, recv, irecv, sendrecv, allreduce, bcast, wait, waitall, barrier
MPI_%	% of total MPI time	send, isend, recv, irecv, sendrecv, allreduce, bcast, wait, waitall, barrier
Count/Time	total calls / application time	send, isend, recv, irecv, sendrecv, allreduce, bcast, wait, waitall, barrier
AvgByte/Time	avg bytes per call / application time	send, isend, recv, irecv, sendrecv, allreduce, bcast
Simplified/Reduced mpiP Metrics		

all_send = send + isend
all_recv = recv + irecv
all_multi = bcast + sendrecv + allreduce
all_wait = wait + waitall + barrier		

for our applications. MpiP collects statistical information about MPI functions (per call site) and produces a report at the end of the application's execution. In this work we aggregate the data for the same MPI routine across all call sites and then compute the metrics as shown in Table I. The functions listed there are the only MPI functions with significant usage across our suite of applications.

Since different applications use different MPI functions, many of the collected metrics in Table I are zero. Further, some proxy/parent applications do not use the same MPI routines, and even use different types of communication primitives (collectives versus peer-to-peer). To correct for this disparity, we reduce and simplify the mpiP data by combining data from categorically similar MPI functions into a generic category, shown at the bottom of the table. Our four categories are: send, receive, multi-way communication, and wait. These categories reduce the mpiP metrics into more consistent category metrics across all of the applications.

For each metric in each domain, then, we collect it for each process (MPI rank), for each run of each application. Although we run each application, both parent and proxy, over a variety of configurations (e.g., number of nodes; number of cores/node), we choose one configuration and size for data analysis since we are trying to understand similarity under equivalent conditions for both proxy and parent application. For each configuration and for each application, we collect data for five distinct runs. Each run's data is kept independent of other runs.

This data then has, for each domain, $|M| * N_p$ data values, where N_p is the number of processes for that run. In an effort to reduce the total data size, we compute the mean, variance, and standard deviation across the metric set for each of the N_p processes and found the variance to be very small (close to zero), excluding the rank 0 process. Rank 0 variance is large relative to the other processes because it is typically responsible for initiating communication and gathering results amongst processes for the final computation. Therefore, we simplify the data analysis and reduce the data by only using that from process 0 and randomly choosing 7 other processes, thus fixing $N_p = 8$ for any size of runs. Future work can explore the benefit of other mechanisms of either sampling or averaging per-process data to make a uniform data set.

Each domain (node and communication) is characterized by a k-vector of metric data, with each run being a data sample. Clustering algorithms are sensitive to large dimensionality data; principal component analysis (PCA) reduces the dimensionality of the data input. We use PCA per domain to extract out the top p principal components, and use their formulae to reduce each data sample (run) to p values, which we represent as a p-length ($p < k$) vector characterizing each run. Note that we use two p-vectors, one each for the node and communication domains (see Section IV for an explanation). Based on initial experimentation, in this paper we chose to use between 5 and 6 PC values (depending on platform) for the node domain, and only 2 PCs for the communication domain (see Section IV for an explanation). The number of PCs chosen for the domains does not need to be equal, but even with data from all four domains, p should probably be kept at less than 10 to avoid high-dimension clustering issues.

We use hierarchical clustering [7] to understand similarity between proxy/parent pairs. We use the elbow method [8] for hierarchical clustering to determine the optimal number of clusters for the hierarchical clustering algorithm. Our hypotheses expect that proxy apps cluster with the real apps that they represent, and that the real apps do not cluster together.

III. Experimental Platform

For this work, we use two Intel-based hardware platforms: Haswell (Xeon E5-2698 v3) and Broadwell (Xeon E5-2695 v4). While we realize these are very similar platforms, they are very common and heavily used in the DOE complex. We desire to extend this methodology to IBM POWER, GPU, and ARM platforms. However, doing so is non-trivial in terms of mapping metrics to events on these platforms and integrating this into our LDMS infrastructure. To include GPUs requires a new LDMS sampler which takes significant effort to develop and to officially integrate into an LDMS distribution. Extension to accelerators and additional platforms will be done in future work.

We perform data collection and use our clustering model to understand similarity of the proxy/parent pairs individually on both Intel architectures, and we also combine the data from both architectures to understand how the behavior and similarity of the proxy/parent pairs changes (or remains the same) with the different architectures. Table II presents the architecture details of these two platforms. Although the Haswell platform is very common, it does have known issues with its performance monitoring unit (PMU) [9], [10], [11]. We are aware of these problems and address this when reporting results in Section IV. In Broadwell, these PMU issues have been addressed and corrected.

We use the Intel 18 compiler for all proxy/parent pairs; compiler flags are kept as consistent as possible across each proxy/parent pair and we replicate the compiler flags that are present in the distribution build files as close as possible.

TABLE II: Hardware Characteristics of Haswell and Broadwell Platforms

Component	Haswell	Broadwell
L1 data cache (per core)	32 KB, 8 way, 64 sets, 64 B line size	same
L1 instr. cache (per core)	32 KB, 8 way, 64 sets, 64 B line size	same same
L2 cache (per core)	256 kB, 8 way, 512 sets 64 B line size	same
L3 cache (shared)	45 MB, 12–16 way, 64 B line size	same except 16–20 way
Memory (per node)	128 GB DDR4-2133 MHz	same except DDR4-2400MHz
Cores/threads	16/32	18/36
Sockets/node	2	2
Total nodes	32	1122
Interconnect	Mellanox FDR Infiniband	Intel OPA and Mellanox ConnectX4
Max Memory BW	68 GB/sec	76.8 GB/sec

As mentioned previously, we execute each proxy and application in several different scaling configurations, but always pin only one process per core. We report data only for a single configuration per application, each of which uses 128 MPI ranks distributed across 8 nodes, using 16 cores per node.

A. Data Collection Tools

We use LDMS (Light-weight Distributed Metric System) [12] to obtain the node domain data from which the metrics are derived. LDMS is a low-overhead, whole-system tool that enables scalable monitoring of large-scale computer systems and applications. It is comprised of a monitoring core and a collection of plugin samplers, each of which is designed to measure a specific component or behavior of the system or application. LDMS takes advantage of RDMA (remote direct memory access), a capability on many network interfaces for directly accessing a designated portion of memory and delivering its contents across the network, without the sending node being interrupted at the processor or O/S level. This is an ideal capability for HPC monitoring purposes since the application can continue execution while the locally created monitoring data is delivered off-node using RDMA.

An LDMS sampler is responsible for collecting a particular metric. At a configured sampling rate the LDMS daemon notifies the sampler to update its metric with the most recent measurement sample. LDMS samplers exist that measure a variety of domains, including:

1) Network-related information: congestion, delivered bandwidth (total), operating system traffic bandwidth, average packet size, and link status
2) Shared file system information (e.g., Lustre): open, closes, reads, writes
3) Memory related information: current free, active
4) CPU information: utilization (user, sys, idle, wait)
5) MPI information: all mpiP metrics (Section II)
6) PAPI events: hardware event counters that the PAPI (Performance Application Programming Interface) [13] interface can access.

In this work, we use the PAPI sampler only. Although the sampler to collect MPI information exists, we were unable to use it on our chosen platforms; therefore, to collect MPI communication information, we use the mpiP library directly.

B. Proxy and Parent Applications

This work is done as part of the DOE Exascale Computing Project (ECP) [14]. Therefore, we use applications that are being used in ECP Application Development [15] projects and use proxy applications that are in the current ECP Proxy App Suite 1.0 [16]. For this work, we chose the following four ECP proxy/parent application pairs:

- SW4lite and SW4 (seismic modeling)
- Nekbone and Nek5000 (thermal transport)
- SWFFT and HACC (cosmology/FFT)
- ExaMiniMD and LAMMPS (molecular dynamics)

We also use the Graph500 benchmark in our experimentation (on one platform only) as a means to help validate the methodology. Graph500 [17] is a data intensive benchmark that represents the analytics problem space. It first constructs an undirected graph, then performs either breadth-first search (BFS) or shortest path algorithms on the graph and times these kernels. We use BFS with R-MAT scale 24 input for this work. We expect the measured characteristics of Graph500 to be very different than the other proxy/parent pairs, since it is an analytics benchmark rather than a scientific computation-based application. Therefore, we also expect Graph500 *not* to cluster with other proxies or parents and to exhibit more dissimilarity with respect to the other proxy/parent clusters.

SW4 [18] is a seismic modeling code that supports a fully 3-D heterogeneous material model that can be specified in several formats. It uses a curvilinear mesh near the free surface to honor the free surface boundary condition on a realistic topography. SW4 solves the seismic wave equations in Cartesian coordinates and is, therefore, appropriate for local and regional simulations, where the curvature of the earth can be neglected. Locations can be specified directly in Cartesian coordinates, or through geographic (latitude, longitude) coordinates. SW4lite [19] is a bare bones version of the SW4 that is intended for testing performance optimization of key numerical kernels, particularly with respect to memory layout and threading. However, it is intended to be representative of the computation, communication, and memory behavior of SW4.

Nek5000 [20], [21] is a spectral element code designed for large eddy simulation (LES) and direct numerical simulation (DNS) of turbulence in complex domains. It simulates thermal transport on a full range of scales set by the geometry encountered within a reactor. Nek5000 has a broad range of applications including vascular flow, ocean modeling, combustion, heat transfer enhancement, stability analysis and MHD (magnetohydrodynamic) flows. Nekbone [22] implements the computationally intensive linear solvers that account for a large percentage of the Nek5000 run time, as well as the communication costs required for nearest-neighbor data exchanges and vector reductions. The Nekbone kernel is embedded in a

TABLE III: Proxy/Parent version information

Proxy	Version	Parent	Version
SW4lite	2.0	SW4	2.0
Nekbone	3.1	Nek5000	17
SWFFT	1.0	HACC	1.0
ExaMiniMD	1.0	LAMMPS	17 Aug 2017

TABLE IV: Proxy/Parent Problems/Input Sizes

Proxy / Parent	Problem/Input size
SW4lite / SW4	LOH.1-h50.in, LOH.1-h50, time=9
Nekbone /	Dim=3; polynomial order=8; spectral multigrid=off max local elements per MPI rank=300
Nek5000	eddy_uv, with Dim=3; polynomial order=8 max local elements per MPI rank=300
SWFFT /	n_repetitions=100; ngx=1024
HACC	steps=100; ngx=1024

ExaMiniMD
/ LAMMPS
	units=lj; nx, ny, nz=100; Timestep=0.005;
	Run=18000 (single- and multinode)

conjugate gradient iteration to solve the 3D Poisson equation. Preconditioning is either a simple diagonal scaling (simpler than Nek5000) or a spectral element multigrid on a block or linear geometry which is more similar to the multigrid structure found in Nek5000.

The Hardware Accelerated Cosmology Code (HACC) framework [23] uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. It simulates the evolution of the Universe from its early times to today to advance our understanding of dark energy and dark matter, the two components that make up 95% of our Universe. HACC implements three phases in it computation: short force evaluation that includes computation and a tree-walk phase, and the long range computations that implements a spectral Poisson solver with an underlying 3D FFT (domain decomposes to 2D). SWFFT [24] is the 3D FFT that is implemented in HACC. Since this FFT accounts for a large portion of the HACC execution time, SWFFT serves as a proxy for HACC. SWFFT replicates the transform and is representative of the computation and communication involved.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [25] is a classical molecular dynamics code that implements potentials for solid-state materials (metals, semiconductors), soft matter (biomolecules, polymers), and coarse-grained or mesoscopic systems. It can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, meso, or continuum scale. Like LAMMPS, ExaMiniMD [26], which is a proxy for LAMMPS, uses spatial domain decomposition. But compared to LAMMPS, ExaMiniMD's feature set is extremely limited, and only three types of interactions (Lennard-Jones/ EAM/SNAP) are available. The SNAP interaction is a much more complicated and computationally expensive potential that attempts to approach quantum chemistry accuracy when modeling metals and other materials. ExaMiniMD and LAMMPS both use neighbor lists for the force calculation. ExaMiniMD is intended to represent both the computation (including memory behavior) and communication that is implemented in LAMMPS.

The problems and input sizes we use for data collection are shown in Table IV and proxy/parent application versions that we use in this work are in Table III. We define problems and input sizes based either on conversations with developers or from development team performance reports. In all cases, we attempt to define problems that are pertinent in the exascale timeframe and we map application problems to their respective proxies to be as consistent as possible. While we realize that the characterization and subsequent clustering results may be sensitive to input, a study of this sensitivity is beyond the scope of this paper and will be done in future work.

C. Statistical Analyses

For our statistical analyses, we use the principal component analysis and clustering algorithms provided by the R Statistical Computing Tool [27]. We use hierarchical clustering, which is an unsupervised machine-learning technique, and we use the elbow method to select the optimal number of clusters to use as a parameter to the clustering algorithm. The hierarchical clustering algorithm is agglomerative and used the Ward [7] cluster criterion.

IV. Results

In this section, we present the results of our analysis that show the (dis)similarity of the proxy/parent pairs with respect to computation, memory behavior, and communication on two distinct architectures and across the two architectures, to understand if the architecture impacts the measured computational behavior. A by-product of our clustering methodology is characterization-related data. We show some of this data to further discuss the results of the clustering analysis.

We use the R Statistical Computing Tool [27] to implement our unsupervised machine-learning-based clustering algorithm. Specifically, we use the hierarchical clustering method in R to compute nearness relationships of the application runs, and then use the *Elbow* method to group them into *K* clusters, selecting the best *K* value. The resulting clusters contain application runs with similar behavior based on the input metrics (computation, memory behavior; and communication). Figures 1 and 5 show the similarity in computation and memory behavior output by the clustering model for the Broadwell and Haswell platforms, respectively. In the dendrograms, the y-axis is the connection height, which is a measure of similarity—the lower the connection height, the higher the similarity of the runs below it. On both architectures, clustering and then applying the elbow method results in selecting five clusters to be best number of clusters.

Figure 1 shows the clustering on the Broadwell system. At the lowest level, the five runs of each application always cluster together, and then in the five clusters all of the proxies cluster with their parents except Nekbone and Nek5000, which cluster together at the first join above the five clusters. There

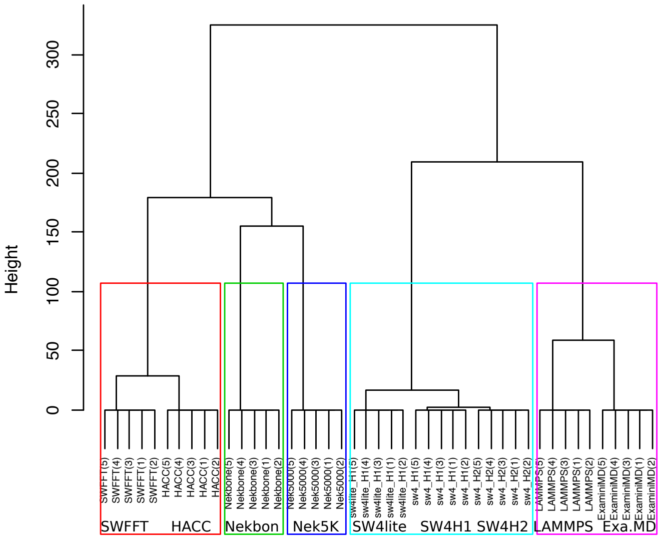

Fig. 1: Computation and Memory Similarity, Broadwell

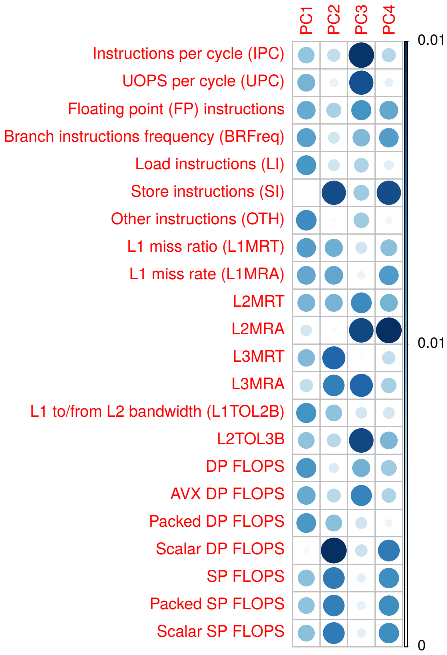

Fig. 2: Principal Component Heatmap, Broadwell

is a large similarity (height) step between the level of five clusters and then further clustering, indicating strong clustering at the five-cluster level. Above this level the clustering rapidly (in terms of height nearness of cluster joins) produces two primary clusters, one containing SWFFT, HACC, Nekbone, and Nek5000; the other comprising SW4lite, SW4, LAMMPS, and ExaMiniMD. Recall that we execute five runs for each proxy/parent application; each of these five runs for each proxy/parent is shown in Figure 1 and they cluster together, which indicates small variance between executions. SW4_H1 and SW4_H2 represent the two inputs that we use for this application (see Table IV). Similarity ranking with respect to computation of the proxy/parent pairs from most to least similar on Broadwell is as follows:

1) SW4/SW4lite
2) HACC/SWFFT
3) LAMMPS/ExaMiniMD
4) Nek5000/Nekbone

Clustering was performed over the four most important principal components produced by PCA. PC1 accounts for 50% of the data set variance, and PC2-4 account for 18, 15, and 10%, respectively (total 93%). Figure 2 shows a heatmap that represents the importance of the metrics to the principal components used in the clustering algorithm. The larger the circle and the darker, the more that metric contributes to the overall variability accounted for by the principal component. PC1 is comprised of many metrics of "medium" importance, the most important being instructions in the *Load* and *Other* categories, *L1 to/from L2 bandwidth*, *DP* (double precision) and *Packed DP FLOPS* (floating-point operations). For PC2, the most relevant metrics are *store* instructions, *L3 miss rate*, and *Scalar DP FLOPS*. Equations to compute the FLOPS categories are based on formulas from the likwid performance tool [28]. PC3 and PC4 pick up IPC, store instructions, and L2 miss ratio as relevant to the applications, and PC4 adds some weight to the FP categories that PC2 includes. Overall, none of the selected metrics are significantly ignored by the PCA, indicating that they are all important.

Figures 3 and 4 present a small portion of the characterization data that is used as input to our clustering model. We present this data to demonstrate how it can be visually interpreted to understand the clusters generated by the model shown in Figure 1. Figure 3 shows the instruction mix of the proxy/parent pairs, where each instruction category is represented as a percentage of the total instructions executed. Through visual inspection of this plot, we see that HACC and SWFTT are very similar, as are SW4 and SW4lite. These are the proxy/parent pairs in Figure 1 that have the smallest clustering height and are, therefore, most similar. The instruction mix of Nek5000, Nekbone, LAMMPS, and ExaMiniMD all look fairly but less similar. We see from the clustering dendogram that Nek5000/Nekbone and LAMMPS/ExaMiniMD both have a relatively large height, so are not as similar as the two other proxy/parent pairs.

Figure 4 shows data on various FLOP categories. We can visually observe from this the similarity between HACC and SWFFT, SW4 and SW4lite, Nek5000 and Nekbone, and LAMMPS and ExaMiniMD. The point here is that for proxy/parent pairs that are highly similar according to their cluster dendogram, it is easy to visually observe that similarity in the underlying hardware metrics. For the pairs that are not as similar, it is harder to see their similarity in the underlying metrics—some of them look very similar, while others do not, and hence the need for a machine-learning mechanism to statistically extract the similarity from the data.

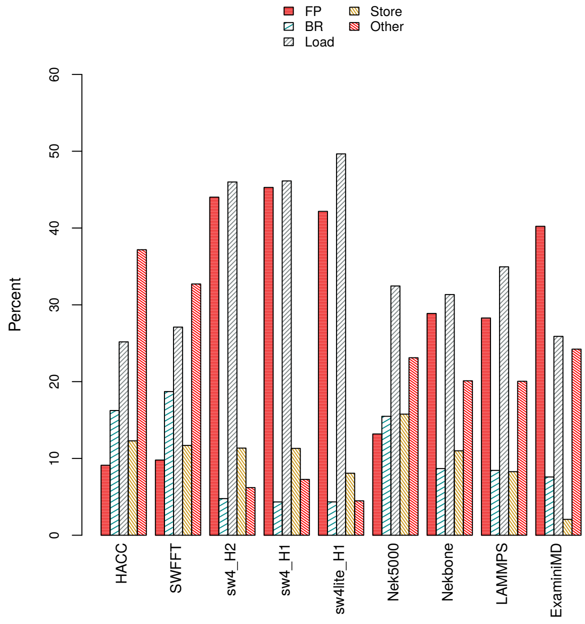

Fig. 3: Instruction Mix, Broadwell

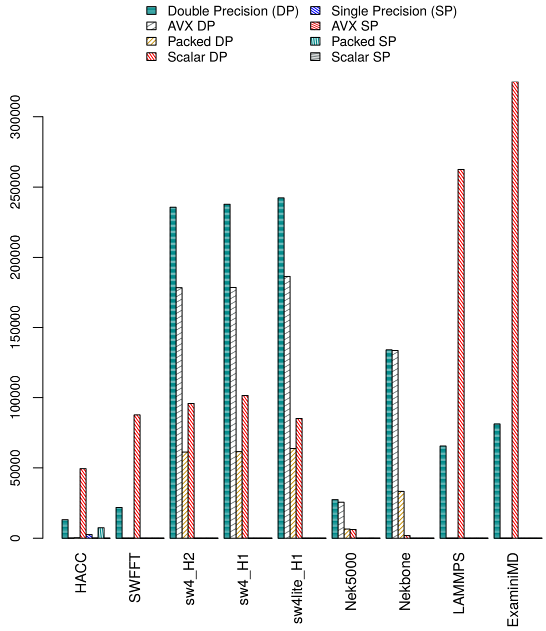

Fig. 4: FP Operations/sec (FLOPS), Broadwell

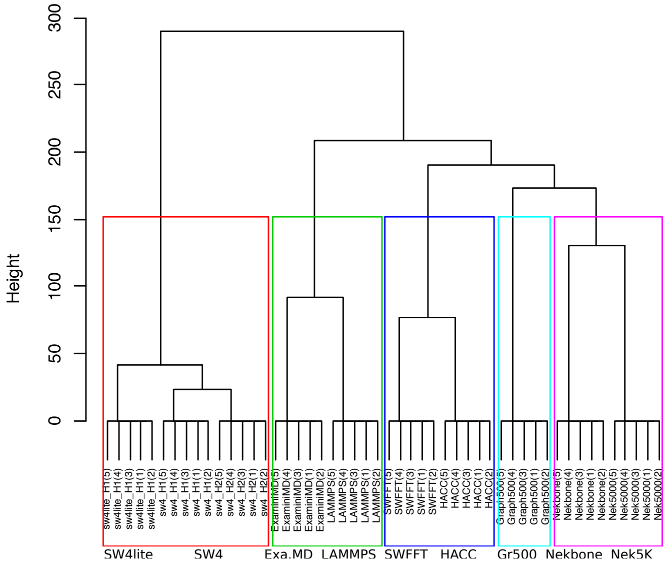

Fig. 5: Computation and Memory Similarity, Haswell

Figure 5 shows the clustering similarity in computation of the four proxy/parent pairs, plus Graph500, on the Haswell system. Graph500 was used as an outlier application that should not cluster tightly with any of the other applications, real or proxy. Five clusters are selected as optimal, with all of the runs of the proxies and parents each forming their own cluster together, and Graph500 forming its own cluster. This is a close-to-ideal result, although the height at which most proxies and parents cluster together is higher than those on the Broadwell system, except for SW4lite and SW4. Above the five-cluster level, everything but SW4 and SW4lite rapidly conglomerate into a single cluster, and the height at which this merges with the SW4/SW4lite cluster shows that SW4/SW4lite is distinct. This is slightly different than on Broadwell, where ExaMiniMD and LAMMPs clustered with SW4lite and SW4, and Nekbone and Nek5000 clustered with SWFFT and HACC. Similarity ranking with respect to computation of the proxy/parent pairs from most to least similar on Haswell is as follows:

1) SW4/SW4lite
2) HACC/SWFFT
3) LAMMPS/ExaMiniMD
4) Nek5000/Nekbone

This is the same ordering as on Broadwell, thus showing perhaps a true ordering in terms of the similarity between the proxies and parents. The largest change in similarity among proxy/parent pairs is in HACC and SWFFT; on Broadwell they are very similar, but on Haswell not as much. Although Graph500 does not remain separate until the final level of clustering, it still clusters by itself at the optimal cluster level (5), and this clustering is only over computation and memory metrics, so it may not be unreasonable for Graph500 to be mixed in among the various proxy/parent pairs.

In the principal component analysis for the Haswell clus-

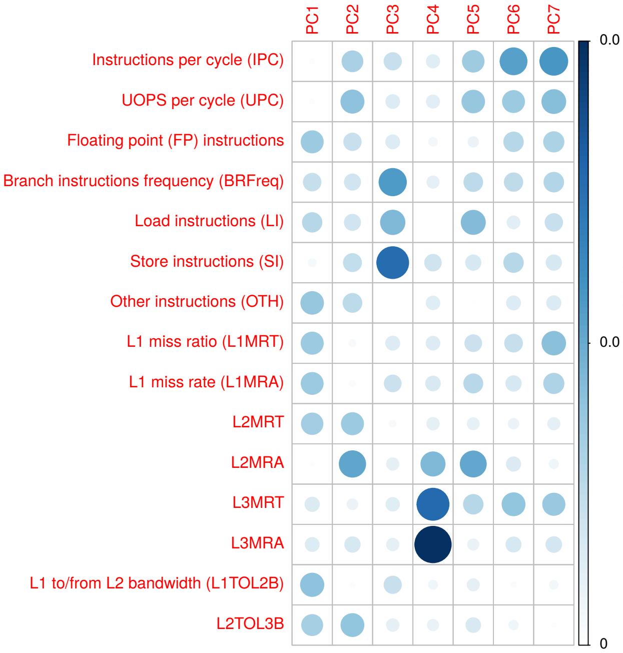

Fig. 6: Principal Component Heatmap, Haswell

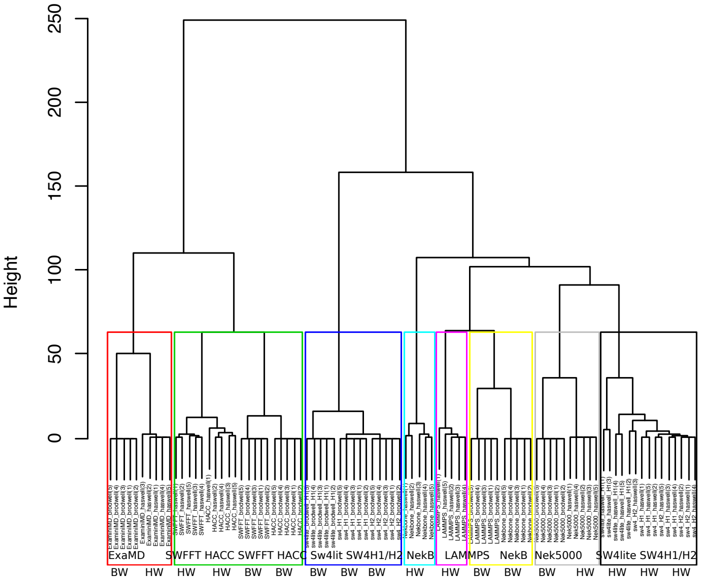

Fig. 7: Computation and Memory Similarity of Proxy/Parent Pairs across Broadwell and Haswell Platforms

tering, seven PCs are selected (in order to achieve above 90% variance explanation). PC1 accounts for 42% of the data set variance, and PC2-7 account for 18, 13, and on down to 3%, respectively. In the PCA (principal component analysis) heatmap in Figure 6, we see that for PC1 the important metrics are *FP* and *other* instructions, *L1 miss rate and miss ratio*, *L2 miss rate*, and *L1 to/from L2 bandwidth*. PC2 important metrics are *L2 miss ratio* and *UPC*. PC3 and PC4 have very strong dependencies, on load/store/branch and L3 cache, respectively, but it should be remembered that these PCs are starting to get much weaker in their contribution to data explanation.

As noted above, the proxy/parent clustering on Haswell, though consistent, is weaker than Broadwell (connected higher up on the height axis). While not depicted here by charts or graphs, due to space, when looking at the characterization data generated on Haswell we do not observe any large differences in metrics between the proxy/parent pairs. However, we do see small differences in many metrics, which accumulate to explain the higher dissimilarity in the clustering. Still, the proxy/parent clustering is a good result.

We also combined all of the data from both the Broadwell and Haswell platforms, and clustered all of the runs together, labeling them with the platform they were run on. This clustering is shown in Figure 7. The elbow method selected eight as the best number of clusters to consider. Some interesting observations arise from this clustering. (1) All of SWFFT and HACC closely cluster together, and also cluster together by platform before clustering with each other across platforms. (2) SW4 and SW4lite strongly cluster with each other by platform, but then the separate platforms do not cluster together until only two clusters remain. (3) Nek5000 clusters with itself across platforms first, but eventually joins Nekbone, which first clustered with LAMMPS on Broadwell, then Haswell. (4) The most architecturally neutral application is Nek5000, followed by ExaMiniMD. HACC. SWFFT, and LAMMPS have roughly equal performance sensitivity to architecture (and larger than Nek5000 and ExaMiniMD), while SW4 and SWlite are the most sensitive to architecture. Finally, with LAMMPS clustering closer to Nekbone, ExaMiniMD clusters by itself across platforms, and does not join LAMMPS in a cluster until everything is in one cluster. While this result shows some expected groupings, it is different from the per-platform clustering analysis and shows that it is safer to perform proxy/parent comparisons per platform rather than across platform, and that different computations can vary widely in their behavior differences across different platforms, at least with respect to the chosen metrics.

Figure 8 shows the clustering that results from the communication (MPI) data, for the Broadwell platform only. Note first the scale of the height axis, and how far away the final two clusterings are from the lower clusterings. This indicates very little similarity at these levels (hierarchical clustering will always connect everything, eventually). Even the connecting of the SW4* cluster and the Nek*/LAMMPS/ExaMiniMD cluster is, relatively, quite high on the axis. In looking at application and proxy implementations, some do use the same MPI communication primitives and style, and these cluster well together. Others, because they were written separately, may try to implement a similar *model* of communication, but they do so with different MPI primitives, and end up having very different aggregate statistics over the mpiP profile data. We view this result as indicating that our basic attempt at abstracting the mpiP data away from specific MPI routines

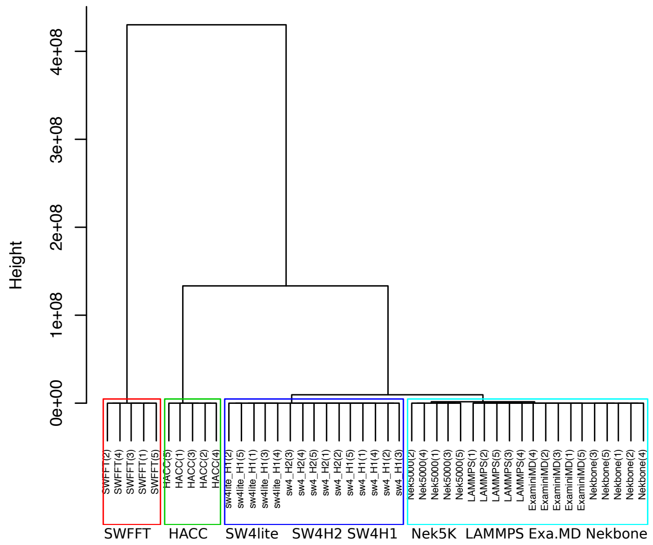

Fig. 8: Communication Similarity, Broadwell

was not enough to actually capture an abstract-enough model of communication where different applications might end up with similar data. We think an approach that attempts to characterize communication patterns is needed in this domain.

V. Related Work

Much related work has been published related to proxy application characterization and comparison of proxies to their parent applications [1], [2], [3], [29], [30], [31], [32], and using proxy applications to project system performance of real applications [4], [33], [34]. We present the most related work below.

Tramm et al. [35] perform a characterization of a Monte Carlo particle transport simulation code, OpenMC, and its proxy, XSBench. Multi-core scaling efficiency, floating point calculation rates, and hardware performance counter profiles are correlated to an efficiecy loss metric that is related to performance as the number of threads are increased. This work is a study on a single proxy/parent application pair and presents a methodology that is manual rather than machine-learning based.

Sreepathi et al. [31] demonstrate the use of Oxbow and PADS, a toolkit and data store infrastructure for application behavior analysis, on various Department of Energy (DOE) co-design centers' mini-applications, as well as High Performance Linpack and High Performance Conjugate Gradient benchmarks. This form of analysis, similar to what we present in this work, aims to provide insight into performance and representativeness of proxy/parent pairs. Although this work is similar to ours, they focus more on characterization of performance, particularly for proxies. Further, they present similarity of proxies, rather than similarity of proxy/parent pairs.

Kim et al. [3] extend their KGen Fortran Kernel Generator tool with the ability to gather descriptive statistics from both the original application and the KGen kernel extracted from the instrumented application. These statistics are used to quantify the performance difference between the kernel and original application, and provide feedback to improve its representativeness of the real application in a way that does not increase the workload of the code. Measurements performed range from the kernel block's elapsed execution time, to PAPI hardware counters depending on the type of analysis a user intends to perform. Their methodology is more manual and based on a simple comparison (difference) of metrics and they only focus on applications and kernels that can be extracted from them using KGen.

Barrett et al. [2], [1] are active in developing proxy applications and in assessing their characteristics in relation to real applications. In [1] they do an in-depth analysis of one real and proxy application pair (Charon/miniFE), while in [2] they propose a methodology whereby a set of performance domains is considered for some pair of proxy and real applications, and comparison and threshold functions are created for each performance domain, which can be very different from domain to domain, and the domains can be different from application to application. For example, for the LAMMPS/miniMD pair (molecular dynamics), their domains are: total time, force calculation time, neighbor list construction time, and interprocess communication time. While the abstract methodology framework is common, since the instantiation for each pair of applications is unique, their work is somewhat orthogonal to ours, as we are proposing a common instantiated methodology.

Islam et al. [32] created the Veritas framework to measure proxy/parent relationships using belief estimation in Dempster-Shafer theory over low level resource measurements of CPU component usage. They then compute the amount that a proxy *covers* the real application in each of the resource categories. The resource categories they used were floating point, branch execution, TLB, L1-L3 caches, memory, and prefetching. They also used PCA to achieve some dimensionality reduction in their data.

Tsuji et al. [4] address the use of proxy applications to inform the use of benchmark applications as a precursor in assessing expected system performance without involving real applications. They propose a new metric, SSSP or Simplified Sustained System Performance, which uses weighting factors derived from proxy application performance applied to performance data from simple kernel benchmarks to compute a metric based on the SSP methodology. This adapted methodology avoids the use of real applications in establishing system performance, which are known to be costly to set up and tune to a given system. They demonstrate the consistency of SSSP with SSP while only utilizing data from proxy applications and simple benchmark kernels. A similar form of performance projection is likewise presented by Sharkawi et al. [34].

VI. Conclusion and Future Work

In this paper, we present an unsupervised machine-learning-based methodology to determine similarity of proxy applications and their respective parents with respect to computation, memory, and communication behavior. Our methodology relies on the collection of hardware-level performance data, which is converted into metrics, reduced in dimensionality using principal component analysis, then used as input to a hierarchical clustering algorithm to determine similarity. We use this methodology to show that the four proxies, SWFFT, SW4lite, ExaMiniMD, Nekbone, are indeed representative of their parent applications (HACC, SW4, LAMMPS, Nek5000, respectively) across two different architectures, Intel Haswell and Broadwell. We show this for one important problem/input size that is mapped consistently across proxy and parent application. SW4 and SW4lite are consistently the most similar across both architectures, and all proxies are more representative of their parents on the Broadwell architecture.

We also looked at the proxy and parent application similarity across both architectures by analyzing and clustering the combined data from both the Haswell and Broadwell systems. Results show that the most architecturally neutral application is Nek5000, while the performance of SW4 and SW4lite is the most sensitive to architecture.

We did apply our methodology to mpiP data collected for each of the proxy/parent pairs. The clustering results revealed almost no similarity in communication behavior for any of the proxy/parent pairs. This occurs because for most of the proxies, the MPI primitives used are different than those used in the respective parent, resulting in very different aggregate statistics over the mpiP data.

In future work, we plan to generate communication pattern heatmap plots from an internal version of mpiP that collects all of the necessary data. We will develop a similarity metric to enable quantitative comparison of these communication patterns. We also plan to migrate the infrastructure to more platforms, including IBM, additional Intel, AMD, and ARM systems. Collecting performance data at the function level to gain a better understanding of overall proxy/parent similarity will also be done. Finally, we plan to develop a GPU sampler for the LDMS infrastructure to enable performance comparison and similarity analysis for GPU-enabled applications.

VII. Acknowledgement

This research was supported by the Exascale Computing Project (ECP), Project Number 17-SC-20-SC, a collaborative effort of two DOE organizations, the 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, to support the nation?s exascale computing imperative.

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

References

[1] P. T. Lin, M. A. Heroux, R. F. Barrett, and A. B. Williams, "Assessing a mini-application as a performance proxy for a finite element method engineering application," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5374–5389, 2015, cpe.3587. [Online]. Available: http://dx.doi.org/10.1002/cpe.3587

[2] R. Barrett, P. Crozier, D. Doerfler, M. Heroux, P. Lin, H. Thornquist, T. Trucano, and C. Vaughan, "Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications," *Journal of Parallel and Distributed Computing*, vol. 75, no. Supplement C, pp. 107 – 122, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001695

[3] Y. Kim, J. M. Dennis, and C. Kerr, "Assessing representativeness of kernels using descriptive statistics," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 818–825.

[4] M. Tsuji, W. T. C. Kramer, and M. Sato, "A performance projection of mini-applications onto benchmarks toward the performance projection of real-applications," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 826–833.

[5] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, ser. WCAE '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1275571.1275600

[6] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPoPP '01. New York, NY, USA: ACM, 2001, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/379539.379590

[7] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *CoRR*, vol. abs/1109.2378, 2011.

[8] A. E. Zambelli, "A data-driven approach to estimating the number of clusters in hierarchical clustering," *F1000Research*, vol. 5, pp. ISCB Comm J–2809, 2016. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC5373427/

[9] https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/700255, "Flops measurement on haswell-ep and broadwell-ep." [Online]. Available: https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/700255

[10] https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/277877, "Interpreting the avx counter results." [Online]. Available: https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/277877

[11] https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/520331, "Understanding l2 miss performance counters." [Online]. Available: https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/520331

[12] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 154–165.

[13] http://icl.cs.utk.edu/papi/, "Papi: Performance application programming interface." [Online]. Available: http://icl.cs.utk.edu/papi/

[14] https://www.exascaleproject.org, "Exascale computing project." [Online]. Available: https://www.exascaleproject.org

[15] https://www.exascaleproject.org/focus_area/application-development, "Ecp application development." [Online]. Available: https://www.exascaleproject.org/focus_area/application-development

[16] https://proxyapps.exascaleproject.org, "Exascale proxy applications." [Online]. Available: https://proxyapps.exascaleproject.org

[17] https://graph500.org, "Graph500." [Online]. Available: https://graph500.org

[18] B. Sjrogreen and N. A. Petersson, "A fourth order accurate finite difference scheme for the elastic wave equation in second order formulation," *Journal Of Scientific Computing*, vol. 52, no. 1, 2011.

[19] https://github.com/geodynamics/sw4lite, “Sw4lite.” [Online]. Available: https://github.com/geodynamics/sw4lite

[20] https://nek5000.mcs.anl.gov, “Nek5000.” [Online]. Available: https://nek5000.mcs.anl.gov

[21] H. M. Tufo and P. F. Fischer, “Terascale spectral element algorithms and implementations,” in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC ’99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331599

[22] https://asc.llnl.gov/CORAL benchmarks/Summaries/Nekbone_Summary_v2.3.4.1.pdf, “Nekbone.” [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/Summaries/Nekbone_Summary_v2.3.4.1.pdf

[23] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley, D. Daniel, P. Fasel, and Z. Lukić, “Hacc: Extreme scaling and performance across diverse architectures,” *Commun. ACM*, vol. 60, no. 1, pp. 97–104, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3015569

[24] https://xgitlab.cels.anl.gov/hacc/SWFFT, “Swfft (hacc).” [Online]. Available: https://xgitlab.cels.anl.gov/hacc/SWFFT

[25] S. Plimpton, “Fast parallel algorithms for short-range molecular dynamics,” *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, Mar. 1995. [Online]. Available: http://dx.doi.org/10.1006/jcph.1995.1039

[26] A. P. Thompson and C. R. Trott, “A brief description of the kokkos implementation of the snap potential in examinimd.” 11 2017.

[27] https://www.r-project.org, “The r project for statistical computing.” [Online]. Available: https://www.r-project.org

[28] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.38

[29] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, “Apples and oranges: a comparison of rdf benchmarks and real rdf datasets,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 145–156.

[30] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, “Jsmeter: Comparing the behavior of javascript benchmarks with real web applications.” *WebApps*, vol. 10, pp. 3–3, 2010.

[31] S. Sreepathi, M. L. Grodowitz, R. Lim, P. Taffet, P. C. Roth, J. Meredith, S. Lee, D. Li, and J. Vetter, “Application characterization using oxbow toolkit and pads infrastructure,” in *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*. IEEE Press, 2014, pp. 55–63.

[32] T. Z. Islam, J. J. Thiagarajan, A. Bhatele, M. Schulz, and T. Gamblin, “A machine learning framework for performance coverage analysis of proxy applications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 46:1–46:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014966

[33] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth, “Navigating an evolutionary fast path to exascale,” in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 355–365.

[34] S. Sharkawi, D. Desota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, “Performance projection of hpc applications using spec cfp2006 benchmarks,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[35] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, “XSBench-The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis,” *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.