Quantifying Architectural Requirements of Contemporary Extreme-Scale Scientific Applications

Jeffrey S. Vetter[1,2(✉)], Seyong Lee[1], Dong Li[1], Gabriel Marin[3], Collin McCurdy[1], Jeremy Meredith[1], Philip C. Roth[1], and Kyle Spafford[1]

[1] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
vetter@computer.org
[2] Georgia Institute of Technology, Atlanta, GA 30332, USA
[3] University of Tennessee–Knoxville, Knoxville, TN 37996, USA

Abstract. As detailed in recent reports, HPC architectures will continue to change over the next decade in an effort to improve energy efficiency, reliability, and performance. At this time of significant disruption, it is critically important to understand specific application requirements, so that these architectural changes can include features that satisfy the requirements of contemporary extreme-scale scientific applications. To address this need, we have developed a methodology supported by a toolkit that allows us to investigate detailed computation, memory, and communication behaviors of applications at varying levels of resolution. Using this methodology, we performed a broad-based, detailed characterization of 12 contemporary scalable scientific applications and benchmarks. Our analysis reveals numerous behaviors that sometimes contradict conventional wisdom about scientific applications. For example, the results reveal that only one of our applications executes more floating-point instructions than other types of instructions. In another example, we found that communication topologies are very regular, even for applications that, at first glance, should be highly irregular. These observations emphasize the necessity of measurement-driven analysis of real applications, and help prioritize features that should be included in future architectures.

1 Introduction

As detailed by several reports [1,2], HPC architectures will continue to change over the next decade in response to efforts to improve energy efficiency, reliability, and performance. At this time of significant disruption, it is critically

Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

S. Jarvis et al. (Eds.): PMBS 2013 Workshops, LNCS 8551, pp. 3–24, 2014.
DOI: 10.1007/978-3-319-10214-6_1

important to understand the requirements of contemporary extreme-scale scientific applications, so that these architectural changes can include features that satisfy these requirements. The proper mapping of these features to these future architectures will ultimately result in the best return on investment in these future systems. For example, in just the past few years, we have seen various new capabilities in contemporary processors and networks (e.g., integrated GPU and CPU, integrated random number generator, transactional memory, fine-grained power management, MPI collective offload, etc.) that have a significant impact on application performance.

In contrast to workload characterization performed in the last twenty years, today's characterizations must be more broad and yet more detailed in order to inform the design of these new architectures. Identifying architecture independent characteristics of applications is challenging, and generating these characteristics using a uniform, crosscutting methodology is vital to prioritization and efficiency.

In this paper, we present a methodology for examining important computation and communication behaviors for a representative set of real-world extreme-scale applications. Our initial toolkit, presented here, allows us to consistently and uniformly measure various behaviors in these applications: instruction mixes, memory access patterns and capacity, point-to-point messaging frequency and payload size, collective frequency, operation, and payload size, and communication topology.

For our applications, we have selected a substantial number of important U.S. Department of Energy (DOE) applications. Also, we have identified several new proxy applications, which are being developed by DOE Co-design centers, and we investigate these "proxy apps" with the same tools in order to identify differences between proxy apps and the applications they are meant to represent. The applications, proxy apps, and benchmarks we studied are summarized in Tab. 2.

With these measurements, we identify a number of observations (in Sect. 5), which we believe can inform decisions about future architectures.

1.1 Key Metrics and Methods

We identified several key metrics (Tab. 1) and methods for measuring those metrics for our study. We focus on processor instructions, memory behavior, and communication using the Message Passing Interface [3] (MPI), which is used by most of our applications. Later sections describe each metric and methodology in detail.

1.2 Related Work

A considerable amount of previous work [4–9] has characterized scientific applications using a variety of metrics and methodologies. This previous work provided detailed information about scientific applications, which typically focused

Table 1. Key metrics and methods

Category	Metrics
Computation	
Instruction mix	Instruction categories and counts
SIMD mix, width	SIMD counts and vector widths
Memory bandwidth	Achieved R/W memory bandwidth per socket
Reuse Distance	Temporal data locality
Communication	
Point-to-Point	Frequency, volume, type, topology
Collective	Frequency, volume, type, operator

on a specific metric, like communication. These existing studies are not sufficient going forward, however. With continued development of applications and algorithms, we need to continuously revisit these questions. And, as mentioned earlier, the new architectural decisions facing HPC are forcing us to answer new questions.

On another front, with the growing importance of benchmarks and proxy applications, we want to clearly distinguish these smaller programs from the more complex existing (or future) applications they are meant to represent. In particular, we want to identify the metrics that they represent well, and, perhaps more importantly, the metrics that they do *not* represent.

1.3 Assumptions and Caveats

Any characterization study comes with a set of assumptions and caveats. In this section, we outline some of these topics. First, although it would be preferred to have an idealized architecture for performing these measurements, running these full applications on a simulator or emulator at scale would be impractical. Instead, we identified a single architecture and software system on which we preformed all of our experiments to ensure consistency and understanding of application measurements. In particular, because processor instructions and semantics differ across platforms, it was very important to use the same processor and compilation system to conduct this analysis. We describe this platform in Sect. 1.4. Second, since our focus is scalable scientific applications, we included measurements of communication behavior and the runtime software, namely MPI, in our analysis. Third, these applications, particularly the proxy apps, are changing rapidly. Our toolkit is built to regenerate this analysis frequently, expecting such rapid changes. In this work, we present a snapshot of the behaviors for a specific version of each application. Finally, many of these applications and benchmarks can be applied to a wide range of problems in terms of both algorithms and data sizes. In our experiments, we selected problem sizes that were representative of typical application experiments; however, for a small set of our measurements, such as reuse distance, we had to reduce the problem

size in order to complete the measurements in a practical amount of time. We identify those constraints in our discussion of the results.

1.4 Experimental Platform

For our application characterizations, we used the Georgia Institute of Technology's Keeneland Initial Delivery System [10] (KID). KID uses the scalable node architecture of the HP Proliant SL-390G7. Each node has two Intel Westmere host CPUs, 24GB of main memory, a Mellanox Quad Data Rate (QDR) InfiniBand HCA, and a local disk. The system has 120 nodes with 240 CPUs.

At the time of our experiments, the KID software environment was based on the CentOS 5.5 Linux distribution. In addition to the CentOS distribution's default software development environment based around the GNU Compiler Collection (GCC), the Intel 2011 and PGI 12 compilers are available to users of the system. To support efficient use of the system's CPUs , math libraries such as the Intel Math Kernel Library (MKL) are also available.

2 Instruction Mix

A program's *instruction mix* captures the number and type of the instructions the program executes when applied to a given input. To collect application instruction mixes from fully optimized x86-64 executable files (also called binaries), our toolkit uses a performance modeling framework called MIAMI. MIAMI uses a PIN [24]-based front-end to recover the application control flow graph (CFG) and record the execution frequency of select control flow graph edges during program execution. After the program finishes, MIAMI recovers the execution frequency of all CFG basic blocks, builds the loop nesting structure, and uses XED [25] to decode the instructions of each executed basic block. We decode the x86 instructions into generic operations that resemble RISC instructions. Thus, arithmetic instructions with memory operands are decoded into multiple micro-ops: one for the actual arithmetic operation, plus one additional micro-op for each memory read and write operation performed by the x86 instruction. Each micro-op has associated attributes such as bit width, data type (integer or floating-point), unit type (scalar or vector), and vector length where applicable.

In our methodology, we aggregate these micro-ops into a few coarser categories as seen in Tab. 3. Load and store operations are all classified as either *MemOps* if they operate on scalar values, or *MemSIMD* if they operate with vector data. Arithmetic operations are classified as floating-point vs. integer, and also as scalar vs. SIMD, resulting in four exclusive categories. The *Moves* category includes scalar and vector register copy operations, as well as data conversions from one data type to another or between different precisions of the same data type. All conditional and unconditional branches, as well as direct and indirect jumps, are classified as *BrOps*. Finally, the *Misc* category includes all other types of instructions, such as traps, pop count, memory fence and other synchronization operations.

Table 2. Applications and kernels

Application	Area	Description	Input Problems
		Benchmarks and Proxy Applications	
HPCC [11]	Benchmark	Collection of kernels to test system-wide and node-level performance	Two nodes (24 MPI tasks), sized to use approximately 50% of memory
AMG [12]	Multigrid Solver	Parallel solver for linear systems on unstructured grids	Built-in Laplace problem on unstructured domain with anisotropy.
Nekbone	Fluid Dynamics	Mini-application of Nek5000, customized to solve basic conjugate gradient solver	Conjugate gradient solver for linear or block geometry
MOCFE [13]	Neutron Transport	Simulates deterministic neutron transport equation parallelized across energy groups, angles, and mesh	Ten energy groups, eight angles, and weakly scaled mesh
LULESH [14]	Hydrodynamics	High deformation event modeling code via Lagragian shock hydrodynamics	Sedov blast wave problem in three spatial dimensions
		Applications	
S3D [15,16]	Combustion	Direct numerical solver for the full compressible Navier-Stokes, total energy, species, and mass continuity equations	Amplitude pressure wave with ethylene-air chemistry on weakly scaled domain
SPASM [17]	Materials	Short-range molecular dynamics	Cu tensile test with the embedded atom method
GTC [18]	Fusion	Particle-in-cell code for studying microturbulence in magnetically confined plasmas	16 toroidal planes, number of domains for plane decomposition varied, 5 particles/cell/domain
ddcMD [19]	Molecular Dynamics	Classical molecular dynamics via flexible domain decomposition strategy	Molten metal re-solidification, 256 MPI processes
LAMMPS [20]	Molecular Dynamics	Large-scale Atomic/Molecular Massively Parallel Simulator	LJ - atomic fluid with Lennard-Jones potential; EAM - Cu with EAM potential; RHODO - rhodopsin protein with long range forces
Nek5000 [21]	Fluid Dynamics	A computational fluid dynamics solver based on the spectral element method	3D MHD
POP [22]	Climate	Ocean circulation model part of the Community Climate System Model [23]	192x128x20 domain with balanced clinic distribution over 8 MPI processes

Collecting instruction mixes from application binaries has both advantages and disadvantages. Working at the binary level reveals the precise instruction stream that gets executed on the machine after all the compiler optimizations are applied. In addition, classifying the semantics of low level machine instructions is less error prone than trying to understand the resulting instruction mix of a high level source code construct. Compilers often need to generate many auxiliary machine instructions to perform a simple source code operation such as an array access. On the other hand, compiler choice and compiler optimizations may affect the reported instruction mixes. In particular, the quality of the register allocator has a direct effect on the number of memory operations in the instruction stream. Other scalar optimizations influence the number of auxiliary instructions that end up in the final application binary.

In our methodology, we strive to be consistent in how we profile applications, making sure that we used the same compiler and optimization flags in all cases. This consistency allows us to more directly compare the instruction mixes from the applications under study.

Table 3. Instruction category descriptions

Category	Description
MemOps	Scalar load and store operations
MemSIMD	SIMD vector load and store operations
Moves	Integer and floating-point register copies; data type and precision conversions
FpOps	Scalar floating-point arithmetic
FpSIMD	Vector floating-point arithmetic
IntOps	Scalar integer arithmetic
IntSIMD	Vector integer arithmetic
BrOps	Conditional and unconditional branches; direct and indirect jumps
Misc	Other miscellaneous operations, including pop count, memory fence, atomic operations, privileged operations

SIMD. Many commodity microprocessors used in today's supercomputers include support for Single Instruction Multiple Data (SIMD) instructions. When executed, an SIMD instruction performs the same operation on several data values simultaneously to produce multiple results. In contrast, a non-SIMD instruction produces at most a single value. On processors that support SIMD instructions, using such instructions is desirable because it increases the amount of data parallelism possible using a given number of instructions. From another perspective, using SIMD instructions places less demand on the memory subsystem for instruction fetches and data loads and stores compared to a sequence of non-SIMD instructions that perform the same sequence of operations. SIMD instructions were introduced for commodity microprocessors in the latter half of the 1990s [26,27] and promoted as support for accelerated graphics and gaming. However, many operations used for graphics and gaming are also useful in scientific computing, making modern SIMD instruction set extensions such as the

Streaming SIMD Extensions 4 (SSE4) [28] an attractive target for developers of scientific applications.

We use an instruction's extension as reported by XED to classify instructions as vector or scalar operations. Some modern compilers commonly generate SSE instructions even for scalar arithmetic, because the SIMD execution path is faster than the x87 pipelines on current x86 architectures. To make the instruction mix metric less dependent on the compiler, we classify SIMD instructions that operate on a single data element as scalar. Therefore, our reported SIMD counts correspond to true vector instructions that operate on multiple data, and the SIMD counts may be lower than a classification based exclusively on instruction extensions.

Table 4. Instruction mix (percentage of all instructions, NOPs excluded)

Target	MemOps%	MemSIMD%	FpOps%	FpSIMD%	IntOps%	IntSIMD%	Moves%	BrOps%	Misc%
HPCC:HPL	0.9	19.2	0.1	60.2	3.1	0.0	15.7	0.8	0.0
HPCC:MPIFFT	24.1	5.7	11.3	11.3	22.5	0.1	18.5	6.4	0.0
HPCC:MPIRandomAccess	28.2	3.3	0.0	0.0	41.1	1.8	10.9	14.6	0.1
HPCC:PTRANS	27.5	1.1	6.7	0.9	36.4	1.1	20.3	6.0	0.0
Graph 500	24.7	0.1	0.0	0.0	37.6	0.0	22.5	15.1	0.0
AMG (setup)	17.3	0.1	0.4	0.0	53.0	0.0	3.1	26.1	0.0
AMG (solve)	29.8	1.3	15.7	0.6	21.3	0.0	20.4	10.9	0.0
MOCFE	31.2	10.1	1.0	6.7	28.8	0.1	10.8	10.9	0.1
Nekbone(1024Weak)	31.3	5.1	0.3	21.2	12.1	0.1	25.3	4.7	0.0
LULESH	31.1	2.2	29.7	4.6	2.2	0.0	28.9	1.2	0.0
S3D	19.1	14.0	3.3	18.3	19.9	2.1	14.4	7.7	1.0
SPASM	31.7	0.4	21.9	0.4	13.5	0.2	24.1	7.8	0.0
GTC	32.7	0.0	7.6	0.3	38.3	0.0	4.4	16.6	0.0
ddcMD	28.6	0.2	34.9	0.3	7.1	0.0	26.7	2.3	0.0
LAMMPS_EAM	36.4	0.0	28.8	0.0	8.6	0.3	20.6	5.2	0.0
LAMMPS_LJ	33.7	0.1	22.6	0.0	10.4	0.0	27.6	5.6	0.0
LAMMPS_RHODO	35.1	0.5	18.5	1.0	14.4	0.2	22.5	7.8	0.0
Nek5000 (MHD)	29.6	2.6	2.4	9.1	23.3	0.1	25.7	7.2	0.1
POP	18.6	15.1	8.4	14.2	20.3	1.2	14.8	7.5	0.0

Results. For this study, we classify SIMD instructions into three categories: memory, integer, and floating-point as seen in Tab. 3. Note that SIMD register copy operations are not reported separately. Instead, they are aggregated together with their scalar counterparts in the *Moves* category.

Table 4 shows the instruction mix captured from benchmarks, proxy apps, and full applications. A few commonalities and distinctions are noteworthy. First, HPL is strikingly different from every other benchmark, proxy app, and full

Table 5. Distribution of data sizes for Memory/Move/Arithmetic Instructions. A/B/C/D/E format represents percentage of instructions working on 8/16/32/64/128 bit width data, respectively. (*In FpOps for HPCC:MPIFFT, E in the A/B/C/D/E format indicates 80 bits.*)

Target	**MemOps**					**Moves**					**IntOps**					**IntSIMD**					**FpOps**					**FpSIMD**				
	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
HPCC:HPL	0	0	1	3	96	0	0	2	0	97	0	0	17	83	0	0	0	100	0	0	0	0	0	100	0	0	0	1	99	0
HPCC:MPIFFT	2	2	9	69	19	7	0	13	36	44	0	0	26	74	0	0	0	86	14	0	0	0	0	95	5*	0	0	5	95	0
HPCC:MPI-RA	3	2	14	71	11	1	0	32	66	1	0	0	15	85	0	0	0	100	0	0	0	0	0	100	0	0	0	0	0	0
HPCC:PTRANS	1	1	55	40	4	0	0	54	30	16	0	0	75	25	0	0	0	100	0	0	0	0	0	100	0	0	0	40	60	0
Graph 500	0	0	32	68	0	0	0	16	84	0	0	0	10	90	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0
AMG (setup)	0	0	61	38	1	0	0	23	67	10	0	0	53	47	0	0	0	100	0	0	0	0	0	100	0	0	0	66	34	0
AMG (solve)	0	0	34	62	4	0	0	1	54	45	0	0	4	96	0	0	0	99	1	0	0	0	0	100	0	0	0	0	100	0
Nekbone	1	1	6	79	14	0	0	6	45	50	0	0	25	75	0	0	0	100	0	0	0	0	0	100	0	0	0	0	100	0
LULESH	0	0	3	90	7	0	0	1	35	64	0	0	48	52	0	0	0	0	100	0	0	0	0	100	0	0	0	41	59	0
S3D	0	0	5	52	42	0	0	12	41	47	0	0	28	72	0	0	0	79	21	0	0	0	0	100	0	0	0	3	97	0
GTC	0	0	36	64	0	0	0	29	54	17	0	0	4	95	0	0	0	100	0	0	0	0	88	12	0	0	0	77	23	0
ddcMD	0	0	4	95	1	0	0	2	17	81	0	0	28	72	0	0	0	68	32	0	0	0	0	100	0	0	0	45	55	0
LAMMPS_EAM	0	0	15	84	0	0	0	11	48	41	0	0	60	40	0	0	0	100	0	0	0	0	0	100	0	0	0	0	0	0
LAMMPS_LJ	0	0	11	88	0	0	0	10	36	53	0	0	62	38	0	0	0	100	0	0	0	0	0	100	0	0	0	0	0	0
LAMMPS_RH	1	1	14	84	1	0	0	12	40	48	0	0	52	48	0	0	0	95	5	0	0	0	0	100	0	0	0	13	87	0
NEK5000 (MHD)	1	0	14	77	8	0	0	14	36	50	0	0	42	58	0	0	0	100	0	0	0	0	0	100	0	0	0	0	100	0
POP	1	0	8	47	45	0	0	10	35	55	0	0	19	81	0	0	0	100	0	0	0	0	0	100	0	0	0	16	84	0

application. It is composed of over 60% floating-point operations; this is nearly double the next highest code (35% in ddcMD). It has the lowest number of branch operations (less than 1%) as well; most codes are many times higher. Virtually every one of its memory and floating-point operations are vectorized, which is unique among our test codes, and it has the lowest fraction (20%) of instructions devoted to memory operations. With respect to instruction mix, HPL has little in common with the "real" computational science applications we studied.

A comparison of instruction mixes can also provide interesting insights into how well a proxy app represents its full application. For instance, consider the Nekbone proxy app intended to represent some characteristics of the Nek5000 application. Though similar in some ways, we also see clear differences: the fraction of floating-point instructions in Nekbone is about double that of Nek5000, and the fraction of integer instructions is about half that of Nek5000. The different solvers in LAMMPS also exhibited some dissimilarities, such as a higher floating-point instruction mix for EAM and a higher integer mix for RHODO, but the similarities across these LAMMPS benchmarks outweighs their differences. For AMG, we separate the setup phase from the solution phase, as the setup phase tended to run longer than the solution phase for the benchmark problems we used. We saw a vast difference in the instruction mixes between these phases, with a much greater integer instruction mix during setup, and many more floating-point instructions during the solution.

Looking at trends across all of our test cases, memory operation mix is — except for HPL — quite similar across the codes. Most comprise 30% to 35% memory instructions, though the fraction of those that are SIMD varies from none to at most half.

The fraction of integer instructions is surprisingly high across a number of applications. A few standout examples in the applications are in GTC and the setup phase of AMG, at 38% and 53%, respectively. In benchmarks, HPCC's MPI-RandomAccess and PTRANS and Graph500 are high as well, around 40% integer instructions. Excluding LULESH (2.2%), the remaining codes are between 7% to 29% integer instructions. Interestingly, though integer instruction count is more than we expected, no code has any significant fraction of vectorized integer instructions; most between 0% and 0.2%, and S3D is the highest at 2%.

Table 5 shows the distribution of memory/move/arithmetic instructions according to their working data sizes, which can provide additional insight about the workloads, especially for designing more specialized hardware. The results in the table indicate that most of memory/arithmetic instructions work on either 32 or 64 bits, as expected.

3 Memory Behavior

3.1 Memory Bandwidth

All memory systems have a limited data transfer speed due to the limited capacity of hardware queues in the processor and the memory controller, the finite

number of buffers and ports in memory DIMMs, as well as the limited width and frequency of memory buses. Due to improvements in computational throughput such as speculative execution and hardware prefetching, applications often become bottlenecked by the transfer capacity of the memory system. This effect is exacerbated as the number of cores on a single chip increases, since those cores compete for limited memory resources.

It is important to understand how applications exercise the memory system. One metric that provides insight is the consumed memory bandwidth, which for many applications is a measure of achieved throughput. Memory bandwidth is defined as the ratio between the amount of data transferred to and from memory, and the time it takes to execute the application.

Achieved memory bandwidth is a performance metric dependent on both the application and the underlying architecture. We use hardware performance counters to measure the number of read and write transactions to memory. Modern micro-processors expose hardware performance events at the memory controller level. These events count read memory requests caused by both data accesses missing in the cache and prefetch requests initiated by the hardware prefetcher, as well as write transactions due to modified cache lines being written back to memory.

In our methodology, we use PAPI [29]-based calipers that record the number of read and write memory transactions as well as the wall clock time it takes to execute the code between the calipers. We compute separate bandwidth results for memory reads and writes.

Results. Each microprocessor in our experimental platform has its own integrated memory controller. As a result, we are interested primarily in the achieved memory bandwidths per socket. Figure 1 presents the read and write memory bandwidths per socket we measured for our test applications. The figure also includes the sustained peak machine bandwidth for one socket, measured for different ratios of read and write memory transactions. To measure these peak bandwidths, we wrote a micro-benchmark that accesses a large block of memory and modifies part of the memory. By varying the amount of memory modified, our micro-benchmark achieves different ratios of memory writes to memory reads. Note that with a write-allocate cache policy, our micro-benchmark can generate write and read memory transactions in a ratio of at most 1:1.

The data in Fig. 1 shows that POP is running very close to the test machine's peak memory bandwidth. Therefore, POP is memory bandwidth limited and would not benefit from an increase in core concurrency without similar increases in available bandwidth. At the other end of the spectrum, LULESH and GTC achieve only a small fraction of the machine's memory bandwidth. Because LULESH does not have MPI support, it was executed only in serial mode, achieving about 2GB/s of combined read and write memory bandwidth per core. GTC, however, does use MPI, and still exhibited a low memory bandwidth even when four GTC processes were run on each socket. The low GTC memory bandwidth results indicate that GTC is likely memory latency limited.

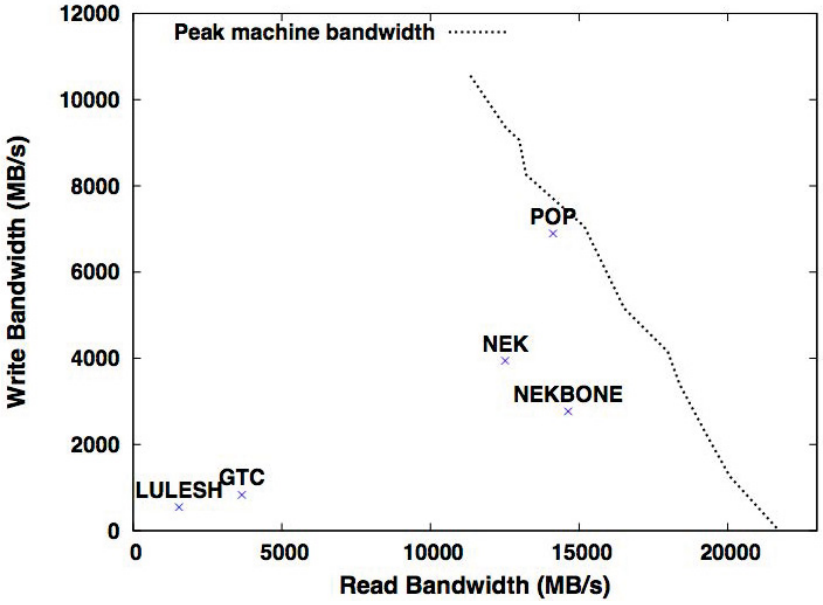


Fig. 1. Measured application bandwidths and sustained peak machine bandwidth per socket

Nek5000 and Nekbone are in between these two extremes. They achieve a significant fraction of the sustained machine bandwidth, with some room for core concurrency scaling. We also note that Nek5000 generates a higher ratio of write memory transactions than its proxy, Nekbone.

3.2 Reuse Distance

Reuse distance is defined as the number of distinctive data elements accessed between two consecutive references to the same element. We use reuse distance as a metric to quantify the pattern of data reuse or program locality. Reuse distance is independent of architecture, because it measures the volume of the intervening data accesses between two accesses. Reuse distance largely determines cache performance. For a fully associative cache under Least Recently Used replacement, reuse distance can accurately measure the number of cache hits or misses, given cache configurations. Reuse distance also allows direct comparison of data behavior across applications. As the memory hierarchy becomes deeper and more adaptive, it is increasingly important to quantify reuse distance to optimize application performance.

Measuring reuse distance is challenging due to its high cost in terms of time and space. For each memory access, we need to check previous memory access records to count distinctive data elements, which is often time consuming. For a program accessing a large amount of data, the space required to save previous

access records is also intimidating. To work around these problems, we use an approximate reuse distance analysis [30] with a Pin-based binary instrumentation tool to measure reuse distance. We use a splay tree to organize the last access record of each block of data. This approach relies on the observation that the accuracy of the last couple of digits of a reuse distance rarely matter [30]. This method significantly reduces the time and space requirements for measuring reuse distance.

Results. Using the tool described in Sect. 3.2, we measured data block reuse distance for 12 applications. We used a data block size of 64 bytes. Despite our optimizations for reducing measurement cost, measuring reuse distance empirically still has a high time and space cost. To make measurement feasible, we had to use smaller problems for some of the applications we studied. Thus, for LAMMPS, we simulated 1,600,000 atoms per process with the LJ and EAM problems (using approximately 575 MB and 498 MB per process, respectively) and 128,000 atoms per process for the RHODO problem (using approximately 474 MB per process).

Figure 2 shows the cumulative reuse distance function for one MPI task for each application. The X axis represents a given reuse distance value x, while the Y axis represents the percentage of application data block accesses that exhibit a reuse distance of x or less under each model. Although the distribution of reuse distance for a specific application is strongly correlated to the input problem size, Fig. 2 shows that the reuse difference (and hence program locality) differs greatly

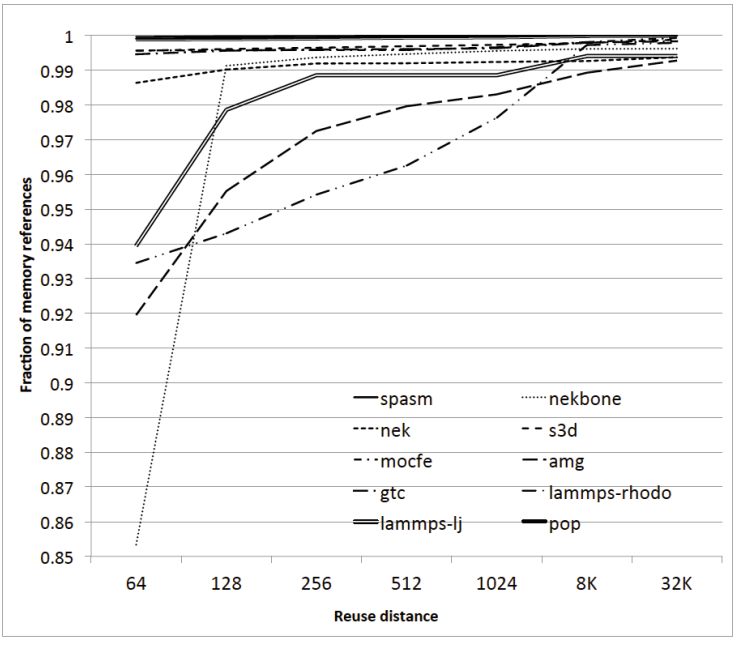


Fig. 2. Cumulative distribution functions (CDF) of reuse distance

between applications. In addition, reuse distance curves often have one or more knees and plateaus, which correspond to different working sets [31]. In many cases, the shape of a specific plot (knees and plateaus) is similar across different inputs [32]. Thus, although we used a single input per program in this study for most programs, our measurements suggest the applications, proxy apps, and benchmarks exhibit a wide diversity of working sets.

Figure 2 shows a substantial difference between Nekbone and Nek5000 with respect to reuse distance. In particular, 98% of Nek5000 memory references have a reuse distance of less than 64, while the corresponding number for Nekbone is only 85%. Upon further investigation, we found that Nekbone's run time is dominated by the conjugate gradient method (CG) and this method does not have good data locality. On the other hand, Nek5000 employs various numerical algorithms, including CG. Some of these algorithms have much better data locality than CG, which explains the higher concentration of reuse distances of less than 64 for Nek5000.

In Fig. 2, each CDF curve exhibits only a few plateaus/knees, showing that the reuse distance is highly concentrated in a few ranges. For example, AMG, Nekbone, and LAMMPS-LJ have 3, 2, and 3 plateaus/knees, respectively. This is consistent with earlier studies of reuse distance [31,32]. In addition, for SPASM, MOCFE, GTC, and POP, more than 99% memory references have reuse distances less than 64. The high concentration of reuse distance in a few ranges suggests that if a cache's size is just large enough to hold a few concentrated ranges, the cache performance will be almost the same as that of a larger cache. However, because of the differences in the concentrated ranges across applications, a dynamic cache with a few configurable effective cache sizes may be desirable for systems with a varied workload.

4 Communication

The efficiency and scalability of communication and synchronization operations is a critical determinant of the overall performance of most parallel programs. In this study, we focus on programs that use MPI for communication and synchronization, for two basic reasons. First, most scalable scientific applications use MPI including the applications we study here. Second, MPI defines an interface that allows straightforward capture of an application's messaging characteristics, so there are many tools available for measuring a program's MPI behavior.

In our methodology, the MPI data transfer operations we consider fall into two categories: *point-to-point* (P2P), in which a single sending MPI task transfers data to a single receiving MPI task, and *collective*, in which a group of processes participate in an operation such as a data broadcast or an all-to-all data exchange.

Table 6. Basic communication characteristics. (P2P and Coll % are the percentage of invocations of each class of MPI subroutines.)

Application	P2P %	Coll %	P2P Subroutines	Coll Subroutines	Comments
AMG	99.7	0.3	Recv, Isend, Irecv	Allreduce, Bcast, Allgather, Scan	In setup, even P2P messages are extremely small (<128 bytes).
Nekbone (linear)	39.8	60.2	Isend, Irecv	Allreduce	P2P or Allreduce gather-scatter implementation chosen dynamically
Nekbone (3D)	88.6	11.4			
MOCFE	44.1	55.9	Isend, Irecv	Allreduce, Reduce	
S3D	99.7	0.3	Isend, Irecv, Send	Allreduce, Bcast	3D Nearest Neighbor on Reg. Grid, periodic BC
SPASM	100.0	negl	SendRecv	Allreduce, Barrier	3D Nearest Neighbor on Reg. Grid, periodic BC
GTC	78.9	21.1	SendRecv	Allreduce	1D Nearest Neighbor along toroid domain
ddcMD	90.0	10.0	Isend, Irecv, Send, Recv	Allreduce, Bcast, Gather, Scatter	Unstructured, flexible domain decomposition
LAMMPS-EAM	99.0	1.0	Send, Irecv	Allreduce	Large P2P messages (>500KB), small collective messages (<64 Bytes); RHODO adds non-neighbor P2P comm. for long-range forces
LAMMPS-LJ	100.0	0.0			
LAMMPS-RHODO	97.7	2.3			
Nek5000	96.6	3.4	Isend, Irecv, Send, Recv	Bcast, Allreduce, Barrier	2D/3D nearest-neighbor communication patterns on an unstructured grid
POP	54.4	45.6	Isend, Irecv	Allreduce	2D Nearest Neighbor

Table 7. P2P communication characteristics. Buffer size columns show histogram bin upper limit

Application	Topology	n	Buffer Size		
			Min	Max	Med.
AMG	Unstructured in 2D extruded to 3D	256	2^3	2^{22}	2^7
Nekbone	Linear	128	2^9	2^9	2^9
Nekbone	3D geometry	128	2^6	2^{22}	2^{14}
MOCFE	3 Dim of Parallelism – Mesh, Energy Group, & Angle	256	2^{13}	2^{14}	2^{14}
S3D	3D Nearest Neighbor	256	2^5	2^{16}	2^{15}
SPASM	3D Nearest Neighbor	256	2^3	2^{13}	2^6
GTC	1D Nearest Neighbor along the toroid domain	256	2^4	2^{19}	2^{18}
ddcMD	3D unstructured	256	2^3	2^{17}	2^{15}
LAMMPS-EAM	3D Nearest Neighbor	96	2^3	2^{22}	2^{20}
LAMMPS-LJ	3D Nearest Neighbor	96	2^{21}	2^{21}	2^{21}
LAMMPS-RHODO	Mainly 3D Nearest Neighbor	96	2^3	2^{22}	2^{17}
Nek5000	2D/3D nearest-neighbor	128	2^3	2^{20}	2^{10}
POP	2D Nearest Neighbor	64	2^9	2^{12}	2^9

In this study, we examined the communication behavior of our test applications running on the KID system's Infiniband interconnection network. Table 6 summarizes the communication behavior of the applications we studied. In the table, the columns indicating percentage of point-to-point and collective communication operations show the percentage of the total MPI operation count.

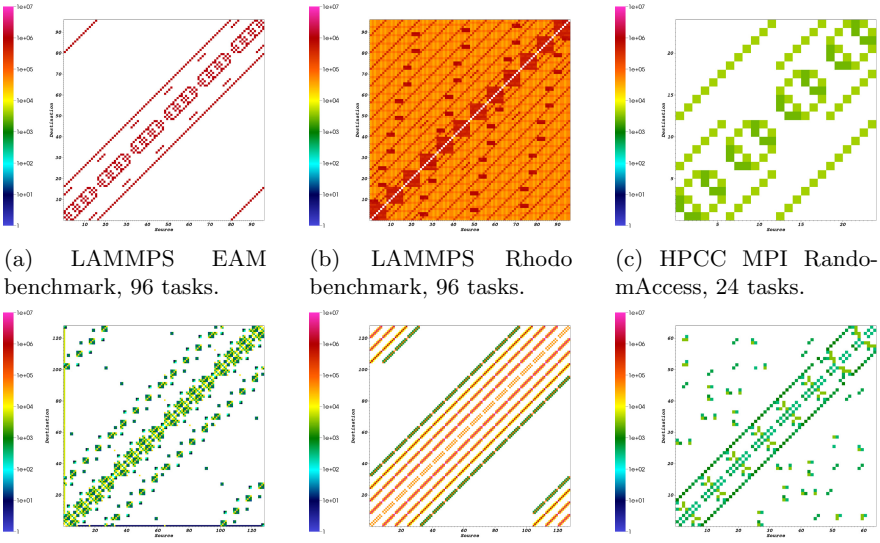
To collect data about application communication behavior and performance, we use mpiP version 3.3. Normally mpiP presents summary statistics about a program's MPI behavior, but we modified mpiP to also collect data about the number of point-to-point operations performed between each pair of program processes and the volume of data transferred in those operations. Our modified mpiP outputs this data in the form of adjacency matrices. Visualizing such matrices is a concise and effective way to communicate the topology of a program's point-to-point communication behavior. For example, visualizing the adjacency matrix for an application whose tasks communicate only with their nearest neighbors in a Cartesian topology (e.g., a 3D stencil operation) produces a distinctive, repeating pattern near the matrix diagonal. With some practice, common communication patterns can be recognized in these visualizations. We also modified mpiP to generate histograms of the data sizes used in point-to-point and collective operations. These histograms give an indication of the type

of demands a program places on a system's interconnection network, such as whether a program performs a large number of collective operations involving small messages.

4.1 Point-to-Point Communication

Figure 3 shows average point-to-point communication volume per iteration for three benchmark problems used in our study. In each case, the data shown was captured during the first ten iterations of the program's main loop. These figures display the communication data as a matrix, such that the block at location (s, d) is colored according to the volume of data sent from MPI rank s to MPI rank d.

Figure 3a shows the average communication volume for the LAMMPS EAM benchmark running on 96 processes. The repeated pattern in the figure reflects a three-dimensional nearest-neighbor communication pattern. Figure 3b also suggests a three-dimensional nearest-neighbor communication pattern, but unlike the pattern for the EAM benchmark the pattern suggests that sub-groups of processes participate in communication topology that is fully-connected within each sub-group. Also, there is a significant amount of point-to-point communication between MPI tasks that are not neighbors in the spatial decomposition. This is to be expected, because unlike the EAM benchmark, the Rhodo benchmark includes long range forces in its computation of potential. These two matrices



(a) LAMMPS EAM benchmark, 96 tasks.

(b) LAMMPS Rhodo benchmark, 96 tasks.

(c) HPCC MPI RandomAccess, 24 tasks.

(d) Nek5000 application, 128 tasks.

(e) Nekbone proxy app, 128 tasks.

(f) POP application, 64 tasks.

Fig. 3. Average volume of point-to-point communication. Color scale is consistent across all plots.

also illustrate how much the communication pattern of the same program can vary depending on the problem input.

Figure 3c presents the average point-to-point communication volume per iteration for the HPCC MPI RandomAccess phase. Because the RandomAccess benchmark performs updates to memory locations selected randomly with a uniform distribution across all processes involved in the benchmark, one expects to see that each MPI task communicated approximately the same amount to each of the other MPI tasks, giving a matrix that is all one color except on the diagonal (since a process need not use MPI operations for updates within itself). Thus, the communication pattern shown in the figure, a nearest neighbor pattern, is counter-intuitive. In fact, the figure is correct and results from the use of an algorithm optimization that organizes the available MPI tasks into a virtual hypercube topology and routes messages along this topology.

Figure 3d and 3e highlight the differences in point-to-point communication patterns between a full application, Nek5000, and a proxy app intended to mimic that application's behavior, Nekbone. The proxy app's communication pattern is much more regular than that of the full application. Note that this difference does not necessarily mean the proxy app is not a valid stand-in for the full application, but it does suggest that the proxy is not a good representative *with respect to communication behavior* for the input set we used.

Finally, Fig. 3f shows the average point-to-point communication volume for the well-studied POP application. This figure suggests POP's primary point-to-point communication pattern is nearest neighbor, but that some processes also communicate with processes that aren't necessarily neighbors. This non-neighbor communication appears somewhat random, and is likely a result of the way that the earth's oceans are mapped to the available MPI tasks.

4.2 Collective Communication

The programs we studied exhibited substantial variety in their collective communication behavior. The programs varied in the number and size distribution of the messages they sent using collective operations, but most that used collectives did so using small message sizes. For example, our three LAMMPS benchmark problems either did not use collectives at all during the main computation phase (LJ benchmark), or sent very little data per operation (EAM and RHODO, which performed collectives using messages with fewer than 64 bytes). Likewise, the "global" phases of the HPCC benchmark exhibited more varied behavior. Both MPIRandomAccess and MPIFFT exhibited a bimodal distribution, with some small collective operations (fewer than 32 bytes) but also larger collective operations (MPIRandomAccess issued over 250,000 collective operations that sent 256KB per process, while MPIFFT issued operations using 256MB per process). In both cases, the operations involving a larger amount of data were all-to-all operations. Like LAMMPS, the collective operations in PTRANS all involved small amounts of data per operation. Our modified version of mpiP reported no collective data transfer operations used within the HPL phase of the benchmark suite. Inspection of the HPCC source code shows that HPL *does* use collective

operations, but it uses its own implementation based on MPI point-to-point operations instead of the MPI collective operations. Because mpiP only collects data about calls to MPI functions, our current methodology cannot detect these data transfers as logically collective operations.

5 Observations

In this section, we make a number of observations from this evidence for future architectures. First, we consider instruction mix.

1. None of the applications makes use of integer SIMD instructions, even though some of the applications do a reasonable amount of integer calculation.
2. About half of tested applications have more integer operations (IntOps + IntSIMD) than floating-point operations (FpOps + FpSIMD).
3. All applications except for LULESH have non-negligible amount of integer computations.
4. MemSIMD is rare, only occurring in S3D, POP, and MOCFE. And in those three cases, it is still lower (by percentage) than non-SIMD mem ops.
5. For all apps except ddcMD, the number of memory operations (MemOps + MemSIMD) are greater than the number of floating-point operations (FpOps + FpSIMD).
6. When FpSIMD is high, the number of branches is always low.

Second, we review the memory behavior of our applications. Not surprisingly, memory behavior has a dramatic impact on performance, but it is also more difficult to measure.

1. POP runs at close to peak machine bandwidth and generates a higher ratio of write memory transactions than the other applications in this study.
2. GTC achieves a very low memory bandwidth utilization, indicating a poor memory access pattern.
3. The reuse distance is highly concentrated in a few ranges, indicating the opportunity for cache architecture improvements.

Third, communication is one of the most important behaviors in determining overall performance for scalable scientific applications.

1. All of the distributed memory applications use the `Allreduce` collective operation with small data payloads (i.e., one double precision number). This analysis reconfirms an earlier observations [9], and has been used to motivate hardware support for collective offload engines in the interconnect.
2. In general, the applications and benchmarks exhibited either a uni-modal collective communication distribution with very small payloads, or a bimodal distribution with both small payloads and very large payloads. Often the large amounts of data sent were used in `Alltoall` operations, such that each process sent a smaller amount of data to each other process, but the aggregate amount of data sent was large.

3. The communication operations for several of the applications we studied were nearly all point-to-point (P2P) operations (by count). This preference for P2P operations appears to be driven mainly by the need for scalability; collective operations, even when implemented with optimal algorithms, can serve as a scalability limitation. Nevertheless, most applications require at least a few collective operations.
4. As expected, the basic P2P communication behavior of the applications that explicitly simulate a physical system is a nearest neighbor communication pattern. The applications differ significantly, however, in how much data is transmitted through those P2P operations, and whether they exhibit an element of non-neighbor communication.
5. The runtime communication selection in Nek5000 and Nekbone reconfirms that a well-optimized collective communication library generally performs better than the P2P-based counterpart.

Finally, aside from specific architecture metrics, we also compare some of the proxy applications against real applications. Benchmarks and proxy applications are very valuable because these kernels provide hardware and software architects with comprehensible code segments that can be simulated and easily rewritten in alternative programming languages. However, because these benchmarks and proxy applications are precisely simplified versions of their real-world counterparts, they also can have different behaviors.

1. Not surprisingly, HPL is a significant outlier from all the applications we tested: practically all (79.4%) of its instructions are memory and floating-point SIMD operations. HPL has more that twice the number of floating-point operations than the other applications, proxy apps, and benchmarks we studied.
2. The proxy app (e.g., Nekbone) and the corresponding full application (e.g., Nek5000) can have different reuse distance distributions. It is important to investigate the full application to understand data locality.
3. Proxy applications and benchmarks tend to have higher rates of SIMD instructions because complex memory access patterns have been removed from compute-intensive loops, and the compiler can optimize and identify SIMD optimizations with higher clarity.
4. Finally, the communication topologies that we measured show that some of the proxy apps do not necessarily represent the communication behavior of the application they are intended to model.

6 Summary

We have presented an empirical analysis of several important scalable scientific applications, benchmarks, and proxy applications. Using a methodology supported by a toolkit of performance tools that allows us to study detailed computation, memory, and communication behavior at varying levels of resolution, we confirmed many of our expectations but also found a number of surprises. In this

time of rapid architectural change for the sake of balancing energy efficiency and reliability against realized performance, the quantitative measurements provided by applying our methodology are critical for finding the right balance point.

References

1. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.C., Barkai, D., Berthou, J.Y., Boku, T., Braunschweig, B., Cappello, F., Chapman, B., Chi, X., Choudhary, A., Dosanjh, S., Dunning, T., Fiore, S., Geist, A., Gropp, B., Harrison, R., Hereld, M., Heroux, M., Hoisie, A., Hotta, K., Jin, Z., Ishikawa, Y., Johnson, F., Kale, S., Kenway, R., Keyes, D., Kramer, B., Labarta, J., Lichnewsky, A., Lippert, T., Lucas, B., Maccabe, B., Matsuoka, S., Messina, P., Michielse, P., Mohr, B., Mueller, M.S., Nagel, W.E., Nakashima, H., Papka, M.E., Reed, D., Sato, M., Seidel, E., Shalf, J., Skinner, D., Snir, M., Sterling, T., Stevens, R., Streitz, F., Sugar, B., Sumimoto, S., Tang, W., Taylor, J., Thakur, R., Trefethen, A., Valero, M., van der Steen, A., Vetter, J., Williams, P., Wisniewski, R., Yelick, K.: The international exascale software project roadmap. International Journal of High Performance Computing Applications **25**(1), 3–60 (2011)
2. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R.S., Yelick, K.: Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA Information Processing Techniques Office (2008)
3. Snir, M., Gropp, W.D., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W. (eds.): MPI-the complete reference (2-volume set) 2nd edn. Scientific and Engineering Computation. MIT Press, Cambridge (1998)
4. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S.: The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
5. Vetter, J.S., Yoo, A.: An empirical performance evaluation of scalable scientific applications. In: SC 2002, Baltimore, MD, USA. IEEE (2002)
6. Shalf, J., Kamil, S., Oliker, L., Skinner, D.: Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 17. IEEE Computer Society (2005)
7. Brightwell, R., Underwood, K.D.: An analysis of the impact of mpi overlap and independent progress. In: Proceedings of the 18th Annual International Conference on Supercomputing, Malo, France, pp. 298–305. ACM (2004)
8. Riesen, R.: Communication patterns. In: 20th International Parallel and Distributed Processing Symposium (IPDPS), 8 p. (2006)
9. Vetter, J.S., Mueller, F.: Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In: International Parallel and Distributed Processing Symposium (IPDPS), Ft. Lauderdale, Florida (2002)
10. Vetter, J.S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., Yalamanchili, S.: Keeneland: Bringing heterogeneous GPU computing to the computational science community. IEEE Computing in Science and Engineering **13**(5), 90–95 (2011)

11. Dongarra, J.J., Luszczek, P.: Introduction to the hpcchallenge benchmark suite. Technical Report ICL-UT-05-01, Innovative Computing Laboratory, University of Tennessee-Knoxville (2005)
12. Brown, P.N., Falgout, R.D., Jones, J.E.: Semicoarsening multigrid on distributed memory machines. SIAM Journal on Scientific Computing **21**(5), 1823–1834 (2000)
13. Smith, M.A., Marin-Lafleche, A., Yang, W.S., Kaushik, D., Siegel, A.: Method of characteristics development targeting the high performance Blue Gene/P computer at argonne national laboratory. In: Proceedings of the International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (MC 2011). American Nuclear Society (2011)
14. Karlin, I., Bhatele, A., Chamberlain, B.L., Cohen, J., Devito, Z., Gokhale, M., Haque, R., Hornung, R., Keasler, J., Laney, D., Luke, E., Lloyd, S., McGraw, J., Neely, R., Richards, D., Schulz, M., Still, C.H., Wang, F., Wong, D.: Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory (December 2012)
15. Chen, J.H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E.R., Klasky, S., Liao, W.K., Ma, K.L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yoo, C.S.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science and Discovery **2**(1) (2009)
16. Spafford, K.L., Meredith, J.S., Vetter, J.S., Chen, J., Grout, R., Sankaran, R.: Accelerating S3D: A GPGPU case study. In: HeteroPar 2009: Proceedings of the Seventh International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Platforms (2009)
17. Germann, T.C., Kadau, K.: Trillion-atom molecular dynamics becomes a reality. International Journal of Modern Physics C **19**(09), 1315–1319 (2008)
18. Lee, W.W.: Gyrokinetic approach in particle simulation. Physics of Fluids **26**, 556–562 (1983)
19. Richards, D.F., Glosli, J.N., Chan, B., Dorr, M.R., Draeger, E.W., Fattebert, J.L., Krauss, W.D., Spelce, T., Streitz, F.H., Surh, M.P., Gunnels, J.A.: Beyond homogeneous decomposition: Scaling long-range forces on massively parallel systems. In: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis, SC 2009. ACM, New York (2009)
20. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics **117**, 1–19 (1995)
21. Fischer, P., Lottes, J., Kerkemeier, S.: Nek5000 website (2008)
22. Smith, R.D., Dukowicz, J.K., Malone, R.C.: Parallel ocean general circulation modeling. Physica D **60**(1–4), 38–61 (1992)
23. Collins, W.D., Blackmon, M.L., Bonan, G.B., Hack, J.J., Henderson, T.B., Kielh, J.T., Large, W.G., McKenna, D.S., Bitz, C.M., Bretherton, C.S., Carton, J.A., Chang, P., Doney, S.C., Santer, B.D., Smith, R.D.: The Community Climate System Model version 3 (CCSM3). Journal of Climate **19**(11), 2122–2143 (2006)
24. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
25. Intel Corporation: XED, http://software.intel.com/sites/landingpage/pintool/docs/53271/Xed/html
26. Intel Corporation: Intel Architecture software developer's manual, vol. 1: basic architecture (1999)

27. Advanced Micro Devices Inc: 3DNow! technology manual (2000)
28. Intel Corporation: Intel SSE4 programming reference (April 2007)
29. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications **14**, 189–204 (2000)
30. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2003)
31. Schuff, D.L., Parsons, B.S., Pai, V.S.: Multicore-aware reuse distance analysis. In: Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems (2010)
32. Ding, C., Zhong, Y.: Reuse distance analysis. Technical Report UR-CS-TR-741, Computer Science Department, University of Rochester (2001)