Exploring Tradeoffs Between Power and Performance for a Scientific Visualization Algorithm

Stephanie Labasan*

University of Oregon

ABSTRACT

Power is becoming a major design constraint in the world of high-performance computing (HPC). This constraint affects the hardware being considered for future architectures, the ways it will run software, and the design of the software itself. Within this context, we explore tradeoffs between power and performance. Visualization algorithms themselves merit special consideration, since they are more data-intensive in nature than traditional HPC programs like simulation codes. This data-intensive property enables different approaches for optimizing power usage.

Our study focuses on the isosurfacing algorithm, and explores changes in power and performance as clock frequency changes, as power usage is highly dependent on clock frequency. We vary many of the factors seen in the HPC context — programming model (MPI vs. OpenMP), implementation (generalized vs. optimized), concurrency, architecture, and data set — and measure how these changes affect power-performance properties. The result is a study that informs the best approaches for optimizing energy usage for a representative visualization algorithm.

1 INTRODUCTION

Power is a central issue for achieving future breakthroughs in high performance computing (HPC). As today's leading edge supercomputers require between 5 and 18 MegaWatts to power, and as the cost of one MegaWatt-year is approximately one million US dollars, supercomputing centers regularly spend over five million US dollars per year, and sometimes exceed ten million US dollars per year. Worse, power usage is proportional to the size of the machine; scaling up to even larger machines will cause power usage (and associated costs) to grow even larger. Applying today's designs to exascale computing would cost hundreds of millions of US dollars to power. As a result, the HPC community has made power efficiency a central issue, and all parts of the HPC ecosystem are being re-evaluated in the search for power savings.

Supercomputers require a varying amount of power. When running programs that stay within the machine's normal operating limits, the amount of power often matches the usage for when the machine is idle. However, programs that engage more of the hardware — whether it is caches, additional floating point units, main memory, etc. — use more power. HPL (High Performance Linpack), a benchmark program that is computationally intensive, has been known to triple power usage, since HPL has been highly optimized and makes intense use of the hardware. However, many visualization programs have not undergone the same level of optimization, and thus only require power near the machine's idle rate. That said, alternate approaches exist that do create opportunities for data-intensive programs — *i.e.,* visualization programs — to achieve energy savings.

*e-mail: slabasan@cs.uoregon.edu

As power usage is highly dependent on clock frequency, reductions in frequency can lead to significant power savings. On the face of it, reducing the clock frequency seems like, at best, a break-even strategy — *i.e.,* running at half the speed should take twice as long to execute. However, visualization programs are different than traditional HPC workloads, since many visualization algorithms are data-intensive. So, while HPC workloads engage floating point units (and thus drive up power), visualization workloads make heavier use of the memory infrastructure.

The data-intensive nature of visualization algorithms creates an opportunity: newer architectures have controls for slowing down the clock frequency, but keeping the memory operating at a normal speed. In this case, power is being devoted to main memory at the same rate (which is good because memory is often a bottleneck), but power is devoted to the CPU at a lower rate (which is also good because the CPU is being under-utilized). As the extreme outcome, then, it is conceivable that slowing down the clock frequency (and keeping the memory operating at full speed) could lead to a scenario where the execution time is the same (since the memory performance dominates), but the power usage drops.

With this study, we explore the efficacy of varying clock speed to achieve energy savings. The achieved effects vary based on myriad factors, and we endeavor to understand those factors and their impacts. Our study focuses on a representative visualization algorithm (isosurfacing), and looks at how that algorithm performs under a variety of configurations seen in HPC settings. We find that these configurations have real impacts on power-performance tradeoffs, and that some approaches lend themselves to better energy efficiency than others.

2 RELATED WORK

2.1 Power

Power is widely viewed as the central challenge for exascale computing, and that challenge is expected to impact exascale software [7], including visualization software [2]. Outside of visualization, many HPC researchers have looked at how to reduce energy usage at different levels ranging from the processor to the full system, including tradeoffs between power and performance. Porterfield et al. [20] looked into the variability in the performance to energy usage at the processor level using OpenMP and MPI. Other research has been dedicated to reduce the total power usage of the system [15, 20, 10, 9]. Ge et al. [12] developed compute-bound and memory-bound synthetic workload to demonstrate that power-performance characteristics are determined by various characteristics in the application.

The study closest to ours was by Gamell et al. [11]. In this work, they investigated the power-performance relationship for visualization workloads. However, our studies are complementary, as they looked at behaviors across nodes and we aim to illuminate the behavior within a node. A second noteworthy study was by Johnsson et al. [14]. In this study, the authors studied power usage on a GPU when different rendering features were enabled and disabled. Our study is different in nature, as we are studying a visualization algorithm, and also we are studying impacts of programming

model, data set, and architectural features, rather than changing the (OpenGL) rendering algorithm itself.

2.2 Visualization

Our study focuses on a traditional isosurfacing algorithm using Marching Cubes [16]. While our study does not preclude using an acceleration structure to quickly identify only the cells that contain the isosurface [6], we did not employ this optimization since we wanted the data loads and stores to follow a more controlled pattern.

Most of our experiments were run using our own implementation of an isosurface algorithm. However, some experiments were run using the isosurfacing algorithm in the Visualization ToolKit (VTK) [21]. Further, we believe that the corresponding results are representative of the class of general frameworks, *e.g.,* OpenDX [1] and AVS [23], and of the end-user tools built on top of such frameworks (*i.e.,* VTK-based ParaView [3] and VisIt [5]).

3 Benchmark Tests

One goal for this study is to identify situations where the clock frequency can be reduced, but the execution time does not increase proportionally. In such cases, energy savings are possible. However, we should only expect the execution time to be maintained if the computation is data-bound. This situation occurs when data load and store requests exceed what the memory infrastructure can deliver.

To get a sense of when programs are data-bound and when they are compute-bound, we created four micro-benchmarks. These benchmarks will also inform the spectrum of outcomes we can expect. The benchmarks are:

- *computeBound*: A compute-bound workload performing several multiplication operations on one variable.
- *computeBoundILP*: The above compute-bound benchmark with instruction-level parallelism. This enables pipelining of multiple instructions.
- *memoryBound*: A memory-bound benchmark that accesses an element in an array and then writes it to another array based on an index.
- *memoryBoundCacheThrash*: The above memory-bound benchmark, but the indices that map the output value have been randomized, removing any benefit of locality.

Figure 1 shows the performance results of our micro-benchmarks with varying CPU clock frequencies. Our original hypothesis was that the *computeBound* workload would double in execution time if run at half the speed, the *memoryBoundCacheThrash* application would have the most consistent runtime across all frequencies, and the *computeBoundILP* and *memoryBound* workloads would have changes in runtime that fall between the two extremes. From the figure, we find that the *computeBound* workload follows our initial premise. The *memoryBoundCacheThrash* workload stays relatively consistent, but there is a slight increase in runtime when run at the lowest frequency. Even with a synthetic data-bound workload, we are not able to achieve perfect consistency in runtime over varying frequencies. This means that we should not expect visualization workloads to achieve perfect consistency in runtime, since they have significantly more computations than the synthetic workload, and since they use cache in a more coherent way.

4 Experimental Setup

The following section details the various parameters in our experiments.

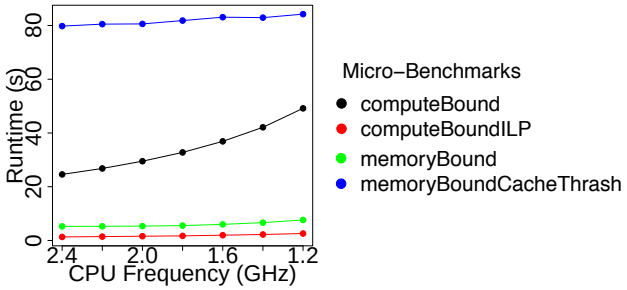

Figure 1: Performance results of our micro-benchmarks with varying frequencies. The *computeBound* workload is directly proportional to the clock speed, while the *memoryBoundCacheThrash* is independent of the change in clock frequency.

Table 1: Increase in runtime for the four micro-benchmarks when slowing down the clock frequency by a factor of 2X. Though *memoryBoundCacheThrash* is synthetically designed to be the most data-intensive workload, it still does not hold constant in runtime across frequencies, *i.e.,* achieve a ratio of exactly 1X.

Benchmark	Time		Ratio
	2.4 GHz	1.2 GHz	
computeBound	24.59s	49.17s	2X
computeBoundILP	1.32s	2.58s	2X
memoryBound	5.25s	7.63s	1.4X
memoryBoundCacheThrash	79.78s	84.22s	1.1X

4.1 Factors Studied

We considered the following factors:

- **Hardware architecture.** Architecture is important, since each architecture has different power requirements at varying clock frequencies, and also different caching characteristics.
- **CPU clock frequency.** As the clock speed slows down, the data-intensive workloads may not slow down proportionally, creating opportunities for power savings.
- **Data set.** Data set dictates how the algorithm must traverse memory. While structured data accesses memory in a regular pattern, unstructured data may have more random arrangements in memory, increasing the data intensity.
- **Concurrency.** Concurrency affects the demands placed on the memory subsystem: more cores are more likely to hit the memory infrastructure's bandwidth limit, while fewer cores are less likely.
- **Algorithm implementation.** The algorithm implementation dictates the balance of computations and data loads and stores. Across implementations, the total number of instructions and the ratios of instruction types will change, which in turn could affect power-performance tradeoffs.
- **Parallel programming model.** The programming model affects how multi-core nodes access data and the necessary memory bandwidth for the algorithm. Specifically, coordinated accesses across cores can reduce cache thrashing, while uncoordinated accesses can increase cache thrashing.

4.2 Configurations

Our study consisted of six phases and 277 total tests. It varied six factors:

- Hardware architecture: 2 options
- CPU clock frequency: 7 or 11 options, depending on hardware architecture
- Data set: 8 options
- Concurrency: 4 options
- Algorithm implementation: 2 options
- Parallel programming model (OpenMP vs. MPI): 2 options

4.2.1 Hardware Architecture

We studied two architectures:

- **CPU1**: A Haswell Intel i7 4770K with 4 hyper-threaded cores running at 3.5 GHz, and 32 GB of memory operating at 1600 MHz. Each core has a private L1 and L2 cache running with a bandwidth of 25.6 Gbytes/s.
- **CPU2**: A single node of NERSC's Edison machine. Each node contains two Intel Ivy Bridge processors, and each processor contains 12 cores, running at 2.4 GHz. Each node contains 64 GB of memory operating at 1866 MHz. Each core has a private L1 and L2 cache, with 64 KB and 256 KB, respectively. A 30 MB L3 cache is shared between the 12 cores. The cache bandwidth for L1/L2/L3 is 100/40/23 Gbytes/s.

Both CPUs enable users to set a fixed CPU clock frequency as part of launching the application. CPU1 uses the Linux `cpufreq-utils` tool, while CPU2 uses an `aprun` command line argument to specify the frequency of a submitted job. Both CPUs are also capable of reporting total energy usage and power consumed (see Section 4.3).

Finally, it is important to note that Intel Haswell processors (*i.e.,* CPU1) do not tie cache speeds to clock frequency, but Intel Ivy Bridge processors (*i.e.,* CPU2) do force their caches to match clock frequency, and thus their caches slow down when clock frequency is reduced.

4.2.2 CPU Clock Frequency

For CPU1, we were able to set the clock frequency at 11 different options, from 1.6 GHz to 3.5 GHz (nominal frequency). For CPU2, the hardware controls only allowed for 7 options, from 1.2 GHz to 2.6 GHz (nominal frequency).

4.2.3 Data Set

In this study, we consider only unstructured meshes, although we consider different sizes, and different cache coherency characteristics. Our study used the following eight data sets:

- **Enzo-1M**: a cosmology data set from the Enzo [19] simulation code originally mapped on a rectilinear grid. We decomposed the data set into 1.13 million tetrahedrons.
- **Enzo-10M**: a 10.5 million tetrahedron version of Enzo-1M.
- **Enzo-80M**: an 83.9 million tetrahedron version of Enzo-1M.
- **Nek5000**: a 50 million tetrahedron unstructured mesh from a Nek5000 [8] thermal hydraulics simulation. Nek5000's native mesh is unstructured, but composed of hexahedrons. For this study, we divided these hexahedrons into tetrahedrons.
- **REnzo-1M, REnzo-10M, REnzo-80M, RNek5000**: Altered versions of the above data sets. We randomize the point indices such that accesses are irregular and locality is not maintained.

We selected an isovalue of 170 for the Enzo data sets, and 0.3 for Nek5000. While "isovalue" could have served as another parameter for our study, we found that varying it did not significantly affect results.

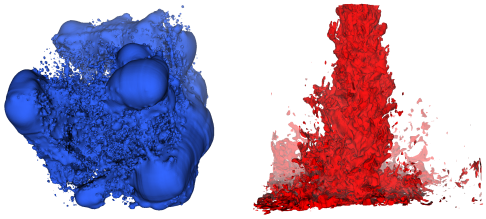

Figure 2: Visualizations of the two data sets used in this study. The Enzo data set is on the left and Nek5000 is on the right.

4.2.4 Concurrency

For CPU1, we ran tests using 1, 2, 3, and 4 cores. For CPU2, no tests of this nature were run.

4.2.5 Algorithm Implementation

The isosurfacing algorithm is a standard visualization technique used to extract a surface from 3D data (in 2D, a plane would be extracted). This surface (or plane) represents points of a constant value, specified by the isovalue. Computing the isosurface involves iterating over all cells in the input data set, comparing each cell to the isovalue, and outputting a new data set consisting of valid cells.

We implemented two different versions of isosurfacing for our study, which target two different use cases:

- **BaselineImplementation**: We wrote our own isosurfacing algorithm, which consisted of only the instructions necessary to perform Marching Cubes on tetahedral data (Marching Tets). Thus, it was efficient for that purpose, especially in terms of minimal numbers of instructions. Though the more common case would make use of the implemented algorithm included in a general-purpose visualization software toolkit (also included in this study for comparison), we wanted to explore the effects of an alternative implementation that could achieve the same algorithm using a minimum number of instructions. We implemented versions of our code to work with both MPI and OpenMP.
- **GeneralImplementation**: Isosurface implemented using a general-purpose visualization software library (VTK), specifically the vtkContourFilter. Generalized frameworks like VTK sacrifice performance to ensure that their code works on a wide variety of configurations and data types. As a result, the performance characteristics of such a framework are different (specifically having an increased number of instructions), and thus the opportunities for power-performance tradeoffs may also be different. The vtkContourFilter does not work with OpenMP, so our only supported programming model for this implementation was with MPI (via our own custom MPI program that incorporated this module).

4.2.6 Parallel Programming Model

We implemented our isosurfacing algorithm using both the OpenMP [4] and MPI [22] parallelism approaches within a node. Depending on the approach, the work of iterating over the cells in the input can be divided among invididual cores in some way in order to increase performance. The approach may impact the percentage of time spent waiting for data being fetched from main memory (depending on if accesses are coordinated or uncoordinated), which may influence power-performance tradeoffs.

With the OpenMP approach, all cores operated on a common data set, meaning that cache accesses can be done in a coordinated way. Our OpenMP implementation used default thread scheduling, and the cells in the data set were split evenly among the threads. With this method, the chunk size is determined by the number of iterations in the loop divided by the number of OpenMP threads. That said, we experimented with many chunking strategies and were not able to locate any that significantly outperformed the default.

With the MPI approach, each core operated on an exact local copy of the data in its own space. In other words, our approach was embarrasingly parallel, and we did not add logic to partition the work across parallel tasks. As a result, the cores were operating independently to compute the isosurface, creating uncoordinated cache accesses.

Regardless of the parallel programming model, the algorithm operated on the same data size, and each compute element (whether that be an OpenMP thread or an MPI rank) iterated over the same number of cells. Given C cores and N cells in the data set, the OpenMP approach divides the work across the cores, such that each core operates on N/C cells, and repeats this assigned work C times. The OpenMP approach coordinates cores and their data accesses. The MPI approach, on the other hand, would have each core operate on N cells, and each core would have uncoordinated data accesses. Further, in order to obtain reliable measurements, we had each algorithm execute ten times, and the reported measurements are for all ten executions.

4.3 Performance Measurements

We enabled PAPI [18] performance counters to gather measurements for each phase of the algorithm. Specifically, we capture `PAPI_TOT_INS`, `PAPI_TOT_CYC`, `PAPI_L3_TCM`, `PAPI_L3_TCA`, and `PAPI_STL_ICY`. (Note that `PAPI_TOT_INS` counts all instructions executed, which can vary from run to run due to CPU branch speculation. Unfortunately, we were not able to count instructions retired, which would be consistent across runs.)

We then derive additional metrics from the PAPI counters:

- instructions executed per cycle (IPC) = `PAPI_TOT_INS` / `PAPI_TOT_CYC`
- L3 cache miss rate = `PAPI_L3_TCM` / `PAPI_TOT_CYC`

On CPU1, we used Intel's Running Average Power Limit (RAPL) [13] to obtain access to energy measurements. This instrumentation provides a per socket measurement, aggregating across cores. On CPU2, we took power and energy measurements using the Cray XC30 power management system [17]. This instrumentation provides a per node measurement, again aggregating across cores.

4.4 Methodology

Our study consisted of six phases. The first phase studied a base case, and the subsequent phases varied additional dimensions from our test factors, and analyzed the impacts of those factors.

4.4.1 Phase 1: Base Case

Our base case varied the CPU clock frequency. It held the remaining factors constant: CPU1, Enzo-10M, four cores (maximum concurrency available on CPU1), the BaselineImplementation, and the OpenMP parallel programming model. The motivation for this phase was to build a baseline understanding of performance.

Configuration: (CPU1, 4 cores, Enzo-10M, BaselineImplementation, OpenMP) $\times$ 11 CPU clock frequencies

4.4.2 Phase 2: Data Set

In this phase, we continued varying clock frequency and added variation in data set. It consisted of 88 tests, of which 11 were studied in Phase 1 (the 11 tests for Enzo-10M).

Configuration: (CPU1, 4 cores, BaselineImplementation, OpenMP) $\times$ 11 CPU clock frequencies $\times$ 8 data sets

4.4.3 Phase 3: Parallel Programming Models

In this phase, we continued varying clock frequency and data set and added variation in parallel programming model. It consisted of 176 tests, of which 88 were studied in Phase 2 (the OpenMP tests).

Configuration: (CPU1, 4 cores, BaselineImplementation) $\times$ 11 CPU clock frequencies $\times$ 8 data sets $\times$ 2 parallel programming models

4.4.4 Phase 4: Concurrency

In this phase, we went back to the Phase 1 configuration, and added variation in concurrency and programming model. It consisted of 88 tests, of which 11 were studied in Phase 1 (the OpenMP configurations using all 4 cores) and 11 were studied in Phase 3 (the MPI configurations using all 4 cores).

Configuration: (CPU1, Enzo-10M, BaselineImplementation) $\times$ 11 CPU clock frequencies $\times$ 4 concurrency levels $\times$ 2 parallel programming models

4.4.5 Phase 5: Algorithm Implementation

In this phase, we studied variation in algorithm implementation. Since the GeneralImplementation was only available with the MPI parallel programming model, we could not go back to Phase 1. Instead, we compared 11 new tests with 11 tests first performed in Phase 3.

Configuration: (CPU1, 4 cores, Enzo-10M, MPI) $\times$ 11 CPU clock frequencies $\times$ 2 algorithm implementations

4.4.6 Phase 6: Hardware Architecture

With this test, we went to a new hardware architecture, CPU2. We kept many factors constant — BaselineImplementation, Enzo-10M, 24 cores — and varied CPU clock frequency and parallel programming model. All 14 tests for this phase were new.

Configuration: (CPU2, 24 cores, Enzo-10M, BaselineImplementation) $\times$ 7 CPU clock frequencies $\times$ 2 parallel programming models

5 Results

In this section, we describe results from the six phases detailed in Section 4.4. Before doing so, we consider an abstract case, as the analysis of this abstract case is common to the analysis of each phase. We also define terms that we use throughout this section.

Assume a visualization algorithm, when running at the default clock frequency of F_D, takes time T_D seconds to run, consumes a total energy of E_D Joules, and requires an average power of P_D Watts (with $P_D = E_D/T_D$). Further assume that same visualization algorithm, when reducing the clock frequency to F_R, takes T_R seconds, consumes a total of E_R Joules, and requires an average of P_R Watts (once again with $P_R = E_R/T_R$). We then define the following terms:

- $F_{rat} = F_D/F_R$. This is the ratio of the clock frequencies. If the clock frequency was slowed down by a factor of two, then $F_{rat} = 2$.
- $T_{rat} = T_R/T_D$. This is the ratio of elapsed time. If the algorithm runs twice as slow, then $T_{rat} = 2$.
- $E_{rat} = E_D/E_R$. This is the ratio of energy consumed. If the energy consumed is reduced by a factor of two, then $E_{rat} = 2$.

- $P_{rat} = P_D/P_R$. This is the ratio of power usage. If the power usage is reduced by a factor of two, then $P_{rat} = 2$.

Note that three of the terms have the value for the default clock frequency in the numerator and the value for the reduced clock frequency in the denominator, but that T_{rat} flips them. This flip simplifies comparison across terms, since it makes all ratios be greater than 1.

We then find these three pairs of terms noteworthy:

- F_{rat} and T_{rat}: When T_{rat} is less than F_{rat}, the data-intensive nature of the visualization algorithm enabled the program to slow down at a rate less than the reduction in clock frequency.

- T_{rat} and E_{rat}: This pair represents a proposition for visualization consumers (*i.e.,* visualization scientists or simulation scientists who use visualization software): "if you are willing to run the visualization (T_{rat}) times slower, then you can use (E_{rat}) times less energy."

- T_{rat} and P_{rat}: This pair represents a related proposition for visualization consumers: "if you are willing to run the visualization (T_{rat}) times slower, then you can use (P_{rat}) times less power when doing so." This power proposition would be useful for those that want to run a computing cluster at a fixed power rate.

5.1 Phase 1: Base Case

Phase 1 fixed all factors except clock frequency, to provide a baseline for future phases. The factors held constant were: BaselineImplementation, OpenMP, 4 cores on CPU1, and the Enzo-10M data set. Table 2 contains the results.

In terms of our three ratios:

- F_{rat} and T_{rat}: At the slowest clock speed (1.6 GHz), F_{rat} was 2.2X, but T_{rat} was 1.84X, meaning that the program was not slowing down proportionally. A purely compute-intensive program that took 1.29s at 3.5 GHz would have taken 2.82s at 1.6 GHz, while our isosurfacing program took 2.40s (*i.e.,* 17% faster).

- T_{rat} and E_{rat}: Energy savings of up to 1.44X can be gained by accepting slowdowns of up to 1.84X. Clock frequencies in between the extremes offer propositions with less energy savings, but also less impact on runtime.

- T_{rat} and P_{rat}: Power savings of up to 2.7X can be gained by accepting slowdowns of up to 1.84X. The power savings are greater than the energy savings since the energy accounts for reduced runtime, while the power only speaks to instantaneous usage. Regardless, such power savings could be useful when running complex systems with a fixed power budget.

5.2 Phase 2: Data Set

Phase 2 extended Phase 1 by varying over data set. Table 3 shows specific results for the REnzo-10M data set (which compares with the Enzo-10M data set in Table 2 of Section 5.1), and Figure 3 shows aggregate results over all data sets.

In terms of our three ratios:

- F_{rat} and T_{rat}: The right sub-figure of Figure 3 shows that the slowdown factor varies over data set. In the worst case, for the Enzo-1M data set, the slowdown factor is at 2.2X — *i.e.,* exactly 3.5 GHz over 1.6 GHz — meaning that it is performing like a computationally-intensive workload. This makes sense, however, since Enzo-1M is our smallest data set, and it has a regular data access pattern.

- T_{rat} and E_{rat}: This tradeoff varies based on data set. The data sets with randomized access patterns (REnzo, RNek) have better propositions, as do large data sets. Also, when comparing Table 3 and Table 2, we can see that the tradeoffs got more favorable with REnzo-10M, with energy savings of 1.7X against slowdowns of 1.4X (where it was 1.44X and 1.84X for Enzo-10M).

- T_{rat} and P_{rat}: Table 3 shows us that the power tradeoff for REnzo-10M is slightly worse than Enzo-10M. We attribute the increase in instantaneous power to increased data intensity (see Table 4).

Table 2: Experiment results for Phase 1, which uses OpenMP and the BaselineImplementation.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
3.5GHz	1X	1.29s	1X	74.3J	1X	57.4W	1X
3.3GHz	1.1X	1.32s	1X	69.4J	1.1X	52.6W	1.1X
3.1GHz	1.1X	1.38s	1.1X	66.7J	1.1X	48.2W	1.2X
2.9GHz	1.2X	1.42s	1.1X	63.4J	1.2X	44.8W	1.3X
2.7GHz	1.3X	1.50s	1.2X	61.5J	1.2X	40.9W	1.4X
2.5GHz	1.4X	1.62s	1.3X	60.9J	1.2X	37.5W	1.6X
2.3GHz	1.5X	1.78s	1.4X	53.7J	1.4X	30.1W	1.9X
2.1GHz	1.7X	1.93s	1.5X	53.8J	1.4X	27.9W	2.1X
2.0GHz	1.8X	1.95s	1.5X	52.1J	1.4X	26.8W	2.2X
1.8GHz	1.9X	2.13s	1.7X	51.1J	1.4X	24.1W	2.4X
1.6GHz	2.2X	2.40s	1.9X	51.4J	1.4X	21.4W	2.7X

Table 3: Experiment results for the REnzo-10M data set in Phase 2, which uses OpenMP and the BaselineImplementation.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
3.5GHz	1X	2.95s	1X	142.7J	1X	48.4W	1X
3.3GHz	1.1X	3.05s	1X	134.8J	1.1X	44.2W	1.1X
3.1GHz	1.1X	3.01s	1X	124.3J	1.1X	41.3W	1.2X
2.9GHz	1.2X	3.33s	1.1X	122.3J	1.2X	36.8W	1.3X
2.7GHz	1.3X	3.23s	1.1X	109.3J	1.3X	33.8W	1.4X
2.5GHz	1.4X	3.22s	1.1X	99.6J	1.4X	30.9W	1.6X
2.3GHz	1.5X	3.48s	1.3X	93.4J	1.5X	26.8W	1.8X
2.1GHz	1.7X	3.49s	1.3X	88.0J	1.6X	25.2W	1.9X
2.0GHz	1.8X	3.79s	1.3X	88.3J	1.6X	23.3W	2.1X
1.8GHz	1.9X	3.79s	1.3X	82.2J	1.7X	21.7W	2.2X
1.6GHz	2.2X	4.19s	1.4X	82.1J	1.7X	19.6W	2.5X

The performance measurements listed in Table 4 help explain the differences between the data sets. Specifically, the L3 miss rate (unsurprisingly) goes up when data sets get larger and their accesses become randomized. This in turn pushes down the number of instructions per cycle (a surrogate for capturing how many stalls are occurring in the pipeline, which is difficult to measure).

Table 4: Performance measurements for the 1.6 GHz experiments from Phase 2. IPC is short for Instructions Per Cycle, and the L3 Miss Rate is the number of L3 cache misses per one million cycles.

Data Set	Time	Cycles	IPC	L3 Miss Rate
Enzo-1M	0.39s	614M	1.42	597
Enzo-10M	2.40s	3.0B	1.89	1027
Enzo-80M	13.2s	18B	2.24	1422
Nek5000	14.3s	20B	1.54	949
REnzo-1M	0.44s	700M	1.17	5420
REnzo-10M	4.2s	6.0B	0.94	10913
REnzo-80M	33.9s	51B	0.78	12543
RNek5000	27.2s	38B	0.81	11593

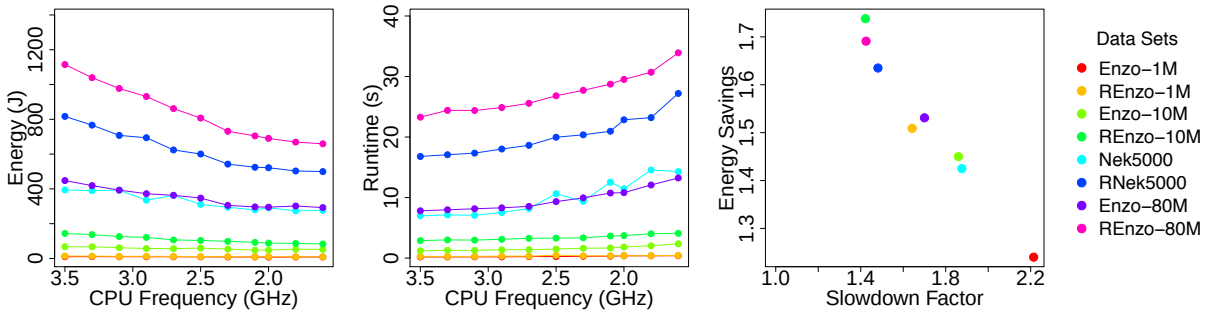

Figure 3: Results from Phase 2, which uses four cores of CPU1 with OpenMP and the BaselineImplementation and varies over data set and clock frequency. The plots are of energy (left) and runtime (middle), as a function of CPU clock frequency. The right figure is a scatter plot of the 1.6GHz slowdown factor versus energy savings for the eight data sets.

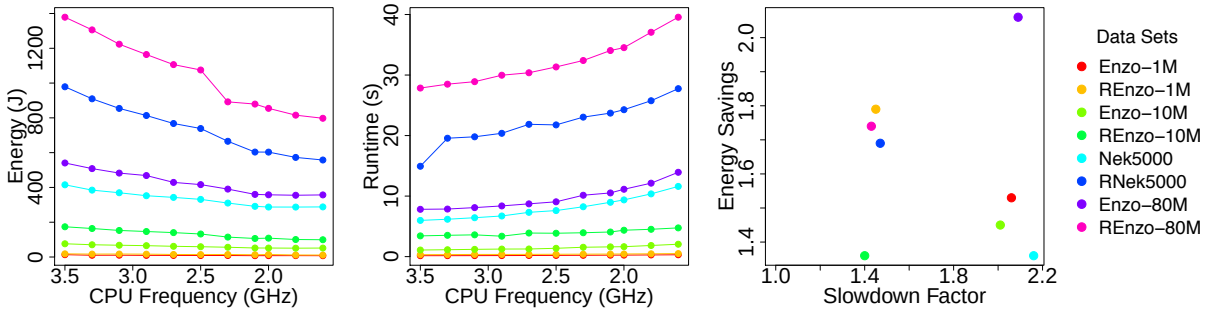

Figure 4: Results from Phase 3, which uses four cores of CPU1 with MPI and the BaselineImplementation and varies over data set and clock frequency. The plots are of energy (left) and runtime (middle), as a function of CPU clock frequency. The right figure is a scatter plot of the 1.6GHz slowdown factor versus energy savings for the eight data sets.

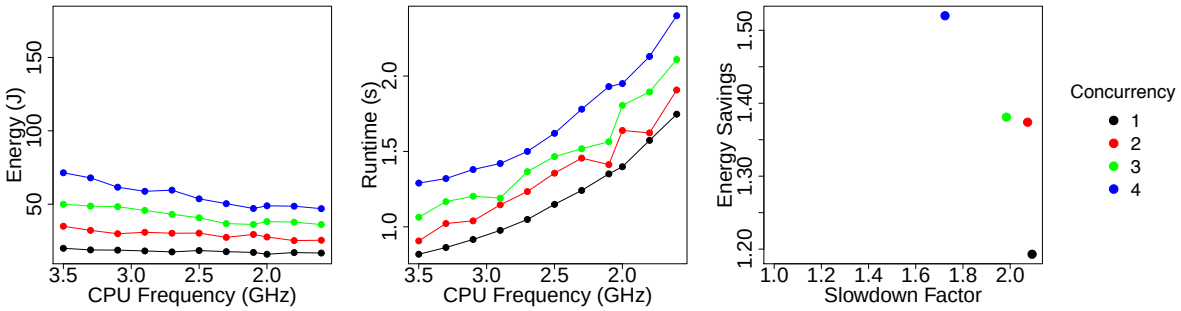

Figure 5: Results from Phase 4's tests with OpenMP, studying Enzo-10M using CPU1 and the BaselineImplementation and varying over clock frequency and concurrency. The plots are of energy (left) and runtime (middle), as a function of CPU clock frequency. The right figure is a scatter plot of the 1.6GHz slowdown factor versus energy savings for the four concurrencies.

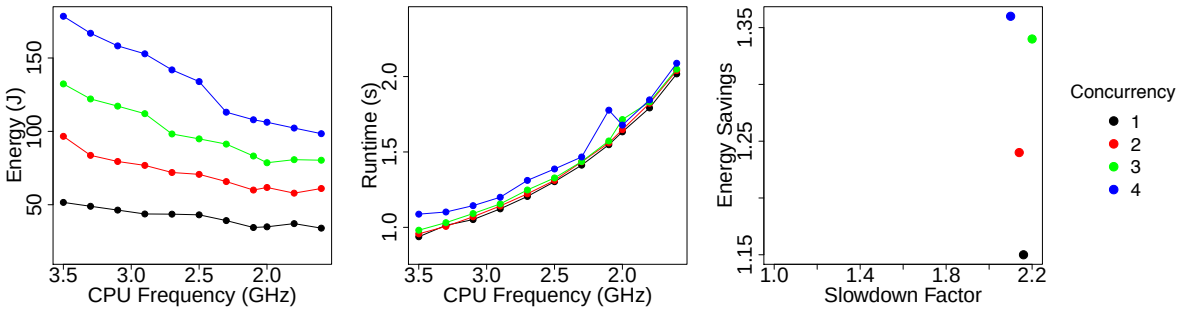

Figure 6: Results from Phase 4's tests with MPI, studying Enzo-10M using CPU1 and the BaselineImplementation and varying over clock frequency and concurrency. The plots are of energy (left) and runtime (middle), as a function of CPU clock frequency. The right figure is a scatter plot of the 1.6GHz slowdown factor versus energy savings for the four concurrencies.

5.3 Phase 3: Parallel Programming Models

Table 5: Experiment results from Phase 3 for the Enzo-10M data set, which uses MPI and the BaselineImplementation.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
3.5GHz	1X	1.08s	1X	74.5J	1X	69.2W	1X
3.3GHz	1.1X	1.12s	1X	70.4J	1.1X	62.7W	1.1X
3.1GHz	1.1X	1.18s	1.1X	67.3J	1.1X	57.0W	1.2X
2.9GHz	1.2X	1.20s	1.1X	66.2J	1.1X	55.0W	1.3X
2.7GHz	1.3X	1.35s	1.3X	63.5J	1.2X	47.1W	1.5X
2.5GHz	1.4X	1.36s	1.3X	59.8J	1.2X	43.9W	1.6X
2.3GHz	1.5X	1.46s	1.4X	55.3J	1.3X	37.8W	1.8X
2.1GHz	1.7X	1.59s	1.4X	51.7J	1.4X	32.6W	2.1X
2.0GHz	1.8X	1.80s	1.7X	55.4J	1.3X	30.8W	2.2X
1.8GHz	1.9X	1.92s	1.7X	52.6J	1.4X	27.4W	2.5X
1.6GHz	2.2X	2.08s	2X	51.8J	1.4X	24.9W	2.8X

Table 6: Experiment results from Phase 3 for the REnzo-10M data set, which uses MPI and the BaselineImplementation.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
3.5GHz	1X	3.46s	1X	179.5J	1X	51.9W	1X
3.3GHz	1.1X	3.48s	1X	166.8J	1.1X	47.9W	1.1X
3.1GHz	1.1X	3.59s	1X	158.9J	1.1X	44.2W	1.2X
2.9GHz	1.2X	3.62s	1X	147.7J	1.2X	40.8W	1.3X
2.7GHz	1.3X	3.78s	1.1X	143.0J	1.3X	37.9W	1.4X
2.5GHz	1.4X	3.88s	1.1X	135.4J	1.3X	34.9W	1.5X
2.3GHz	1.5X	4.00s	1.1X	116.2J	1.5X	29.1W	1.8X
2.1GHz	1.7X	4.18s	1.3X	108.0J	1.7X	25.8W	2X
2.0GHz	1.8X	4.29s	1.3X	109.8J	1.6X	25.6W	2X
1.8GHz	1.9X	4.52s	1.3X	105.0J	1.7X	23.2W	2.2X
1.6GHz	2.2X	4.62s	1.4X	95.5J	1.9X	20.7W	2.5X

Phase 3 extended Phase 2 by varying over parallel programming model. Figure 4 shows the overall results for all eight data sets using MPI; it can be compared with Figure 3 of Section 5.2, which did the same analysis with OpenMP. Tables 5 and 6 show the results using MPI on the Enzo-10M and REnzo-10M data sets, respectively. Table 2 of Section 5.1 and Table 3 of Section 5.2 are also useful for comparison, as they showed the results for these same data sets using OpenMP.

In terms of our three ratios:

- F_{rat} and T_{rat}: The right sub-figure of Figure 4 shows two clusters: one grouping (made up of the randomized data sets) slows down only by a factor of ˜1.4, while the other grouping (made up of the non-randomized data sets) slows down in near proportion with the clock frequency reduction. This contrasts with the OpenMP tests seen in Figure 3, which showed more spread over these two extremes. We conclude that the randomized data sets create significantly more memory activity for MPI than for OpenMP, which is supported by our performance measurements. Taking REnzo-80M as an example, the MPI test had over 48,000 L3 cache misses per million cycles, while the OpenMP test had less than 12,500.

- T_{rat} and E_{rat}: Renzo-10M with MPI gave the largest energy savings of any test we ran, going from 179.5J to 95.5J. That said, its starting point was higher than OpenMP, which went from 142.7J to 82.1J. Overall, energy savings were harder to predict with MPI, but were generally better than the savings with OpenMP (again because it was using more energy to start with).

- T_{rat} and P_{rat}: the MPI tests used more power, but saw greater reduction when dropping the clock frequency. For the Enzo-10M data set, the MPI test dropped from 72.2W (3.5GHz) to 25.3W (1.6GHz), while OpenMP dropped from 57.4W to 21.4W. MPI's increased power usage likely derives from activity with the memory system, and increased L3 cache misses.

Summarizing, our performance measurements show that the MPI approach uses the memory infrastructure less efficiently, leading to increased energy and power usage, but also creating improved propositions for reducing energy and power when reducing clock frequencies.

5.4 Phase 4: Concurrency

Phase 4 did not build on Phase 3, but rather went back to Phase 1 and extended it by considering multiple concurrency levels and programming models. Figure 5 shows plots of our results with OpenMP and Figure 6 shows the results with MPI. Table 7 contains data that complements the figures.

Higher levels of concurrency cause more memory requests to be issued, leading to saturation of the memory infrastructure. As a result, runtimes steadily increase. Because some processes may have to wait for the memory infrastructure to satisfy requests, we observed energy savings by slowing down the clock frequency, making waiting processes consume less power and having their memory requests satisfied more quickly (relative to the number of cycles, since cycles lasted longer). Table 7 shows this trend, in particular that the four core configurations overwhelm their memory (as seen in the increase in L3 cache misses and in the reduction in instructions per cycle), while the one core configurations fare better.

In terms of our three ratios:

- F_{rat} and T_{rat}: The right sub-figures of Figures 5 and 6 show that the slowdown factor varies over concurrency. Lower concurrencies (which have faster runtimes) have higher slowdowns, because their memory accesses are being supported by the caching system. Higher concurrencies (which have longer runtimes) have lower slowdowns, because the cache was not keeping up as well at high clock frequencies (since more processors were issuing competing requests).

- T_{rat} and E_{rat}: The tradeoff between slowdown and energy varies quite a bit over concurrency. With OpenMP, a single core suffers over a 2X slowdown to receive a 1.2X energy savings. But, with four cores, the slowdown improves to 1.86X and the energy savings improve to 1.45X. With MPI, the trends are similar, but less pronounced.

- T_{rat} and P_{rat}: As seen in Table 7, the power savings get better as more cores are used, but not dramatically so. With one core, both OpenMP and MPI provide 2.5X power improvements by dropping the clock frequency. Going up to four cores raises this power improvement to 2.85 (MPI) and 2.68 (OpenMP).

5.5 Phase 5: Algorithm Implementation

Phase 5 once again went back to Phase 1 as a starting point, this time extending the experiments to consider multiple algorithm implementations. The factors held constant were: OpenMP, 4 cores on CPU1, and the Enzo-10M data set. Table 8 contains the results.

With GeneralImplementation, runtime and clock frequency are highly correlated, *i.e.,* reducing the clock frequency by 2.2 causes the workload to take 2.1X longer to run. This relationship between frequency and runtime is characteristic of a compute-intensive workload, depicted by our *computeBound* micro-benchmark. In

Table 7: Experiment results from Phase 4. IPC is short for Instructions Per Cycle, and L3 is the number of L3 cache misses per one million cycles.

Configuration	Time	Energy	Power	IPC	L3 Miss Rate
MPI/1/3.5 GHz	0.90s	24.4J	26.9W	2.37	5652
MPI/1/1.6 GHz	2.0s	21.1J	10.8W	2.41	3788
OpenMP/1/3.5 GHz	0.83s	19.9J	23.9W	2.06	1697
OpenMP/1/1.6 GHz	1.74s	16.7J	9.6W	2.11	931
MPI/4/3.5 GHz	0.96s	69.5J	72.2W	2.07	10476
MPI/4/1.6 GHz	2.02s	51.2J	25.3W	2.37	3456
OpenMP/4/3.5 GHz	1.29s	74.3J	57.4W	1.51	3351
OpenMP/4/1.6 GHz	2.40s	51.1J	21.4W	1.89	1027

Table 8: Experiment results for GeneralImplementation of Phase 5. These results compare with BaselineImplementation, whose corresponding results are in Table 2.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
3.5GHz	1X	16.06s	1X	1056J	1X	65.8W	1X
3.3GHz	1.1X	16.57s	1X	992J	1.1X	59.9W	1.1X
3.1GHz	1.1X	17.64s	1.1X	950J	1.1X	53.9W	1.2X
2.9GHz	1.2X	19.00s	1.3X	928J	1.1X	48.8W	1.3X
2.7GHz	1.3X	20.85s	1.3X	914J	1.2X	43.9W	1.5X
2.5GHz	1.4X	21.82s	1.4X	876J	1.2X	40.1W	1.6X
2.3GHz	1.5X	24.01s	1.4X	784J	1.3X	32.7W	2X
2.1GHz	1.7X	26.09s	1.7X	763J	1.4X	29.3W	2.2X
2.0GHz	1.8X	27.43s	1.7X	768J	1.4X	28.0W	2.4X
1.8GHz	1.9X	30.67s	1.9X	764J	1.4X	24.9W	2.6X
1.6GHz	2.2X	34.17s	2.1X	756J	1.4X	22.1W	3X

contrast, the BaselineImplementation exhibited behavior closer to data-intensive in our previous phases.

The explanation for the difference between the two implementations is in the number of instructions. While both issue the same number of loads and stores, the GeneralImplementation issues 102 billion instructions, while the BaselineImplementation issues only 7 billion. These additional instructions change the nature of the computation (from somewhat data-intensive to almost entirely compute intensive), as well as making the overall runtimes and energy consumption much higher. Of course, these instructions add value for general toolkits, in terms of supporting more data models and algorithms. The takeaway from this study is that the approach from general toolkits appears to tilt the instruction mix (at least for isosurfacing).

Interestingly, the E_{rat} and P_{rat} ratios are still favorable, at 1.4X and 3X, respectively. This is because the relationship between clock frequency and energy consumed is not strictly linear. As a result, even compute-intensive workloads can benefit from clock frequency reductions, although their T_{rat}'s will still match the clock frequency reduction.

5.6 Phase 6: Architecture

Phase 6 did not build on any previous phases. Instead, it explored CPU2, whose results do not translate to any of the previous CPU1 experiments. The factors held constant were: MPI, 24 cores on CPU2, Enzo-10M, and the BaselineImplementation. Table 9 contains the results.

CPU2 is significantly different than CPU1 in that it contains Ivy Bridge processors, while CPU1 contains Haswell processors. On Haswells, the core (compute units, private L1 and L2 caches) and uncore (shared L3 cache, main memory) are on separate clock domains, so slowing down the frequency only applies to the speed of the executing instructions and accessing L1 and L2 caches. On Ivy Bridge, core and uncore share the same clock frequency, and so data-intensive workloads cannot benefit with respect to T_{rat}.

Table 9 shows that, while P_{rat} is better at lower clock frequencies, E_{rat} is worse. Restated, while the power dropped, its drop was not steep enough to offset the increases in runtime, and so overall energy usage goes up. This does not match the results in Phase 5, where a compute-bound workload created a similar "T_{rat} equals F_{rat}" situation. As explanation, we again note the non-linear relationship between power and clock frequency (which varies over architecture).

Table 9: Experiment results from Phase 6, which uses CPU2 with MPI and the BaselineImplementation.

F	F_{rat}	T	T_{rat}	E	E_{rat}	P	P_{rat}
2.4	1X	2.265	1X	549	1X	242.4	1X
2.2	1.1X	2.479	1.1X	558	1X	225	1.1X
2.0	1.2X	2.695	1.2X	571	1X	211.9	1.1X
1.8	1.3X	3.024	1.3X	573	1X	189.5	1.3X
1.6	1.5X	3.385	1.5X	631	0.9X	186.4	1.3X
1.4	1.7X	3.836	1.7X	668	0.8X	174.1	1.4X
1.2	2X	4.466	2X	697	0.8X	156	1.6X

6 Conclusion and Future Work

We conducted a study exploring the tradeoffs between power and performance when reducing clock frequencies. We summarize the results of our findings by phase:

- Phase 1 confirmed our basic hypotheses about reducing clock frequencies: (i) Isosurfacing is sufficiently data-intensive to slow the impact from reduced clock frequencies. (ii) Clock frequency reductions can create options for visualization consumers to choose between finishing an algorithm quickly using more energy, or slowly using less energy. (iii) Clock frequency reductions decrease power usage, creating options for visualization consumers wanting to balance system-wide power usage.

- Phase 2 showed that the tradeoffs between energy and runtime get increasingly favorable as data complexity goes up (either due to size of increased irregularity in data access).

- Phase 3 showed that MPI's less coordinated memory accesses affect energy and power tradeoffs compared to OpenMP.

- Phase 4 showed that the tradeoffs between execution time and energy are most favorable when the memory infrastructure is being stressed, and that this scenario exists at higher concurrencies (or, alternatively, is less likely to exist when some of a node's cores are not being used).

- Phase 5 showed that general-purpose implementations of visualization algorithms shift the instruction mix such that the tradeoffs between execution time and energy are less favorable.

- Phase 6 showed the importance of having an architecture where the memory infrastructure can be controlled separately from the CPU.

In terms of future work, we would like to explore additional visualization algorithms. While we feel isosurfacing is representative of many visualization algorithms — *i.e.,* those characterized by iterating on cells one-by-one and producing a new output — other algorithms have different properties. In particular, particle advection problems perform data-dependent memory accesses, which may produce even more favorable propositions for energy and power savings. Further, algorithms like volume rendering require both significant computation and irregular memory accesses (especially for unstructured grids), making it unclear how it would be affected by changes in clock frequency.

ACKNOWLEDGEMENTS

The authors thank Laura Carrington and Ananta Tiwari of San Diego Supercomputing Center, who provided excellent advice and expertise. This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, Program Manager Lucy Nowell. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, Feb. 1995.

[2] S. Ahern et al. Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization, July 2011.

[3] J. Ahrens, B. Geveci, and C. Law. Visualization in the paraview framework. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, pages 162–170, 2005.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[5] H. Childs et al. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 357–372. Oct. 2012.

[6] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *Visualization and Computer Graphics, IEEE Transactions on*, 3(2):158–170, 1997.

[7] J. Dongarra, P. Beckman, et al. The International Exascale Software Roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.

[8] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[9] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *J. Parallel Distrib. Comput.*, 68(9):1175–1185, Sept. 2008.

[10] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM.

[11] M. Gamell et al. Exploring power behaviors and trade-offs of in-situ data analytics. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 77:1–77:12, New York, NY, USA, 2013. ACM.

[12] R. Ge, X. Feng, and K. W. Cameron. Improvement of power-performance efficiency for high-end computing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12*, IPDPS '05, pages 233.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[14] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller. Power efficiency for software algorithms running on graphics processors. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 67–75. Eurographics Association, 2012.

[15] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 79–90, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, July 1987.

[17] S. Martin and M. Kappel. Cray xc30 power monitoring and management. *Proceedings of CUG*, 2014.

[18] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[19] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints*, Mar. 2004.

[20] A. Porterfield, R. Fowler, S. Bhalachandra, and W. Wang. Openmp and mpi application energy measurement variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[21] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 93–ff. IEEE Computer Society Press, 1996.

[22] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference: The MPI Core, 2nd edition*. MIT Press, Cambridge, MA, USA, 1998.

[23] C. Upson et al. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.