Understanding the Performance of HPC Applications

Brian Gravelle

March 2019

Abstract

High performance computing is an important asset to scientific research, enabling the study of phenomena such as nuclear physics or climate change, that are difficult or impossible to be studied in traditional experiments or allowing researchers to utilize large amounts of data from experiments such as the Large Hadron Collider. No matter the use of HPC, the need for performance is always present; however, the fast-changing nature of computer systems means that software must be continually updated to run efficiently on the newest machines. In this paper, we discuss methods and tools used to understand the performance of an application running on HPC systems and how this understanding can translate into improved performance. We primarily focus on node-level issues, but also mention some of the basic issues involved with multi-node analysis as well.

1 Introduction

In the modern world, supercomputing is used to approach many of the most important questions in science and technology. Computer simulations, machine learning programs, and other applications can all take advantage of the massively parallel machines known as supercomputers; however, care must be taken to use these machines effectively. With the advent of multi- and many- core processors, GPGPUs, deep cache structures, and FPGA accelerators, getting the optimal performance out of such machines becomes increasingly difficult. Although many modern supercomputers are theoretically capable of achieving tens or hundreds of petaflops, applications often only reach 15-20% of that peak performance.

In this paper, we review recent research surrounding how users can understand and improve the performance of HPC applications. Starting with a knowledge of the architecture of computer systems and theory behind parallel algorithms is vital for developers trying to improve the performance of their applications. Deep understanding of both of these topics will allow the developers to determine what programming techniques and hardware functionalities are useful to their situation. By carefully examining the implementation of their application and how it uses the hardware, a developer can make improvements to the performance. Understanding how an application uses the hardware can be achieved through a variety of methods as simple as timing different sections of the program or as complex as using performance analysis tools to collect detailed counts of hardware events associated with the application. From this information, the developer can get a sense of places the application can make better use of the system and then use knowledge of the algorithms and hardware to propose performance improvements for the application. These potential improvements can then be verified through the same methods of collecting data on the application.

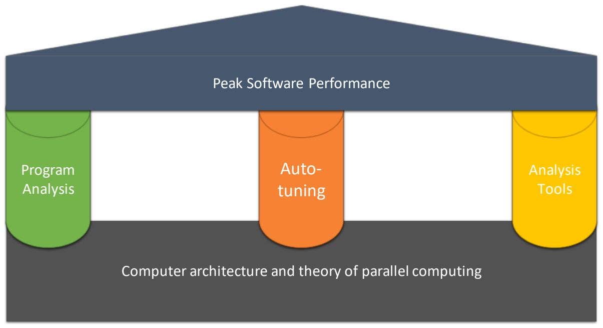

Peak Software Performance
Program Analysis
Auto-tuning
Analysis Tools
Computer architecture and theory of parallel computing

Figure 1: *Our view of the performance analysis area. The whole building rests on a foundation of knowledge concerning architecture and theory (Section 2), while pillars of Program Analysis (Section 3), Analysis Tools (Section 4), and Autotuning (Section 5) support the roof of peak performance.*

We view performance analysis largely as an iterative process of studying an application, forming a hypothesis of how to improve it, then testing the hypothesis and checking the result. This method has been used for decades however, there is significant room for improvement. Numerous techniques exist to inform developers about which areas of the application use the most time, make inefficient use of cache, don't vectorize the computation, or have a variety of other features indicative of potential improvements. Unfortunately, developers are on their own to establish what tests to run, how to connect spots of bad performance to the larger algoritmic issues at the root, and what improvements to try once problem areas are found. The connection between analysis and optimization is held together through the ad-hoc knowledge of HPC developers. Data collection attributes metrics to ultra-specific areas of the application and must be interpreted by the developers without indication of which issues are most important. Autotuning exists in a vacuum in which arbitrary optimizations are applied without considering the hardware or algorithmic issues. The deficiencies in modern performance analysis make it seem more like a cairn[1] than the Greek temple we pictured (Figure 1). Skilled artists can construct beautiful impossibly balanced Cairns, but most people suffer many collapses and must hold the rocks in place lest they fall again. Performance analysis of HPC is in need of standard practices and workflows that enable consistent and reproducible root-cause analysis rather than the existing bespoke efforts of a limited number of experts.

Guided by our diagram of performance analysis (Figure 1), we begin our paper by laying out foundational information about HPC systems and parallel algorithms for the reader (Section 2). Building upon this foundation allows the developer to reason about the performance of their application. This reasoning can then be combined with source code modeling and by-hand analysis to make predictions about how the application can be improved (Section 3). These predictions can be tested with simple timers or more in-depth analysis. Such analysis often requires tools (Section 4) to gather in-depth information about how the system is being used. Often this process of analyzing, improving, and re-analyzing an application can be tedious work with similar optimizations used in many applications. Tedious work often calls for automation, so autotuners (Section 5) are often used to automatically test and apply optimizations to kernels or full applications. In this paper, we review all these areas of performance analysis with a particular focus on node-level performance.

1.1 Example Problem

Throughout this paper, we use a single common example problem to discuss the methods, tools, and issues of performance analysis. This application is a miniature stencil problem loosely based on solving second order PDEs, a common computational problem in HPC. Multiple versions of the code and inputs are available on Github[2].

Our stencil application is written in C, and based on a regular rectangular mesh as depicted in Figure 2. Each cell holds four values named *avg*, *sum*, *pde*, and *dep*. Two, *sum* and *avg* are very simple calculations. At each timestep, the *avg* value of each point is updated to be the average of the neighbor cells and itself while *sum* is a summantion of a fraction of each of the neighbor cells. These are shown in Algorithm 1. For additional complexity, we use the other two variables, *pde* and *dep*. Control flow for *pde* is roughly the same and for the others, but significantly more complex computation is performed based more closely on actual second order PDEs. See Algorithm 2 for details. The *dep* variable depends on *avg* and *sum* for

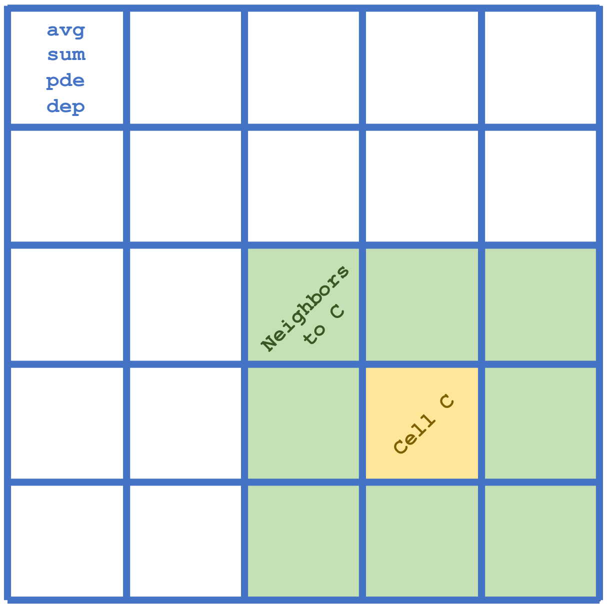


Figure 2: *The simple stencil used as an example throughput this paper.*

[1]Human-made stack of stones; `https://en.wikipedia.org/wiki/Cairn`

[2]`https://github.com/mastino/stencil_ex`

its computation instread of constants, but is otherwise the same as *pde*. These four variables and the stencil pattern provide a range of performance challenges to explore throughout our survey.

```
for v in mesh vertices do
    v.volume = v.volume / count(neighbors);
    for n in neighbors do
        v.avg += n.heat;
        v.sum += n.sum / count(neighbors);
    end
    v.heat /= count(neighbors);
end
```

Algorithm 1: *avg* and *sum* updates for the stencil example

```
C = 0.25; // arbitrary constant
dt = 1.0; // change in each timestep
for v in mesh vertices do
    v.pde = -2*dt*dt * v.pde * C;
    for n in neighbors do
        dist_squared = get_distance_squared(n, v);
        v.pde = (-2*dt2 * n.pde) / ((dist2 + 1.0) * C);
    end
end
```

Algorithm 2: *pde* updates for the stencil example

```
C = 0.25; // arbitrary constant
dt = 1.0; // change in each timestep
for v in mesh vertices do
    v.dep = -2*dt*dt * v.dep * C;
    for n in neighbors do
        dist_squared = get_distance_squared(n, v);
        v.dep = (-1*n.avg*dt2 * n.dep) / ((dist2 + n.sum) * C);
    end
end
```

Algorithm 3: *dep* updates for the stencil example.

To study node-level issues of modern systems, we parallelize the application using openMP threads. This threading will allow us to explore important issues such as scaling and contention for data. Additionally, we implement both Array of Structures and Structures of Array storage of the data. In AoS, a C struct is defined with a single array to describe the mesh. Each array index corresponds to a struct with all four variables. SoA is the opposite; there is one struct with four arrays. Depending on the algorithms and architectures in use, these data arrangements have significant performance impacts.

2 Background

Understanding performance analysis and optimization relies foundationally on understanding the architectures and theory used in HPC applications. In this section, we discuss how major elements of modern computer architectures and parallel computing influence the performance of an application. We pay particular attention to node-level features and the areas that require the most amount of effort to achieve optimal performance.

Looking to resources such as the books by Hennessy and Patterson [85, 55] can help developers get a fundamental understanding of the systems they use. Such books will discuss both the complexities of modern architectures and simpler models that can be used to understand them. Here we will limit the discussion to the most important models from these classic texts that describe modern parallel computers and how developers can use those models to improve their applications.

2.1 Understanding Node-Level Performance

Current computing systems consist largely of two pieces: the memory and the processor. The memory is where data is stored while the computation is going on, and the processor is the part that actually does mathematics. Computers operate in a cycle of reading data (memory), changing data (processor), and writing data (memory). We use memory and processor architectures as the fundamental categories when discussing node-level architecture.

2.1.1 Memory Architecture

Since the 1980s the performance of both memory and processors have been improving dramatically; however, the performance of processors has far outpaced that of memory (Figure 3). This gap has increased the effect of memory latency on performance, hardware architects can mitigate the latency by inserting caches between main memory and the processor. Modern systems usually have two or three levels of cache with the largest, slowest levels shared between several or all cores in a multiprocessor and the smallest, fastest local to only one or two. Reuse of data in the caches combined with software or hardware prefetching into the cache allows the applications to minimize the effects of memory latency.

When data is requested, those bytes, along with nearby ones are brought into the smallest and fastest cache. The cache benefits come from *data locality.* Data adjacent to each other are brought into memory together, even if only one element is needed, so applications that use data close together benefit from *spatial locality.* Since data is saved in the cache after it is used, applications can benefit from *temporal locality* by reusing data befor eit is evicted. As the cache is filled, which data are saved and which are evicted depends on the size, the associativity (how data is placed in the cache), and the eviction protocol (which line gets removed next). In conjunction with tracking the large memory space of an application, keeping track of exactly what data is and isn't in cache is a difficult problem. Thankfully the Three C's model of caching, [57], exists to help developers figure out when important data may be missing from the cache. This model presents cache misses as occurring for three reasons. First, there are Compulsory misses caused when a new section of memory is reached in your application. Prefetching can mitigate the effect of these, but they cannot be avoided since the data is simply not in the cache yet. The second C, capacity, describes when useful data was evicted because the cache ran out of space. When iterating over a memory area that is significantly larger than the cache size, capacity misses may become a

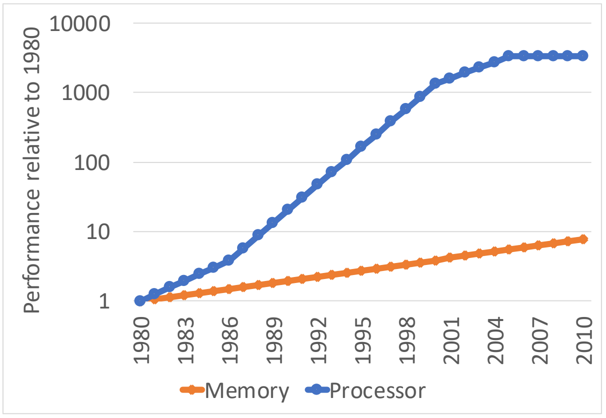


Figure 3: *The widening gap in memory and processor (per core) performance. Based on data from [55].*

problem and can be avoided through blocking (breaking the working set into cache sized bites). Lastly, Conflict misses occur when multiple chunks of data are mapped to the same line in the cache. These occur when the application alternates between sections of memory that are fairly far apart. With modern many-way associative caches, these are less of an issue and largely indistinguishable capacity misses.

In general, memory optimization is not an issue of using less memory as one may expect, but using memory effectively. To do this the developer must arrange the data structures and computation to fit nicely into the caches, have predictable access patterns and be reused frequently before being evicted from the cache.

2.1.2 Processor Architecture

Once the memory subsystem has brought the data into the processor, it is time to perform the computation. When considering the performance of a processor, the first question people ask is "How fast can it run?" The most straightforward way to get a number is to look at the documentation; count cores, clock speed, and FLOPS[3] per cycle; and do a little multiplication as in Equation 1. Until recently this method would have given you a fairly accurate approximation of the peak performance for your system. Modern processors have multiple types of floating point and vector instructions, frequency scaling, and other complications that make the process more difficult but still possible. In [36], the authors go through the process of analyzing a system for peak FLOPS/s considering multiple types of vector instructions and how the frequency changes with the number of cores in use. Benchmarking (Section 2.3) is often better suited to this arduous task.

$$Peak\ FLOPS/s = cores * clockrate * (FLOPS\ per\ cycle) \tag{1}$$

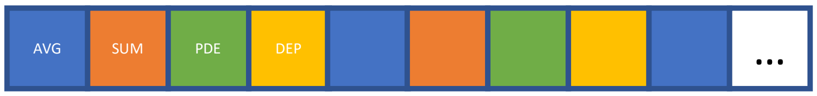
That being said, the peak performance isn't necessarily the most important thing to know when optimizing an application for a new system. The key is understanding how the hardware computes on data and how the details of an application impact potential performance. Processors perform computation in pipelined steps: loading an instruction, getting data, performing computations, and writing results. As the one instruction finishes a step the next one starts that step and so on so that several instructions are being worked on ("in-flight") at a time. These pipelines can be fairly deep, up to 20 steps in some cases and some steps (especially complex floating point operations such as division) may take multiple cycles. For the multi-cycle instructions, there are often multiple units so that two can be "in-flight" at a time. On XEON processors this often means that there are two vector floating point units per core.

The pipeline and varied latency of the instructions impacts how computationally intense code should be designed. A pipeline is most efficient if it can be fully filled at all times, meaning that the computer knows exactly what instructions are coming next and can put them in the proper order. Branch statements that redirect the program can force a compiler to insert no-ops or cause the pipeline to be flushed to avoid executing incorrect code. Understanding this problem, programmers can avoid putting *if* or *break* statements in major computational loops to keep the control flow predictable. Similarly, an algorithm can be redesigned without division or trigonometry to prevent long-latency instructions from interfering with the pipeline. Also, memory requests that can't be filled from L1 often force the pipeline to pause for several cycles as the lower caches are accessed. Reordering of the instructions in the CPU and careful planning of data structures can avoid this memory latency. Finally, making sure there are enough operations or threads to use both floating point units can help avoid under-utilizing the the hardware features.

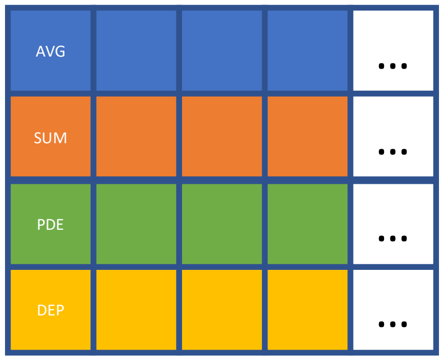
Vector instructions offer a way to improve the performance of computationally intense applications. When a developer or compiler knows that an identical series of operations are going to occur for adjacent memory elements, they can use vector instructions to have those operations occur simultaneously. These instructions act like standard floating point operations but on several operands simultaneously. However, they are most effective if the operands are adjacent in memory and aligned with cache lines to be able to be loaded into the vector registers. Such restrictions can be difficult to implement, but extremely helpful, on data that is structured as a sparse array or spread out across memory. For our stencil example, if the data is organized as an Array of Structures (Figure 4(a)) then the elements of one variable must be gathered into the vector registers before the operation is performed. Using the structure of arrays format (Figure 4(b)) allows the

[3]floating point operations

elements to be loaded directly into the registers. In other cases, AoS maybe be more efficient, so it can be worth the effort to try both implementations.



(a) AoS memory layout.



(b) SoA memory layout.

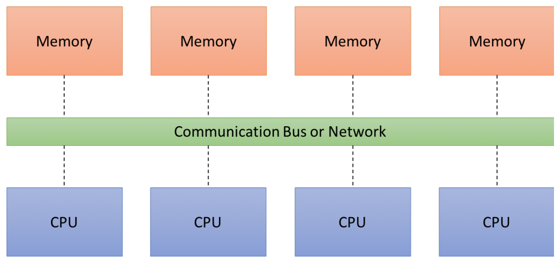
Figure 4: *Arrangement of memory for our stencil example*

Similar to vector instructions are Graphics Processing Units. GPUs were originally developed to accelerate graphics processing but were soon discovered to be useful for more problems, particularly in scientific computing. GPUs are off-core accelerators that work in conjunction with a CPU to solve a computational problem. Developers looking to use GPUs should carefully organize the memory and avoid branch statements as they do for vector instructions in CPUs. Also, it is important to consider the time it takes for data to move into GPU memory. The speedup of using a GPU compared to a CPU needs to be enough to offset the significant overhead of transferring data. The general purpose use of GPUs has had an enormous impact on scientific computing in recent years; we discuss several examples in Section 3.

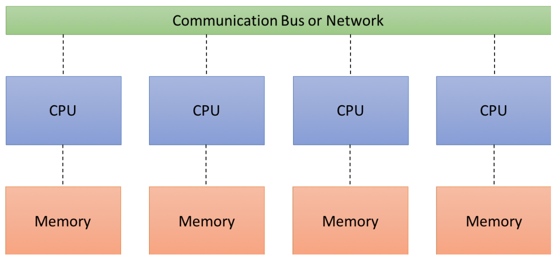
Understanding the complexities of modern architecture is vital to be able to effectively tune code to a system, but understanding the system or a model of it is only half the challenge. One must also be able to study the application and check how it is using the architecture and locate areas that should be targeted for improvement. Methods for this analysis are discussed in Section 3 and empirical analysis tools are handled in Section 4. If the developer can connect the high-level issues from analyzing the algorithm and program with details about the hardware and data from the empirical analysis, then they will have a good chance of finding ways to improve the application.

2.2 Parallel Hardware and Computation

For parallel systems, there are two basic ways that the processors and memory are organized: shared memory (Fig. 5(a)) vs distributed memory (Fig. 5(b)). The differences between these two designs inform many of the choices in application design and parallel decomposition that developers must make. In shared memory parallel computers, all of the processors access the same memory space, so all of the data is visible to all of the processing units. When using a shared memory programming scheme, the programmer can assume that all of the data is accessible to all of the threads of computation. This assumption makes programming much easier but requires the programmer to avoid having multiple threads write to the same part of memory at one time. While avoiding contention in the application it is also necessary to ensure that the application isn't accidentally serialized by overuse of locking. Another consideration when sharing data is that the caches (especially L1 and L2) of shared memory machines are usually private, meaning that although the hardware can provide access to all the data, doing so may impede effective use of the caches that mitigate the high cost of memory access.



(a) A diagram of a shared memory system



(b) A diagram of a distributed memory system

Figure 5: *General types of parallel computing architectures*

In distributed memory systems, processors or nodes do not share an address space. Each one maintains separate memory and messages are used to communicate between the different processes. This method is dominant in large HPC clusters where shared memory nodes are networked into a larger distributed memory cluster. Communication and moving data between the processes of a distributed memory program takes more overhead and is more challenging to program than a shared memory system, but is necessary for large scale applications. The main concern in distributed memory programming is the communication overhead, so programmers must be careful to minimize the amount of communication and overlap it with computation to ensure maximum performance.

Both shared and distributed memory systems allow multiple computations to occur concurrently and independently except for where the programmer want synchronization to occur. This type of computation is referred to as MIMD: Multiple Instruction, Multiple Data. This acronym and the others (SISD, SIMD, and MISD) are part of Flynn's Taxonomy [43] which is pictured in Figure 6.

Alternatively, SIMD style architectures (Single Instruction Multiple Data) allow the programmer or compiler to specify that a single set of operations should be applied to several pieces of data at once. Originally, this method was pursued in the vector processors that were used in early supercomputers and has since has found a home in GPUs and vector instructions integrated into standard processors. GPUs take this method to the extreme, with large vector lengths and complex architectures for handling dependencies, conditionals, and other complications to vector processing. Finding areas to exploit these architectures can greatly improve the performance of an application as it allows the processor to complete numerous computations simultaneously.

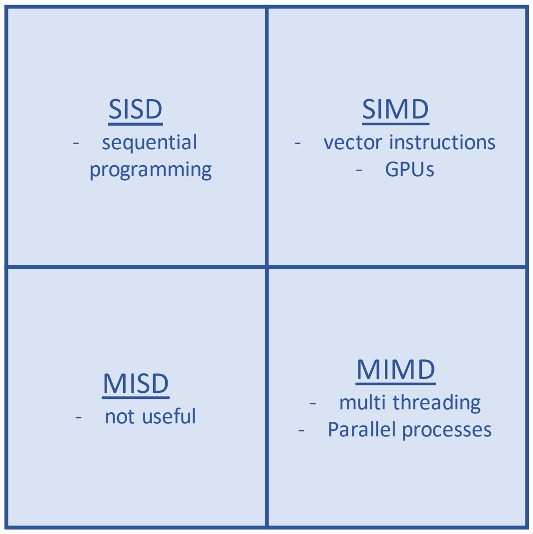
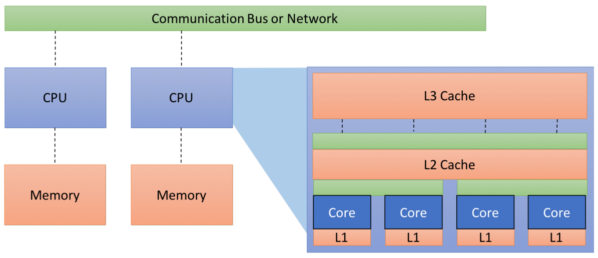

SISD
- sequential programming
SIMD
- vector instructions
- GPUs
MISD
- not useful
MIMD
- multi threading
- Parallel processes

Figure 6: *A table of Flynn's taxonomy of parallel computing.*

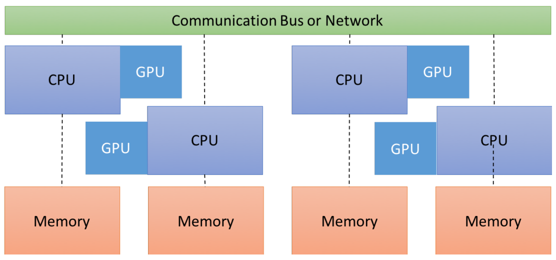
Modern clusters combine all of these designs allowing them to scale beyond the limitations of shared memory while still exploiting it for low-level computation. Figure 7(a) shows an example of a distributed cluster of shared memory nodes and Figure 7(b) shows a similar cluster with GPUs attached to each node. Recent supercomputers such as Summit[4] and Sierra[5] have 6 and 4 GPUs on each node respectively. Heterogeneous computing of this sort complicates programming since data must be transferred between the CPU and GPU memory systems and different programming models are often used for the different processors.

When using clusters with multiple layers of parallelism the developer must carefully plan how to best use each level. Large pieces of the computation can be decomposed across the distributed processes while nested parallelism is spread across the shared memory threads in each process. Finally, the low-level matrix and

[4]https://www.olcf.ornl.gov/summit/
[5]https://computation.llnl.gov/computers/sierra



(a) A cluster of shared memory multi-core processors



(b) A heterogeneous cluster of CPUs and GPUs

Figure 7: *Distributed computer system arrangements*

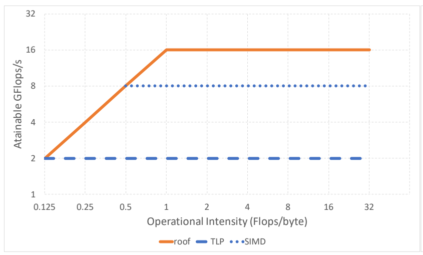
vector operations can be accelerated with the vector architectures provided the data structures are organized to efficiently load those registers. This coordination between several levels of parallel computation is the primary source of modern supercomputing performance.

2.3 Benchmarks

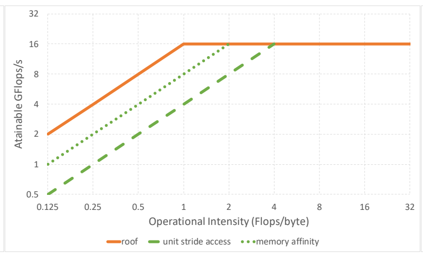
One method of better understanding an architecture is to use benchmarks to explore the constraints of one particular aspect of a system. Benchmarks are small applications or application kernels that are designed to stress a particular aspect of the hardware or be representative of a type of application. They are used by hardware designers to help guide improvements, application users to choose the best systems for their workloads, and application developers to understand the most effective ways to use a particular piece of hardware, and tool experts to validate measurements.

Some benchmarks stress particular features to help understand the potential of a processor. For example the STREAM benchmark [75] is widely used to obtain realistic memory bandwidth estimates for a system. Similarly, the WBTK micro-benchmarks [62] can help predict memory performance but additionally provides guidance about types of access patterns that can be troublesome in a particular platform. Other benchmarks focus on different hardware aspects such as LINPACK [37] which performs dense linear algebra operations that stress the floating point units of the system, the NAS benchmarks [7] which were an early study of parallel architectures, and Rodina [21] which is intended for heterogeneous computers.

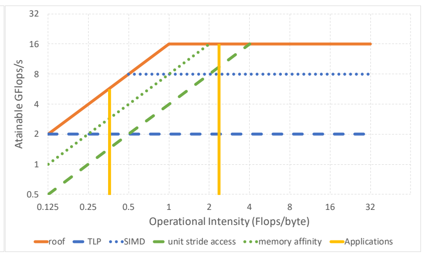
The Roofline benchmark [110, 35, 98, 74] takes benchmarking a step further. It calculates a maximum computation rate based on the memory system and the processing speed and allows an application developer to use this information to guide performance optimization. The Roofline model was first presented as a method for comparing an application's performance to the potential of the system and suggesting optimizations that would best benefit it. Algorithms are defined using a metric called Operational Intensity with units of FLOPS per Byte of data moved. This metric indicates how many Floating point operations can occur for each byte of memory accessed. Computationally intensive algorithms have a high Operational Intensity and are compute bound, while memory intensive applications have a lower Operational Intensity.



(a) Computational Ceilings.



(b) Bandwidth Ceilings.



(c) Full roofline with example applications in memory bound and compuation bound regions.

Figure 8: *Example of an architectural roofline.*

Figure **??** is an example roofline pulled from the original paper [110]. The angled lines on the left of the plot indicate cache and memory performance, while the horizontal lines indicate processing speed. The top-most lines indicate absolute peak performance while the lower ones indicate optimizations (i.e. cache-blocking, use of vector instructions, etc) that will allow the application to move into that section. An algorithm has a particular arithmetic intensity and is therefore marked as a vertical line along which implementations can move depending on the performance. Changes to the algorithm will move that vertical line along the x-axis, impacting possible peak performance. In this example, the developer for Kernel 1 should focus on memory optimizations and enabling thread level parallelism but should ignore SIMD optimizations. Kernels 2 need threading and SIMD instructions and probably won't benefit from memory optimizations.

Often benchmarks are small pieces of code that are intended to focus on a particular aspect of a machine. If a developer is looking for a broader analysis of a system, then they may choose to use a mini-application instead. Mini-applications (such as those available from the Mantevo Project [56] or LLNL[6]) are small applications that are taken from or representative of particular full applications. These programs allow performance analysis of particular algorithms and systems without the effort required to run a full application but with more of the complexities than in regular benchmarks.

In all cases, benchmarks are designed to help a developer better understand how applications interact with hardware. This understanding can be targeted at particular aspects of hardware (STREAM and NAS), types of applications (mini-apps), or some combination of the two (Roofline).

2.4 Understanding power and energy

As supercomputers approach exaflop speeds, power and energy use have become increasingly important issues. Power is an instantaneous measure of how much electricity is being used and creates challenges for supercomputers as many systems are theoretically capable of drawing more power than is within supply and thermal limits. Energy is the amount of electricity used over a period of time and corresponds to how much the electric bill costs. Sometimes keeping both of these low can be conflicting goals. Options such as increasing the threads in use on a CPU or raising the clock rate can cause the application to finish faster thus using less total electricity but may cause spikes in the power above what is available to the node. Controlling and measuring both metrics is increasingly challenging as systems add accelerators, vector units, and frequency scaling.

Currently, many applications can be easily optimized for energy use using the "race to sleep" method; energy, being dependant on how long the application runs, can be minimized by finishing the application as soon as possible. In this scenario, optimizing for performance is identical to optimizing for energy use. In [6], the authors break this paradigm for a specific set of circumstances. They run the programs exactly the same over a range of frequencies and show that there is not always a direct connection between time and energy use.

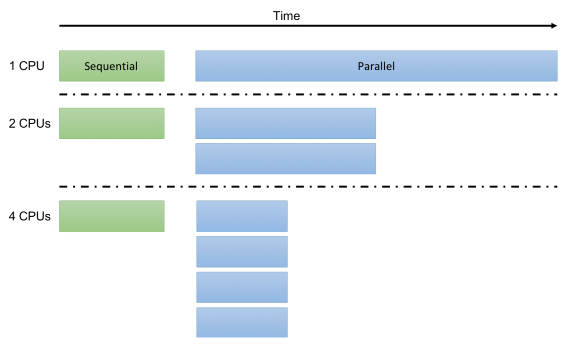
In an effort to enable better energy use for HPC, several authors have tried to model and benchmark it similar to time. Choi et al. [27, 26] developed a roofline model that related time and energy issues to guide developers and tuners in improving the energy use for their applications. Ilic et al. [60] produce a more detailed model based on deep analysis of memory systems and CPUs at the core, chip, and package level.

A great deal more work is needed in the area to fully understand energy and power use especially as new architectures complicate the problem. This topic should be an interesting area of research in the near future but is largely beyond the scope of our current review.

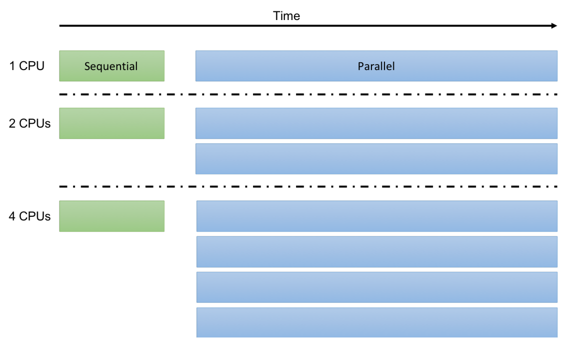
2.5 Parallel Algorithm Models

For HPC applications, an important first step is to parallelize the algorithm. This involves decomposing it into sub-tasks that can be computed concurrently with one another and adjusting the algorithm and data structures accordingly. Parallel tasks can take several different forms depending on the algorithm. For example, a graph algorithm could be decomposed into tasks based on nodes or converted into a matrix and tasks could be based on the parallelism inherent in linear algebra. Considering multiple approaches to parallel computation gives the developer options once implementation and target hardware details become available.

[6]https://computation.llnl.gov/projects/co-design/proxy-apps



(a) A diagram of a strong scaling



(b) A diagram of a weak scaling

Figure 9: *Strong scaling where the problem size is static compared to weak scaling with a problem size that increases with the number of threads.*

In even the most parallel algorithms a certain amount of work will remain sequential. Amdahl [3] noticed that an algorithm can be separated into parallel and sequential sections. Based on this one can calculate the potential speedup by dividing the fraction of the program that can be parallelized (f_p) by the number of processors (N) (or infinity if you would like the max) and add that to the fraction of the program that is sequential (f_s) and taking the reciprocal of the result (see equation 2). For a more accurate calculation, the overhead (O) can also be included (equation 3), but the time (T) must also be added into the equation.

$$S = \frac{1}{f_s + \frac{f_p}{N}} \tag{2}$$

$$S = \frac{T}{T * f_s + T * \frac{f_p}{N} + O} \tag{3}$$

Amdahl's Law is concerned with strong scaling in which more parallel tasks are used to operate on the same size of data (Figure 9(a)). Amdahl demonstrated that speedup is thus asymptotically limited to how much of the program must run sequentially. Originally, this argument caused researchers to take a pessimistic view of parallel computing; however, the real advantage of parallel computation comes from Gustafson-Baris' Law and weak scaling (Figure 9(b)). Instead of attempting to do the same problem on a larger computer, weak scaling uses a larger computer to do a larger problem. In this law, the sequential part of the program is assumed to be the same no matter how large the program or computer gets and the parallel section grows arbitrarily with the computer size. Both methods of understanding parallelization offer important insights into how to effectively design and optimize a parallel application.

Gustafson-Baris' and Amdahl's Laws assume that some arbitrarily parallelizable section of code exists to be broken up among many processors. An alternative to this assumption is the Work-Span model. This model uses a task graph (Fig 10) to describe the application with tasks and dependencies defined as a directional graph. The *Work* is the total number of tasks that must be performed (i.e. the sequential time) while the *Span* is the largest number of concurrent tasks (i.e. the maximum number of tasks that can be performed in parallel). The longest path of dependencies through the graph is the *critical path* which is the shortest possible time that the application can be completed. These tasks can take the form of high-level sections of the application such as input, initialization, and output; or lower level details such as the additions necessary to reduce a vector dot product into a final sum.

Based on these different understandings of parallelism, the developer can begin to adapt the algorithm to parallel computation. The main part of this process is task decomposition - dividing the tasks into independent parts that can be run concurrently. The process of decomposition generally doesn't have one right answer but will produce several options for the developer to consider. In our stencil example, the tasks could be defined as computing new values for each row (as many tasks as there are rows) or computing new values for each mesh variable (as many tasks as there are variables). Furthermore, these decompositions could be nested with individual processes tasked with computing on each variable and threads within the processes handling individual rows. Much more information on parallel programming can be found various texts [96] [49] [76].

When considering different decomposition options, the developer must also think about the types of hardware available. Some applications work better on GPUs and other are better to be distributed across a cluster. Different levels of decomposition can be spread to different types of hardware. Large tasks can be split to across the distributed memory nodes with the subtasks allocated to threads within those processes. Numerous identical operations (i.e. matrix operations) can be distributed to SIMD hardware such as vector operations in a CPU or offloaded to a GPU. It is also necessary to consider how the levels of decomposition impact each other. For example, if our stencil example with AoS data structures is parallelized by variable, then each thread will operate on one data element from each struct in the array. As a result the cache coherency hardware will think the threads are sharing all the data in a cache line. When the threads are spread to different cores, this false sharing will cause constant updates to the local caches rendering the L1 caches useless. Comparing different decompositions can help identify how to best fit them together.

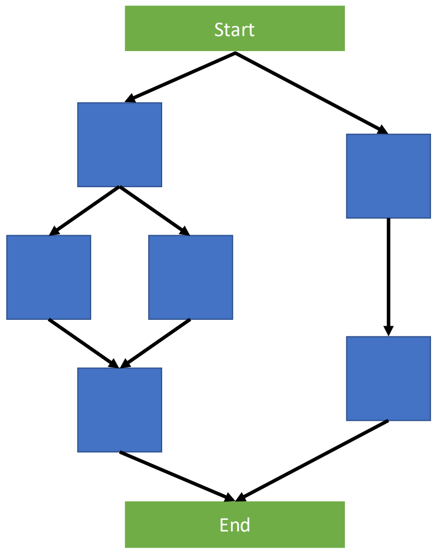


Figure 10: *An example of a task graph with a critical path of 3 and a span of 3.*

3 Performance Analysis and Optimization

Building on the foundation of computer architecture and parallel theory discussed in the previous section, a programmer can adjust an application for maximum efficiency. The process of locating and fixing performance bottlenecks is called performance analysis and optimization. In this section, we discuss many of the ways developers can analyze their applications including several case studies from literature.

3.1 Predicting Performance Improvement

Predicting the amount of improvement that a particular optimization can provide is an important area for performance analysis. Often changes to the code are not trivial, so knowing which should be attempted is vital for the work efficiency of the developer. Significant work has been done in an attempt to predict how improvements to an application will impact the performance; most of it involves looking at how much particular sections of code contribute to the total runtime of an application.

$$S = \frac{T}{T * f_u + T * \frac{f_i}{Speedup}} \tag{4}$$

For example, we discussed several equations for predicting the performance gains from parallelizing an application in Section 2, such as Amdahl's Law and Gustafson-Baris' Law. These both provide upper bounds for the improvement derived from turning a serial application into a parallel one. In Figure 11, we show the scaling of our stencil example compared to the ideal scaling based on Amdahl's Law. This figure shows that our stencil is close to perfect scaling but not quite there. Amdahl's Law can be adapted beyond parallelism to explore speedups of various parts of the application. In Equation 4, we have the Law with serial and parallel fractions adjusted to be the *fraction unchanged* (f_u) and *fraction improved* (f_i) with the node count replaced by a generic speedup. This simple equation can help a programmer identify sections that can have a significant impact on performance and how much they must improve that section by to realize their performance goals.

Empirical methods of predicting performance improvements also exist. Critical path analysis can be used to determine upper bounds on the improvement to an overall program that can come from improving one function [58]. More recently, causal profiling seeks to predict how improving one area of a program will impact the overall performance. Coz [30] does this by slowing all other areas of the application to create an artificial speedup of the targeted section. This method aims to warn developers away from improving sections of the code that are not on the critical path and thus, don't impact total performance. OMP-WHIP [15] and Task Prof [114] have similar goals but are more targeted towards predicting how parallelism will impact the performance. Both tools create task graphs to determine the best way to parallelize the application. These predictive measures of profiling can be helpful to developers who are deciding what part of the program to improve next.

These methods can point a developer to parts of the program that could have significant impact of performance, or warn them away from optimizing insignificant areas of the code. However these methods are not effective at indicating how those sections can be improved or if it is possible to improve them at all.

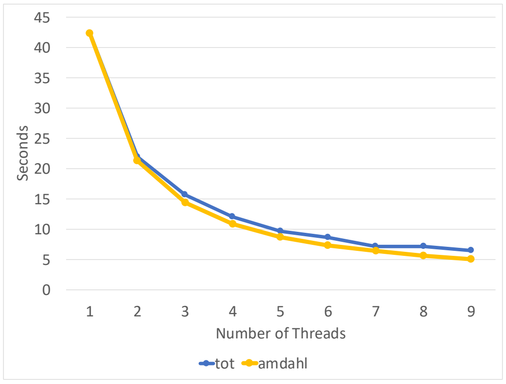


Figure 11: *Timings for the stencil application vs the number of threads compared to the ideal scaling based on Amdahl's Law.*

3.2 Algorithmic Modeling and Analysis

Although performance prediction can be useful many developers would like to the improvements realized in the application. One of the simplest ways to optimize an application is for the developer to look through the code and try to find ways that it can be improved. Often this technique will be the first step (or even a side thought in the initial development) in optimizing an application and lead to simple, but important, steps of reducing redundant computation, overlapping communication and computation, and generally attempting to eliminate obvious inefficiencies. In literature, this method often is presented as algorithm or code analysis, in which the authors explain certain aspects of an algorithm or implementation in great detail and show how those aspects connect to the hardware and impact performance. The authors will often follow that explanation up with related improvements and performance results.

Commonly, applications are limited by the rate at which memory can be loaded into the processor. This fact has brought about the development of deep memory hierarchies and complicated optimization schemes for fitting data into those caches. Additionally, it means that many of the by-hand analyses can focus on how data is loaded into the processor rather than on the complexities of the processor itself. For example in [107], the authors identify that the algorithm used for sparse matrix A^TAx computation is memory bound, and that, in standard implementations, the A matrix is read twice. They reorganize the algorithm to perform the computation by reading the matrix in only once then carefully analyze the resulting program to develop upper bounds for their implementation. Included are two analyses, one which ignores conflict and capacity misses and another which considers capacity to help identify blocking size for the autotuner Sparsity [61]. The prior analysis is compared to actual times and an upper bound guided by empirical data from hardware counters. The hardware counters suggest a close but lower bound than that of the analysis likely caused by ignoring buffer sizes and associativity of the cache.

In this example, careful analysis of the memory use is performed based on a strong knowledge of how the processor uses the memory and what assumptions can be made. Indeed, the authors make significant assumptions about the memory, ignoring conflict misses, buffer size, associativity, and other details that impact performance. In the fifteen years since these papers were written, the complexity of caches and processors has significantly increased. To perform such analysis on a modern system, one must also disregard vector instructions, cache coherence, non-uniform memory access, or focus purely on the level one cache. Such broad stroke assumptions can still inform important optimizations like the reworking of the memory access and blocking as Vuduc et al. did [107], but more advanced methods are needed if static analysis is to contribute beyond the early stages of algorithm design.

Similar methods can be used automatically rather than manually to apply these techniques to larger codes. For example, data locality patterns can be predicted for full runs of a program based on data from small sample runs [33]. If developers are looking to move to GPUs the performance can be automatically predicted based on estimations from existing CPU codes [77]. Lastly, ExaSAT [103] is designed to model software and hardware to predict upper bounds on the performance of particular applications of architectures that haven't been built yet. This goal is accomplished in two parts. First, the application is statically analyzed using the ROSE compiler framework[7] [86]. Then that information is combined with architecture specifications given by the user to produce a reasonable approximation of the upper bounds. To achieve this approximation the authors focus on analyzing the data movement of the application but also include some discussion of the computation and communication. These examples demonstrate how automating the manual analysis techniques can reduce the work required to study large applications.

Ideally, static analysis will enable the developer to understand how implementation decisions affect the use of hardware and how adjustments can be made to the implementation to better use the system's resources. Whether it is performed by-hand or with an automated tool, this type of static analysis enables developers to gather ideas for improving performance. These ideas can be broad algorithm changes, slight tweaks to improve the efficiency of a loop, or anything in between, but they will remain simple hypotheses until tested.

3.3 Empirical Performance Analysis

A common method for performance analysis and optimization is the basic scientific method. A hypothesis is made about an improvement to the code, and adjustment is made, and the hypothesis is tested. For example, a developer may predict that the AVX-512 vector instructions are faster than the AVX2 instructions. This hypothesis can be tested with either intrinsic commands or Intel compiler flags. Timers, hardware counters or other metrics (see Section 4) can be used to compare the versions and determine the effectiveness of the optimization. Based on these metrics the developer can then choose their preferred version. We call this method empirical analysis.

Empirical analysis follows directly from the static analysis discussed above. Gropp et al. analyze CFD codes in [51], count memory and computational operations and use knowledge of hardware to predict how fast the runtimes will be. They show that the memory bandwidth is generally the limiting factor as the estimations based on those counts best fit reality. Based on this information, the authors attempt and evaluate a variety of optimizations to improve cache and memory performance [50]. Specifically, they move from an AoS data arrangement to an SoA which improves spatial locality since the values of mesh points are used in conjunction with each other. Additionally, they use blocking to focus on sections of the mesh that fit within the cache rather than trying to use the entire mesh at once. Results from hardware counters confirm the reduction in cache and TLB misses, so the hypothesis is empirically validated.

Rosales [91] use this technique effectively to adapt s Lattice Boltzmann Code to the Intel Xeon Phi architecture. First, the author considers vector instructions which are initially unsuccessful but provide significant improvement once the data structures are adjusted to fit an SoA design rather than an AoS. The author also addresses the alignment of memory and distribution of work between the Xeon Phi and its Host Xeon processor. In each case, a rate of computation (in GFLOPS) demonstrates how the adjustments have impacted the performance. A similar analysis is performed in [89]; Reguly et al. adapt unstructured mesh computations to heavily vectorized hardware. They include both Intel Xeon Phi and GPUs in the paper, adding OpenMP, vector intrinsics, openCL to the existing MPI implementation to work with the new architectures. They provide detailed explanations of the logic behind the optimizations, discussing which sections of code have the performance bound by memory, computation, and control flow.

In [22], the authors focus on improving the uniprocessor performance of a parallel CFD application. Their first step is to use Gprof to determine where the application spends most of its time. This process allowed the authors to determine 10 subroutines that could be further analyzed for performance improvements. Once these routines are determined, the authors try out several loop optimizations. First, permutations are applied to the loops to ensure that the matrix elements are accessed in the same order as they are stored in memory (Fig. 12). Each of the arrays shown in Fig. 12 is 4D with the innermost dimension holding only 4 elements. The static size of the resulting innermost loops allows the authors to fully unroll the loops, avoiding branch penalties and maximizing the instruction level parallelism. In some similar situations, the innermost loop varies to be 3 or 4. The authors code two versions of those loops, each with the full 3 or 4 level unrolling, to achieve the same goals. These optimizations are targeted at maximizing the performance of the caches and the processor pipeline as is discussed in Section 2.

[7]http://rosecompiler.org/

```
! Original Loop
DO 100 J = 1, JM
DO 100 K = 1, KM
DO 100 L = 1, LM
  ...QBAR(*, J, K, L)...
  ...Q(*, J, K, L)...
  ...Q1(*, J, K, L)...
100 CONTINUE
```

(a) Original loop

```
! Permuted Loop
DO 100 L = 1, LM
DO 100 K = 1, KM
DO 100 J = 1, JM
  ...QBAR(*, J, K, L)...
  ...Q(*, J, K, L)...
  ...Q1(*, J, K, L)...
100 CONTINUE
```

(b) Permuted loop in [22] to iterate over the innermost values of the arrays

Figure 12: *Loop permutations to maximize cache performance in 4D arrays in [22]*

Another major performance improvement made in [22] is to the mesh generation. The original code would generate the mesh in a standalone application, write that mesh to disk and then read it off disk to begin the actual computation. Just as the improvements to processor have outpaced those to memory, the processor performance has similarly outpaced IO performance, so this method was a serious bottleneck to the application. The authors resolve the problem by incorporating the mesh generation into the main application which will eliminate the need for writing it out to disk.

White et al. [109] use this empirical method to explore memory performance of unstructured mesh applications. The authors show how heavy use of object oriented programming resulted in large fractions of the time being spent computing memory addresses. The authors then discuss how precomputnig the locations of the data, inlining various data access functions, and reordering of the mesh can reduce the time spent on such aspects of the application. Additionally, they discuss how the need for flexibility in the application resulted in the redirections which helps the reader understand the needs of HPC applications beyond performance issues.

As GPUs have become more common in HPC and non-HPC scientific applications, performance analysis involving porting applications to GPU has become common. In [68, 67], Lacasta et al. show the importance of mesh reordering in GPU performance as the GPUs are very sensitive to data layout. In particular, they discuss unstructured meshes and how the edges and cells need to be reordered so that the memory access patterns are convenient for GPU access. Alternatively, in Jespersen's analysis [63] of the CFD application OVERFLOW, the authors are forced to adjust the algorithm to remove data dependencies that impede moving to GPU. Liu, Qin, and Li [72] discuss how HPC techniques and commercial GPUs can be used to enable users to use laptops and workstations to run applications that were previously limited to large clusters. They motivate the move to personal computers with the important study and prediction of flooding rivers in which emergency response groups made not have access to large systems. This application shows both how GPUs have impacted HPC and how improvements in HPC can have an impact on other computing platforms.

In addition to GPUs, CPU architectures have advances to include SIMD instructions which take careful work to efficiently use (see Section 2). Hadade et al. [52] discuss using such architectures (in particular Intel Broadwell, Skylake, KNC, and KNL) to perform CFD applications. Although similar to GPU computation, CPU SIMD presents some different challenges including shorter vector lengths. This difference causes the authors to move from a (SoA) data format to an (AoS), the opposite of how similar codes are optimized for GPU. CPUs offer efficient SIMD gather instructions that allow such an organization to be effective. They go further for some data items; moving to an Array of Structures of Arrays (AoSSoA). This setup differs from AoS in that the items in the structures are arrays with lengths pegged to the size of the SIMD registers for optimal loading efficiency. These differences between modern CPUs and GPUs demonstrate the importance of optimizing an application for a particular machine.

In most of these case studies, the authors use hypotheses based on expert knowledge about the memory structure or processor architecture and how the application will use those systems. The authors then verify those claims by demonstrating performance improvements based on timing of the applications. A common approach to help guide the expert interpretation of the application and hardware is to use performance counters to report information on CPU and memory activity. Vetter and Yoo [105] present a thorough analysis of eight HPC applications based on the computational, memory, and MPI communication performance. The data is collected through function level profiling which allows for intricate details of each application to be analyzed. More on this type of collection can be found in Section 4. However the data is collected it relies on the skill of the experts at understanding how that data connects to the implementation and their creativity at finding alternative implementations.

3.4 Analysis of Parallel Applications

Many of the above examples target parallel codes, but the optimizations discussed could be used for serial codes just as well. When working on parallel applications there are often problems and overheads that need to be addressed in addition to the challenges that face both serial and parallel applications. These include minimizing the overheads of managing threads or processes, handling shared data, and balancing the workload across threads or processes.

The saying goes that there is “no such thing as a free lunch”, and threading is the same. Initializing and coordinating threads takes time, which can limit the improvements that come from multi-threading [17]. Similarly, initializing and using processes incurs overhead. It takes significant time to move data from one processor node to another, so it is important to minimize how much communication is performed as the authors do in [94]. Most of the improvements for thread or process overheads are limited to the implementations of the parallel libraries, but developers can compare different libraries, limit the creation of new threads, and avoid implicit or explicit barriers to maximize the efficiency of the parallel library they have selected.

When multiple threads are attempting to work on the same data, contention occurs that can interfere with the parallelism being exploited in an application. Critical sections are areas of the code that must be performed sequentially; only one thread can enter that section at a time. When considering critical sections, the developers must consider how often the data is in contention, how often the lock is in contention (indicating false sharing if different than the data), and how much time is spent in the locked region [92]. In [48], the authors reorganize the data structures to prevent contention from interfering with the parallelism in a shared memory system. A variety of optimizations exist to limit the impacts of contention, for example, padding variables to not share the same cache line can prevent false sharing and performing gather-type operations (i.e. summation) with a tree algorithm.

In parallel applications, several threads or processes run simultaneously and the overall time that the program takes is determined by the longest running thread or set of threads, called the critical path. If a developer improves a section of the application that is not on the critical path, then it is unlikely the overall computation time will be improved. As a result, an important part of analyzing parallel applications is establishing what parts of the computation are on the critical path and what are not. Early work in this area by Yang and Miller [112] established using Program Activity Graphs based on the execution of a program to determine which sections comprised the critical path. Later, this method was applied to shared memory applications [58].

If some threads or processes have significantly more work to do than other threads, then the machine is not being used efficiently and severe performance penalties can result particularly if scaled to large numbers of threads or processes. Du Bois et al. [38] provide a convenient metric for identifying which threads limit the computation time. They define thread criticality as a measure of time that shows how much a thread operates while other threads are waiting (Equation 5). The authors of [99] work to attribute the imbalance to specific functions using the callpath analysis provided by HPCToolkit (more in Section 4).

$$thread\ criticality[id] = \sum_{i=0}^{N-1} \begin{cases} \frac{time\ for\ section}{num\ threads\ working} & thread[id]\ is\ working \\ 0 & thread[id]\ is\ not\ working \end{cases} \tag{5}$$

All of these challenges to parallelization manifest themselves in sub-optimal scaling performance. As the number of threads in use increases, the performance ideally grows linearly with that number; however, the speedup is usually far less than this ideal. Speedup Stacks [41] offer a way to identify which problems are limiting the scaling speedup. This method of analysis uses profiling to collect time taken for overhead, waiting on locks or barriers, cache interference, and the actual program. The speedup stack then demonstrates how the execution time is divided between these operations so the developer can devote time to the most troublesome areas. Often scaling problems only appear once the application is scaled to larger sizes which can be time consuming and expensive. Extra-P [18] provides automatic analysis that can find scaling problems from application runs with fewer threads or processes.

In the lead-up to Exascale computing, the amount of parallelism used by architectures and applications has increased and will likely continue to increase dramatically. Single node parallelism, in particular, offers many new challenges with GPUs and many-core CPUs (i.e. Intel KNL) straining the existing solutions to parallel software. Interested readers are directed to [69] for more information on programming such systems. The increased parallelism will likely lead to significantly more data available for empirical analysis, which will increase the challenges for experts trying to connect that data to meaningful improvements in the application.

3.5 Improvement of Large Applications

One of the most exciting aspects of HPC is the large applications that can use an entire supercomputer and produce results with major scientific impact. Such applications are developed and optimized by large teams of domain scientists and HPC experts targeting specific supercomputers. These applications require many types of optimization from the algorithm level to the operation on individual nodes of the machine.

In one example, the authors of [104] present a complete overhaul of the earthquake simulator SeisSol. This work targets homogenous systems of Intel Haswell or KNL nodes and incorporates algorithmic, cache, communication, and IO optimizations. The authors make significant algorithmic changes so that important matrices can fit within the L1 cache of the processors and use autotuning (see section 5) to optimize the matrix storage technique (sparse, dense, or block partitioned) and order of matrix multiplications. These optimizations improve cache performance and reduce the number of FLOPs required by the application. They make additional changes to overlap communication, IO, and computation. Overall the improvements allowed the researchers to gain important insights into the devastating 2004 Sumatra-Andaman Earthquake that caused tsunamis around the Indian Ocean.

When the first Petascale system, Roadrunner, was constructed at Los Alamos National Lab, the unique architectural features required significant changes to applications to allow for efficient computation. When porting the plasma simulation application VPIC to Roadrunner [16], the developers made significant changes to how computation and communication were managed among the CPUs, how memory was stored and accessed, and how SIMD instructions were used. These optimizations resulted in a sustained performance of 0.374 Pflop/s for the application.

Recently, such work is often concerned with coordinating massive parallelism and heterogeneous resources. A recent Gordon Bell Prize contestant [12] performed a new algorithm for Lattice Quantum Chromodynamics simulations on Sierra, Summit and Titan supercomputers. This simulation required that several types of computation be performed in conjunction with each other, some using CPU resources and some using the associated GPUs. The authors go to great lengths to ensure that the GPUs can communicate with each other without going through the CPU so that unnecessary CPU overhead is avoided. Additionally, the Monte Carlo part of the application produces numerous significant jobs with varied completion rates, motivating the authors to develop a job scheduler that does not require Monte Carlo jobs to wait for one another. At the node level, the authors use an on-line autotuner (see section 5) to adjust each computation to the particular conditions of each node. The authors cover every detail of the application from the physics algorithm used, to the management of communication, down to the cache performance of each GPU, allowing for dramatic improvement in the time to solution for their physics results.

These are only three examples of the many exciting scientific applications that HPC makes possible. Without the performance analysis techniques discussed in the rest of this section, many such applications would take an infeasible amount of time to complete.

4 Measurement Approaches

The performance analysis techniques in Section 3 require the collection of some form of data to understand the application. Such data can be as simple as timing different runs of the application or as complicated as recording each call to functions of interest, but they all require some kind of tool to collect that data. There are generally three ways for tools to gather data about an application. One is through sampling metrics at given intervals or specific events, another is through instrumenting the application, and lastly, there is tracing events by recording the order in which they occur. Each method has its own advantages and disadvantages and they can often be used in tandem to gain greater insight. These tools interface with the application being tested and what we refer to as "low level libraries" to connect the metric to particular aspects of the program.

4.1 Low Level Measurements

User-oriented performance analysis tools rely on many lower level libraries, OS functions, and hardware capabilities to function. Each of these provides information or capabilities that permit the tool to gather data and present it to the user.

Most software developers have, at some point, attempted to time their code through a language's timing functions or the Linux "time" command. These methods are useful even in advanced performance analysis and many tools use the same underlying OS timers to provide wall clock and CPU times (see Table 1) for the target application. Additionally, the OS often provides basic information on memory (i.e. through "/proc/meminfo") that tools can poll for information on memory use. These utilities provide a fundamental starting place that all of the other measurement techniques can build on.

Table 1: *Types of time used in performance analysis.*

Name	Definition
Wall Clock	Time according to "the clock on the wall" i.e. normal human time from the start of the program until its termination
CPU Time	Time the program is runnning on each CPU i.e. four CPUs for 10 minutes (wall clock) is 40 CPU minutes
User Time	Time spent in the user space
Kernel Time	Time spent in kernel space (system calls)

Timers can give a good indication of what areas take up the most time (called hotspots). Amdahl's Law tells us that improving these areas will have the most impact, but they may not correspond to places that need improvement. Given their importance, hotspots are often kernels of computation that are the focus of optimization efforts early in development. For this reason, performance analysis experts brought in later may find nothing to do in the biggest hotspots, but they also may find opportunities for the most improvement.

For more detailed information on events happening in the hardware at the node level, a user can access hardware counters or performance counters. Often these are accessed through the Performance Application Programming Interface (PAPI) [101]. PAPI allows tools to easily access hardware counters that Intel and other manufacturers include in their CPUs. Originally intended purely for in-house verification these counters tend to change with each new release of a processor, so it is important to keep PAPI updated and be aware of what counters are available on your system. The counters use registers to count specific hardware events such as cycles, cache access/ misses, floating point operations and much more. A user or tool can use hardware counter information to find areas with high cache miss rates or frequent stalls and determine the best way to alleviate those problems.

Some examples of counters include:

- PAPI_TOT_CYC (total cycles) - useful as a proxy for time that ignores the tool overhead
- PAPI_L2_TCA and PAPI_L2_TCM (level 1, 2, or 3 cahce acceses/ misses) - useful for determining how well each level of cache is used
- PAPI_SP_VEC and PAPI_DP_VEC (single and double precision vector instructions) - useful for determining if the program is successfully vectorized
- PAPI_TOT_INS PAPI_LST_INS and PAPI_FP_OPS (total number of instructions, number of load/store instructions, and number of floating point operations) - useful for computing computation intensity

Recently, the RAPL counters [32] were added to Intel processors and incorporated into PAPI provide information on power and energy use. If RAPL counters are insufficient, WattProf [88] is a PCIe extension that has multiple sensors to collect information on different aspects of the system. Alternatively, Rahman et al. [87] estimate energy use based on hardware counters for cache operations, CPU stalls, and floating point operations.

These performance metrics would not be useful if it weren't possible to tie performance issues back to specific parts of the code. The simplest way to make these connections is to include debugging information in the compiled application. Usually, compilers will remove line numbers and function names from a binary application, but the '-g' option ('-gopt' in PGI) will leave this information. Tools can connect the measurement of a metric back to a program counter location and the debugging information allows them to go the next step to the actual function, loop, or even line of code responsible.

The debugger information can provide detailed information about where the measurement was taken, but often functions are called from multiple parts of the code. To find the callpaths to a particular measurement you must "unwind" the callstack with a library such as Libunwind[8]. Unwinding the callpath allows the tool and the user to differentiate between function calls made from different sections of the application. For example, many parts of an application may call MPI_BARRIER, but the overuse may only occur in one area. Using the callpaths allows the user to know which call to MPI_BARRIER causes the issue.

Parallel libraries or language extensions can require additional effort to separate where the measurements are taken. For OpenMP parallel API, OMPT [40] provides an interface so that measurement tools can better understand what measurements correspond to particular OpenMP pragmas. NVidia provides a similar function for CUDA users through nvprof [9] which can be connected to other tools or through its own user interface.

In addition to single application analysis, there are tools aimed at monitoring and analyzing the performance of a whole HPC system. One such tool, the Lightweight Distributed Metric Service (LDMS) [2] collects the system information at regular intervals on the order of seconds. Available data includes information on CPU usage (time in user, kernel, and idle), memory use (allocated, cached, buffers), and virtual memory statistics (file information, paging info), and job information. It collects, aggregates, analyzes, and displays this data so that administrators can better understand how well the system is being used. Such information can be useful to system administrators trying to understand how the systems are used but is largely beyond the scope of this discussion.

These libraries and tools collect information based on the hardware or OS. Such raw data can be useful but is usually best understood through a more user-oriented tool. These tools interface with the low-level measurements discussed above to collect information through sampling, instrumentation or tracing. Once the data is collected the tools can present it to the user directly or perform preliminary analysis to help direct the user.

4.2 Sampling Application Performance

The first method of collecting performance information is sampling which takes advantage of hardware interrupts to record values for a measurement and the approximate location in the code. These interrupts occur at scheduled intervals ranging from a few milliseconds to a few seconds and whenever the counters in question overflow. Sampling can be performed without significant changes to the build process, so it allows for easy tool workflow. Changing the frequency of the interrupts allows the user to adjust the overhead to their current needs, minimizing overhead. The simplicity and low overhead of this method make it excellent for getting a high-level overview of the application which can help find some optimizations or indicate which sections are best suited for additional analysis and optimization.

However, depending on how the application is written and the sampling frequency, it can be possible to see significant inaccuracies in the results. Sampling attributes the whole counter value to the spot where the sample is taken. This method statistically tends to be accurate but can over-represent certain spots depending on the sampling frequency. For example, a low sample frequency or small function sizes can cause samples to miss important functions by representing only the larger higher level ones. Similarly, OpenMP and MPI Barriers can falsely accumulate counts for metrics such as floating point operations if the sample

[8]https://www.nongnu.org/libunwind

[9]`https://docs.nvidia.com/cuda/profiler-users-guide/index.html`

intervals happen to fall when the thread or process is waiting for others to complete. However, these issues are not necessarily present for all applications or machines and are merely potential challenges that users of sampling tools should be aware of. If you think such errors are occurring, run several experiments with a variety of sample frequencies (preferably prime numbers) and compare the results.

Several of the tools in this survey make use of sampling. Open|SpeedShop primarily collects information through sampling, while HPCToolkit augments it with binary and source analysis to help connect the data back to the source code. TAU allows the user to choose between sampling and multiple instrumentation options.

4.3 Instrumentation of Applications

Instrumentation works by inserting library or API calls directly into the source or binary application to make measurements precisely where the user or tool would like. Instrumentation has the great advantage of being able to selectively study an area of an application rather than look at the entire execution. This feature is very useful for being able to perform focused analysis on a particular piece of an application. Unfortunately, instrumentation incurs a heavy overhead (100s or 1000s of times normal), so users generally cannot instrument every function in an application. Ideally, sampling can be used to get an overview of the application and instrumentation to explore specific areas of interest.

Instrumentation can be added in several different ways. The first option is to manually add calls to all of the areas of interest. Obviously a challenge for large applications, the manual method can be useful if the user has very specific areas that they are interested in measuring. Caliper exclusively uses manual instrumentation while TAU allows users to do so but the developers often recommend using automatic methods first. For automatic instrumentation, parsers can be used to search through the application and add instrumentation calls to the source code, as in PDT [71], which is available in TAU. Finally, there is compiler instrumentation which targets the code during compilation so that the compiler optimizations are taken into consideration during the measurement. This is TAU's default method of instrumentation. In the case of automatic instrumentation, TAU allows the users to define sections that should be included or ignored to limit the analysis to areas of interest and minimize overhead.

4.4 Tracing

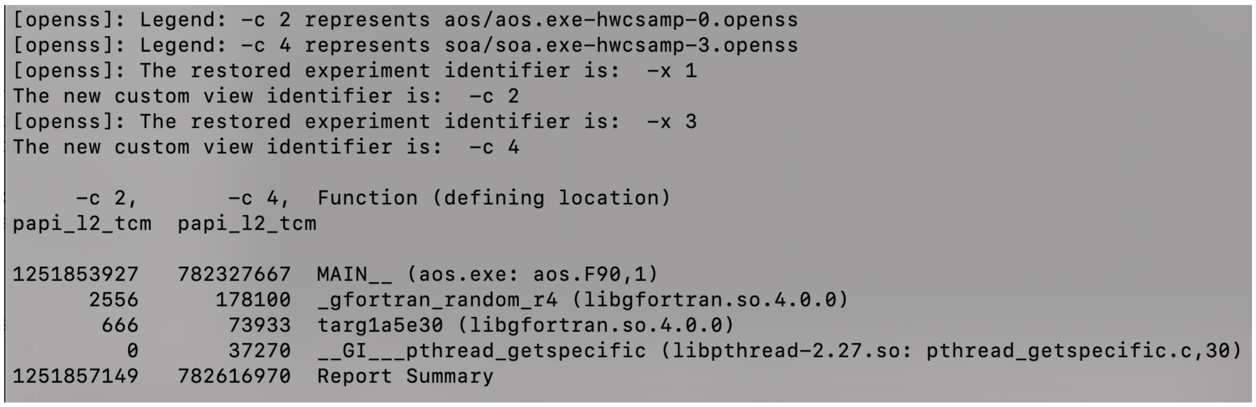
While sampling and instrumentation focus on associating events with specific areas of code, tracing is focused on associating events with the time and order in which they occurred. At each instance of each traced event (i.e. memory access, MPI calls) the event and timestamp are recorded. This data allows an analyst to look at how events are related and is particularly useful for examining how MPI ranks interact to minimize communication overhead. Since each individual event must be recorded, tracing adds frequent calls to the performance analysis program which results in a large overhead to the experiment and produces a large amount of data. For this reason, it is not recommended as an initial approach to performance analysis.

Chen and Stenstrom [24] use tracing to identify important critical sections in multithreaded applications. They instrument the pthreads library so that threading calls can be traced and used to determine which critical sections lie on the critical path of an application. This analysis guides programmers who want to know how to minimize the performance loss due to synchronization between threads.

4.5 Open|SpeedShop

Open|SpeedShop (OSS) [93] is a user-oriented tool designed to be easily usable by non-experts and easily expandable by tool developers. This application is developed by the Krell Institute in collaboration with LANL, LNLL, and SNL. OSS is compatible with all major processor types including ARM systems and GPUs and provides both graphical and command line interfaces that have the same functionality.

By default six experiments are distributed with the application; two that sample with simple timers, one that allows the user to interact with PAPI counters, and three tracing experiments (MPI, IO, and floating point exceptions). The resulting data can be explored through a graphic interface that includes some basic analysis functions. One particularly useful function is the ability to compare two versions of the same application. We show an example of the command line output when such a comparison is made in


```
[openss]: Legend: -c 2 represents aos/aos.exe-hwcsamp-0.openss
[openss]: Legend: -c 4 represents soa/soa.exe-hwcsamp-3.openss
[openss]: The restored experiment identifier is:  -x 1
The new custom view identifier is:  -c 2
[openss]: The restored experiment identifier is:  -x 3
The new custom view identifier is:  -c 4

     -c 2,       -c 4,  Function (defining location)
papi_l2_tcm  papi_l2_tcm

1251853927   782327667  MAIN__ (aos.exe: aos.F90,1)
      2556      178100  _gfortran_random_r4 (libgfortran.so.4.0.0)
       666       73933  targ1a5e30 (libgfortran.so.4.0.0)
         0       37270  __GI___pthread_getspecific (libpthread-2.27.so: pthread_getspecific.c,30)
1251857149   782616970  Report Summary
```

Figure 13: *A comparison of AoS and SoA versions of sample code with Open|SpeedShop.*

Figure 13. This output shows that the AoS version (in column 2) has more L2 misses than the SoA version (in column 4). Simple analysis such as this can provide deep insights into the application for a user who is familiar with it.

4.6 TAU

The TAU performance system [95] is another mainstay of performance analysis on HPC systems. It provides users a full complement of measurement tools including timers, sampling, instrumentation, and tracing, which enables users to study everything from small single node performance to scaling tests of large MPI applications. It is also available across most platforms including Intel, IBM, and ARM. Additionally, through Paraprof [10], TAU gives users advanced visualization options so that they may understand the data it produces.

An excellent way to start TAU is through sampling. For most applications, sampling provides the best balance between speed and accuracy. The program is compiled normally (with debug options) and run through the 'tau_exec' execution program. Arguments such as mpi, openmp, and PAPI are included to help TAU find the relevant Makefile and TAU libraries. More advanced users can set the sampling frequency to ensure their application is adequately covered. For instrumentation, the application must be compiled with the tau compiler wrapper then run as usual. The compiler wrapper uses an underlying compiler (set when the Makefile is configured) and adds instrumentation to the code. Optionally, the user can use a selection file to indicate which parts should and should not be instrumented. Instrumentation involves a great deal of overhead, especially if you attempt to instrument the entire application. The application can then be run normally in most cases with the added calls collecting data. While primarily focused on profiling the application, TAU can also provide some tracing functionality. These trace results tend to be quite large, but can be viewed with the proprietary VAMPIR [80] from TU Dresden or the free Jumpshot [20, 111] from Argonne.

TAU's data collection interface is based entirely in the command line through configure commands and generated makefiles. As a result, it is not particularly user friendly and the development team has been working to improve this situation. The TAUCommander tool[10] provides a significantly more user-friendly commandline interface to the full TAU and Paraprof toolsets with excellent re-usability of previous configurations. In this paper, we use the original, but highly recommend TAUCommander to new users.

4.7 HPCToolkit

HPCToolKit [1] is a well-developed tool with a large GUI interface for viewing results. It aims to provide as much information about the application with as little interference with the source code and build process as possible. To achieve this goal, the developers at Rice University couple instrumentation with binary analysis to be able to accurately connect metrics to sections of code.

[10]taucommander.paratools.com

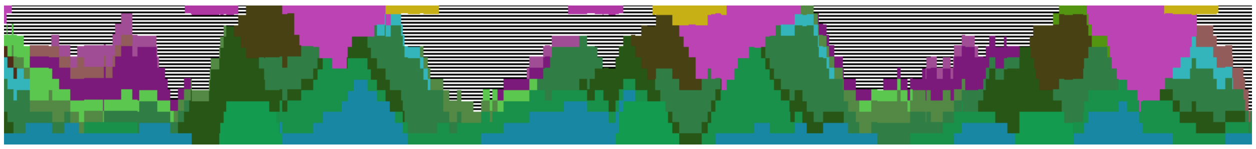


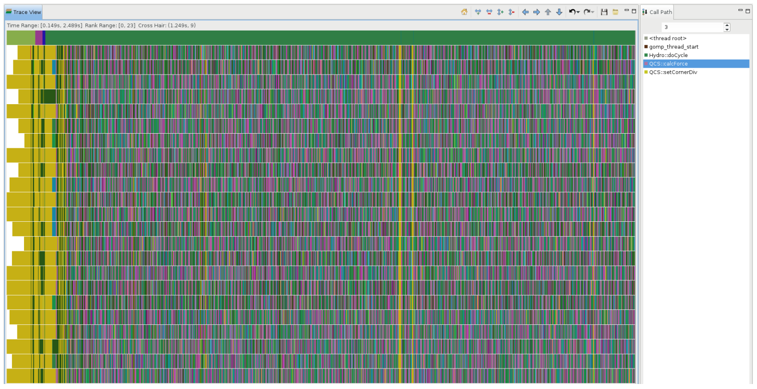
Figure 15: *A summary of the cycle usage over time with the calcForce function highlighted with stripes*

HPCToolkit makes use of several underlying measurement libraries to offer a wide range of performance metrics to the user. In addition, to measuring time with standard timers, the user can make use of Perf or PAPI to access hardware counters. The tool can also be used with a wide variety of languages (C, C++, Fortran, CUDA), parallel libraries (OpenMP, MPI, Cilk, TBB, ...), and trace callpaths into those libraries even down into the kernel. These functionalities give great insight to analysts, in particular when the problems involve a poor choice of library or runtime environment.

Additionally, in [73], the authors extend HPCToolkit to include data-centric analysis. The type of measurement tracks data structures (typically large arrays) rather than sections of code as is typical in these tools. Data-centric analysis is particularly useful for identifying locality issues across caches and NUMA sections of a processor.

HPCToolkit also provides a tracing functionality, but it is not the same as the tracing provided by other tools. This tracing records a timestamp and call path for each process and thread for each sample taken. By doing so, HPCToolkit can show the user a high-level view of what the application is doing at any time during the run and at any depth of the callstack. This view enables the user to identify places where there is contention, load imbalance, insufficient parallelism, and many other challenges that are not available in a fine-grained view.

The tracing tool in HPCToolkit provides a unique and useful method for getting an overview of how the program executes. These results can be viewed using the trace viewer application which allows the user to explore the execution along three axes: real time of samples, thread, and callpath depth. Figure 14 shows the basic tracing view for PENNANT. The x-axis represents time, while each row is one thread. The slivers of color are individual samples with each color as a different callpath and callpath depth can be manipulated with the menu on the right.



Time Range: [0.149s, 2.489s] Rank Range: [0, 23] Cross Hair: (1.249s, 9)

3

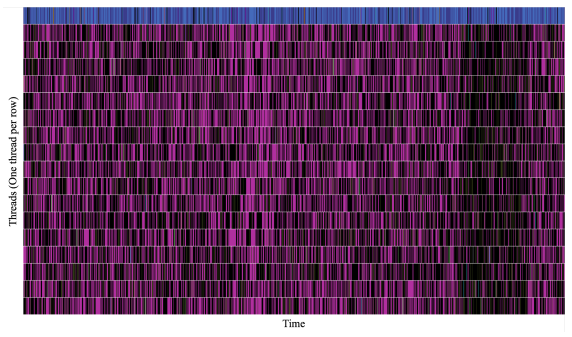
■ <thread root>
■ gomp_thread_start
■ Hydro::doCycle
■ QCS::calcForce
■ QCS::setCornerDiv

Figure 14: *A section of the trace view for the PENNANT mini-app based on HPCToolkit data.*

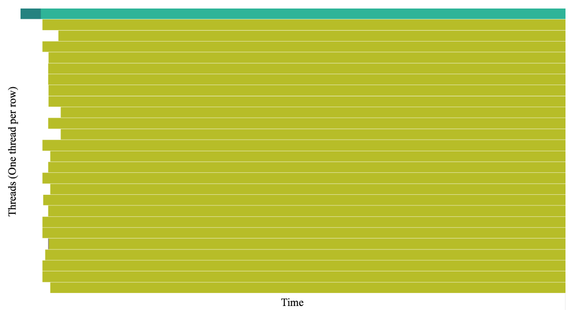
Another view of the trace shows summaries of each time-step so that the user can easily see what function is the biggest user of resources at any one time. Figure 15 shows part of this view with the *calcForce* function highlighted with stripes. This function uses roughly 25% of the cycles for any one section, suggesting that is could be a good target for optimization.

For an example of using the tracing tool to find bottlenecks, we use an intentionally slow version of matrix multiply. Listing 1 shows a matrix multiply with the innermost loop parallelized and a barrier after each inner loop. This version has significant overhead associated with allocating work to threads and coordinating them at the barrier.

A zoomed in version of the trace view is available in Figure 16(a). In this view, we adjusted the colors to show functions associated with OMP barriers as black. So all the time in black is essentially wasted indicating that we may want to adjust our use of OpenMP. Seeing this inefficiency, we adjust our code as shown in Listing 2. Figure 16(b) shows how the program is impacted. This trace is very boring because the deepest layer of the callpath is all computation, so we know we have succeeded in improving the application efficiency.



(a) A section of the trace view for matrix multiply with OMP barrier in black.



(b) A section of the trace view for the efficient version of matrix multiply.

Figure 16: *Examples of HPCToolkit's tracing output*

Listing 1: matrix multiplication with omp barrier

```
for (i=0; i<N; i++){
  for (j=0; j<M; j++){
    #pragma omp parallel for private(k)
    for (k=0; k<P; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
    #pragma omp barrier
  }
}
```

Listing 2: matrix multiplication without omp barrier

```
#pragma omp parallel for private(i,j,k)
for (i=0; i<N; i++){
  for (j=0; j<M; j++){
    for (k=0; k<P; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

4.8 Caliper

Caliper [13] aims to provide a flexible, easily extensible tool geared for the needs of modern HPC development. It is designed as an API that can link the measurement, data collection, and analysis aspects together. Universal library calls are added to the application at key points, allowing the user to choose different measurement options (i.e. timers, PAPI counters, or no measurement) at runtime. Since the function calls and compilation are universal, new services (i.e. for new architectures or systems) can be offered without impacting previously instrumented applications. Similarly, when a new service is added it must write data in the standard output formats; providing maximum compatibility with analysis and visualization tools. Caliper provides the abstraction between measurement and analysis that has the potential to significantly ease the development of new tools for the future of HPC.

One result of the increased use of HPC is the development of various libraries, in particular, numerical or other low-level computational systems, that allow developers to focus on the unique aspects of their application trusting that the underlying functions will be maintained and optimized for various systems. This method of abstraction provides great benefits to the productivity of software developers but can create challenges for performance engineers. Often these libraries will contain some of the most significant pieces of

Table 2: Features of the tools discussed here.

Tool	Sampling	Instrumentation	Tracing	GUI
Caliper		X		
HPCToolkit	X		X	X
Open| Speedshop	X		X	X
TAU	X	X	X	X

computation in an application but are hidden from analysis by traditional tools. Caliper provides a solution to this challenge by allowing the users to enable or disable the instrumentation of the program at runtime. The authors suggest that this method would allow the distribution of instrumented libraries that can then be easily analyzed during application development and not during production runs.

Lastly, Caliper provides API functions to instrument specific data structures, rather than points in the algorithm. This instrumentation allows for data-centric analysis that explores whether or not data is accessed efficiently, where troublesome latencies occur or similar questions. While not widely used, several other tools provide similar functionality and the increasing gap between computation and memory performance will likely make such analysis more important in the future.

Caliper has many possibilities for extension and improvement on existing analysis work-flows, but there is a significant gap in the data analysis area. This area is a work in progress with very exciting development by the team. Additionally, Caliper is integrated with VTune and TAU, allowing it to collect data based on those tools' functions and display the results through their interfaces.

4.9 Tools Wrapup

The tools discussed here provide a wide range of services that are often complementary. Some tools, in particular, TAU, OpenSpeedshop, and HPCToolkit, provide excellent overviews of the application, so the user can see how it functions as a whole and which areas are best targeted for improvement. This information can then be used to inform a more targeted study of a section of code with a tool such as Caliper or gain more detailed information. Although the tools provide many overlapping services each has unique attributes. Table 2 provides a list of the tools we discussed here and the features they provide.

Evaluating data based on the results from several tools can be difficult, so Score-P [66] provides an instrumentation library that can be used to interact with several tracing and profiling tools. This includes TAU and Vampir (a tracing tool associated with TAU) along with Scalasca [46], a tracing tool for applications running of thousands of processors, and Periscope [47], for online evaluation of large-scale applications.

The measurement tools and techniques discussed in this section provide a vital but limited service to developers. They are purely instruments of measurement that can provide the user with a metric associated with a section of an application. Although a high miss rate associated with a particular function may indicate that function should be improved, this may not be the case. There could be broader algorithmic issues causing the high miss rate, a poor ordering of steps in the application causing cache evictions, or maybe the function is where data is initialized and the misses are compulsory. Current measurement tools cannot diagnose why the performance is poor or how it can be improved. These issues are left to the user.

5 Autotuning

When analysis works, it leads the developer to changes that improve the application's performance. In general, such changes are made manually and are very specific to the application or the systems involved. Some optimizations are broadly explained in Section 3, but for the most part the details relate to the exact algorithms and data structures which are beyond the scope of this review. Instead, we discuss the automated process of optimizing an application, autotuning.

An alternative to determining the best options through analysis is to try many or all of the possible optimizations and choose the best one. By automatically generating code variants and testing each one, a tool can tune the code to a particular architecture. Autotuning can be Incorporated into compilers but usually requires too much time for compiler users and is kept to separate applications. Generally, this process takes significant time, but minimal human effort.

Several types of tools exist. Libraries that make use of tuned code or tune it at installation time, tools that a developer can use to generate tuned code, tools that target application level variations, and Domain Specific Languages (DSLs) that take a high-level description of an algorithm and use compilers and/or autotuners to transform it into efficient code. Most commonly these tools are used for improving the speed of applications, but can additionally be utilized for reducing the power or energy footprint.

5.1 Libraries

A common use for autotuning is to incorporate it into a library that can be called from other programs. At install time the autotuning algorithm can be run, so the library is tuned for that particular system. Such libraries offer an alternative to hand tuning a library for each system and can be called from many different applications.

An early example of this method came from the Basic Linear Algebra Subprograms (BLAS). BLAS is a standard interface for fundamental linear algebra functions that is commonly used across many areas of scientific computing. Vendors could distribute version optimized for their systems allowing performance portability at a cost of licensing. ATLAS (Automatically Tuned Linear Algebra Software[11]) [108] provides an alternative by autotuning the functions at install time. This method allows users to work with optimized libraries across numerous machines without maintaining separate libraries (i.e. between IBM and Intel systems) or paying the vendors for hand tuned versions. Similar to ATLAS, FFTW [44] provided an autotuning platform for Fast Fourier Transform calculations. The success of FFTW resulted in its function API becoming the standard for FFTs in scientific computing. Additionally, vendor distributed libraries such as Intel's MKL and Cray's LibSci for dense linear algebra along with NVIDIA's cuFFT have incorporated autotuning to improve their performance. These examples show how early autotuning spread from a research idea into wide use in HPC.

5.2 Autotuning Code Generators

Often one will want to tune some code that is not available in library form. This is a great opportunity for code generation-style autotuning. These tools generally get hints from the developers of what optimizations to try and then iterate through the possible combinations. Code generating autotuners offer the flexibility of being applied to potentially any application but at the cost of requiring the programmer to annotate the application and taking the time to explore the possible tuning options.

Listing 3: Original loop for matrix vector multiplication

```
void KERNEL(int size){
    float A[size][size]; float B[size]; float C[size];
    initialize_arrays(A, B, C, size);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i] += A[i][j] * B[j];
        }
    }
}
```

A common target for optimization is computationally heavy loops that take up a significant fraction of the runtime. Such loops generally iterate over a large section of memory and have opportunities for improving cache performance or using vector instructions to improve performance. Some optimizations include unrolling of loops to eliminate branches and encourage vectorization and blocking to improve cache performance. Additionally, loop fusion takes multiple similar loops and integrates them into one, so that data is reused better than performing the loops separately. Using Listing 3 as the starting point we show

[11]math-atlas.sourceforge.net

unrolling (Listing 4) and tiling (Listing 5). These loop transformations are well-studied and often used by optimizing compilers or human tuners [53].

Listing 4: Unrolled inner loop for matrix vector multiplication

```
void KERNEL(int size){
    float A[size][size]; float B[size]; float C[size];
    initialize_arrays(A, B, C, size);
    int j = 0;
    int unroll_depth = 8;
    for (int i = 0; i < size; i++) {
        for (j = 0; j < size; j+=unroll_depth) {
            C[i] += A[i][j]   * B[j];
            C[i] += A[i][j+1] * B[j+1];
            C[i] += A[i][j+2] * B[j+2];
            C[i] += A[i][j+3] * B[j+3];
            C[i] += A[i][j+4] * B[j+4];
            C[i] += A[i][j+5] * B[j+5];
            C[i] += A[i][j+6] * B[j+6];
            C[i] += A[i][j+7] * B[j+7];
        }
        // finish the part that doesn't fit in the unroll
        for (j = (j - unroll_depth) + 1; j < size; j++) {
            C[i] += A[i][j] * B[j];
        }
    }
}
```

For a given loop there is a vast number of combinations of loop transformations that may improve the performance; too many for hand tuners or compilers to reasonably explore. Autotuners such as CHiLL [23], Orio [83], [54], and POET [113] provide a way to explore all or many of the possibilities. These tools take hints from the developer about which transformations to try and what types of input may be expected and then test the variations using exhaustive or optimizing search techniques to explore the possibilities. Orio has the advantage of including optimizations such as OpenMP pragmas, variations of compiler flags, and GPU versions for CUDA and OpenCL [19]. The process is time-consuming but can provide an optimal loop for a given machine.

Listing 5: Tiled loop for matrix vector multiplication

```
void KERNEL(int size){
float A[size][size]; float B[size]; float C[size];
initialize_arrays(A, B, C, size);
int i, j, it, jt;
int tile_size  = 8;
for (it = 0; it < size; it+= tile_size) {
    for (jt = 0; jt < size; jt+= tile_size) {

        for (i = it; i < min(it+tile_size, size); i++) {
            for (j = jt; j < min(jt+tile_size, size); j++) {
                C[i] += A[i][j] * B[j];
            }
        }
    }
}
}
```

While CHiLL and Orio are designed to work on arbitrary loops, some researchers have preferred to focus on more specific domains. For example, Datta et al. [31] present a code-generating autotuning framework that specifically targets stencil algorithms commonly used in structured mesh computations. Similarly, Gates et al. [45] tune the data layout to improve the performance of batch Cholesky Factorization on GPUs. A more general case is OpenTuner [5] which is a framework for creating program specific autotuners.

The above examples are all 'empirical' autotuners. This type generates different code variations and the runs the results to see which is the fastest. Empirical tuning is advantageous because the tool and developer can take a naive approach to the architectural details, assuming the empirical results will take those details into account. An alternative is to use analytic or model based autotuning which will automatically develop models of the system and application and produce a version of the application that is predicted to be the best by those models. The advantage to this approach is that it avoids the time required to run numerous empirical experiments and allows tuning for architectures that the user cannot yet access. This approach is taken in compilers to improve results and also in some autotuning systems [115], [28], [61]. These models tend to be limited by a lack of knowledge concerning different inputs or other run-time characteristics. Other methods use the model to restrict the search area and then use empirical tuning as a final step. For example, the authors of [70] use a model of the program to limit the search space and then explore those options empirically. The authors in [11] utilize machine learning based on empirical runs to create a more accurate model of the program. The PLuTo system [14] combines the methods to optimize programs for both data locality and parallelism.

Online autotuning [102] offers a way to hide the overhead of empirical tuning. At runtime the autotuner explores the search space, switching code variants between loop iterations and keeping track of the results. As the program progresses the overall performance improves although some iterations maybe less good than others. Search techniques play an important role in mitigating the risk of using particularly bad variants.

5.3 DSLs

Domain specific languages combine some of the characteristics of autotuned libraries and code generated autotuners. Domain specific constructs are used to enable high-level programming of an application. The compiler can then use information inherent to those constructs or assumptions based on the domain to make optimizations that would not otherwise be possible.

As an alternative to tuned or autotuned linear algebra libraries, there are several DSLs that will generate linear algebra code from higher level descriptions. BTO Blas [82] takes matrix operations as input and explore various loop arrangements and transformations to find the ideal computation. The Tensor Algebra Compiler [65] provides a similar service for dense and sparse tensors (matrices with more than 2 dimensions). In [42], the authors look at mathematical optimization rather than code-based changes. The possible re-orderings of the operations are considered to find the least computationally intensive way to get the result.

Pochoir [100] and Patus [29] are DSLs for stencil computations. The compiler provided by Pochoir translates high-level stencil descriptions into parallel Cilk code with the aim of improving parallelism while maintaining cache efficiency. Patus takes a different approach, using code generation and autotuning to transform the high-level descriptions into efficient code.

Similar to DSLs are frameworks that can target multiple diverse architecture types. Specifically, GPUs require significantly different programming techniques or languages than CPUs which inhibits performance portability. Frameworks such as Raja [59] and Kokkos [39] take one c++ based constructs and can use those to create multiple versions for threaded CPU, SIMD CPU, or GPU backends. NVidia provides similar functionality through Thrust [84]. Such frameworks provide a simple way for programmers to target multiple architectures and parallelization techniques without individually writing several versions of the application. Additionally, Apollo [9] allows online autotuning to choose the best variant at runtime.

5.4 Application level tuning

Often there are many ways to get a result in an application and depending on the input or the architecture one may be more efficient than another. For example, storing intermediate results can improve performance, unless the problem size is too large to fit in some level of cache or memory. A memory constraint that is unknown until runtime could make it more advantageous to choose an algorithm that recalculates intermediates many times. Choosing the correct algorithm can be done by application level autotuning.

Application level autotuning consists of selecting the ideal code variant from a selection based on information about architecture or input. Orio, the empirical loop transformation tuner mentioned above, achieves a simple version of this technique with conditional blocks based on input sizes. A more advanced version of this method is used in [4] and [90] which targets different levels of the memory hierarchy with the variants and choosing the problem size cutoffs with other autotuning methods.

Information beyond problem size, such as meta information about matrices that are to be solved, can create enough input to justify using machine learning to build a model for the selection of code variants. In [79] and [34] the authors allow developers to define arbitrary variants through autotuning or by hand along with heuristics about the inputs that can then be used to build the machine learning model. The Lighthouse project [81], [97], [78], takes a different approach; it targets applications (i.e. the many sparse linear solvers available in PETSc) that have many well-defined algorithms and provides the machine learning model that the developer can incorporate into an application to avoid making difficult and often under-informed choices about which solution is ideal.

While inputs can be important to the choice of variant, other factors play significant roles as well. The authors of [64] discuss how speedup and efficiency can be at odds when approaching the limits of strong scaling and present a tuning framework that chooses variants based on the number of threads available while optimizing both speedup and efficiency.

Obviously, a great deal of relevant information about input values may not be available until runtime, so autotuners such as ActiveHarmony [102] and ADAPT [106] can do so online. These tools test variant at runtime, using the program iterations as individual experiments. As they collect information, they select a default variant to be used.

5.5 Power and Energy Tuning

As HPC has progressed towards Exascale computing the importance of energy efficiency has become nearly equal to that of raw performance. Autotuners have followed this progression and many now include multi-objective optimization to minimize both performance and power or energy use. Multi-objective optimization is a difficult problem to solve as the objectives may at times are conflicting or several possibilities may be considered optimal. This optimization problem is discussed in detail with respect to performance tuning in the work by Balaprakash et al. [8]. Rather than address the complex optimization issues, the ANGEL approach [25] uses simpler single objective optimization in a hierarchical method to target first one objective and then another.

Despite the interest in energy targeted tuning, many codes achieve their lowest energy usage when they complete as soon as possible, so the easiest way to tune for energy use is often to tune for performance. When peak power use is important, there may be more use for multi-objective tuning since faster code tends to use more instantaneous power; however having peak power as a constraint on single objective optimization is a simpler way to achieve the same goal. This is not to say that power and energy tuning should be ignored; as supercomputers consume ever more energy and heterogeneous computing complicates the relationship of time and energy use, power and energy tuning may soon become a vital aspect of HPC.

In this survey and in the broader research, autotuning tends to exist separately from other aspects of performance analysis and optimization. Most of the autotuning applications merely time the kernel and select the fastest version. They treat the search as an arbitrary optimization problem without taking into account the wealth of performance analysis knowledge that developers have built over the years. These tools act as a student of performance analysis may; randomly trying optimizations that they have heard work well without fully understanding the context surrounding the optimization and application. An extremely difficult but potentially rewarding research aim would be to incorporate knowledge of performance analysis into autotuners.

6 Conclusion

Performance analysis and optimization for High Performance Computing is a well-developed and exciting field that has many opportunities for new work. Here we discussed background issues of architecture and algorithms, methods of performance analysis, techniques used for measurement, and autotuning. Many of the topics in this review are too often viewed as separate research areas rather than a set of tools and techniques that can be combined by a knowledgeable developer to improve any HPC application on any piece of hardware. Unfortunately, modern performance analysis relies on the knowledge and experience of those analyzing the application without best practices or automated tools to guide a developer through the process of developing and testing potential optimizations. Taking this view of performance analysis and optimization, we can see that algorithms, architecture, analysis tools, and autotuning can be developed into a complementary method of root cause analysis to enable developers to produce the best possible HPC applications.

References

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701.

[2] Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., and Tucker, T. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2014), pp. 154–165.

[3] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), ACM, pp. 483–485.

[4] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. *PetaBricks: a language and compiler for algorithmic choice*, vol. 44. ACM, 2009.

[5] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), ACM, pp. 303–316.

[6] Austin, B., and Wright, N. J. Measurement and interpretation of micro-benchmark and application energy use on the cray xc30. In *Energy Efficient Supercomputing Workshop (E2SC), 2014* (2014), IEEE, pp. 51–59.

[7] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[8] Balaprakash, P., Tiwari, A., and Wild, S. M. Multi objective optimization of hpc kernels for performance, power, and energy. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems* (2013), Springer, pp. 239–260.

[9] Beckingsale, D., Pearce, O., Laguna, I., and Gamblin, T. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International* (2017), IEEE, pp. 307–316.

[10] Bell, R., Malony, A. D., and Shende, S. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *European Conference on Parallel Processing* (2003), Springer, pp. 17–26.

[11] Bergstra, J., Pinto, N., and Cox, D. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012* (2012), IEEE, pp. 1–9.

[12] Berkowitz, E., Clark, M., Gambhir, A., McElvain, K., Nicholson, A., Rinaldi, E., Vranas, P., Walker-Loud, A., Chang, C. C., Joó, B., et al. Simulating the weak death of the neutron in a femtoscale universe with near-exascale computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), IEEE Press, p. 55.

[13] Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. Caliper: performance introspection for hpc software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE Press, p. 47.

[14] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)* (2008).

[15] Boushehrinejadmoradi, N., Yoga, A., and Nagarakatte, S. A parallelism profiler with what-if analyses for openmp programs. In *A Parallelism Profiler with What-If Analyses for OpenMP Programs* (2018), IEEE, p. 0.

[16] Bowers, K. J., Albright, B. J., Bergen, B., Yin, L., Barker, K. J., and Kerbyson, D. J. 0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE Press, p. 63.

[17] Bronevetsky, G., Gyllenhaal, J., and De Supinski, B. R. Clomp: Accurately characterizing openmp application overheads. *International journal of parallel programming 37*, 3 (2009), 250–265.

[18] Calotoiu, A., Hoefler, T., Poke, M., and Wolf, F. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), ACM, p. 45.

[19] Chaimov, N., Norris, B., and Malony, A. Toward multi-target autotuning for accelerators. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on* (2014), IEEE, pp. 534–541.

[20] Chan, A., Gropp, W., and Lusk, E. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming 16*, 2-3 (2008), 155–165.

[21] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), Ieee, pp. 44–54.

[22] Che, Y., Zhang, L., Xu, C., Wang, Y., Liu, W., and Wang, Z. Optimization of a parallel cfd code and its performance evaluation on tianhe-1a. *Computing and Informatics 33*, 6 (2015), 1377–1399.

[23] Chen, C., Chame, J., and Hall, M. Chill: A framework for composing high-level loop transformations. Tech. rep., Technical Report 08-897, U. of Southern California, 2008.

[24] Chen, G., and Stenstrom, P. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of the international conference on high performance computing, networking, storage and analysis* (2012), IEEE Computer Society Press, p. 71.

[25] Chen, R. S., and Hollingsworth, J. K. Angel: A hierarchical approach to multi-objective online auto-tuning. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (2015), ACM, p. 4.

[26] Choi, J., Dukhan, M., Liu, X., and Vuduc, R. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International* (2014), IEEE, pp. 447–457.

[27] Choi, J. W., Bedard, D., Fowler, R., and Vuduc, R. A roofline model of energy. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 661–672.

[28] Choi, J. W., Singh, A., and Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM sigplan notices* (2010), vol. 45, ACM, pp. 115–126.

[29] Christen, M., Schenk, O., and Burkhart, H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 676–687.

[30] Curtsinger, C., and Berger, E. D. C oz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 184–197.

[31] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE Press, p. 4.

[32] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design* (2010), ACM, pp. 189–194.

[33] Ding, C., and Zhong, Y. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices* (2003), vol. 38, ACM, pp. 245–257.

[34] Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.-M., and Amarasinghe, S. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 379–390.

[35] Doerfler, D., Deslippe, J., Williams, S., Oliker, L., Cook, B., Kurth, T., Lobet, M., Malas, T., Vay, J.-L., and Vincenti, H. Applying the roofline performance model to the intel xeon phi knights landing processor. In *International Conference on High Performance Computing* (2016), Springer, pp. 339–353.

[36] Dolbeau, R. Theoretical peak flops per instruction set on modern intel cpus.

[37] Dongarra, J. J. The linpack benchmark: An explanation. In *Supercomputing* (1988), Springer, pp. 456–474.

[38] Du Bois, K., Eyerman, S., Sartor, J. B., and Eeckhout, L. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. *ACM SIGARCH Computer Architecture News 41*, 3 (2013), 511–522.

[39] Edwards, H. C., Trott, C. R., and Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing 74*, 12 (2014), 3202–3216.

[40] Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., and Lorenz, D. Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging. In *International Workshop on OpenMP (IWOMP 2013)* (2013).

[41] Eyerman, S., Du Bois, K., and Eeckhout, L. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on* (2012), IEEE, pp. 145–155.

[42] Fabregat-Traver, D., and Bientinesi, P. Application-tailored linear algebra algorithms: A search-based approach. *The International Journal of High Performance Computing Applications 27*, 4 (2013), 426–439.

[43] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE transactions on computers 100*, 9 (1972), 948–960.

[44] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[45] Gates, M., Kurzak, J., Luszczek, P., Pei, Y., and Dongarra, J. Autotuning batch cholesky factorization in cuda with interleaved layout of matrices. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International* (2017), IEEE, pp. 1408–1417.

[46] Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper. 22*, 6 (Apr. 2010), 702–719.

[47] Gerndt, M., and Ott, M. Automatic performance analysis with periscope. *Concurr. Comput. : Pract. Exper. 22*, 6 (Apr. 2010), 736–748.

[48] Gonnet, P. Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism. *Journal of computational chemistry 33*, 1 (2012), 76–81.

[49] Grama, A., Kumar, V., Gupta, A., and Karypis, G. *Introduction to parallel computing.* Pearson Education, 2003.

[50] Gropp, W. D., Kaushik, D. K., Keyes, D. E., and Smith, B. Performance modeling and tuning of an unstructured mesh cfd application. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000), IEEE Computer Society, p. 34.

[51] Gropp, W. D., Kaushik, D. K., Keyes, D. E., and Smith, B. F. Toward realistic performance bounds for implicit cfd codes. In *Proceedings of parallel CFD* (1999), vol. 99, Citeseer, pp. 233–240.

[52] Hadade, I., Wang, F., Carnevale, M., and di Mare, L. Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures. *Computer Physics Communications 235* (2019), 305–323.

[53] Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., and Khan, M. M. Loop transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing* (2009), Springer, pp. 50–64.

[54] Hartono, A., Norris, B., and Sadayappan, P. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–11.

[55] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[56] Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. Improving Performance via Mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories, 2009.

[57] Hill, M. D., and Smith, A. J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers 38*, 12 (1989), 1612–1630.

[58] Hollingsworth, J. K. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (1998), 1029–1040.

[59] Hornung, R. D., and Keasler, J. A. The raja portability layer: overview and status. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[60] Ilic, A., Pratas, F., and Sousa, L. Beyond the roofline: cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Transactions on Computers 66*, 1 (2017), 52–58.

[61] Im, E.-J., Yelick, K., and Vuduc, R. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications 18*, 1 (2004), 135–158.

[62] Jalby, W., Lemuet, C., and Le Pasteur, X. Wbtk: a new set of microbenchmarks to explore memory system performance for scientific computing. *The International Journal of High Performance Computing Applications 18*, 2 (2004), 211–224.

[63] Jespersen, D. C. Acceleration of a cfd code with a gpu. *Scientific Programming 18*, 3-4 (2010), 193–201.

[64] Jordan, H., Thoman, P., Durillo, J. J., Pellegrini, S., Gschwandtner, P., Fahringer, T., and Moritsch, H. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (2012), IEEE, pp. 1–12.

[65] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 77.

[66] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[67] Lacasta, A., Morales-Hernández, M., Murillo, J., and García-Navarro, P. An optimized gpu implementation of a 2d free surface simulation model on unstructured meshes. *Advances in engineering software 78* (2014), 1–15.

[68] Lacasta, A., Morales-Hernández, M., Murillo, J., and García-Navarro, P. Gpu implementation of the 2d shallow water equations for the simulation of rainfall/runoff events. *Environmental Earth Sciences 74*, 11 (2015), 7295–7305.

[69] Levesque, J., and Vose, A. *Programming for Hybrid Multi/Manycore MPP Systems*. Chapman and Hall/CRC, 2017.

[70] Lim, R. V., Norris, B., and Malony, A. D. Autotuning gpu kernels via static and predictive analysis. *arXiv preprint arXiv:1701.08547* (2017).

[71] Lindlan, K. A., Cuny, J., Malony, A. D., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing, ACM/IEEE 2000 Conference* (2000), IEEE, pp. 49–49.

[72] Liu, Q., Qin, Y., and Li, G. Fast simulation of large-scale floods based on gpu parallel computing. *Water 10*, 5 (2018), 589.

[73] Liu, X., and Mellor-Crummey, J. A data-centric profiler for parallel programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for* (2013), IEEE, pp. 1–12.

[74] Lo, Y. J., Williams, S., Van Straalen, B., Ligocki, T. J., Cordery, M. J., Wright, N. J., Hall, M. W., and Oliker, L. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems* (2014), Springer, pp. 129–148.

[75] McCalpin, J. D. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what 22* (1995).

[76] McCool, M., Robison, A., and Reinders, J. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[77] Meng, J., Morozov, V. A., Vishwanath, V., and Kumaran, K. Dataflow-driven gpu performance projection for multi-kernel transformations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 82.

[78] Motter, P., Sood, K., Jessup, E., and Norris, B. Lighthouse: an automated solver selection tool. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering* (2015), ACM, pp. 16–24.

[79] Muralidharan, S., Shantharam, M., Hall, M., Garland, M., and Catanzaro, B. Nitro: A framework for adaptive code variant tuning. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International* (2014), IEEE, pp. 501–512.

[80] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. Vampir: Visualization and analysis of mpi resources.

[81] Nair, R., Bernstein, S., Jessup, E., and Norris, B. Generating customized sparse eigenvalue solutions with lighthouse. In *Proceedings of the Ninth International Multi-Conference on Computing in the Global Information Technology* (2014).

[82] Nelson, T., Belter, G., Siek, J. G., Jessup, E., and Norris, B. Reliable generation of high-performance matrix algebra. *ACM Transactions on Mathematical Software (TOMS) 41*, 3 (2015), 18.

[83] Norris, B., Hartono, A., and Gropp, W. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462. Also available as Preprint ANL/MCS-P1392-0107.

[84] NVIDIACorporation. Nvidia's thrust gpu library, https://developer.nvidia.com/thrust. Accessed: 2018-11-3.

[85] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface.* Newnes, 2013.

[86] Quinlan, D. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters 10*, 02n03 (2000), 215–226.

[87] Rahman, S. F., Guo, J., and Yi, Q. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (2011), ACM, pp. 107–116.

[88] Rashti, M., Sabin, G., Vansickle, D., and Norris, B. Wattprof: A flexible platform for fine-grained hpc power profiling. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on* (2015), IEEE, pp. 698–705.

[89] Reguly, I. Z., László, E., Mudalige, G. R., and Giles, M. B. Vectorizing unstructured mesh computations for many-core architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores* (2014), ACM, p. 39.

[90] Ren, M., Park, J. Y., Houston, M., Aiken, A., and Dally, W. J. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), ACM, pp. 280–291.

[91] Rosales, C. Porting to the intel xeon phi: Opportunities and challenges. In *Extreme Scaling Workshop (XSW), 2013* (2013), IEEE, pp. 1–7.

[92] Sahelices, B., Ibáñez, P., Viñals, V., and Llabería, J. M. A methodology to characterize critical section bottlenecks in dsm multiprocessors. In *European Conference on Parallel Processing* (2009), Springer, pp. 149–161.

[93] Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., and Cranford, S. Open— speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming 16*, 2-3 (2008), 105–121.

[94] Shaw, D. E. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *Journal of computational chemistry 26*, 13 (2005), 1318–1328.

[95] Shende, S. S., and Malony, A. D. The tau parallel performance system. *The International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[96] Snyder, L., and Lin, C. *Principles of Parallel Programming, ePub*. Pearson Higher Ed, 2011.

[97] Sood, K., Norris, B., and Jessup, E. Iterative solver selection techniques for sparse linear systems.

[98] Suetterlein, J. D., Landwehr, J., Marquez, A., Manzano, J., and Gao, G. R. Extending the roofline model for asynchronous many-task runtimes. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on* (2016), IEEE, pp. 493–496.

[99] Tallent, N. R., Adhianto, L., and Mellor-Crummey, J. M. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[100] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 117–128.

[101] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[102] Tiwari, A., and Hollingsworth, J. K. Online adaptive code generation and tuning. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 879–892.

[103] Unat, D., Chan, C., Zhang, W., Williams, S., Bachan, J., Bell, J., and Shalf, J. Exasat: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications 29*, 2 (2015), 209–232.

[104] Uphoff, C., Rettenberger, S., Bader, M., Madden, E. H., Ulrich, T., Wollherr, S., and Gabriel, A.-A. Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 21.

[105] Vetter, J. S., and Yoo, A. An empirical performance evaluation of scalable scientific applications. In *Supercomputing, ACM/IEEE 2002 Conference* (2002), IEEE, pp. 16–16.

[106] Voss, M. J., and Eigemann, R. High-level adaptive program optimization with adapt. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 93–102.

[107] Vuduc, R., Gyulassy, A., Demmel, J. W., and Yelick, K. A. Memory hierarchy optimizations and performance bounds for sparse a t ax. In *International Conference on Computational Science* (2003), Springer, pp. 705–714.

[108] Whaley, R. C., and Petitet, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35*, 2 (February 2005), 101–121. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[109] White, B. S., McKee, S. A., de Supinski, B. R., Miller, B., Quinlan, D., and Schulz, M. Improving the computational intensity of unstructured mesh applications. In *proceedings of the 19th annual international conference on Supercomputing* (2005), ACM, pp. 341–350.

[110] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM 52*, 4 (2009), 65–76.

[111] Wu, C. E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., and Gropp, W. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing* (November 2000).

[112] Yang, C.-Q., and Miller, B. P. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed* (1988), IEEE, pp. 366–373.

[113] Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. Poet: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–8.

[114] Yoga, A., and Nagarakatte, S. A fast causal profiler for task parallel programs. *arXiv preprint arXiv:1705.01522* (2017).

[115] Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. A comparison of empirical and model-driven optimization. *ACM SIGPLAN Notices 38*, 5 (2003), 63–76.