CHAPTER 3

Top-Down Software Optimization

The tuning of a previously unoptimized hardware/software combination is a difficult task, one that even experts struggle with. Anything can go wrong here, from the proper setup to the compilation and execution of individual machine instructions. It is, therefore, of paramount importance to follow a logical and systematic approach to improve performance incrementally, continuously exposing the next bottleneck to be fixed.

This chapter provides such a framework. We will talk very little here about what and how to tune but, rather, leave that to subsequent chapters to consider in detail. We will instead specify the necessary requirements for the workload, application, and benchmarking; and we will provide a systematic staged tuning process, the so-called *top-down approach*. In this process, the performance tuning is considered at three different levels: system, application, and microarchitecture. Each level will be tuned iteratively to convergence, possibly exposing further bottlenecks at other levels.

The Three Levels and Their Impact on Performance

Most people think about performance tuning of HPC applications as the process of tuning the actual source code, but as we shall see, this is only part of the story.

We discussed latency and throughput in Chapter 2. Let us have a look at the typical access latency and throughput for different components in an HPC system that was discussed there. This information is summarized in Table 3-1, with a few numbers deliberately rounded to the nearest order of magnitude.

Table 3-1 shows a trend of diminishing latency and increasing throughput as we move closer and closer to the execution of instructions. Indeed, the whole process might be thought of as a pipeline provisioning data to the processor core, delivering it through the cache hierarchy from the operating system memory, or even farther away from the external node's memory or a hard disk.

Performance follows the weakest-link paradigm: if one stage of the pipeline does not work according to expectations, the rest of the pipeline will starve. While optimizing this pipeline, we should start with the biggest potential bottlenecks first—at the top of this list, working our way down, as shown in Figure 3-1. Indeed, it makes little sense to start working on the branch misprediction impact while the application spends most of its time in the network communication or cache misses. Once we have made sure data is available in the cache, a continuously occurring branch misprediction does have a huge relative impact.

Table 3-1. *Memory Technologies and Their Latency and Throughput (to the Order of Magnitude)*

Component	Typical Latency	Typical Throughput
Local SATA HDD	~1 ms	100 MB/s
Local SATA SSD	~1 ms	500 MB/s
1GB Ethernet	~15 us	100 MB/s
10GB Ethernet	~4 us	1 GB/s
Infiniband FDR	1.5 us	~6.5 GB/s
Local memory (loaded)	~250 ns	~100 GB/s
Local memory (idle)	~60 ns	0 GB/s
Remote memory (idle)	~100 ns	0 GB/s
QPI (intersocket)	~100 ns*	~64 GB/s
L3 cache access	10-25 ns	~160 GB/s
L2 cache access	~5 ns	~160 GB/s
L1 cache access	2-3 ns	~240 GB/s
XOR instruction	~6 ns	~2.5 Ginstruction/s
Branch misprediction	~7 ns	-
SIMD Division instruction	~16 ns	~1 Ginstruction/s

**QPI remote connection latency is hardly observable on the backdrop of the remote memory latency mentioned above.*

Considering Table 3-1, the tuning of a system/software combination may be intuitively broken down into three stages, which are roughly ordered according to the data flow and their *impact time*—that is, the time impact that an inefficiently working part could make on the execution:

- *System*: This is the computer hardware and system software as such and all that brings it to life: the hard disk, the network interfaces, the memory, the BIOS, the operating system, the job manager, the cooling system, and the processor. All of these components require proper setup and configuration for the considered application workload to deliver the expected performance.
- *Application*: This is the part that the user is most exposed to, since this is what he writes or modifies as source code. The application level comprises the algorithmic implementation, the use of external application programming interfaces (APIs), locks, heap, stack, and so on. One central point of the application level is the proper management of data and the access thereto. In particular, this includes the parallelization in two flavors: the shared and distributed memory programming.
- *Microarchitecture*: For most people this is the most obscure level. It is concerned with the efficient use of the processor-internal resources by the application. For example, how efficient is the processor interpreting the strange hex numbers in your binary? How many instructions does the processor complete per cycle? Does an instruction wait most of the time for another one to complete? Is the processor able to predict the conditional branches in your code? Generally, one does not want to know about all of this, but this is where the battle is decided at the last stage of the optimization process.

It is important to understand that bottlenecks in the higher levels may hide bottlenecks in the lower ones. On the other hand, improvements in the lower levels can create bottlenecks at the higher levels. Figure 3-1 shows an overview of the individual levels.

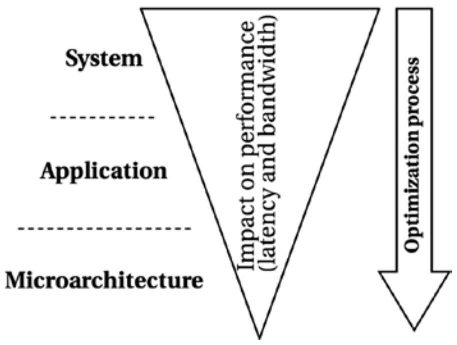

Figure 3-1. *Bottleneck levels and their impact on performance of applications*

System Level

Before worrying about the code of your application, the most important and impactful tuning can be achieved looking at the system components of the compute node, interconnect, and storage. No matter how advanced and skillfully implemented an algorithm is, a wrongly configured network, a forgotten file I/O, or a misplaced memory module in a NUMA system can undo all the effort you put into careful programming.

In many cases, you will be using a system that is in good shape. Particularly if you are a user of an HPC compute center, your system administrators will have taken care in choosing the components and their sound setup. Still, not even the most adept system administrator is immune to a hard disk failure, the cooling deficits of an open rack door, or a bug in a freshly installed network driver. No matter how well your system seems to be maintained, you want to make sure it really does perform to its specification.

You'll find a detailed description of the system tuning in Chapter 4, but here we give an overview of the components and tools. For an HPC system, the hardware components affecting performance at a system level are mostly as follows:

- *Storage and file systems*: As the most of HPC problems deal with large amounts of data, an effective scaling storage hierarchy is critical for application performance and scaling. If storage is inadequate in terms of bandwidth or access latency, it may introduce serialization into the entire application. Taking into account Amdahl's Law (discussed in Chapter 2), this should be considered as the first optimization opportunity.

- *Cluster interconnection hardware and software*: HPC applications do not only demand high bandwidth and low latency for point-to-point exchanges. They also demand advanced capabilities to support collective communications between very large numbers of nodes. A single parameter set wrongly here may completely change the relevant performance characteristics of the network.

- *Random access memory (RAM)*: The RAM attached to the integrated memory controller of a CPU comes in packages called dual-lnline memory modules (DIMMs). The memory controller supports a number of channels that can be populated with several DIMMs. At the same time, different specifications of DIMMs may be supported by the memory controller, such as DIMMs of different sizes in the same channel. Asymmetry in either size or placement of the DIMMs may result in substantial performance degradation.

- *Platform compute/memory balance*: As discussed in Chapter 2, each system has its compute/memory performance balance that can be visualized by the Roofline model. Depending on the specific platform configuration (including the number of cores, their speed and capabilities, and the memory type and speed), the application may end up being memory or compute bound, and these specific platform characteristics will define the application performance.

- *Basic input-output system (BIOS)*: The BIOS is used to bootstrap the system (that is, starting the OS without having full knowledge of the components used), but more importantly, it is also used to configure certain hardware features that can only be set at the boot time. Examples for such features are:
    - *NUMA mode*: Does the BIOS present the system memory as local to a socket or as one homogeneous memory region? Inefficient memory initialization may introduce significant system-level bottlenecks for particular applications.
    - *Processor and RAM frequencies*: The central processor unit (CPU) and RAM can operate under different frequency policies. The CPU, for instance, will try to assume a low-frequency state if no activity is detected, so as to save energy. Latest CPU and RAM specifications need to be supported by the BIOS in order to give the best performance. At the same time, CPU frequency variations driven by desire of saving power may lead to unpleasant load-imbalance issues.
- *Operating system (OS)*: The OS seems somewhat misplaced in the hardware category, since it is indeed software. But once you access the memory, you are actually interacting with the OS, since it will abstract the true memory away from you. So, to some degree, the OS is a proxy to hardware and should be treated in the same category. The OS should be kept up to date, and the version installed should support the features of the CPU and the rest of the system that are essential for performance. For instance, the use of the advanced vector extensions (AVX) and NUMA must be supported by the OS. Apart from this, the most critical point from the OS perspective is the drivers that allow hardware components to be operated from the user space. Examples of this are InfiniBand network cards, hard disk interfaces, and so forth.

All of these components need to be tested and benchmarked. A detailed guide on how to identify, find root causes for, and fix system level bottlenecks is provided in Chapter 4.

■ **Note** System-level performance impact 2x–10x.

Application Level

After the bottlenecks at the system level are successfully cleared, the next category we enter is the application level: we are actually getting our hands on the code here! Application-level tuning is more complicated than system level because it requires a certain degree of understanding of algorithmic details. At the system level, we dealt with standard components—CPUs, OS, network cards, and so on. We rarely can change anything about them, but they need to be carefully chosen and correctly set up. At the application level, things change. Software is seldom made from standard components: most of its functionality is different from all other software. The essential part causing this differentiation is the algorithm(s) used and the implementation thereof.

Note that optimization should not mean a major rewrite. You don't want to change the general algorithm as such. A finite difference program should remain that way, even if finite elements might be more suitable. We are, rather, talking about optimizing the algorithm at hand and the plethora of smaller algorithms that it is built from.

Working Against the Memory Wall

As explained in Chapter 2, performance of modern HPC systems comes from two main sources: SIMD vectorization and parallelization. Both need to be considered at the application level. One central problem still needs to be addressed, however: the divergence of processor and memory performance. Moore's Law promises doubling of the number of transistors on a fixed silicon area roughly every two years.[1] This implies to some degree a doubling of performance as well, because when you talk about doubling the number of processing cores on a chip, you have twice the available space. Even if the number of cores doesn't double, there might be other uses for these additional transistors, such as the AVX1 and AVX2 instruction sets, each of which doubles the floating point operations that can be processed per cycle. Note also that the ever-faster, ever-bigger, and increasingly more efficient caches are part of this development.

When you leave the boundaries of the processor, though, there is no such rapid development. Dynamic RAM (DRAM) performance grows at 1.2x in the same time as the CPU performance grows 2x. The observation that this would lead to a starving CPU was first put forward by W. A. Wulf and S. A. McKee in 1994.[2] It did not come out quite as bad as predicted—more cache levels, larger cache sizes, integrated memory controllers, and more memory channels in combination with the CPU hardware prefetchers mitigated this predicted trend to some degree. Still, there is increasing pressure on the memory subsystem, and so application tuning should focus there. Chapter 8 deals with the respective optimization techniques in detail.

The impact of proper data management may be estimated to be in the order of the cache latency at different levels compared to the latency of RAM access:

$$S = \frac{L_{RAM}}{L_{cache_n}}$$

$$S = \frac{L_{RAM}}{L_{cache_n}}$$

This ratio ranges between 2x and 5x.

■ **Note** Data layout and access performance impact: 2–5x.

The Magic of Vectors

Once data is readily available in the cache, computation itself might become the bottleneck. Now, SIMD vectors come into play. As described in Chapter 2, a SIMD instruction can execute the same arithmetic operation on different elements of a SIMD vector at the same time, as shown in Figure 3-2. Usually, the compiler does a decent job vectorizing code even in a very complex environment, but there are reasons it might not be able to vectorize your code. The Intel Compiler has some very useful reporting that will tell you exactly why the compiler cannot vectorize a particular loop. In the figure, `vmulpd` two SIMD AVX vectors containing four double elements each or one SIMD vector and a memory reference. The assembly code shows that the compiler already unrolls the loop by 4.

C source

```
for(int i=0;i<size;i++){
  c[i]=a[i]*b[i];
}
```

Generated machine instructions

```
..B1.7:
        vmovupd    ymm0, YMMWORD PTR [r15+rdx*8]
        vmovupd    ymm2, YMMWORD PTR [32+r15+rdx*8]
        vmovupd    ymm4, YMMWORD PTR [64+r15+rdx*8]
        vmovupd    ymm6, YMMWORD PTR [96+r15+rdx*8]
        vmulpd     ymm1, ymm0, YMMWORD PTR [rbx+rdx*8]
        vmulpd     ymm3, ymm2, YMMWORD PTR [32+rbx+rdx*8]
        vmulpd     ymm5, ymm4, YMMWORD PTR [64+rbx+rdx*8]
        vmulpd     ymm7, ymm6, YMMWORD PTR [96+rbx+rdx*8]
        vmovupd    YMMWORD PTR [rcx+rdx*8], ymm1
        vmovupd    YMMWORD PTR [32+rcx+rdx*8], ymm3
        vmovupd    YMMWORD PTR [64+rcx+rdx*8], ymm5
        vmovupd    YMMWORD PTR [96+rcx+rdx*8], ymm7
        add        rdx, 16
        cmp        rdx, 10000
        jb         ..B1.7
```

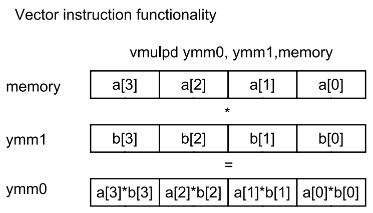

Figure 3-2. *Example for an automatic vectorization by the compiler in C source code, and the resulting assembly instructions*

The impact of vectorization on performance may be estimated by the number of vector elements of a given type that can be processed in parallel. For double precision/AVX, the possible speedup is four times; for single precision, it's eight times.

■ **Note** Vectorization performance impact (double precision): 4x.

Distributed Memory Parallelization

The most important parallelization technique in HPC is *distributed memory parallelization* that enables communication between processes that may not share a common address space (although they can, of course). The benefit of this is immediately clear: you can communicate across physically different computers and gain access to the full power of the massively parallel HPC clusters.

As in the shared-memory approach (discussed in the next section), there is need for a robust library that would abstract all the low-level details and hide from the user the differences between various interconnects available on the market. So, back in the early 1990s, a group of researchers designed and standardized the Message Passing Interface (MPI).[3] The MPI standard defines a language-independent communications protocol as well as syntax and semantics of the routines required for writing portable message-passing programs in Fortran or C/C++; nonstandard bindings are available for many other languages, including C++, Perl, Python, R, and Ruby. The MPI standard is managed by the MPI Forum[4] and is implemented by many commercial and open-source libraries.

The MPI standard was widely used as a programming model for distributed memory systems that were becoming increasingly popular in the early 1990s. As the shared memory architecture of individual systems became more popular, the MPI library evolved as well. The latest MPI-3 standard was issued in September 2012. It added fast remote memory access routines, nonblocking and sparse collective operations, and some other performance-relevant extensions, especially in the shared memory and threading area. However, the programming model clearly remains the distributed memory one with explicit parallelism: the developer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI primitives.

The performance improvement that can be gained from distributed memory parallelization is roughly proportional to the number of compute nodes available, which ranges between 10x and 1000x for the usual compute clusters.

■ **Note** Distributed memory parallelization performance impact: 10–1000x.

Shared Memory Parallelization

The next level to look at is the *shared memory parallelization*. In contrast to the distributed memory programming, where the parallelization unit is normally a process with its own, unique address, space, shared memory programming deals with parallel execution flow in a common address space. Generally, the execution needs to take place on the same physical system. Although processes can also participate in shared memory communication, we generally think about *threads* here.

How do you make a program utilize all processors in a shared memory system? There are multiple libraries providing application program interfaces, or APIs, such as POSIX Threads,[5] that help create and manage multiple application threads. Unfortunately, a lot of threading APIs are either operating system specific (and thus not portable to other OS), or use unique features of the underlying hardware, or are simply too low-level. This is why the HPC community has been building open, portable, and hardware-agnostic programming interfaces to implement threading support in the most popular programming languages: C, C++, and Fortran. The demand from developers for a cross-platform, easy-to-use, threading API helped OpenMP[6] to become the most popular threading API by far. OpenMP consists of a set of compiler directives, as well as library routines and environment variables, that influence the program runtime behavior.

The most recent development of the OpenMP moved the OpenMP API beyond traditional management of pools of threads. In the OpenMP specification version 4.0, released in July 2013, you find support for SIMD optimizations, as well as support for accelerators and coprocessors that architecturally better fit into the distributed memory system type discussed earlier in this chapter. Chapter 5 discusses OpenMP and other threading-related optimization topics, including how to deal with the application-level bottlenecks specific to the shared memory systems programming.

The performance improvement for shared memory parallelization is roughly proportional to the number of cores available per compute node, which is from 10x to 20x in modern server architectures.

■ **Note** Shared memory parallelization performance impact: 10x–20x.

Other Existing Approaches and Methods

So far we have discussed the most popular and widely used parallel programming models for the shared and distributed memory architectures—namely, MPI and OpenMP. However, there are a couple of other methods worth mentioning.

Partitioned Global Address Space (PGAS) is a model that assumes a global memory address space that is logically partitioned, with each portion being local to each process or thread. The PGAS approach attempts to combine the advantages of the MPI programming style for distributed memory systems with the data referencing semantics used in programming shared-memory systems. The PGAS model is the basis for Unified Parallel C,[7] Coarray Fortran[8] (now a part of the Fortran standard), as well as more experimental interfaces and languages.

The SHMEM (Shared Memory) library provides a set of functions similar to MPI.[9] It is available for C and Fortran programming languages. SHMEM routines support remote data transfer, work-shared broadcast and reduction, barrier synchronization, and atomic memory operations.

Intel Thread Building Blocks (TBB)[10] and Intel Cilk Plus[11] aim at making threading and SIMD kind of parallelism easier to use. They represent a new wave of the programming interfaces being developed to address the increased need for parallelization that has reached the mainstream.

Another emerging programming model, applicable for processing large data sets in the so-called Big Data applications, using a parallel, distributed algorithm on a cluster, is MapReduce.[12] A MapReduce program consists of a `Map()` procedure that usually performs filtering and sorting of large arrays of data, and a `Reduce()` procedure that performs a summary or other reduction operation on the results of the `Map()` operation. The MapReduce system middleware—for example, open-source Apache Hadoop[13]—orchestrates the distributed memory servers, runs various tasks in parallel, manages all communication and data transfers between the parts of the system, and provides transparent redundancy and fault tolerance.

One thing to keep in mind when working at the algorithm level is that *you do not need to reinvent the wheel*. If there is a library available that supports the features of the system under consideration, you should use it. A good example is the standard linear algebra operations. Nobody should program a matrix-matrix multiplication or an eigenvalue solver if it is not absolutely necessary and known to deliver a great benefit. The vector-vector, matrix-vector, and matrix-matrix operations are standardized in the so-called Basic Linear Algebra System (BLAS),[14] while the solvers can be addressed via the Linear Algebra Package (LAPACK)[15] interfaces, for which many implementations are available. One of them is Intel Math Kernel Library (Intel MKL), which is, of course, fully vectorized for all available Intel architectures and additionally offers shared memory parallelization.[16]

Microarchitecture Level

Having optimized the system and the algorithmic levels, let's turn now to the problem of how the actual machine instructions are executed by the CPU. According to Table 3-1, microarchitectural changes have the least individual impact in absolute numbers, but when they are accumulated, their impact on performance may be large. Microarchitectural tuning requires a certain understanding of the operation of the individual components of a CPU (discussed in detail in Chapter 7). Here, we restrict ourselves to a very limited overview.

Addressing Pipelines and Execution

The most important features of a modern CPU that need to be addressed at the microarchitectural level are as follows:

- *Pipelining*: The concept of pipelines is addressed at various points in this book, but they play a special role in the design of a CPU. Pipelines are probably the most impactful design pattern in modern computer architecture. The idea is based on the principle of an assembly line: one stage of the pipeline provides input to the following stages. Each stage is specialized in a particular task, which reduces complexity and increases performance. However, a stall at a particular stage may easily spread across the pipeline, both up (for lack of resources) and down (for lack of tasks to address).
- *Out-of-order (OOO) execution*: This is the ability of the CPU to reorder the instructions of a program according to the readiness of the required resources. If *instruction1* depends on the input parameters that are not yet available, the CPU scheduler might schedule execution of the following *instruction2* that meets all dependency requirements.
- *Superscalarity*: Superscalarity describes the implementation of instruction-level parallelism within the CPU. A superscalar CPU features multiple independent pipelines of the same or different

capabilities. The scheduler routes instructions to these pipelines depending on what type the instructions are, and tries to execute them in parallel. In the current Intel architecture codenamed Haswell, for example, the CPU can execute two FMA operations at the same time, reaching throughput of 0.5 cycles/FMA. The total number of instructions that can be executed in parallel is 4/cycle.

- *Branch prediction*: A real problem in pipelined processors is conditional branches, which are jumps to a different part of the instruction flow based on the decision computed at runtime. In this case, the pipeline has to stop issuing instructions until the decision criterion is available. In order to circumvent this problem, a special unit in the CPU predicts the criterion based on the earlier decisions. A special cache is available to store these decisions. In this way, the CPU pipeline can continue operating speculatively, assuming continuation of the instruction flow at the predicted position. If the prediction was wrong, all instructions following the wrongly predicted branch are invalid and the complete pipeline has to be flushed for the execution flow to continue with the correct instruction.

Microarchitectural performance tuning is made more difficult because the actual implementation of the technologies just described can and will change with every processor generation, and might differ considerably across different vendors. Intel's CPUs offer particular hardware functionality to access the information necessary to perform microarchitectural tuning, the so-called performance monitoring unit (PMU). The PMU offers measures that keep track of what exactly happens in the chip—for instance, how many branch predictions have been done and how many have failed. Although you can access the PMU explicitly, it is much more convenient to use a tool that does the PMU programming for you, such as Intel VTune Amplifier XE,[17] Likwid,[18] or the Perf[19] command accessing the PMU via the Linux kernel.

The impact of the microarchitectural optimization can be estimated by the product of the depth of the pipelines and the number of pipelines in the modern processor, ranging in the 10x to 20x area.

■ **Note** The performance impact of microarchitectural tuning can be up to 10x–20x.

Closed-Loop Methodology

One of the most critical factors in the tuning process is the way you load the system. There is some ambivalence in the use of the terms *workload* and *application*. Very often, they are used interchangeably. In general, *application* means the actual code that is executed, whereas *workload* is the task and data that you give to the application. For instance, the application might be `sort.exe`, and the workload might be some data file that contains the names of persons.

Workload, Application, and Baseline

In the current context, we would like to take a simpler view, considering both application and workload in combination simply as the workload. This combination needs to fulfill a number of criteria to be suitable for our purposes:

1. The workload should be *measurable*—that is, there should be quantifiable metric that represents performance of the application. Such a metric can be obvious ones, like execution time or GFLOPS, or more specialized, like simulated nanoseconds/day or transactions/s.

2. Measurement of the performance metric must be *reproducible*. Upon repetitive runs of the application, the resulting numbers need to be consistent. Also, the stress exerted by the application on the system needs to be reproducible.

3. The workload should be *static*—that is, it must not vary over time, and it needs to result in the same performance, regardless of when the workload is executed. In practical terms, performance observed should not vary beyond 1 to 2 percent.

4. The workload must be *representative* of the load imposed upon the system under normal operating conditions. In other words, it should stress those parts of the system that are loaded under normal operation.

In most cases, a real application (or part thereof) and a real compute task will be used for benchmarking. This need not be the case, however, as generating representative stress might be too time-consuming and the application itself might not be designed for benchmarking. Instead, you can consider an artificial benchmark that represents the real situation but gives more detailed information about the performance of individual fractions of the code and executes much faster. A good example is CERN's HEP-SPEC benchmark,[20] a subset of SPEC that mimics the system stress exerted on the CERN computing center.

One thing that must not be forgotten is to establish a *baseline* performance of the workload before you start tuning. Without the baseline, there is no objective starting point against which to compare any consequent potential improvement.

Iterating the Optimization Process

The top-down approach provides structured prioritization of the tuning tasks at hand. We now come to the second important part of this methodology: the *closed-loop* concept. While working at one level, we execute the following scheme:

1. *Gather performance data*: Collect performance data in the metric(s) agreed.

2. *Analyze the data and identify issues*: Focus on the most time-consuming part(s). Begin by looking for unexpected results or numbers that are out of tolerance. Try to fully understand the issue by using appropriate tools. Make sure the analysis does not affect the results.

3. *Generate alternatives to resolve the issue*: Remove the identified bottlenecks. Try to keep focused on one step at a time. Rate the solutions on how difficult they are to be implemented and on their potential payback.

4. *Implement the enhancement*: Change only one thing at a time in order to estimate the magnitude of the individual improvement. Make sure none of the changes causes a slowdown and negated other improvements. Keep track of the changes so you can roll them back, if necessary.

5. *Test the results*: Check whether performance improvements are up to your expectations and that they remove the identified bottleneck.

After the last step, you restart the cycle to identify the next bottleneck (see Figure 3-3). Clearly, this loop is normally infinite, for the time to stop is determined by the amount of time left to do the job.

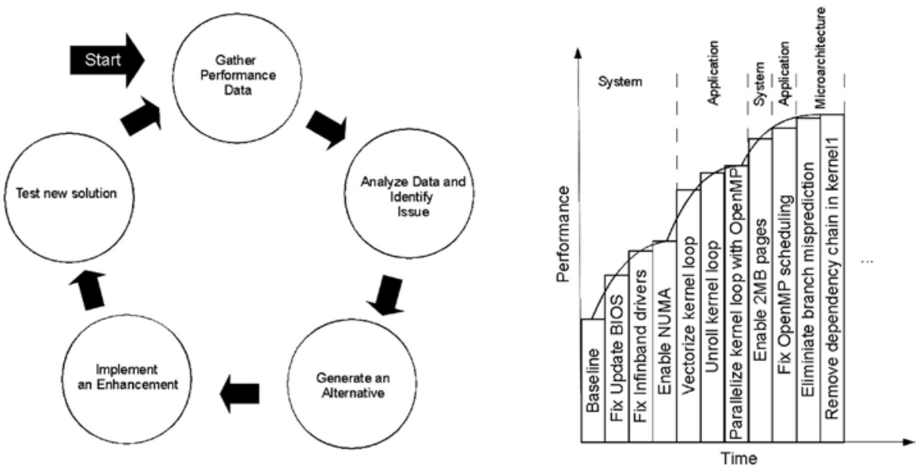

Figure 3-3. *Left: The closed-loop iterative performance optimization cycle. Right: Example performance gains by tuning through various levels*

The right graph in Figure 3-3 shows an artificial performance optimization across different levels. Note that at each level the performance saturates to some degree and that we switch levels when other bottlenecks become dominant. This can also mean going up a level again, since successful tuning of the application level might expose a bottleneck at the system level.

For example, consider an improvement in the OpenMP threading that suddenly causes the memory bandwidth to be boosted. This might very well expose a previously undiscovered bottleneck in the systems memory setup, such as DIMMs in the channels of the memory controllers having different sizes, with the resulting decreased memory bandwidth.

Summary

The methodology presented in this chapter provides a solid process to tune a system consistently with the top-down/closed-loop approach. The main things to remember are to investigate and tune your system through the following different levels:

1. System level (see Chapter 4)
2. Application level, including distributed and shared memory parallelization (see Chapters 5 and 6)
3. Microarchitecture level (see Chapter 7)

Keep iterating at each level until convergence, and proceed to the next level with the biggest impact as long as there is time left.

References

1. G. E. Moore, "Cramming More Components onto Integrated Circuits" *Electronics* 38, no. 8 (19 April 1965): 114–17.
2. W. A. Wulf, and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," 1994, `www.eecs.ucf.edu/~lboloni/Teaching/EEL5708_2006/slides/wulf94.pdf`.
3. MPI Forum, "MPI Documents," `www.mpi-forum.org/docs/docs.html`.
4. "Message Passing Interface Forum," `www.mpi-forum.org`.
5. The Open Group, "Single UNIX Specification, Version 4, 2013 Edition," 2013, `www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=12310`.
6. OpenMP.Org, "OpenMP," `http://openmp.org/wp`.
7. UPC-Lang.Org., "Unified Parallel C," `http://upc-lang.org`.
8. Co-Array.Org, "Co-Array Fortran," `www.co-array.org`.
9. "SHMEM," Wikipedia, the free encyclopedia, `http://en.wikipedia.org/wiki/SHMEM`.

10. Intel Corporation, "Intel Threading Building Blocks (Intel TBB)," http://software.intel.com/en-us/intel-tbb.

11. Intel Corporation, "Intel Cilk Plus," http://software.intel.com/en-us/intel-cilk-plus.

12. G. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, December 2004.

13. "Welcome to Apache Hadoop!," http://hadoop.apache.org.

14. NetLib.Org, "BLAS (Basic Linear Algebra Subprograms)," www.netlib.org/blas/.

15. NetLib.Org, "LAPACK—Linear Algebra PACKage," www.netlib.org/lapack/.

16. Intel Corporation, "Intel Math Kernel Library," http://software.intel.com/en-us/intel-mkl.

17. Intel Corporation, "Intel VTune Amplifier XE 2013," http://software.intel.com/en-us/intel-vtune-amplifier-xe.

18. "Likwid - Lightweight performance tools," http://code.google.com/p/likwid/.

19. "perf (Linux)" Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Perf_(Linux).

20. HEPiX Benchmarking Working Group, "HEP-SPEC06 Benchmark," https://w3.hepix.org/benchmarks/doku.php.