6.10 Multiprocessor benchmarks and performance models

As we saw in COD Chapter 1 (Computer Abstractions and Technology), benchmarking systems is always a sensitive topic, because it is a highly visible way to try to determine which system is better. The results affect not only the sales of commercial systems, but also the reputation of the designers of those systems. Hence, all participants want to win the competition, but they also want to be sure that if someone else wins, they deserve it because they have a genuinely better system. This desire leads to rules to ensure that the benchmark results are not simply engineering tricks for that benchmark, but are instead advances that improve performance of real applications.

To avoid possible tricks, a typical rule is that you can't change the benchmark. The source code and data sets are fixed, and there is a single proper answer. Any deviation from those rules makes the results invalid.

Many multiprocessor benchmarks follow these traditions. A common exception is to be able to increase the size of the problem so that you can run the benchmark on systems with a widely different number of processors. That is, many benchmarks allow weak scaling rather than require strong scaling, even though you must take care when comparing results for programs running different problem sizes.

The figure below gives a summary of several parallel benchmarks, also described below:

- *Linpack* is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the Linpack benchmark. The DGEMM routine in the example in COD Section 3.5 (Floating Point) represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark. It allows weak scaling, letting the user pick any size problem. Moreover, it allows the user to rewrite Linpack in almost any form and in any language, as long as it computes the proper result and performs the same number of floating point operations for a given problem size. Twice a year, the 500 computers with the fastest Linpack performance are published at www.top500.org. The first on this list is considered by the press to be the world's fastest computer.
- *SPECrate* is a throughput metric based on the SPEC CPU benchmarks, such as SPEC CPU 2006 (see COD Chapter 1 (Computer Abstractions and Technology)). Rather than report performance of the individual programs, SPECrate runs many copies of the program simultaneously. Thus, it measures task-level parallelism, as there is no communication between the tasks. You can run as many copies of the programs as you want, so this is again a form of weak scaling.
- *SPLASH* and *SPLASH 2* (Stanford Parallel Applications for Shared Memory) were efforts by researchers at Stanford University in the 1990s to put together a parallel benchmark suite similar in goals to the SPEC CPU benchmark suite. It includes both kernels and applications, including many from the high-performance computing community. This benchmark requires strong scaling, although it comes with two data sets.
- The *NAS (NASA Advanced Supercomputing) parallel benchmarks* were another attempt from the 1990s to benchmark multiprocessors. Taken from computational fluid dynamics, they consist of five kernels. They allow weak scaling by defining a few data sets. Like Linpack, these benchmarks can be rewritten, but the rules require that the programming language can only be C or Fortran.
- The recent *PARSEC (Princeton Application Repository for Shared Memory Computers) benchmark suite* consists of multithreaded programs that use *Pthreads* (POSIX threads) and OpenMP (*Open MultiProcessing*; see COD Section 6.5 (Multicore and other shared memory multiprocessors)). They focus on emerging computational domains and consist of nine applications and three kernels. Eight rely on data parallelism, three rely on pipelined parallelism, and one on unstructured parallelism.
- On the cloud front, the goal of the *Yahoo! Cloud Serving Benchmark* (YCSB) is to compare performance of cloud data services. It offers a framework that makes it easy for a client to benchmark new data services, using Cassandra and HBase as representative examples. [Cooper, 2010]

Pthreads: A UNIX API for creating and manipulating threads. It is structured as a library.

Figure 6.10.1: Examples of parallel benchmarks (COD Figure 6.16).

Benchmark	Scaling?	Reprogram?	Description
Linpack	Weak	Yes	Dense matrix linear algebra [Dongarra, 1979]
SPECrate	Weak	No	Independent job parallelism [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Strong (although offers two problem sizes)	No	Complex 1D FFT Blocked LU Decomposition Blocked Sparse Cholesky Factorization Integer Radix Sort Barnes-Hut Adaptive Fast Multipole Ocean Simulation Hierarchical Radiosity Ray Tracer Volume Renderer Water Simulation with Spatial Data Structure Water Simulation without Spatial Data Structure
NAS Parallel Benchmarks [Bailey et al., 1991]	Weak	Yes (C or Fortran only)	EP: embarrassingly parallel MG: simplified multigrid CG: unstructured grid for a conjugate gradient method FT: 3-D partial differential equation solution using FFTs IS: large integer sort
PARSEC Benchmark Suite [Bienia et al., 2008]	Weak	No	Blackscholes—Option pricing with Black-Scholes PDE Bodytrack—Body tracking of a person Canneal—Simulated cache-aware annealing to optimize routing Dedup—Next-generation compression with data deduplication Facesim—Simulates the motions of a human face Ferret—Content similarity search server Fluidanimate—Fluid dynamics for animation with SPH method Freqmine—Frequent itemset mining Streamcluster—Online clustering of an input stream Swaptions—Pricing of a portfolio of swaptions Vips—Image processing x264—H.264 video encoding
Berkeley Design Patterns [Asanovic et al., 2006]	Strong or Weak	Yes	Finite-State Machine Combinational Logic Graph Traversal Structured Grid Dense Matrix Sparse Matrix Spectral Methods (FFT) Dynamic Programming N-Body MapReduce Backtrack/Branch and Bound Graphical Model Inference Unstructured Grid

The downside of such traditional restrictions to benchmarks is that innovation is chiefly limited to the architecture and compiler. Better data structures, algorithms, programming languages, and so on often cannot be used, since that would give a misleading result. The system could win because of, say, the algorithm, and not because of the hardware or the compiler.

While these guidelines are understandable when the foundations of computing are relatively stable—as they were in the 1990s and the first half of this decade—they are undesirable during a programming revolution. For this revolution to succeed, we need to encourage innovation at all levels.

Researchers at the University of California at Berkeley have advocated one approach. They identified 13 design patterns that they claim will be part of applications of the future. Frameworks or kernels implement these design patterns. Examples are sparse matrices, structured grids, finite-state machines, map reduce, and graph traversal. By keeping the definitions at a high level, they hope to encourage innovations at any level of the system. Thus, the system with the fastest sparse matrix solver is welcome to use any data structure, algorithm, and programming language, in addition to novel architectures and compilers.

Performance models

A topic related to benchmarks is performance models. As we have seen with the increasing architectural diversity in this chapter—multithreading, SIMD, GPUs—it would be especially helpful if we had a simple model that offered insights into the performance of different architectures. It need not be perfect, just insightful.

The 3Cs for cache performance from COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) is an example performance model. It is not a perfect performance model, since it ignores potentially important factors like block size, block allocation policy, and block replacement policy. Moreover, it has quirks. For example, a miss can be ascribed due to capacity in one design, and to a conflict miss in another cache of the same size. Yet 3Cs model has been popular for 25 years, because it offers insight into the behavior of programs, helping both architects and programmers improve their creations based on insights from that model.

To find such a model for parallel computers, let's start with small kernels, like those from the 13 Berkeley design patterns in the figure above. While there are versions with different data types for these kernels, floating point is popular in several implementations. Hence, peak floating-point performance is a limit on the speed of such kernels on a given computer. For multicore chips, peak floating-point performance is the collective peak performance of all the cores on the chip. If there were multiple microprocessors in the system, you would multiply the peak per chip by the total number of chips.

The demands on the memory system can be estimated by dividing this peak floating-point performance by the average number of floating-point operations per byte accessed:

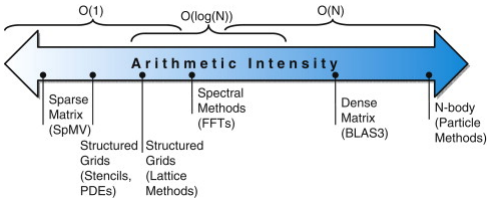
$$\frac{\text{Floating-Point Operations/Sec}}{\text{Floating-Point Operations/Byte}} = \text{Bytes/Sec}$$

The ratio of floating-point operations per byte of memory accessed is called the *arithmetic intensity*. It can be calculated by taking the total number of floating- point operations for a program divided by the total number of data bytes transferred to main memory during program execution. the figure below shows the arithmetic intensity of several of the Berkeley design patterns from the figure above.

Arithmetic intensity: The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory.

Figure 6.10.2: Arithmetic intensity, specified as the number of float-point operations to run the program divided by the number of bytes accessed in main memory [Williams, Waterman, and Patterson 2009] (COD Figure 6.17).

Some kernels have an arithmetic intensity that scales with problem size, such as Dense Matrix, but there are many kernels with arithmetic intensities independent of problem size. For kernels in this former case, weak scaling can lead to different results, since it puts much less demand on the memory system.



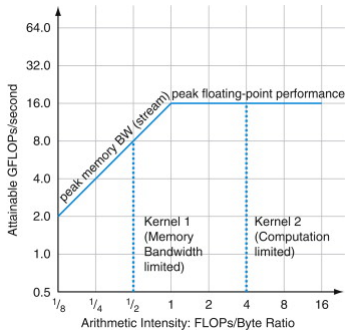
The roofline model

This simple model ties floating-point performance, arithmetic intensity, and memory performance together in a two-dimensional graph [Williams, Waterman, and Patterson 2009]. Peak floating-point performance can be found using the hardware specifications mentioned above. The working sets of the kernels we consider here do not fit in on-chip caches, so peak memory performance may be defined by the memory system behind the caches. One way to find the peak memory performance is the Stream benchmark. (See the *Elaboration* in COD Section 5.2 (Memory technologies)).

The figure below shows the model, which is done once for a computer, not for each kernel. The vertical Y-axis is achievable floating-point performance from 0.5 to 64.0 GFLOPs/second. The horizontal X-axis is arithmetic intensity, varying from 1/8 FLOPs/DRAM byte accessed to 16 FLOPs/DRAM byte accessed. Note that the graph is a log-log scale.

Figure 6.10.3: Roofline Model [Williams, Waterman, and Patterson 2009] (COD Figure 6.18).

This example has a peak floating-point performance of 16 GFLOPS/sec and a peak memory bandwidth of 16 GB/sec from the Stream benchmark. (Since Stream is actually four measurements, this line is the average of the four.) The dotted vertical line in color on the left represents Kernel 1, which has an arithmetic intensity of 0.5 FLOPs/ byte. It is limited by memory bandwidth to no more than 8 GFLOPS/sec on this Opteron X2. The dotted vertical line to the right represents Kernel 2, which has an arithmetic intensity of 4 FLOPs/byte. It is limited only computationally to 16 GFLOPS/s. (This data is based on the AMD Opteron X2 (Revision F) using dual cores running at 2 GHz in a dual socket system.)



For a given kernel, we can find a point on the X-axis based on its arithmetic intensity. If we draw a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line. We can plot a horizontal line showing peak floating-point performance of the computer. Obviously, the actual floating-point performance can be no higher than the horizontal line, since that is a hardware limit.

How could we plot the peak memory performance, which is measured in bytes/second? Since the X-axis is FLOPs/byte and the Y-axis FLOPs/second, bytes/second is just a diagonal line at a 45-degree angle in this figure. Hence, we can plot a third line that gives the maximum floating-point performance that the memory system of that computer can support for a given arithmetic intensity. We can express the limits as a formula to plot the line in the graph in the figure above:

Attainable GFLOPs/sec = Min (Peak Memory BW x Arithmetic Intensity, Peak Floating-Point Performance)

The horizontal and diagonal lines give this simple model its name and indicate its value. The "roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity. Given a roofline of a computer, you can apply it repeatedly, since it doesn't vary by kernel.

If we think of arithmetic intensity as a pole that hits the roof, either it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth, or it hits the flat part of the roof, which means performance is computationally limited. In the figure above, kernel 1 is an example of the former, and kernel 2 is an example of the latter.

Note that the "ridge point", where the diagonal and horizontal roofs meet, offers an interesting insight into the computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance.

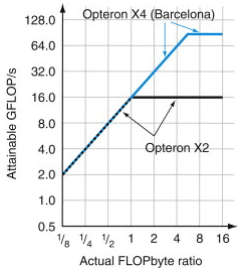
Comparing two generations of opterons

The AMD Opteron X4 (Barcelona) with four cores is the successor to the Opteron X2 with two cores. To simplify board design, they use the same socket. Hence, they have the same DRAM channels and thus the same peak memory bandwidth. In addition to doubling the number of cores, the Opteron X4 also has twice the peak floating-point performance per core: Opteron X4 cores can issue two floating-point SSE2 instructions per clock cycle, while Opteron X2 cores issue at most one. As the two systems we're comparing have similar clock rates—2.2 GHz for Opteron X2 versus 2.3 GHz for Opteron X4—the Opteron X4 has about four times the peak floating-point performance of the Opteron X2 with the same DRAM bandwidth. The Opteron X4 also has a 2MiB L3 cache, which is not found in the Opteron X2.

In the figure below the roofline models for both systems are compared. As we would expect, the ridge point moves to the right, from 1 in the Opteron X2 to 5 in the Opteron X4. Hence, to see a performance gain in the next generation, kernels need an arithmetic intensity higher than 1, or their working sets must fit in the caches of the Opteron X4.

Figure 6.10.4: Roofline models of two generations of Opterons (COD Figure 6.19).

The Opteron X2 roofline, which is the same as in the figure above, is in black, and the Opteron X4 roofline is in color. The bigger ridge point of Opteron X4 means that kernels that were computationally bound on the Opteron X2 could be memory-performance bound on the Opteron X4.



The roofline model gives an upper bound to performance. Suppose your program is far below that bound. What optimizations should you perform, and in what order?

To reduce computational bottlenecks, the following two optimizations can help almost any kernel:

1. *Floating-point operation mix.* Peak floating-point performance for a computer typically requires an equal number of nearly simultaneous additions and multiplications. That balance is necessary either because the computer supports a fused multiply-add instruction (see the Elaboration in COD Section 3.5 (Floating Point)) or because the floating-point unit has an equal number of floating-point adders and floating-point multipliers. The best performance also requires that a significant fraction of the instruction mix is floating- point operations and not integer instructions.

2. *Improve **instruction-level parallelism** and apply SIMD.* For modern architectures, the highest performance comes when fetching, executing, and committing three to four instructions per clock cycle (see COD Section 4.10 (Parallelism via instructions)). The goal for this step is to improve the code from the compiler to increase ILP. One way is by unrolling loops, as we saw in COD Section 4.12 (Going faster: Instruction-level parallelism and matrix multiply). For the x86 architectures, a single AVX instruction can operate on four double precision operands, so they should be used whenever possible (see COD Sections 3.7 (Real stuff: Streaming SIMD extensions and advanced vector extensions in x86) and 3.9 (Going faster: Subword parallelism and matrix multiply)).

To reduce memory bottlenecks, the following two optimizations can help:

1. *Software prefetching.* Usually the highest performance requires keeping many memory operations in flight, which is easier to do by performing **predicting** accesses via software prefetch instructions rather than waiting until the data are required by the computation.
2. *Memory affinity.* Microprocessors today include a memory controller on the same chip with the microprocessor, which improves performance of the **memory hierarchy**. If the system has multiple chips, this means that some addresses go to the DRAM that is local to one chip, and the rest require accesses over the chip interconnect to access the DRAM that is local to another chip. This split results in non-uniform memory accesses, which we described in COD Section 6.5 (Multicore and other shared memory multiprocessors). Accessing memory through another chip lowers performance. This second optimization tries to allocate data and the threads tasked to operate on that data to the same memory-processor pair, so that the processors rarely have to access the memory of the other chips.

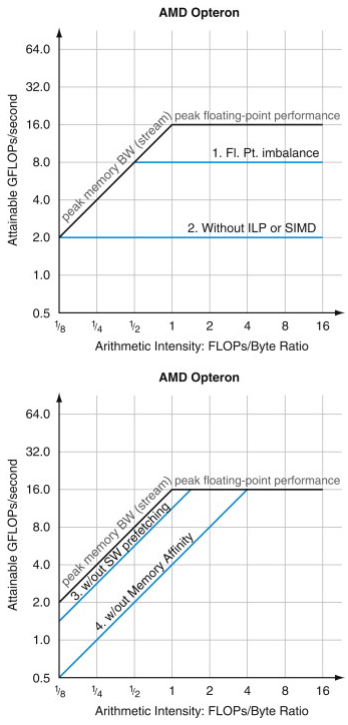
The roofline model can help decide which of these two optimizations to perform and the order in which to perform them. We can think of each of these optimizations as a "ceiling" below the appropriate roofline, meaning that you cannot break through a ceiling without performing the associated optimization.

The computational roofline can be found from the manuals, and the memory roofline can be found from running the Stream benchmark. The computational ceilings, such as floating-point balance, can also come from the manuals for that computer. A memory ceiling, such as memory affinity, requires running experiments on each computer to determine the gap between them. The good news is that this process only need be done once per computer, for once someone characterizes a computer's ceilings, everyone can use the results to prioritize their optimizations for that computer.

The figure below adds ceilings to the roofline model in COD Figure 6.18 (Roofline model …), showing the computational ceilings in the top graph and the memory bandwidth ceilings on the bottom graph. Although the higher ceilings are not labeled with both optimizations, they are implied in this figure; to break through the highest ceiling, you need to have already broken through all the ones below.

Figure 6.10.5: Roofline model with ceilings (COD Figure 6.20).

The top graph shows the computational "ceilings" of 8 GFLOPs/sec if the floating-point operation mix is imbalanced and 2 GFLOPs/sec if the optimizations to increase ILP and SIMD are also missing. The bottom graph shows the memory bandwidth ceilings of 11 GB/ sec without software prefetching and 4.8 GB/sec if memory affinity optimizations are also missing.



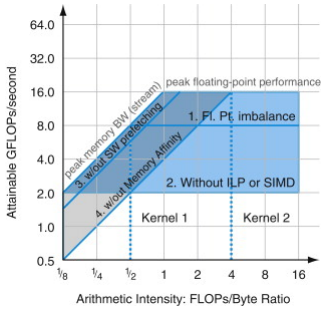
The width of the gap between the ceiling and the next higher limit is the reward for trying that optimization. Thus, the figure above suggests that optimization 2, which improves ILP, has a large benefit for improving computation on that computer, and optimization 4, which improves memory affinity, has a large benefit for improving memory bandwidth on that computer.

The figure below combines the ceilings of the figure above into a single graph. The arithmetic intensity of a kernel determines the optimization region, which in turn suggests which optimizations to try. Note that the computational optimizations and the memory bandwidth optimizations overlap for much of the arithmetic intensity. Three regions are shaded differently in the figure below to indicate the different optimization strategies. For example, Kernel 2 falls in the blue trapezoid on the right, which suggests working only on the computational optimizations. Kernel 1 falls in the blue-gray parallelogram in the middle, which suggests trying both types of optimizations. Moreover, it suggests starting with optimizations 2 and 4. Note that the Kernel 1 vertical lines fall below the floating-point imbalance optimization, so optimization 1 may be unnecessary. If a kernel fell in the gray triangle on the lower left, it would suggest trying just memory optimizations.

Figure 6.10.6: Roofline model with ceiling, overlapping areas shaded, and the two kernels from COD Figure 6.18 (Roofline Model …) (COD Figure 6.21).

Kernels whose arithmetic intensity land in the blue trapezoid on the right should focus on computation optimizations, and kernels whose arithmetic intensity land in the gray triangle in the lower left should focus on memory bandwidth optimizations. Those that land in the blue-gray

parallelogram in the middle need to worry about both. As Kernel 1 falls in the parallelogram in the middle, try optimizing ILP and SIMD, memory affinity, and software prefetching. Kernel 2 falls in the trapezoid on the right, so try optimizing ILP and SIMD and the balance of floating-point operations.



Thus far, we have been assuming that the arithmetic intensity is fixed, but that is not really the case. First, there are kernels where the arithmetic intensity increases with problem size, such as for Dense Matrix and N-body problems (see COD Figure 6.17 (Arithmetic intensity, specified as the number of float-point operations …)). Indeed, this can be a reason that programmers have more success with weak scaling than with strong scaling. Second, the effectiveness of the **memory hierarchy** affects the number of accesses that go to memory, so optimizations that improve cache performance also improve arithmetic intensity. One example is improving temporal locality by unrolling loops and then grouping together statements with similar addresses. Many computers have special cache instructions that allocate data in a cache but do not first fill the data from memory at that address, since it will soon be over-written. Both these optimizations reduce memory traffic, thereby moving the arithmetic intensity pole to the right by a factor of, say, 1.5. This shift right could put the kernel in a different optimization region.

While the examples above show how to help programmers improve performance, architects can also use the model to decide where they should optimize hardware to improve the performance of the kernels that they think will be important.

The next section uses the roofline model to demonstrate the performance difference between a multicore microprocessor and a GPU and to see whether these differences reflect performance of real programs.

Elaboration

The ceilings are ordered so that lower ceilings are easier to optimize. Clearly, a programmer can optimize in any order, but following this sequence reduces the chances of wasting effort on an optimization that has no benefit due to other constraints. Like the 3Cs model, as long as the roofline model delivers on insights, a model can have assumptions that may prove optimistic. For example, roofline assumes the load is balanced between all processors.

Elaboration

An alternative to the Stream benchmark is to use the raw DRAM bandwidth as the roofline. While the raw bandwidth definitely is a hard upper bound, actual memory performance is often so far from that boundary that it's not that useful. That is, no program can go close to that bound. The downside to using Stream is that very careful programming may exceed the Stream results, so the memory roofline may not be as hard a limit as the computational roofline. We stick with Stream because few programmers will be able to deliver more memory bandwidth than Stream discovers.

Elaboration

Although the roofline model shown is for multicore processors, it clearly would work for a uniprocessor as well.

PARTICIPATION ACTIVITY 6.10.1: Check yourself: Benchmarks for parallel computers.

1) The main drawback with conventional approaches to benchmarks for parallel computers is that the rules that ensure fairness also slow software innovation.
   - ○ True
   - ○ False

Provide feedback on this section