

## 6.13 Fallacies and pitfalls

(Original section<sup>1</sup>)

“ For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. ... Demonstration is made of the continued validity of the single processor approach ...  
Gene Amdahl, *“Validity of the single processor approach to achieving large scale computing capabilities,” Spring Joint Computer Conference, 1967*

The many assaults on parallel processing have uncovered numerous fallacies and pitfalls. We cover four here.

### Fallacy: Amdahl's law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that a multiprocessor machine had broken Amdahl's Law. To try to understand the basis of the media reports, let's see the quote that gave us Amdahl's Law [1967, p. 483]:

‘ A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

This statement must still be true; the neglected portion of the program must limit performance. One interpretation of the law leads to the following lemma: portions of every program must be sequential, so there must be an economic upper bound to the number of processors—say, 100. By showing linear speed-up with 1000 processors, this lemma is disproved; hence the claim that Amdahl's Law was broken.

The approach of the researchers was just to use weak scaling: rather than going 1000 times faster on the same data set, they computed 1000 times more work in comparable time. For their algorithm, the sequential portion of the program was constant, independent of the size of the input, and the rest was fully parallel—hence, linear speed-up with 1000 processors.

Amdahl's Law obviously applies to parallel processors. What this research does point out is that one of the main uses of faster computers is to run larger problems. Just be sure that users really care about those problems versus being a justification to buying an expensive computer by finding a problem that simply keeps lots of processors busy.

### Fallacy: Peak performance tracks observed performance.

The supercomputer industry once used this metric in marketing, and the fallacy is exacerbated with parallel machines. Not only are marketers using the nearly unattainable peak performance of a uniprocessor node, but also they are then multiplying it by the total number of processors, assuming perfect speed-up! Amdahl's Law suggests how difficult it is to reach either peak; multiplying the two together multiplies the sins. The roofline model helps put peak performance in perspective.

### Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.

There is a long history of parallel software lagging behind parallel hardware, possibly because the software problems are much harder. We give one example to show the subtlety of the issues, but there are many examples we could choose!

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the Silicon Graphics operating system originally protected the page table with a single lock, assuming that page allocation is infrequent. In a uniprocessor, this does not represent a performance problem. In a multiprocessor, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has a significant impact on overall parallel performance. This performance bottleneck persists even for task-level parallelism. For example, suppose we split the parallel processing program apart into separate jobs and run them, one job per processor, so that there is no sharing between the jobs. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the jobs—so even the independent job performance is poor.

This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminated the problem.

### Fallacy: You can get good vector performance without providing memory bandwidth.

As we saw in the Roofline model, memory bandwidth is quite important to all architectures. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the DAXPY performance of the vector sequence used, since it was memory limited. The Cray-1 performance on Linpack jumped when the compiler used blocking to change the computation so that values could be kept in the vector registers. This approach lowered the number of memory references per FLOP and improved the performance by nearly a factor of two! Thus, the memory bandwidth on the Cray-1 became sufficient for a loop that formerly required more bandwidth, which is just what the Roofline model would predict.

(<sup>\*1</sup>) This section is in original form.

 [Provide feedback on this section](#)