9

High-Performance and Parallel Computing

9.1 Code optimization

To use resources efficiently, the time saved through optimizing code has to be weighed against the human resources required to implement these optimizations. There is no need to worry about the speed of non-critical parts. And sometimes writing well performing code is no more work than writing in badly performing style.

Memory addresses are numbered in a linear manner. Even when an array has two or more indices, its elements are ultimately stored in physical memory in a one-dimensional fashion. The fastest accessible index of an array is for Fortran the first (leftmost) index and for C the last (rightmost) index. Fortran stores data row-wise, C column-wise. Although this is not part of the language standard, it is the universally accepted practice of compiler writers. Reading data along any other index requires jumping between distant memory locations, leading to cache misses.

Obviously, large data should not be unnecessarily duplicated. In C, arrays can only be passed to a function by the address of its first element, but some other languages allow this confusion to happen.

An effort-free way to take advantage of parallelism on any level is index-free notation, available in some languages, such as arithmetic with an array, e.g. `a(:)=b(:)+1`.

Optimization by the Compiler. Almost any compiler provides options for automatic speed optimization. A speedup by a factor of five is not unusual, and is very nice since it requires no work on our part (see example in chapter 8.1). In the optimization process the compiler might decide to get rid of unused variables, pull out common subexpressions from loops, rearrange formulas to reduce the number of required floating-point operations, inline short subroutines, rearrange the order of a few lines of code, and more. The compiler optimization cannot be expected to have too large of a scope; it mainly does local optimizations.

Rearrangements of formulas can spoil roundoff and overflow behavior. If an expression is potentially sensitive to roundoff or prone to overflow, setting otherwise redundant parenthesis might help. Some compilers honor them.

It can make a difference whether subroutines are in the same file or not. Compilers usually do not optimize across files or perform any global optimization.

At the time of compilation it is not clear (to the computer) which parts of the program are most heavily used or what kind of situations will arise. The optimizer may try to accelerate execution of the code for all possible inputs and program branches, which may not be the best possible speedup for the actual input. (This is the one reason why JIT-compilers can, in principle, provide better optimization results than static compilers.)

Another trick for speedup: If available, try different compilers and compare the speed of the executable they produce. Different compilers sometimes see a program in different ways. For example, a commercial and a free C compiler may be available.

Over time, compilers have become more intelligent and mature, taking away some of the work programmers need to do to improve performance of the code. The compiler most likely understands more about the computer's central processor than we do, but the programmer understands her code better overall and will want to assist the compiler in the optimization.

Performance Analysis Tools. Hidden or unexpected bottlenecks can be identified and improvements in code performance can be verified by measuring the execution time for the entire run or the fraction of time spent in each subroutine. Even for a single user the execution time of a program varies due to other ongoing processes, say by 5%. Thus, a measured improvement by a few percent might merely be a statistical fluctuation. For *profiling*, a program is compiled and linked with appropriate options enabled. The profiler then collects information while the program is executing. Profiling analysis reveals how much time a program spent in each function, and how many times that function was called. If you simply want to know which functions consumes most of the cycles, this is the way to find out.

9.2 Parallel computing

Modern CPUs are intrinsically parallel. In fact, purely serial CPUs are no longer manufactured. Fully exploiting the parallel hardware can be a formidable programming challenge.

A computer cluster with many processor or a CPU with several processor cores can execute instructions in parallel. The memory can either still be shared among processors or also be split among processors or small groups of processors ("shared memory" versus "distributed memory"). An example of the former is a multicore processor. For very large clusters only the latter is possible. When the memory is split, it usually requires substantially more programming work that explicitly controls the exchange of information between memory units.

As for a single processor core, a parallel calculation may be limited by 1) the number of floating-point operations, 2) memory size, 3) the time to move data between memory and processor, and 4) input/output, but communication between processors needs to be considered also. For parallel as well as for single processors moving data from memory to the processor is slow compared to floating-point arithmetic. If two processors share a calculation, but one of them has to wait for a result from the other, it might take *longer* than on a single processor. Transposing a matrix, a simple procedure that requires no floating-point operations but lots of movement of data, can be particularly slow on parallel machines.

Parallel computing is only efficient if communication between the processors is meager. This exchange of information limits the maximum number of processors that can be used economically. The fewer data dependencies, the better the "scalability" of the problem, meaning that the speedup is close to proportional to the number of processors.

If the fraction of the runtime that can be parallelized is p, then the execution time on N cores is $t_N = \frac{p}{N}t_1 + (1-p)t_1$. The speedup is $S = t_1/t_N = 1/(\frac{p}{N}+1-p)$; this commonsense relation is called Amdahl's law. Suppose a subroutine that takes 99% of the run time on a single processor core has been perfectly parallelized, but not the remaining part of the program, which takes only 1% of the time. Amdahl's law shows that 4 processors provide a speedup of 3.9 (subproportional), and 100 processors provide a speedup of 50. No matter how many processors are used, the speedup is always less than 100, demanded by the 1% fraction of non-parallelized code. For a highly parallel calculation, small non-parallelized parts of the code create a bottleneck. Essentially, Amdahl's law says that the bottleneck will soon lie somewhere else.

When the very same program needs to run only with different input parameters, the scalability is perfect. No intercommunication between processors is required during the calculation. The input data are sent to each processor at the beginning and the output data are collected at the end. Computational problems of this kind are called "embarrassingly parallel."

Parallelization of a program can be done by compilers automatically, but if this does not happen in an efficient way—and often it does not—the programmer has to provide instructions, by placing special commands into the source code (compiler directives).

```
!$OMP PARALLEL              !$OMP PARALLEL DO
print *,'Hello world'       do i=1,10
!$OMP END PARALLEL             ...
```

Figure 9-1: *Examples of parallel programming for a shared memory machine using compiler directives for OpenMP. The left example spawns a task over several threads. The right example is a parallelized loop.*

A common standard for parallelization of programs on shared memory machines is OpenMP (Multi-Processing), and many compilers offer intrinsic support for OpenMP. It specifies, for example what loop should be parallelized, and variables can be declared as public (shared through main memory) or private (limited in scope to one core). A widely accepted standard for parallelization on distributed memory machines is

MPI (Message Passing Interface), http://www.mpich.org. Message passing is communicating between multiple processors by explicitly sending and receiving information.

"Thinking Parallel." Summing numbers can obviously take advantage of parallel processing. But in the sequential implementation

```
s=a[0];
for(i=1;i<N;i++) s=s+a[i];
```

the dependency of `s` on the previous step spoils parallelization. The sequential implementation disguises the natural parallelizability of the problem. In Fortran and Matlab a special command, `s=sum(a)`, makes the parallelizability evident to the compiler. The same languages also offer index-free notation, such as matrix multiplication `A*B` and elementwise multiplication `A.B` (in Matlab notation). The right side of Figure 9-1 is replaced by `do concurrent (i=1:10)` in Fortran.

Concurrency refers to processes or tasks that can be executed in any order or, optionally, simultaneously. Concurrent tasks may be executed in parallel or serially, and no interdependence is allowed. Concurrency allows processors to freely schedule tasks to optimize throughput.

Distributed or grid computing involves a large number of processors located at multiple locations. It is a form of parallel computing, but the communication cost is very high and the platforms are diverse. Distributed computer systems can be realized, for example, between computers in a computer lab, as a network of workstations on a university campus, or with idle personal computers from around the world. Tremendous computational power can be achieved with the sheer number of available processors. Judged by the total number of floating point operations, distributed calculations rank as the largest computations ever performed. In a pioneering attempt, SETI@home utilizes millions of personal computers to analyze data coming from a radio telescope listening to space.

☐ Executables are typically submitted to a computer cluster through a job scheduler, such as *Condor*, *Portable Batch System (PBS)*, and *Slurm*.

9.3 Hardware acceleration

Graphics Processing Units (GPUs). CPUs have to carry out versatile tasks, in addition to floating point arithmetic. Additional dedicated floating-point units can be of benefit for numerically intensive calculations. Graphics Processing Units, which, as the name indicates, evolved from graphics hardware, provide massively parallel computational capabilities. GPUs consist of many (up to thousands of) cores and each core can execute many threads concurrently. In fact, each core processes threads in groups no smaller than 32 (a "warp").

The many floating point units are only part of the reason why GPUs achieve high throughput; exploiting memory structures is necessary to keep them busy. Physically the memory technology of a GPU is not much different from that of a CPU; there is fast on-chip memory and slow off-chip memory, but how memory is organized and used is different. For example, part of the memory can be dedicated as read-only memory. And communication among a block of threads is through fast shared memory. The programmer has a more detailed control of memory organization and access patterns.

Programming GPUs efficiently can be difficult, but the payoff is potentially high. GPUs work well only with data-level parallel problems. And they are only efficient if many FLOPs are carried out for each byte of data moved from main memory to GPU memory. For example, matrix addition is not worthwhile on GPUs, but matrix multiplication is. Successfully ported applications are known to have led to a hundred-fold speedup compared to a CPU core (e.g. the gravitational N-body problem). Languages for GPU programming are CUDA and OpenCL. A modern GPU can carry out calculations with single and double precision numbers, but, unlike a modern CPU, the latter is slower.

In CUDA, a C function call `VecEval(C)` becomes `VecEval<<<1,N>>>(C)` to evaluate the same function on N threads. The function can use a thread specific index with `int i = threadIdx.x`.

Other accelerators. Another type of numerical co-processor is the "Many Integrated Core Architecture" (currently the Xeon Phi). It has many cores, but is easier to program than GPUs.

Special purpose computers. For particularly large and important problems, specialized hardware optimized for the problem at hand can be considered. Examples of special purpose hardware are GRAPE, for the numerical solution of the N-body problem, and Deep Blue, for chess playing. So far, such attempts have turned out to be short-lived, because the performance of mainstream processors has increased so rapidly. By the time the specialized system is developed and built, its performance gain barely competes with the newest mainstream computing hardware. However, since Moore's law has slowed down this approach may become fruitful.

☐ Intel and AMD are the manufacturers of CPUs. nNvidia and AMD build GPUs.

Recommended Reading: Kirk & Hwu, *Programming Massively Parallel Processors: A Hands-on Approach* is a pioneering book on GPU computing. The aforementioned textbook by Patterson & Hennessy also includes an introduction to GPUs.