Lecture 13: Software and Hardware Issues in Computational Physics

Ammar H. Hakim (ahakim@pppl.gov) [1]

[1]Princeton Plasma Physics Laboratory, Princeton, NJ

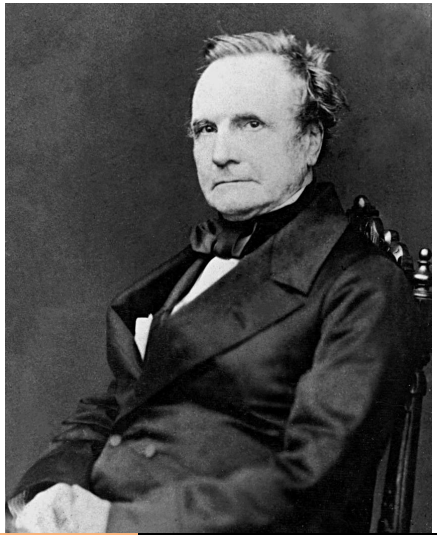
Princeton University, Course APC523, 2020

Goal: Hardware and software for Computational Physics

Our computation physics codes must run somewhere: Making code work on modern hardware and writing *long-lived* and *usable* software is highly non-trivial task. **Difficult and under appreciated art!**

- Modern computer hardware is changing: new architectures are emerging (too) rapidly.
  - Pressure on hardware: make chips *faster* but *consume less energy*. Contradictory goals.
  - New directions: many (100s or 1000s) more low-power "cores" with lower clock speed. Funded by Exascale Project in the US (https://www.exascaleproject.org). Aims to build machines that do 10^{18} FLOPS!
- Software is expensive, even (and especially) when it is free!
  - Software development is *labor intensive*. Takes time, and humans get tired, need to sleep, eat, take vacations (and hide from viruses).
  - More importantly: writing good code is an *art*. Can't be learned only from books. Need to *apprentice* yourself with a Master Craftsman. Process is slow, can take years to perfect art.

Some History: Charles Babbage, (1791-1871)



- Considered to be the "Father of the computer". Designed mechanical machines to compute values of polynomial functions. *Not completed in his lifetime*. Built by Science Museum, London in 1991 (printer he designed constructed in 2000)
- Accomplishment of greatest genius was the invention of the "Analytical Engine". General purpose computer, with all essential ideas now found in modern computer hardware worked out in detail.

See links on lecture website. See also cyberpunk novel "The Difference Engine". Explores dystopian alternate history in which Analytical Engine built.

Some History: John von Neumann, (1903-1957)

- Polymath of great genius: mathematician, physicist and computer scientist. Modern computer architecture (von Neumann architecture) named after him.
- Influential machine designed at Institute of Advanced Study from 1945-1951; 40-bit "word" with two 20-bit instructions; 1024 words of memory.
- Also wrote first and highly influential paper on error analysis on Gaussian elimination with Herman Goldstine: created field of numerical analysis. See review in SIAM Rev. **53** (4), pp.607-682.

Many others, some with Princeton background

The history of computing is complex and many people have played key role, many with Princeton background (like von Neumann who was at IAS)

- Alan Turing. Towering genius in theoretical computer science, got his math Ph.D from Princeton University. Invented "Turing machines", an abstract model for universal computing machine. Very influential and part of modern CS curriculum. Worked with von Neumann on philosophy of AI and machine computability.
- Not so well known: Charles Sander Peirce ("Purse"). Realized that logical operations could be carried out by **electrical circuits, way back in 1886**, decades before such machines were built! See letter in which *first logic circuit* is described to his former student Allan Marquand of Princeton. (Science Library near GIS and book stacks). Rather sad letter, written when C.S Peirce was very poor and had to borrow money from Marquand. (Art library named after Marquand).

A note on computer speeds and prefixes

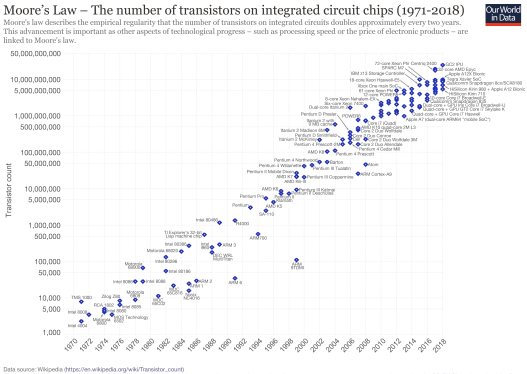
Just a reminder

- Giga = billion (10^9)
- Tera = trillion (10^{12})
- Peta = 10^{15}
- Exa = 10^{18} (billion times faster that Gigascale)

Typical processors are in the GHz range. (My MacBook Pro clocks at 2.8 GHz and has 4 cores). Typical laptop drives are in terabyte range. Largest current machines provide 10s of Petaflops computing powers with new exascale machines available in a 2-5 years.

What drives single processor performance?

Moore's "Law" (more of an industry observation) "The number of transistors incorporated in a chip with approximately double every 24 months"



- Ultimately, it boils down to the smallest feature sizes that can be lithographed. At one point, people thought it would bottom out at 28 nm. However many companies have 10 nm and 7 nm in mass production, with 5 nm around the corner.
- *Eventually, individual processors are not going to get much faster*. It is hard to shrink semiconductor feature sizes any further. Fundamental limits from physics (speed of light, EM cross-talk, heat generation, ...

At present, processor hardware architecture is in significant flux

- There is a trend away from faster, complex, chips to chips with simpler, less fast "cores" (many-core machines)
- Power consumption is a huge issue with major supercomputer installations. If we go along the path we are on now, a small power-plant will be required to simply run the cooling systems of large machines.
- With the rise of handheld devices (smartphones, tablets, ...) there is a drive towards low-cost and low-power chips. Significant work in ARM chips is going on. (These are Reduced Instruction Set Computer (RISC) chips, designed to be cheap and low power)
- Use of Graphics Processing Units (GPUs) in HPC is also increasing.
- It is incredibly hard for software to keep up with hardware. Software is written by humans, who need to eat, sleep, take vacation, etc. Also, software is *extremely complicated*, more so than the hardware it runs on.

A major applied mathematics, computational physics and software engineering challenge is to design algorithms which take advantage of this complicated zoo of emerging hardware.

Two major types of parallel programming

To remain abstract, I will use "Processing Unit" (PU) as a short-hand. It could mean independent CPUs, threads or cores.

- In *Shared memory* parallel programming, different PUs may run different code, but have access to the same *memory*. Synchronization of read/writes is a major issue.
- Shared memory systems usually use "threads". These are separate code execution paths. A CPU runs a thread for a bit, stops it, and then runs another thread. Complex programs can have hundreds of threads.
- In the second type, *distributed memory*, each PU executes the same code, but on their own portions of the data. Communication is done via *messages*. This is the most common pattern in code which solves PDEs (either via grid or PIC methods).

How much can parallel algorithms help? Amdahl's Law

Amdahl wrote an important paper in 1967 ("Validity of the single processor approach to achieving large scale computing capabilities"). Basically, he advocates using *serial* processing. Paper starts out: "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 and of the weaknesses of the multiple processor approach in terms of application to real problems and their attendant irregularities"

This paper has the statement to what is now known as "Amdahl's Law". It is a very good (and quick) read.

Incidentally: Amdahl never actually writes any equations in his paper, but states it in words! (Perhaps he didn't know how to typeset equations?)

Amdahl's Law shows the limitation of parallel programming with *constant work loads*

Let a process take unit time to run. Then, ideally, with p PUs, the run time should be $1/p$. Amdahl points out that in reality, only a fraction f can sped up, the remaining $1 - f$ is "irreducibly" serial. Hence, the run-time is

$$T_a(p, f) = f/p + (1 - f)$$

Hence, *speedup* will be

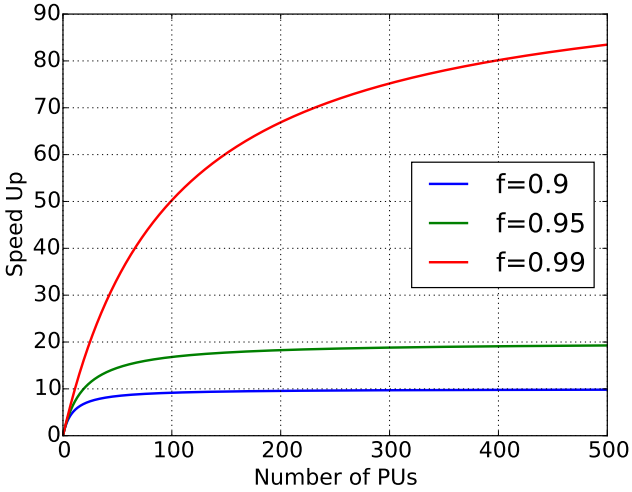
$$S_a(p, f) = \frac{1}{f/p + (1 - f)}$$

Note that if $f < 1$, then

$$S_a(\infty, f) = \frac{1}{1 - f}$$

which is independent of number of PUs!

Amdahl's Law for various parallelizable fractions



Some comments on Amdahl's Law

- Amdahl's law is *optimistic* in the sense that it assumes that the parallelizable fraction speedup scales linearly. Things get worse with (inevitable) overhead say from messaging, etc.
- Amdahl's law is *pessimistic* in that it assumes that *we want faster results for the same problem*. This is not true for most problems of interest.

In reality, if someone gives us a bigger computer, we want to do a *bigger problem*. This is crucial to getting around the limitations of Amdahl's Law.

Assume that if f is the parallelizable fraction of a serial job, then with p PUs we will run a p times bigger job. Then, the speedup will be

$$S_g(p, f) = (1 - f) + fp$$

Note that now, the speedup is linear in p! Essentially, the cost of the serial part is amortized by going to a bigger problem. This is called *Gustafson's Law*.

These two speedup laws distinguish *strong* and *weak* scaling

The *strong scaling* of a problem is the actual speedup one can achieve for a problem of fixed size, but increasing PUs. Amdahl's law bounds such a scaling. (In general, actual strong-scaling will be *worse* than predicted by Amdahl's Law)

The *weak scaling* of a problem is the actual speedup one can achieve for by problem with size increasing linearly with number of PUs. Gustafson's law bounds such a scaling. (In general, actual weak-scaling will be *worse* than predicted by Gustafson's Law)

In general, most real-life problems will scale in a more complicated manner. For example, for some problems are I/O and/or communication bound. Others are bound by codes which are not parallel at all (for example, some old legacy serial code might be needed).

The Corollary of Modest Potential

This term was coined in an important paper "Type Architectures, Shared Memory, and the Corollary of Modest Potential" by L. Snyder in 1986. (Ann. Rev. Comput. Sci. 1986. 1:289-317).

Consider a time-dependent 3D problem on a rectangular grid, with n number of cells in each direction. Then, assuming that the number of time-steps to get to a given time is proportional to n, the computational time will scale as

$$T_m(n) \sim n^4$$

(There are n^3 cells, and n steps to take). Hence, even if we get linear scaling from Gustafson's law, the scaling of n with number of PUs, p, will be sub-linear. In this particular case,

$$n \sim p^{1/4}$$

Hence, to do a problem 100 times bigger, we will need 10^8 more computing resources!

Can be *worse* than predicted by The Corollary of Modest Potential

Note that most algorithms are not *parallel in time*! This means, that for a 3D problem, we can only use $\sim n^3$ more PUs, and need to run for a factor of n longer.

What do all of these (seemingly pessimistic) scaling laws teach us?

This *does not* mean that things are hopeless! In fact, what this means is that parallel algorithms must go hand-in-hand with better physics models, which do not require such high resolutions. For example,

- Instead of solving Navier-Stokes (NS) equations, one solves Reynolds Averaged NS (RANS) equations, or uses Large Eddy Simulation (LES) techniques
- Or, instead of tracking very particle in a large system, one performs statistical averaging and uses expensive particle/kinetic calculations to provide occasional validation
- Or, instead of doing Vlasov-Maxwell (6D, with plasma frequency time-scales), one uses (in appropriate limits) gyrokinetic equations (5D, with turbulent fluctuation time-scales), or fluid or other appropriate approximations.

Conclusion: Ammar's Corollary of Cynical Progress

It is easy to fill up the biggest computer with FLOPs and bytes. It is *much* harder to do useful science.

Another way to put it: in a Darwinian process with pressure to use bigger-and-bigger machines, one may select for less-than-desired traits

Anatomy of a Very "Simple" Problem

Numerical Methods 101: Write a function to multiply two square matrices

```
for (auto i=0; i<N; ++i)
  for (auto j=0; j<N; ++j) {
    C(i,j) = 0.0;
    for (auto k=0; k<N; ++k)
      C(i,j) += A(i,k)*B(k,j);
  }
```

Recall that

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

The solution in C++ is very simple:
However, how efficient is this solution?

There is a long history of linear algebra libraries

- Designing *efficient* linear algebra routines is not trivial, even for such "simple" matrix-matrix multiply problems.
- Most widely used libraries are BLAS (Basic Linear Algebra System) on top of which is built LAPACK (Linear Algebra Package).
- BLAS provides "basic" routines (matrix-vector, matrix-matrix multiplication, etc). LAPACK provides linear solvers, eigensystem finders, Singular Value Decomposition (SVD), etc.
- Created originally by Jack Dongara in 1970s. Continues to be developed and optimized. Many platforms (Apple, Intel, ...) supply their own highly optimized BLAS/LAPACK libraries. Other "self-tuning" implementations also exist (see ATLAS).
- Modern libraries using very clever C++ techniques are also being created and extensively used. The best example I know of is the Eigen C++ library (see `http://eigen.tuxfamily.org/`).

Lets compare our hand-written code with Eigen and BLAS. (Look at code)

What do Eigen/BLAS know that we don't?

Cache usage and use of SIMD[1] instructions. Modern processors have a hierarchy of memory locations they can access, with *very* different access speeds.

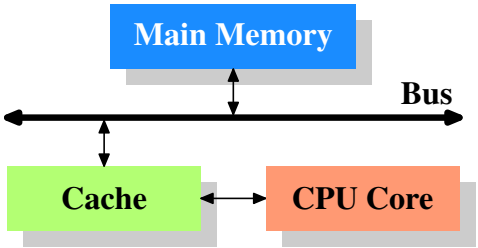


Figure: A very simple cache configuration

A "cache" is an area of memory which is close to the CPU core. The memory access to the cache is very fast, compared to access to main memory. The difference comes about as main memory is Dynamic RAM (DRAM) v/s cache is SRAM. DRAM circuits are tiny (hence more of them can be on a single chip) v/s SRAM circuits which are very large. However, DRAM essentially relies capacitive discharge, which is very slow.

[1] Single Instruction Multiple Data

Eigen and BLAS preload as much data as possible in the fast cache

- Using clever techniques, these libraries preload as much of the matrix data as possible into the cache
- Once loaded, the CPU can fetch this data very rapidly to do the multiplication and sums.
- On modern CPUS FLOPS are basically free. For example, most operations take a single (even fewer) cycle. It is the memory access which is very expensive (5-20 cycles for cache, and 100s of cycles for main memory). Hence, to improve performance, one must make sure that cache is used in an optimal manner.

Why is our matrix-matrix multiply code so slow?

Is it because of the data access pattern in matrix-matrix multiply?

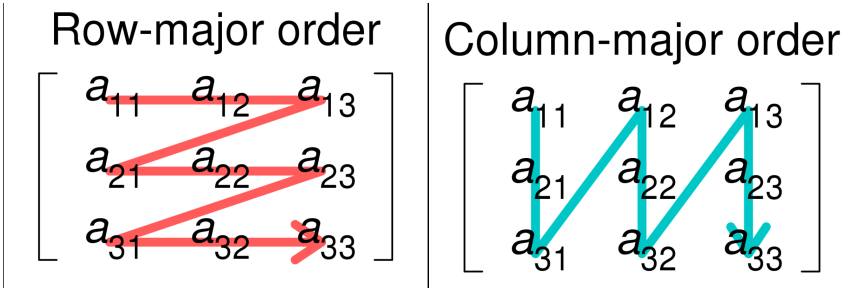


Figure: In row major order (left) the *rows* are kept contiguously. To be cache friendly one must increment the column index (j) faster. In column major order (right) the *columns* are kept contiguously. To be cache friendly one must increment the columns i index faster. For matrix-matrix multiply this is impossible, as one or either indexing will incur a cache miss!

So how to overcome this seemingly "impossible" situation?

```
// transpose
Matrix T(N,N);
for (auto i=0; i<N; ++i)
  for (auto j=0; j<N; ++j)
    T(i,j) = B(j,i);

for (auto i=0; i<N; ++i)
  for (auto j=0; j<N; ++j) {
    C(i,j) = 0.0;
    for (auto k=0; k<N; ++k)
      C(i,j) += A(i,k)*T(j,k);
  }
```

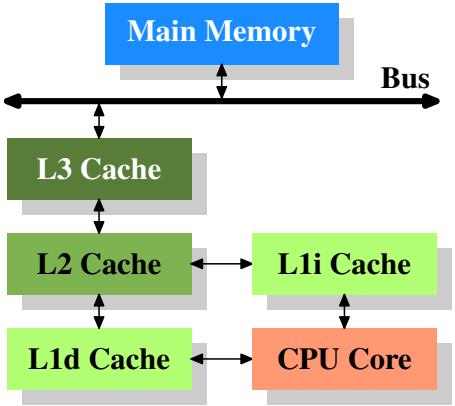
The problem is that the matrix-matrix product is an inner product of columns of A with the rows of B. So, lets copy the *transpose* of B matrix to a temporary matrix T. I.e. $T = B^T$. Then, the matrix-matrix multiply becomes

$$C_{ij} = \sum_k A_{ik} T_{jk}$$

Now, note that if the data is stored in *row* major order, both matrices are accessed in a cache friendly manner!

This is problem and chip dependent: likely does not matter much for "small" matrix sizes.

Modern CPUs very complicated: multiple levels of caches, SIMD ...



A more realistic processor with three levels of cache. Caches further from the CPU core are slower, but larger. On my MacBook Pro I have 32KB L1 Cache, 256KB L2 Cache and 6MB L3 Cache. In contrast I have 16GB of main memory!

Each level of cache has it own access timing and its own latency

Operation		Time	Rel. cost
Data access	Internode	10 µs	100 000
(Latency)	Memory	50 ns	500
	L2 Cache	5.0 ns	50
	L1 Cache	1.0 ns	10
Data movement	Internode	5.0 ns	50
(32-bit)	Memory	0.5 ns	5
	L2 Cache	0.2 ns	2
	L1 Cache	0.1 ns	1
Single precision	FLOP	0.1 ns	1

Figure: Table I in Phys. Plasmas **15**, 055703 (2008)

Writing code which is cache friendly is not easy

Very problem *and* machine dependent!

- In general, be aware of where the data you want comes from. Modern CPUs "prefetch" data, i.e. if you ask for element `a[i]` in an array, very likely the CPU has also fetched element `a[i+1]` into the cache. So, use it!
- In most multi-dimensional structured arrays (say a matrix) at least one index is a "stride" away. This can cause a huge cache miss.
- This is partly unavoidable. Some newer codes use "space filling curves" to index an array. This ensures that *most* (not all) array accesses are from close-by in memory.
- If you are using particles (PIC codes) you must try and keep all the particles in a cell close by. This may require periodically sorting the particles, as otherwise they will drift arbitrarily far away, killing cache performance.
- If there is a cache aware library USE IT!! These types of optimizations are very difficult and time consuming, and are best left to experts.

Additional complications come about as modern CPUs allow the same operation to be applied to more than one number

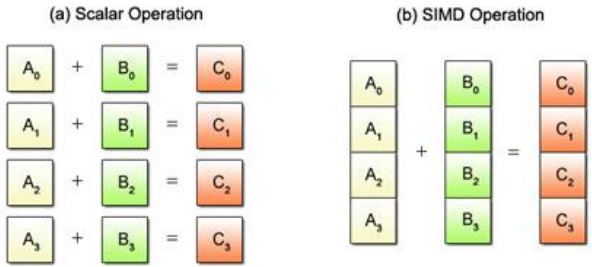


Figure: Scalar operations are applied sequentially to one register after another (left). SIMD operations apply the same operation ("Single Instruction") to "Multiple Data" locations. For this the CPU provides special registers that can store 4 (or more) floating point numbers.

SIMD and "Vectorization"

- Modern processors have vector instructions that make it possible to do operations on *all* elements of a vector *simultaneously*.
- Total size of each vector can be 64 bits (MMX), 128 bits (XMM), 256 bits (YMM) and, 512 bits (ZMM).
- Often, we can rearrange code to ensure same floating-point operation is done on multiple data elements at the same time.
- This depends on the instruction sets available on the CPU you have: these instruction sets go by strange names like "SSE2", "AVX", "AVX512" etc.
- However, programming with these instructions sets/registers is not easy! Compilers can often vectorize automatically, but this is best done by a human who understands the algorithm in detail.
- Note: not all algorithms are suitable for vectorization! Luckily, many PDE and particle solvers can be vectorized, resulting in significant gain in speed ($2\times$ to $4\times$).

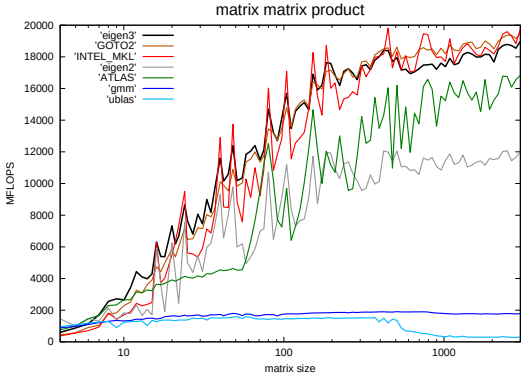
SIMD programming is done via CPU "intrinsics"

These are functions which the CPU vendor provides (Intel, AMD, ARM ...). Unfortunately, there is no standardization (say between Intel/AMD and ARM), and the calls look bizarre and complicated.

The most detailed description on how to optimize C++ code (cache, SIMD, ...) is on Agner Fog's blog. See https://www.agner.org/optimize/. Excruciating details but his manual on "Optimizing Software in C++" worth consulting if you care about speed.

He has a C++ library that abstracts always much of the gory details of SIMD programming. See https://github.com/vectorclass. See also NSIMD library https://github.com/agenium-scale/nsimd.

Cache-awareness and SIMD usage makes Eigen very fast



Eigen performance with matrix size for a matrix-matrix multiplication. For other benchmarks see http://eigen.tuxfamily.org/index.php?title=Benchmark

Some final thoughts

For serious computational work, thought must be given to algorithm and code optimization.

- Hands down, it is more optimal to use a *"better" algorithm* than optimize a crappy one. Example: FFTs will beat naive N^2 algorithms for any useful N.
- However, even good algorithms need major effort in optimizing for daily use in production code. (If you only care about a small "throw-away" application then this effort is not worth it). If your code is widely used, then pressure on optimization is higher.
- Remember that not all algorithms will benefit from cache and SIMD vectorization as much as did matrix-matrix multiplication. However, in general it is a **good idea to be aware of memory access patterns in your code**.
- Despite complexity, SIMD vectorization is often needed to make good use of modern processors. **Use libraries!** Hand-written SIMD code is ugly and likely hard to maintain.

Next lecture: some guidelines on software engineering, programming languages, parallel programming and perhaps an overview of GPU programming.