Lecture 14: 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.
- 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.

How fast (in principle) is my Mac?

Mid 2015, Intel Core i7 chip. See See this page for info on chips which ship with different Macs. Strangely, this information is not in the "About this Mac" tab and one needs to go digging for it.

- Clock-speed is 2.8 GHz. So we have 2.8 GFLOPS/core (assuming a FLOP can be done in 1 cycle).
- However, SIMD can potentially do 4 FLOPS per cycle (for `float`. For `double` SIMD is 2 FLOPS per cycle). Giving 11.2 GFLOPS/core.
- Often there are are multiple FPUs (floating point units) on a core. Mine has 2 FPUs (AFAICT. The specs say total 8 "threads" which I assume means 2 FPUs per-core), but some chips have 4 FPUs. So that brings it to 22.4 GFLOPS/core.
- With 4 cores this gives 89.6 GFLOPS (half of that for `double` precision numbers) total.

Our Eigen double matrix-matrix multiply peaked at about 8 GFLOPS on a single core. Float matrix-matrix multiple peaked at about 15 GFLOPS.

Eigen's performance comes from "tiling" the matrix and using SIMD instructions. (Most PDE solvers can't always use such tricks. Linear algebra is very specialized. Best left to experts).

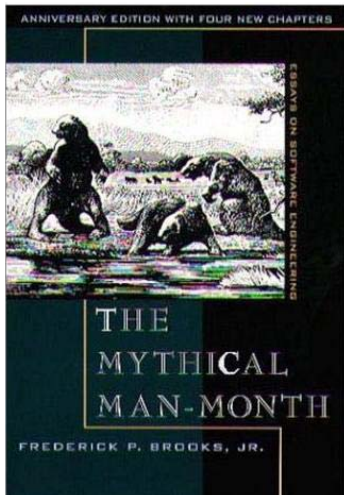
It is hard to achieve anything close to peak-performance

- Memory access will slow down a program by a lot: FLOPS are not the only thing that matter!
- If you get between 10-20% peak you are in (very) good shape. More likely, 5% peak is reasonable.
- My team's code (`Gkeyll`) gets 500 GFLOPS for some compute kernels on an Intel Skylake chip (peak-performance of 3 TFLOPS). Does not use SIMD vectorization (due to our code structure SIMD is hard to use). This is pretty good.
- Similar performance is obtained on a Mac: 12 GFLOPS on a 4-core Mac. (Skylake chips have 48 cores and are $3\times$ faster).

Study Agner Fog's optimization manuals if you are serious about optimization. PU PICSciE (Research Computing) offers workshops.

Some (brief) notes on software engineering

If you read only one book on Software Engineering, it should be the “Mythical Man Month”.



TL;DR Version:

- Writing large-scale software programs is hard.
- Techniques that work on small projects don’t scale to large projects. It is important to learn by looking at good, large code bases.
- Most of physicists won’t work on such projects, but these are now becoming increasingly important. Consider the SCIDAC program, which supports huge codes, running into 100K+ (even millions of) lines of code (LOC), with multiple developers, are massively parallel and are designed to solve very complex physical problems.

Some (brief) notes on software engineering

Large-scale computational physics software is, in some ways, even harder, as it is not clear if one is solving the equations correctly (*verification*) and if one has the correct model in the first place (*validation*). *Regression testing* and careful benchmarks are essential to build confidence.

Some general recommendations for "small" projects (thesis level)

- Unless you are working on an existing "old" code, use modern C++. It is hard to find libraries, and general support for high-level things in Fortran (scripting, flexible input files, ...). Hard to find jobs.
- Don't invent your own. There are a huge amount of existing math libraries (LAPACK/BLAS, GSL, PETSC, Eigen, Boost, FFTW, SuperLU, HDF5 ....), and it is unlikely you can write a better routine in a short amount of time.
- Use a version control system, *even for small projects*. Git is a good option, and code can be hosted for free on github or bitbucket.
- Use an automated build system. CMake has a lot of momentum, and is a very good choice. There are other possible choices too. I like Waf a lot.

A reasonable stack: C++ and Eigen with CMake build system, with HDF5 or NETCDF output format, GSL, PETSC, ..., and post-processing using Python (Anaconda Python is an excellent free distribution with everything you need to do viz in 2D and 3D).

Programming languages for computational physics

Somewhat personal biased views:

- Modern large-scale computational physics, at least the core computational kernels, should be done in C++: produces highly optimized code, and has huge industry support, with excellent compilers available on every conceivable platform. Vast majority of time-critical code *anywhere* is written in C or C++.
- Why not C++? It is difficult to learn, and really ugly.
- Modern C++ (post 2011) has fixed some historical issues, with very elegant new features useful in computational physics added recently (including in-built parallelism, with GPU and other accelerator array data-structures being proposed as part of a future standard).
- An ideal combination: C++ called from a scripting language like Python or Lua. Done extensively, for example in video games and major Internet appliances. My own code uses C++ and LuaJIT.

JIT: Just-in-Time compilers generate machine code on-the-fly. Can be very efficient but need some care.

Two major types of parallel programming in computational physics

To remain abstract, I will use "core" as a short-hand. It means an independent execution unit. Depends on context (CPU core, thread, etc)

- In *Shared memory* parallel programming, different cores may run different or same code, **but have access to the same memory**. Synchronization of read/writes to memory is a major issue. Needs locking mechanism.
- Shared memory systems usually use "threads". These are separate code execution paths. A core 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 core executes the same (or less often, different) code, but on their **own portions of the memory**. Can't directly touch memory owned by others. Communication is done via *message passing*. This is the most common pattern in code which solves PDEs (either via grid or particle methods).

Not the only type of parallel or distributed programming!

Concurrency is important, not just in computational physics

Think of Facebook, Twitter, WhatsApp or Amazon AWS. These are *massively concurrent* systems with millions of people connected at the same time, commenting, sending messages etc. Extremely difficult problem!

- Many specialized libraries for such applications. Example, ZeroMQ (also based on message passing. Almost magical library for concurrency) and Redis (in-memory concurrent database. Insanely beautiful. Used in Twitter "Timeline" feature).
- Specialized languages developed for massively concurrent systems. Example, Erlang. Designed to be fault tolerant and massively distributed. Used in WhatsApp. Other example is the Go programming language.
- These specialized languages are *functional languages* that have *immutable* datastructures, making concurrency much easier to implement.

The Message Passing Interface (MPI) Standard

MPI is the de-facto standard for writing distributed memory programs, i.e. code which runs on a cluster of computers. Unlikely to go away even with future architectures.

These programs have a (mostly) single instruction set that runs on each core, however, on different portions of the data. Messaging is done by explicitly sending *messages* between cores.

Problem: You are given a (huge) array of numbers, a_i for $0 \leq i < N$. Compute the sum $\sum_i 1/2^{a[i]}$. Test this with $a[i] = i$. (i.e. compute the sum $\sum_{i=0}^{N-1} 1/2^i$)

This is a type of *map-reduce* pattern: apply a function to each element (*map* operation), and find its sum (*reduce* operation). An old concept from LISP, made famous by Google, and now used extensively for "Big Data" applications. Also, many numerical codes work in this way.

We will use this example to learn some MPI. Not trivial problem. Try this for homework on your own.

Basic design pattern of an explicit, parallel PDE solver

Each simulation begins with

- Domain decomposition of grid
- Initialization of fields on local portion of grid

and then in a loop

- Update local portion of fields using algorithm (RK updates, finite-differences, upwinding, etc.)
- Communicate with neighboring cores to “synchronize” field data before next time-step is taken.

Periodically, field data is written out. Here there are two choices

- Each core writes to its *own* file. These can be combined as post-processing to do analysis/visualization
- All cores write to the *same* file concurrently. Avoids need of combining later, but can be difficult to do. (Although there are libraries like HDF5 and NETCDF which you can/should use).

Domain decomposition in 2D/3D

We saw an example of 1D domain decomposition. The idea was to split the domain up into approximately equal pieces ("load balanced"), and each piece is then handled by a separate core.

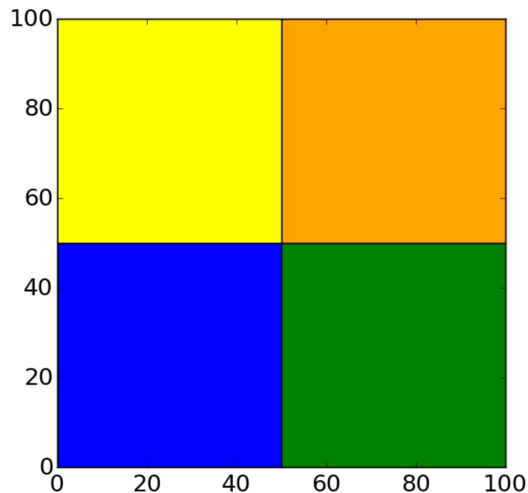
Domain decomposition is much harder in higher dimensions, specially when using unstructured grids.

Question: You are given a 100×100 grid. How will you split equitably into p pieces?

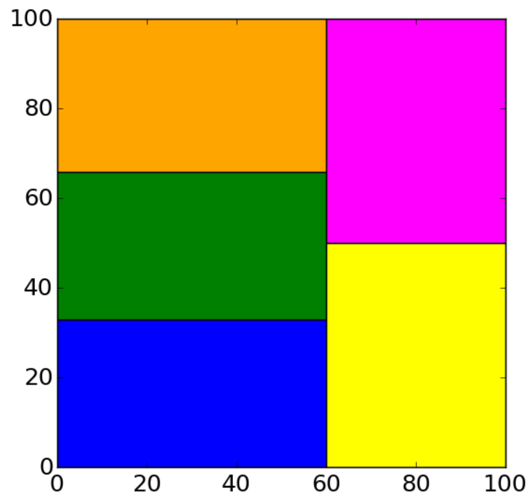
In general, this is a non-trivial problem. Imagine $p = 5$. How would you do this? Some constraints: each sub-domain should have about the same volume, the surface area of each domain should be as small as possible. This is an *integer linear programming* problem.

Often it is simpler to use a *Cartesian product* decomposition, by specifying the number of "cuts" in each direction.

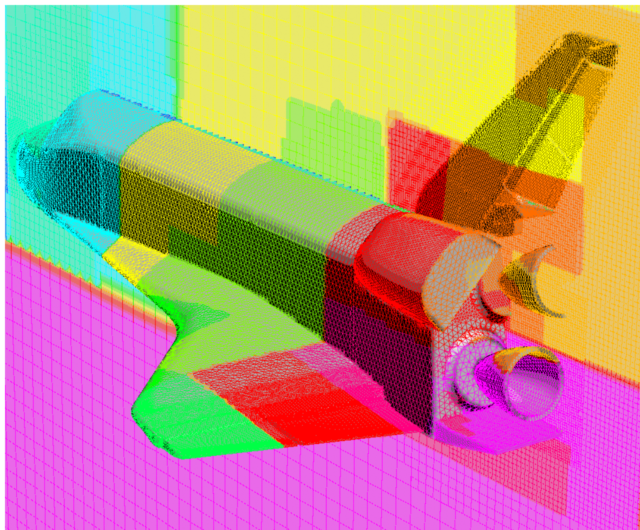
An example of a Cartesian decomposition into four sub-domains



An example of a general decomposition into five sub-domains



For unstructured grids domain decomposition is more complex



Domain decomposition on the exterior of the space shuttle. Notice the complicated sub-domain shapes. To create these decomposition is a non-trivial problem, and libraries should be used. Communication between sub-domains is very complex in such situations, and requires significant book-keeping. (Caveats about unstructured meshes)

Use Ghost/Skin cells to communicate between sub-domains

Consider the finite-difference approximation to $g = f_{xx}$

$$g_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2}$$

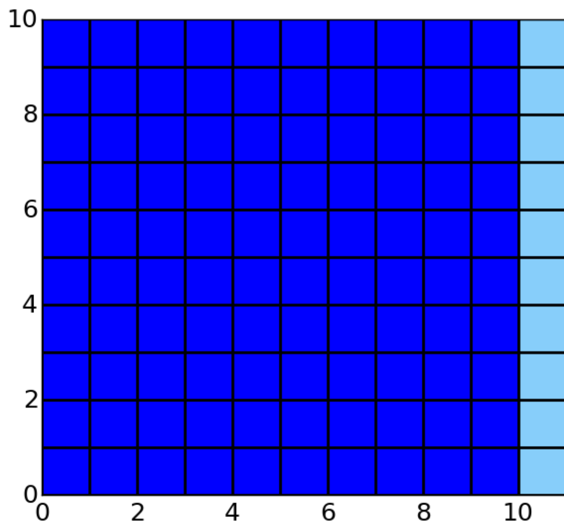
Now let I be the last cell on a sub-domain boundary. Then, we have

$$g_I = \frac{f_{I+1} - 2f_I + f_{I-1}}{\Delta x^2}$$

Note we do not know what f_{I+1} is as it is *outside* the sub-domain. I.e. it either lies outside the physical boundary or on another sub-domain.

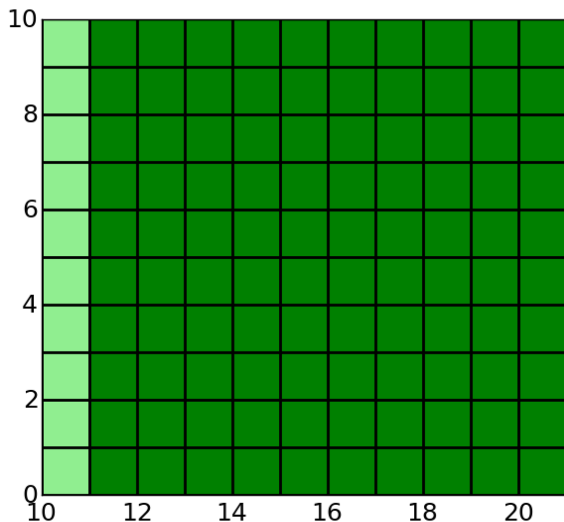
These outside cells are called "ghost cells". They are very useful for applying boundary conditions and are *essential* for synchronization across cores when doing a parallel job.

Ghost cells are the layer of cells *outside* sub-domain



The layer of cells on the right (pale blue) are the *ghost cells* for the sub-domain (blue). They lie *outside* the sub-domain.

Skin cells are layer of cells *inside* sub-domain



The layer of cells on the left (pale green) are the *skin cells* for the sub-domain (blue). They lie *inside* the sub-domain.

To synchronize copy data from skin-cells to ghost-cells

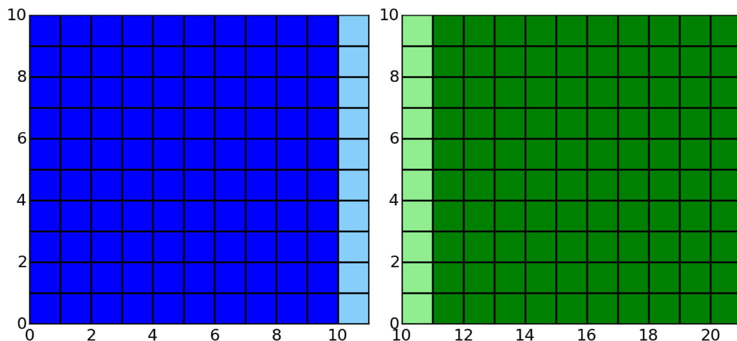


Figure: Before each time-step we must copy data from pale green region on right sub-domain into the pale blue region on left sub-domain.

Ghost/Skin cell distribution is determined by the *stencil*

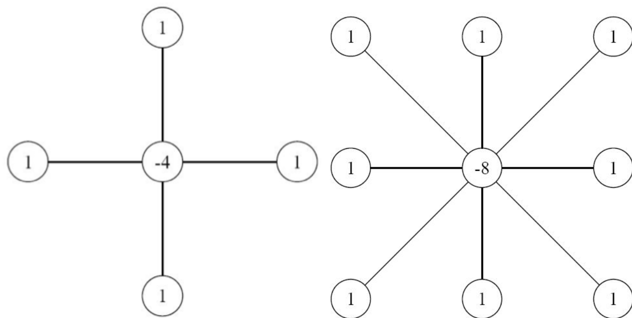
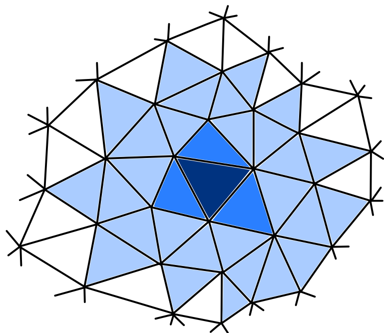


Figure: 5-point stencil (left) and 9-point stencil (right) for a Laplacian in 2D. Note that the 9-point stencil needs corner cells. This means that parallel communication must involve neighbors which share a *corner* (and not just face neighbors). This can significantly complicate communication.

For unstructured meshes, things can get *very* nasty



To update dark blue cell, some scheme would need only face neighbors. However, some higher-order methods would need a bigger stencil, shown here in pale blue for a third order finite-volume scheme. The ghost/skin cell determination and parallel communication is very complicated.

Alice and Bob eat lunch

Imagine Alice is in one city, and Bob in another. How can we ensure Alice eats lunch *before* Bob?

Note that we can't rely on clocks, as clocks may not be accurate, and not every event (in life as well a computer) has a time-stamp associated with it.

Obvious solution: Bob *waits* for Alice to call him after she eats lunch. Then Bob eats lunch. (Blocking receive)

Bob keeps doing his stuff, and every time he gets hungry, he checks if Alice left him the *correct* message. If yes, he eat, or else he goes back to doing other things. (Non-blocking receive)

If Alice never calls, Bob will starve to death.

MPI Communicators, rank and blocking/non-blocking calls

- Cores can be grouped to form communicators (`MPI_Comm`). When program starts, a global communicator with all cores is created for you (`MPI_COMM_WORLD`). You can create sub-communicators if you want (rarely need to do this).
- The rank of a cores is an integer *starting from 0*. Each core has a unique rank.
- For many communication methods, there are *blocking* and *non-blocking* versions.
- When you use a *blocking* version of a call, the code waits for the operation to be completed.
- When you use a *non-blocking* version of a call, the code immediately continues, *even if the operation is not complete*.
- Non-blocking calls can be more efficient, as you can overlap communication and computation. However, it can lead to nasty bugs. In particular, non-blocking receives are very confusing. You *must not* use the "received" data before it is actually received.
- MPI provides means of waiting for a send or receive to finish. These must be used to get synchronization correct.

To send/receive one needs to use MPI_Send and MPI_Recv

There are two versions of each call

- *Blocking* versions MPI_Send and MPI_Recv. The blocking calls will *block* till the send/receive are completed.
- *Non-blocking* versions MPI_Isend and MPI_Irecv. The non-blocking calls will *return immediately* even if the send/receive are not completed yet.

How to determine if the send/receive are actually completed or not?

Each non-blocking class returns to the caller a MPI_Request "token". This can be used in the MPI_Wait method to wait till the operation with that "token" is completed.

Think of this as a "will call" system. You buy tickets online and print out a receipt at home. Before the show you go to the "will call" counter with the receipt and pick up your tickets.

The anatomy of MPI_Send

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm)
```

- `buff` Is the pointer to the array you want to send
- `count` Is the size of the array you are sending
- `datatype` Is the type of data you are sending (doubles, floats, ...)
- `dest` Is the rank of the *destination* core you want to send the array to
- `tag` is a unique tag attached to each message (this distinguishes multiple message between the same pair of communicating cores)
- `comm` Is the communicator group you are using

The `MPI_Recv` signature is analogous. Note that *a send must be balanced by a receive or the code will hang!*.

With `MPI_Irecv` call you *must* wait before you use the data

After calling a `MPI_Irecv` (non-blocking call) you can do other things. However, *before* you use the data you are expecting, you must wait by calling the `MPI_Wait` with the "token" which `MPI_Irecv` gave you.

In addition, you must not send any more data before the receive is completed. Otherwise terrible things will happen!

In short: when you can, use *blocking* calls. Reasoning about non-blocking calls is hard and can lead to very subtle bugs which may only show up when you run on a large machine. Remember Murphy's Law: "Anything that can go wrong, will go wrong"

Summary of distributed programming with MPI

- The standard pattern of distributed programming is: decompose, initialize and then in a loop, do local updates, and then communicate ghost/skin cell data.
- You only need to know a few methods to write most solvers in parallel. `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Allreduce`, `MPI_Send` and `MPI_Recv` will cover 90% of your needs.
- You must carefully look at your scheme's stencil to determine which of your neighbors you need to send/receive data to/from. *Avoid* schemes with more that one or two ghost/skin-cells. I.e. don't use a scheme which has a wide stencil.

What we did not cover ...

- OpenMP programming for shared memory parallelism. This is a “directives” based model, in which you insert “pragma”s into the code, telling the compiler which parts of the code to parallelize
- Compiler will generate threads to run the loops etc in parallel. Easy to use. Not much control. Also, launching threads can be expensive.
- Alternate to OpenMP shared memory programming is to use MPI for shared memory: more complex but full control and no threads are launched.
- When using C++ you can also just use C++ `std::thread` and concurrency built into the language. Difficult at first, but very powerful.
- GPU programming: this is a world onto itself. Most codes will need major rewrites to work on GPUs. PU PICSciE (Research Computing) offers workshops.