To those who know nothing about high-performance computation

Denis Jacob Machado

1 What is a supercomputer?

Supercomputers are fundamental tools for scientific discovery. They can explore things that are too big, too fast, too expensive, too dangerous, or too impractical to investigate in a lab.

Most supercomputers have over 100,000 processing cores and leverage a lot of parallelization (*i.e.*, breaks a big problem into smaller problems to be computed simultaneously and independently from each other across many processors).

FLOPS

Computer performance is measure in "floating-point operations per second" (FLOPS).

An average PC ranges from 2–7.5 Gigaflops. Supercomputers today can easily surpass 10 Petaflops (*i.e.*, 10,000,000 billion operations per second). This means that a conventional supercomputer does more before breakfast than your computer does in a lifetime (*i.e.*, by 10:00 AM, a supercomputer of 10 Petaflops will have performed 144,000 quadrillion operations while your computer will have performed about 0.108 quadrillion operations).

2 What is the difference between supercomputers and cluster computers?

A supercomputer is a term that describes computers used to solve problems which require processing capabilities nearly as big as anyone can build at a given time. What people call supercomputers change over time, and it is essential to keep in mind that supercomputers of yesteryear aren't that "super" anymore.

Cluster computers are loosely coupled parallel computers where the computing nodes have **individual** memory and instances of the operating system, but typically share a file system, and use an explicitly programmed high-speed network for communication. The term "loosely" refers to the technicalities of how such machines are constructed.

The vast majority of modern supercomputers are cluster computers, but this has not always been the case, and it may not always remain that way.

But... What about servers?

A server is a singular entity. For most regular tasks, a dedicated server equipped with specific optimizations is enough. But servers only get so big. After a certain point, it doesn't make sense to keep upgrading to a bigger server. Sometimes there are no bigger servers, and sometimes a single server isn't the most efficient option. For sites like these, we need to go even bigger than the hugely powerful enterprise-grade server hardware we use. The best option for those is a server cluster.

3 Why so many processing cores?

Traditionally, the software has been written for serial computation (Fig. 1):

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

However, the world saw a considerable increase in the importance of parallelism in the 2000's due to the move of processor manufacturers away from their traditional methods of making faster individual processor cores to the new trend of making processor chips with more processing cores on them. The more processing

Figure 1: The serial problem.

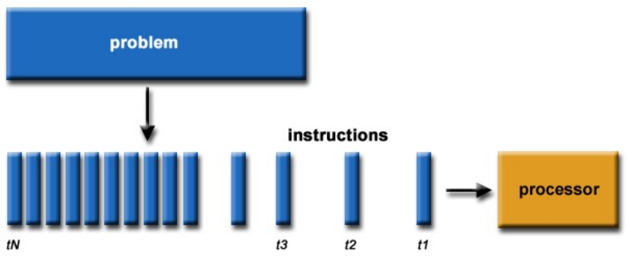
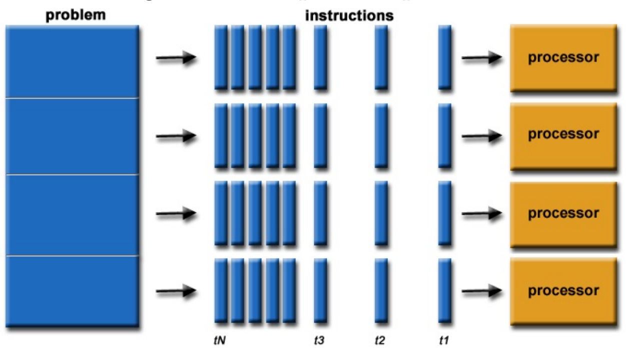


Figure 2: The parallel problem.



cores, the faster it is to process big tasks that can be divided into smaller subtasks. This is what parallelism is all about (Fig. 2).

In the simplest sense, parallel computing is the simultaneous use of multiple computer resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed

The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource

Typical computer resources:

- A single computer with multiple processors/cores
- An arbitrary number of such computers connected by a network

Some types of parallel computing

1. **SIMD** (Single Instruction Multiple Data): Same operations on different data. Also called "Data-parallel"
2. **SPMD** (Single Program Multiple Data): Same program, different data
3. **MIMD** (Multiple Instruction, Multiple Data): Different programs, different data

SPMD and MIMD are essentially the same because any MIMD can be made SPMD. SIMD is also equivalent, but in a less practical sense. The **Message Passing Interface (MPI)** is primarily for SPMD/MIMD. The **High Performance Fortran (HPF)** is an example of a SIMD interface.
[Click here to learn more about the types of parallel computing and what is Flynn's Classical Taxonomy. Click here to learn more about MPI, HPF, and more.]

4 What is the difference between processor and core?

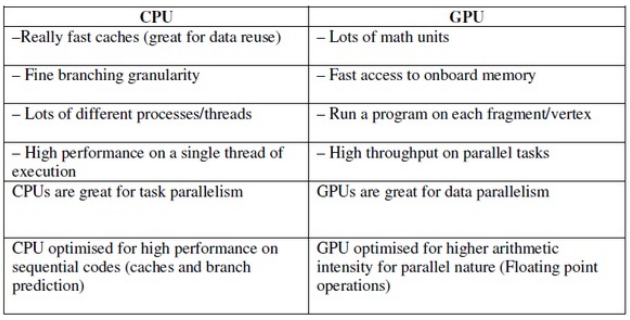
A core is usually the basic computation unit of the processor – and please keep in mind that there are multiple types of processors, such as "Central Processing Units" (CPUs) or "Graphical Processing Units" (GPUs).

Each core in a processor can run either a single program context or multiple ones if it supports hardware threads such as Intel CPUs with hyperthreading). No matter if the core has multiple threads or not, it is key that it can run maintaining the correct program state, registers, and proper execution order, and performing the operations through arithmetic logic units (ALUs). For optimization purposes, a core can also hold on-core caches with copies of frequently used memory chunks.

A CPU, for example, may have one or more cores to perform tasks at a given time. These tasks are usually software processes and threads that the operational system (OS) schedules. Note that the OS may have many threads to run, but the CPU can only run T such tasks at a given time, where $T = number\ of\ cores \times threads\ per\ core$. The rest would have to wait for the OS to schedule them whether by preempting currently running tasks or any other means.

In addition to the one or many cores, the CPU will include some interconnect that connect the cores to the outside world, and usually also a large "last-level" shared cache. There are multiple other key elements required to make a CPU work, but their exact loca-

Figure 3: Comparison between CPU and GPU.



CPU	GPU
–Really fast caches (great for data reuse)	– Lots of math units
– Fine branching granularity	– Fast access to onboard memory
– Lots of different processes/threads	– Run a program on each fragment/vertex
– High performance on a single thread of execution	– High throughput on parallel tasks
CPUs are great for task parallelism	GPUs are great for data parallelism
CPU optimised for high performance on sequential codes (caches and branch prediction)	GPU optimised for higher arithmetic intensity for parallel nature (Floating point operations)

tions may differ according to design. You'll need a memory controller to talk to the memory, I/O controllers (*e.g.*, display, PCIe, USB). In the past, these elements were outside the CPU, in the complementary "chipset," but the most modern design has integrated them into the CPU.

Besides, the CPU may have an integrated graphics processing unit (GPU), and pretty much everything else the designer wanted to keep close for performance, power and manufacturing considerations (for differences between CPU and GOU, see Fig. 3). CPU design is mostly trending into what's called system on chip (SoC).

This is a "classic" design, used by most modern general-purpose devices (client PC, servers, and also tablet and smartphones). You can find more elaborate designs, usually in the academy, where the computations is not done in basic "core-like" units.

Processor, core, and thread

- Processor (one or many, located in a socket in the motherboard)
  - Core (one or many, located in a processor)
    - Thread (one or many, located in a core)

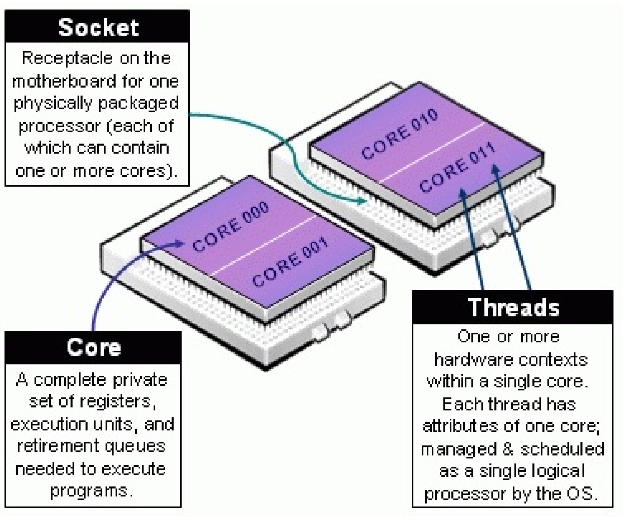
Also see Fig. 4.

On Linux, the command `nproc` returns the *total number of threads*, calculated as *number of processors* × *number of cores* × *threads per core*.

5 What are "threads" (really)?

A thread is an independent set of values for the processor registers (for a single core). It includes the "Instruction Pointer" (a.k.a., "Program Counter"), that controls what executes in what order, and the "Stack Pointer", which control the memory allocated to each thread and avoids they interfering with each other. Another way to think about a thread is to imagine it as

Figure 4: Processor (in a socket), cores, and threads.


Socket
Receptacle on the motherboard for one physically packaged processor (each of which can contain one or more cores).
CORE 010
CORE 011
CORE 000
CORE 001
Core
A complete private set of registers, execution units, and retirement queues needed to execute programs.
Threads
One or more hardware contexts within a single core. Each thread has attributes of one core; managed & scheduled as a single logical processor by the OS.

the software unit affected by control flow (e.g., functions "call", "loop", "goto"), because those instructions operate on the Instruction Pointer, and that belongs to a particular thread.

You should note that threads are often scheduled according to some prioritization scheme (although some systems have one thread per processor core and no scheduling is needed since every thread is always running).

In fact, the value of the Instruction Pointer and the instruction stored at that location is sufficient to determine a new value for the Instruction Pointer. For most instructions, this advances the Instruction Pointer by the size of the instruction, but control flow instructions change the Instruction Pointer in other, predictable ways. The sequence of values the Instruction Pointer takes on forms a path of execution weaving through the program code, giving rise to the name "thread".

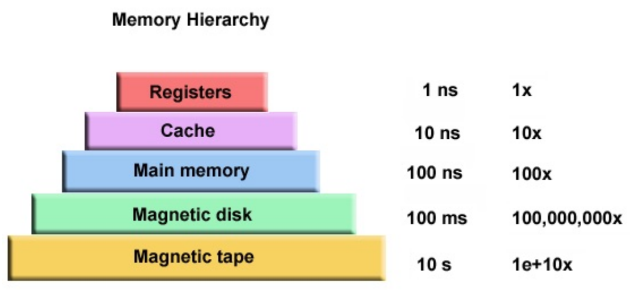
6 What does I/O have to do with parallelization?

I/O is short for input/output (pronounced "eye-oh"). The term I/O is used to describe any program, operation or device that transfers data to or from a computer and to or from a peripheral device. Every transfer is an output from one device and an input into another (Fig. 5). Devices such as keyboards and mouses are input-only devices while devices such as printers are output-only. A writable CD-ROM is both an input and an output device.

6.1 Bad news

- I/O operations are generally regarded as inhibitors to parallelism
- I/O operations require orders of magnitude more time than memory operations

Figure 5: Memory organization in computer architecture.



- Parallel I/O systems may be immature or not available for all platforms
- In an environment where all tasks see the same file space, write operations can result in file overwriting
- Read operations can be affected by the file server's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers

6.2 Good news

Parallel file systems are available. For example:

- GPFS: General Parallel File System (IBM). Now called IBM Spectrum Scale
- Lustre: for Linux clusters (Intel)
- HDFS: Hadoop Distributed File System (Apache)
- PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)

The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

6.3 For those about to I/O

The following are some guidelines for programmers about to I/O, that the interested user might want to check out on the programs they are using to select among different alternatives:

- Rule #1: Reduce overall I/O as much as possible
- If you have access to a parallel file system, use it
- Writing large chunks of data rather than small chunks is usually significantly more efficient
- Fewer, larger files performs better than many small files
- Confine I/O to specific serial portions of the job and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks
- Aggregate I/O operations across tasks - rather than having many tasks perform I/O, have a subset of tasks perform it

7 What does I/O have to do with genome assembly?

I/O matters, and it makes little difference what a computer's computational capability may be if the I/O port is unable to keep up with the processor.

I/O bottleneck

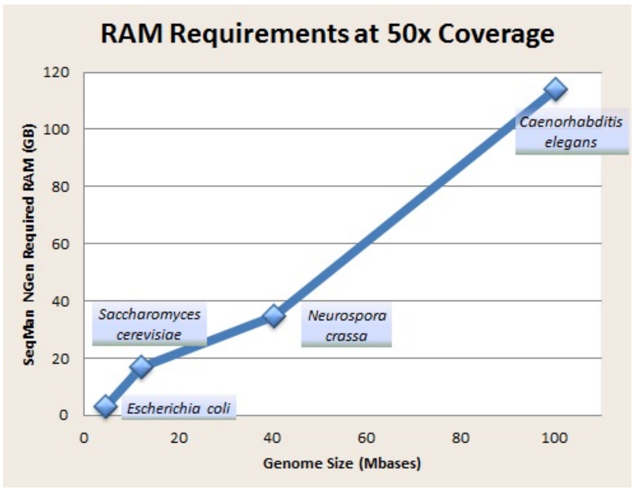
An I/O bottleneck is a problem where a system does not have fast enough input/output performance.
Many problems associated with I/O bottlenecks happen because data transaction processes simply have not matched the advancements in processor speed and memory capacity. Some point to Moore's law, which refers to the doubling of transistors on a circuit, which also supported the tremendous growth in computer processing and memory storage throughout the late 20th and early 21st centuries. When I/O cannot keep up, significant I/O bottlenecks emerge.

In bioinformatics, you will need to always care about I/O problems. Especially after the development of next-generation sequencing (NGS; *i.e.*, second-generation) technologies, since genome assembly has become one of the most computational and I/O intensive analyses done with genomic data (read more here.)

Most rigorous *de novo* genome assemblers require adequate random-access memory (RAM) to cluster sequence data into "contigs". Several factors contribute to memory usage, some of which are obvious, like the number of reads, and others that are more difficult to estimate, like genome complexity. The most important factors include:

- Number of reads
- Genome Size
- Read Length

Figure 6: Memory requirements during *de novo* assembly.



- Genome Complexity
- Read accuracy

NGS users generally have access to paired-end Illumina data, at least 100bp in length, with at least 50× coverage across the genome.

Based on these assumptions, Matthew Keyser did a small experiment (see original post at DNASTAR here) where he downloaded four Illumina 2 × 100 paired-end datasets from the Short Read Archive (SRA), for four organisms (*E. coli*, *S. cerevisiae*, *N. crassa*, and *C. elegans*) and used SeqMan NGen to restrict input reads to obtain 50× coverage before assembly. RAM usage was monitored during assembly so that peak max committed memory (physical RAM + disk paging files) could be recorded.

The graph in Fig. 6 plots memory usage *vs.* genome size and reveals to us a rough "rule of thumb" that when using "typical" Illumina data at "typical" coverages, 1GB RAM is required for every 1Mbase of the genome, a much more linear relationship than expected.

Class discussion

Read the following questions and think about them before we talk about them in class:

1. Which is better for phylogenetics, lots of threads or lots of RAM?
2. Which is better for phylogenetics, CPUs or GPUs?
3. Which is better for genome assembly, lots of threads or lots of RAM?
4. Which is better for genome assembly, CPUs or GPUs?

This document was created on May 22, 2018 and last updated on October 19, 2018.