Supercomputers and the message-passing interface

What is a supercomputer?

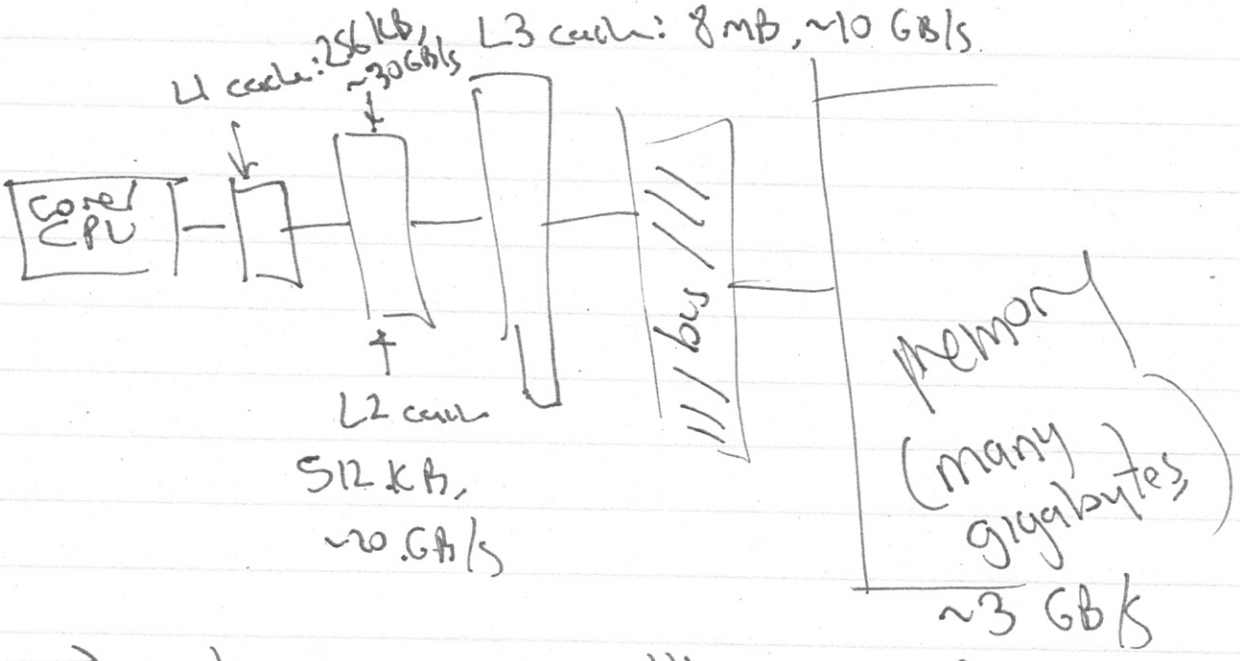
This is a question that is challenging to answer, and it evolves over time. It's useful to consider how a "computer" and a "supercomputer" differ.

"computer": a handful of computing cores on a CPU, connected to some memory via a "bus" that transfers data. Sometimes there are more than 1 CPU:

- each CPU has 4 cores; 2 CPUs talk to memory over the bus. All cores share this memory, but each core often has its own "piece".

- Why multiple cores? CPUs consume more power with higher clock frequency (2 GHz $\simeq 2\times10^9$ ops/second, so higher freq. = faster computation). Power consumption $\propto$ (frequency)^2 or more, so having more but slower cores is overall better for performance and power usage.

note that each CPU has its own very fast memory called "cache": faster is always more expensive, so you have less of it.



→ each core in a multi-core CPU typically has its own L1/L2 cache, but may share L3 cache.

A <u>supercomputer</u> is often built out of commodity hardware (i.e., similar to a modern desktop computer), though some are quite exotic. In general, there are a few characteristics:

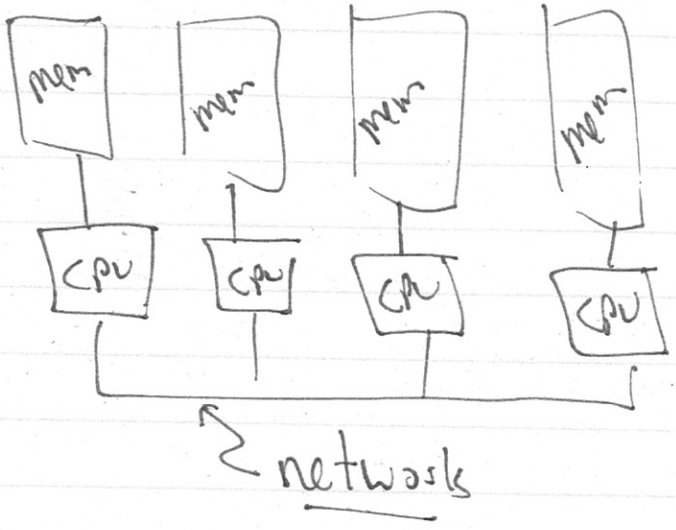
- <u>many</u> and/or <u>very fast</u> CPUs (or GPUs, co-processors, FPGAs, ...)
- a high-bandwidth, low-latency interconnect between CPUs/computing "nodes"
- a fast, parallel file system.

Historically, supercomputers were classified as "shared" and "distributed" memory architectures:

Shared:

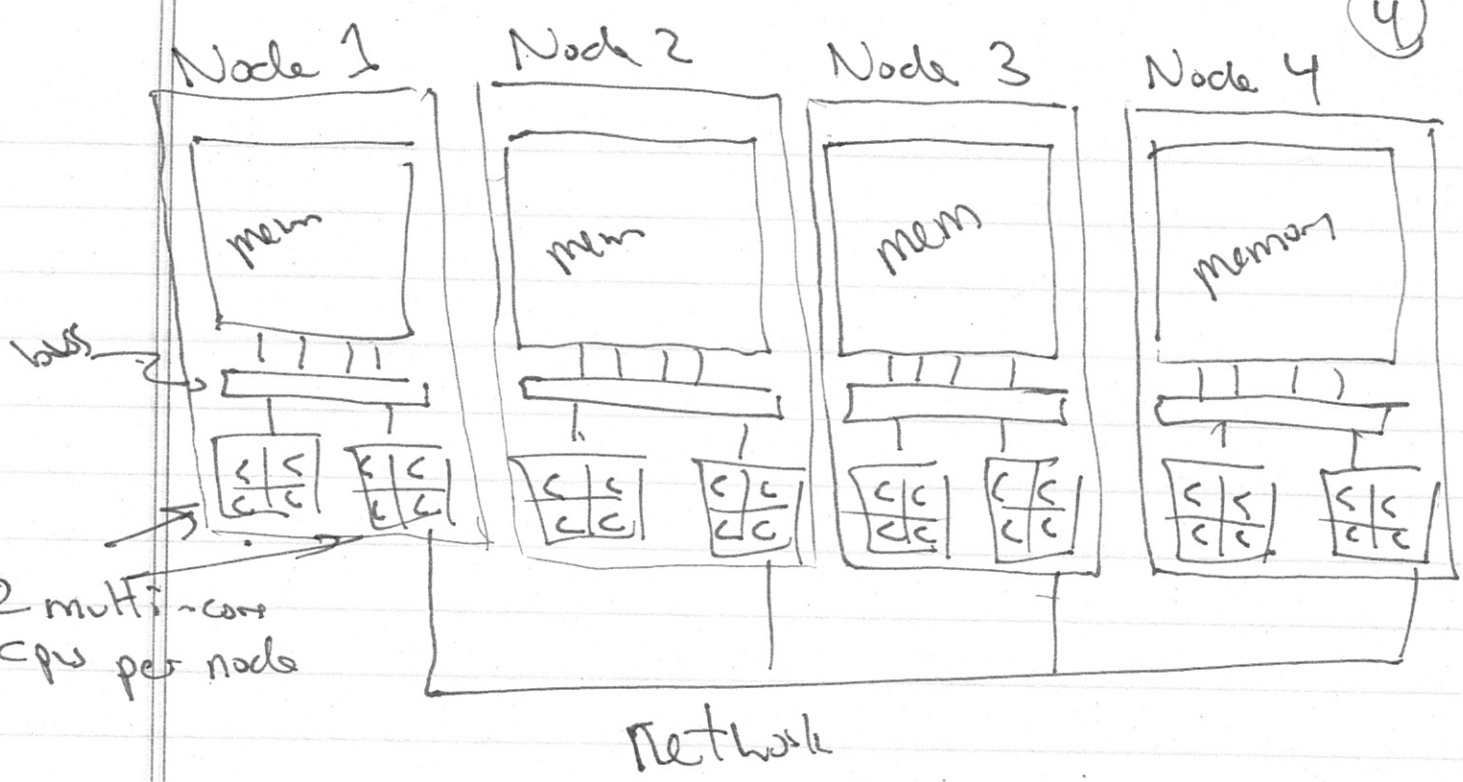
all CPUs have access to all memory.

Distributed:



each CPU has access to a block of memory directly, but not all: need to communicate somehow.

modern supercomputers are often composed of nodes of 1 or more multi-core CPUs that have shared access to local memory, and are connected to other nodes via a network. These are "distributed shared memory" architectures.



This architecture is driven by money and power (as are so many things):

- gaming/PC market pushes for "commodity" chips w/ moderate clock speed and many cores: great for MS Word, less great for supercomputing.

- lower clock speed but many cores saves electricity and generates less heat (which must be removed) → electricity cost is a huge cost as fraction of supercomputer price tag.

One of the fundamental challenges in modern supercomputing is mapping problems onto architectures (breaking it up effectively so it runs quickly).

What models are available to us?

Shared-memory parallelism: "shared memory" machines let you keep only 1 copy of data and all CPUs access it. "Threaded" model for programming → can tell each core what to do simply. But, shared-memory computers get challenging ($$) to build as you go to more CPUs/cores; impractically expensive beyond ~512-1024 cores.

distributed memory (really, "distributed shared memory") machines require problem to be broken up into many parts, and the programmer has to put a lot of thought into getting the different pieces to talk to each other. (message-passing models). More difficult to do, but can scale to much bigger problems because you can use commodity hardware.

task-based parallelism: different processes/cores run different tasks, and data is accessed centrally or handed from task to task. This is analogous to an assembly line, where the individual workers each do something different.

data parallelism: data is distributed to different processes/nodes, which each do the same thing to their own piece of the data. This is analogous to painting all rooms in a house by assigning several painters to each take care of one room.

Note that these are not hard lines: you can combine different types of parallelism, since they are often complementary.

Tools for shared memory / task based parallelism include OpenMP and pthreads (or in Python, the "threading" and "multiprocessing" modules).

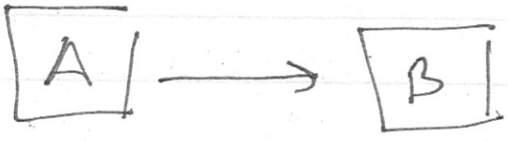
Tools for distributed memory / data-based parallelism are many, but the most common is the Message Passing Interface, or MPI. This is accessed through mpi4py in Python.

How does MPI work?

An MPI communicator (pool of MPI tasks) is of a given size (# of tasks in pool). Each task is assigned a rank (unique identifier). Think of a task as a process that can do some computation. The number of tasks (size) is typically constant.

Tasks communicate with each other by sending messages composed of data, and by receiving those messages. This communication can be point-to-point (i.e., processor A sends process B information), or a more global process (e.g., all processes send process A a specific piece of information, or all processes communicate to find the global minimum $ value in some dataset).

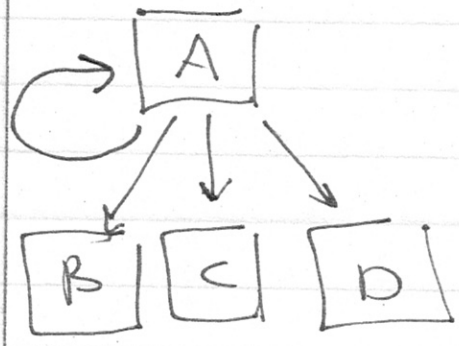
Typical tasks look like this (arrow indicates information flow between processes).



A sends, B receives.

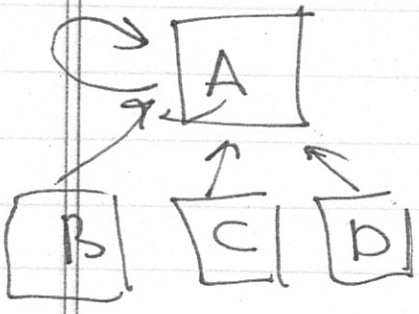
(this is "point-to-point" communication)

Collective communication



"Scatter" A breaks up data, gives different parts to itself, B, C, D.

"Broadcast" A sends copies of entire dataset to itself, B, C, D.



"Gather": A collects pieces of dataset from all tasks, including itself, and reassembles them. This is the inverse of a scatter operation.

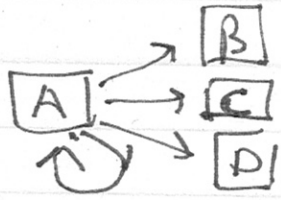
"All gather": as above, but all tasks get copies of the entire dataset in the end.

"Reduce": A collects data from all tasks and performs a global operation of some sort. (sum, find minima/maxima, product, Boolean operations, etc.). At the end, only A has the result of the operation.

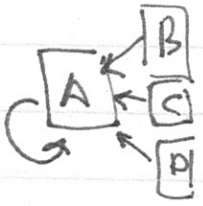
"All reduce": as above, but all tasks end up with the result of the global operation.

An example of a simple MPI program might behave like this

(1) program begins: task A sends initialization data to all tasks; all start computing.

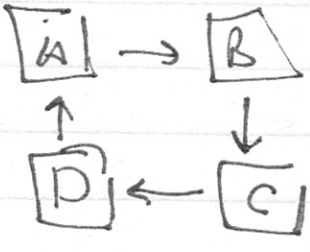


(2) At a later time, all CPUs call a "reduce" operation to find the minimum value of a quantity across the tasks; A is in charge. A "barrier" is used to ensure no task rushes ahead.

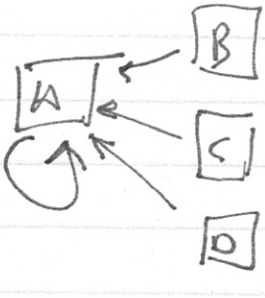


(3) A broadcasts the data back out; computation resumes.
(note: (2) + (3) is equivalent to an "AllReduce")

(4) Each task shares information with one other



(5) All tasks give data back to A using a "gather" operation; A writes to disk.



Success!

The biggest challenge is often figuring out how to map a problem to a supercomputer.

Small group exercise: think about how you might parallelize one of the following things, make pseudo-code, and report back:

(using the operations we discussed)

1. the solution of the heat equation on a 2D cartesian grid.
2. numerical integration of a function $f(x)$ using the trapezoidal rule
3. calculating all prime numbers between arbitrary N_{small} and N_{big}
4. calculating the forces between all stars in a galaxy using a direct sum technique.
5. building a graph of relationships between people in facebook to try to find social groups
6. multiplying two very large matrices together (too big to store on 1 core).

mpi4py → go to the code!

comm =
- MPI.COMM_WORLD → the communicator
- size: comm.Get_Size() → # of tasks
- rank: comm.Get_rank() → my rank

point-to-point:

Send() / Recv(): send and receive memory buffers (such as numpy arrays).

send()/recv() can do the same w/ arbitrary python objects (lists, objects, dictionaries, etc.) but are much slower.

→ these are "blocking" methods: the caller cannot move on until the data has accomplished what it's supposed to, and can be safely reused!

python also allows for non-blocking sends/receives: Isend() / Irecv(). This allows communication and computation to be overlapped to help performance. This is powerful but dangerous.

collective communications

- all the things we talked about before:

Bcast() / bcast()	-	broadcast
Scatter() / scatter()	-	scatter
Gather() / gather()	-	gather op
Allgather() / Allgather()	-	"all gather"
Reduce() / reduce()	-	reduce
Allreduce() / allreduce()	-	"all reduce"

just memory buffers (e.g. numpy arrays) — arbitrary Python objects → slow!

these require different arguments; see documentation and examples, particularly Jeremy Bejarano's mpi4py tutorial.

One further method: Barrier(): mostly useful for send/receive calls, since the collective calls build it in. This method keeps the code from moving on until all tasks reach that point. Very helpful.

Last things to do:

1. Work through examples in mpi-examples directory.
2. implement Monte Carlo integration in parallel.
3. implement Traveling Salesman in parallel.