CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

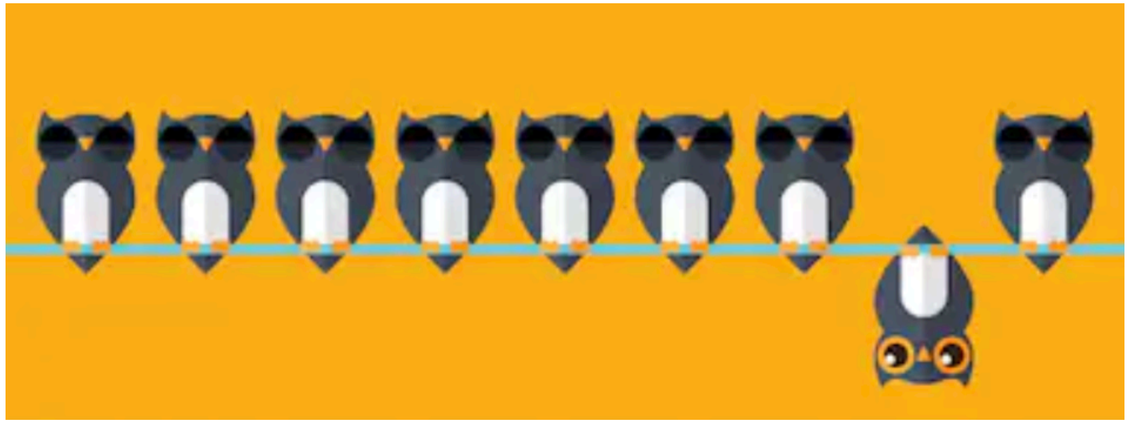
"Computers are getting smarter all the time. Scientists tell us that soon they will be able to talk to us. (And by 'they', I mean 'computers'. I doubt scientists will ever be able to talk to us.)" (Dave Barry)

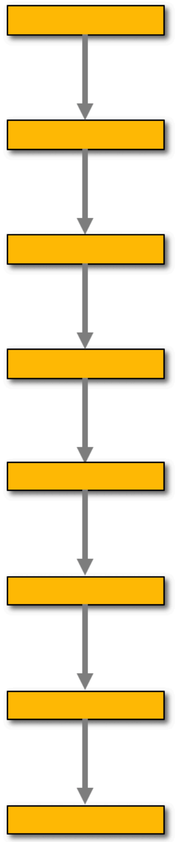
Stanford University

Example of a Parallel Computation

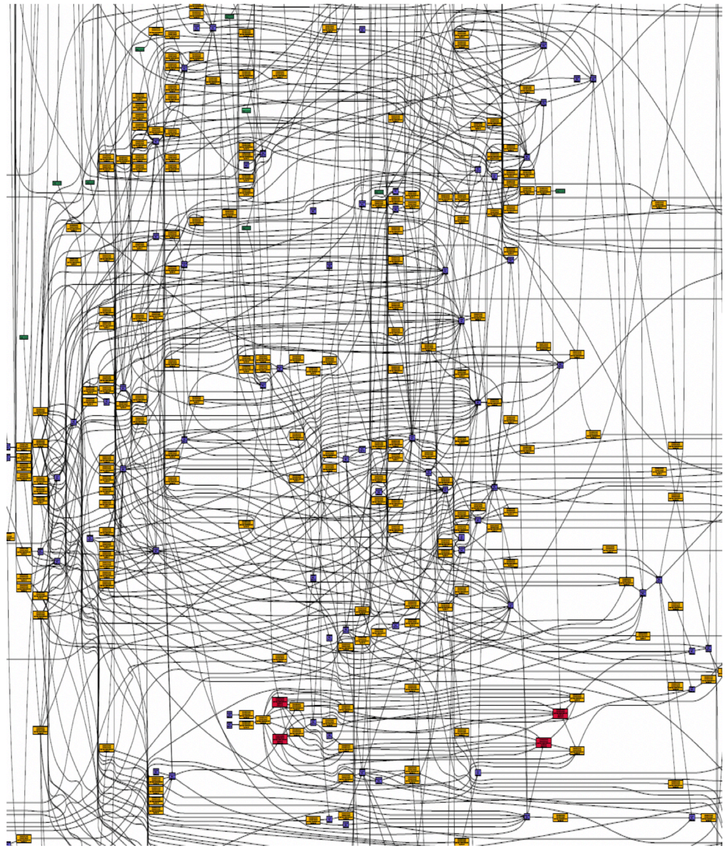
Parallel programs often look very different from sequential programs

- Parallel programming requires a different type of thinking.
- It’s more complicated and difficult to debug that sequential programs.





Sequential



Parallel

- Sequential programming is easy to understand.
- Parallel programming requires understanding dependencies between computational tasks and how data flow from one task to another.

Example: program to sum numbers

- Assume we want to calculate the sum of n numbers.
- This seems simple enough with a sequential code.

```
for (int i = 0; i < n; ++i) {
  float x = ComputeNextValue();
  sum += x;
}
```

Parallel computing

- We have p cores that can compute and exchange data.
- Can we accelerate our calculation by splitting the work among the cores?

Split the work

- First, we split the work across the p processing units (PU).
- Each PU computes a chunk of the sum.
- The chunk size is b.

```
int r; /* thread number */
int b; /* number of entries processed */
int my_first_i = r * b;
int my_last_i = (r + 1) * b;
for (int my_i = my_first_i; my_i < my_last_i; my_i++) {
  float my_x = ComputeNextValue();
  my_sum += my_x;
}
```

Final sum

- Not that simple
- At this point, each core has computed a partial sum.
- All these partial sums need to summed up together.
- We have p numbers that need to be added using p PUs or cores.

Sequential approach

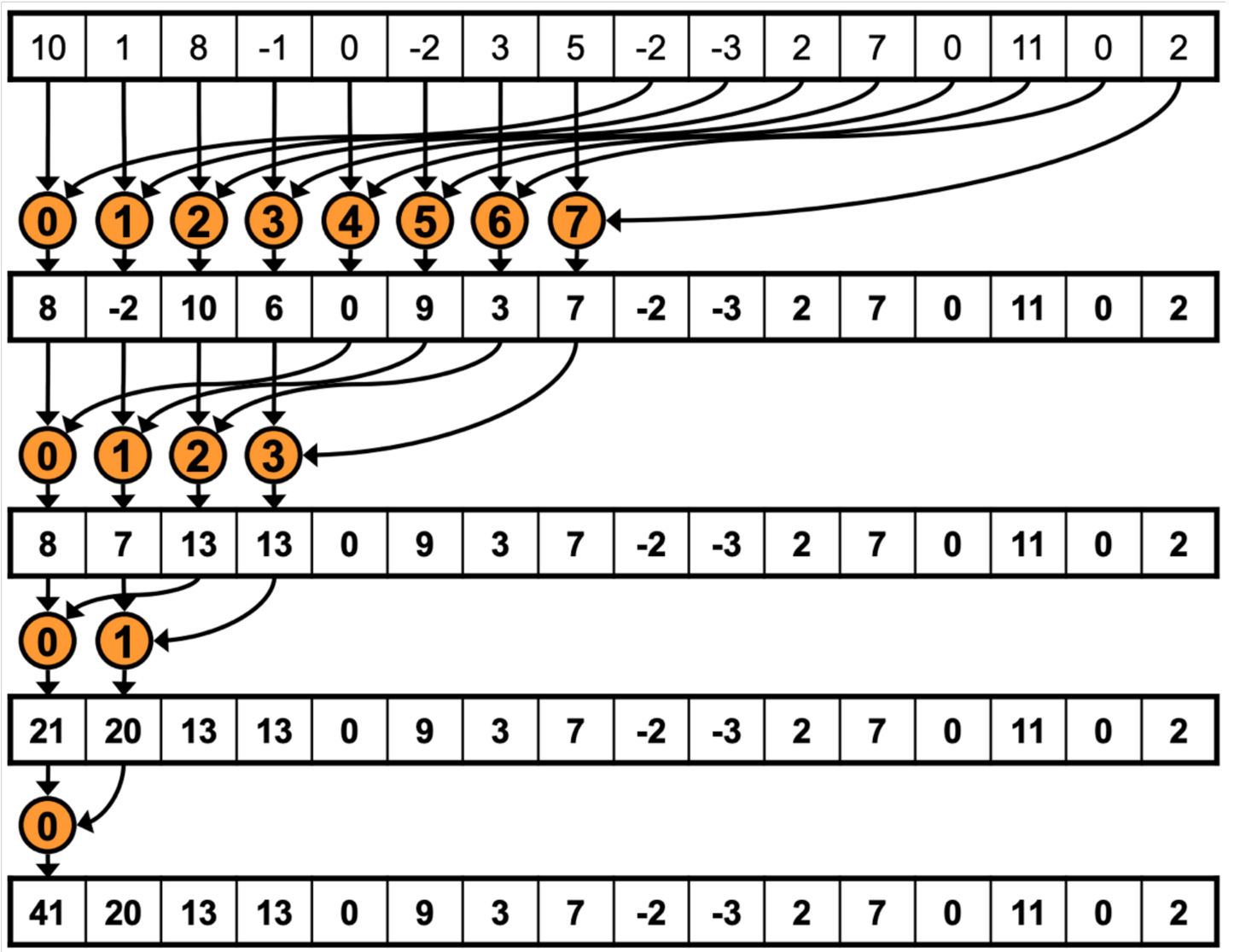
- The simplest.
- Have one main thread do all the work.
- Thread = PU or core doing work.
- ReceiveFrom: receive data from remote thread.
- SendTo: send data to main thread (with ID 0).

```
if (r == 0) /* main thread */ {
  float sum = my_sum;

  for (int r_mote = 1; r_mote < p; ++r_mote) {
    float sum_r;
    ReceiveFrom(&sum_r, r_mote);
    sum += sum_r;
  }

} else /* worker thread */ {
  SendTo(&my_sum, 0);
}
```

Sequential bottleneck

- But that may not be enough.
- **If we have many cores, this final sum may take a lot of time.**
- This is true for example on GPUs where the number of cores is very large.
- Example: NVIDIA H100 SXM5 has 16,896 FP32 cores.
- In many applications, it may become comparable with the size of the loop.

- Binary tree can be used to perform the reduction efficiently.
- $p = 8$ in this example.
- Number of passes or stages is $\ln_2(p) + 1$.
- This algorithm is used to compute reductions on GPUs.



Parallel algorithms

- This simple example illustrates the fact that it is difficult for a compiler to parallelize a program.

- Instead the programmer must often re-write his code having in mind that multiple cores will be computing in parallel.

- Key questions:

1. How can the workload be distributed across the different cores? That is: what is each core going to compute?

2. How can we estimate the runtime of an execution in parallel?

3. What are the performance bottlenecks?

Shared Memory Processors

Memory organization

- Despite the availability of many cores, performance is often driven by memory accesses.
- **Assigning 2x the number of cores to a calculation does not typically result in a 2x speedup!**
- Memory access times need to be accounted for and in most cases are the bottleneck.

Problems with memory access time

- **Memory access time** can not be reduced at the same rate as the processor clock period. This leads to an increased number of machine cycles for a memory access.

- **The speed of signal transfers** within the wires is a limiting factor. For example, a 3 GHz processor has a cycle time of 0.33 ns. Assuming a signal transfer at the speed of light, a signal can cross a distance of ~10 cm in one processor cycle. This is not significantly larger than the typical size of a processor chip. **Wire lengths become an important issue.**

- The physical size of a processor chip limits the **number of pins** that can be used, thus limiting the bandwidth between CPU and main memory.

- This may lead to a processor-to-memory performance gap which is sometimes referred to as **memory wall.**

- Many hardware improvements have been made to mitigate these shortcomings.

- They mostly involve using multiple cores and a distributed hierarchical memory.

Overview of key components of the architecture

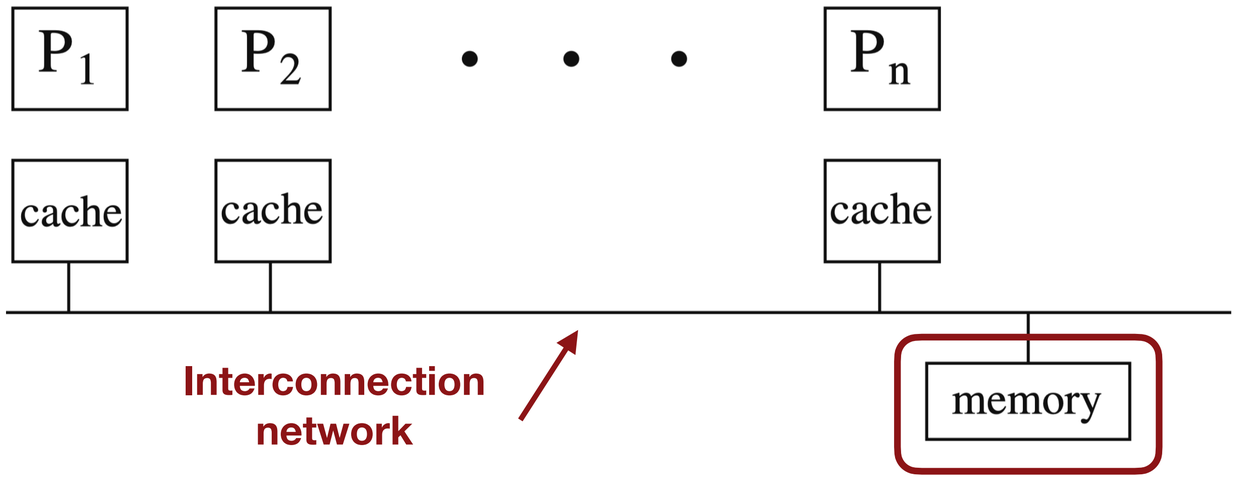
Components of a multicore computer node:

1. A number of processors or cores (single or multiple CPU sockets)
2. A shared physical memory (single or distributed global memory)
3. Cache memory
4. An interconnection network to connect the processors with the memory

Typically designed to deliver performance for both sequential and parallel programs.

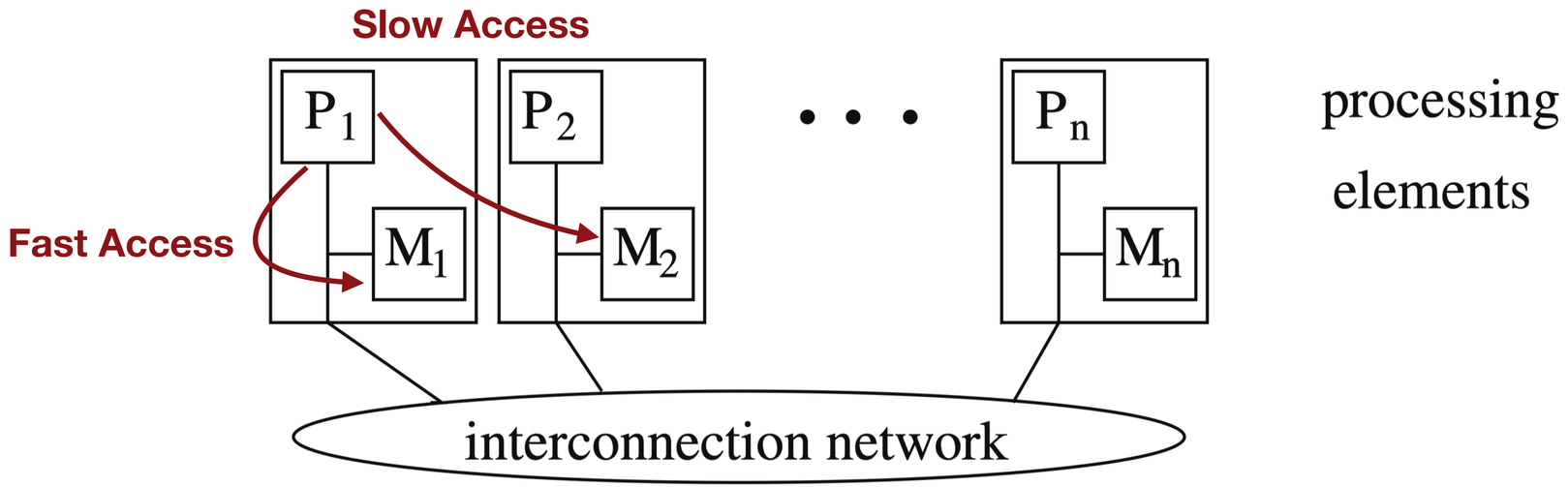
Symmetric multiprocessors

- The simplest but not the one with highest performance.
- A single global memory is shared among all processors.
- All processors have the **same access time** to memory = symmetric.



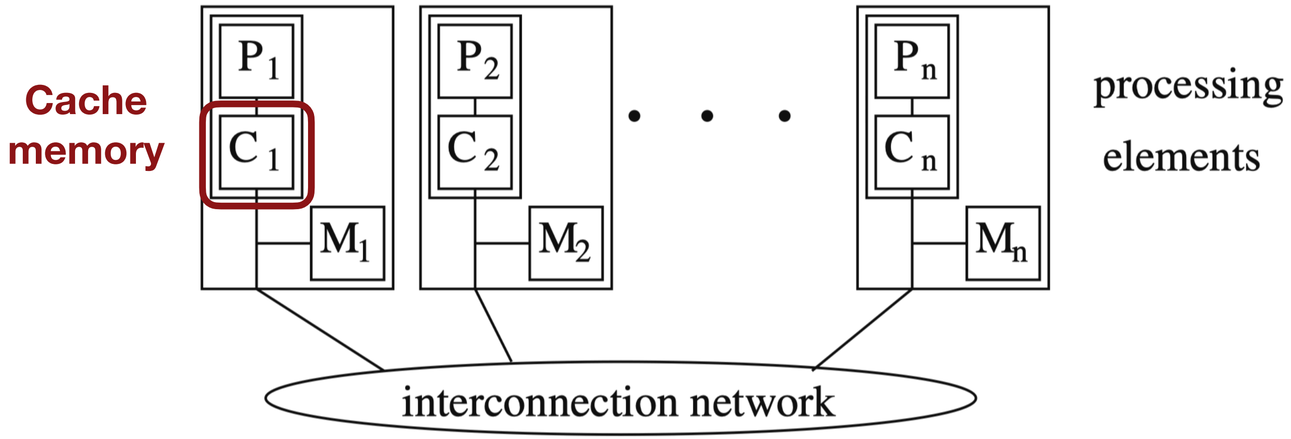
NUMA

- **Non-uniform memory access** = NUMA
- It is difficult to ensure high performance with a single shared memory.
- Instead, in most systems, the memory is **distributed,** and processors are connected to the different memories using a network.



CC-NUMA

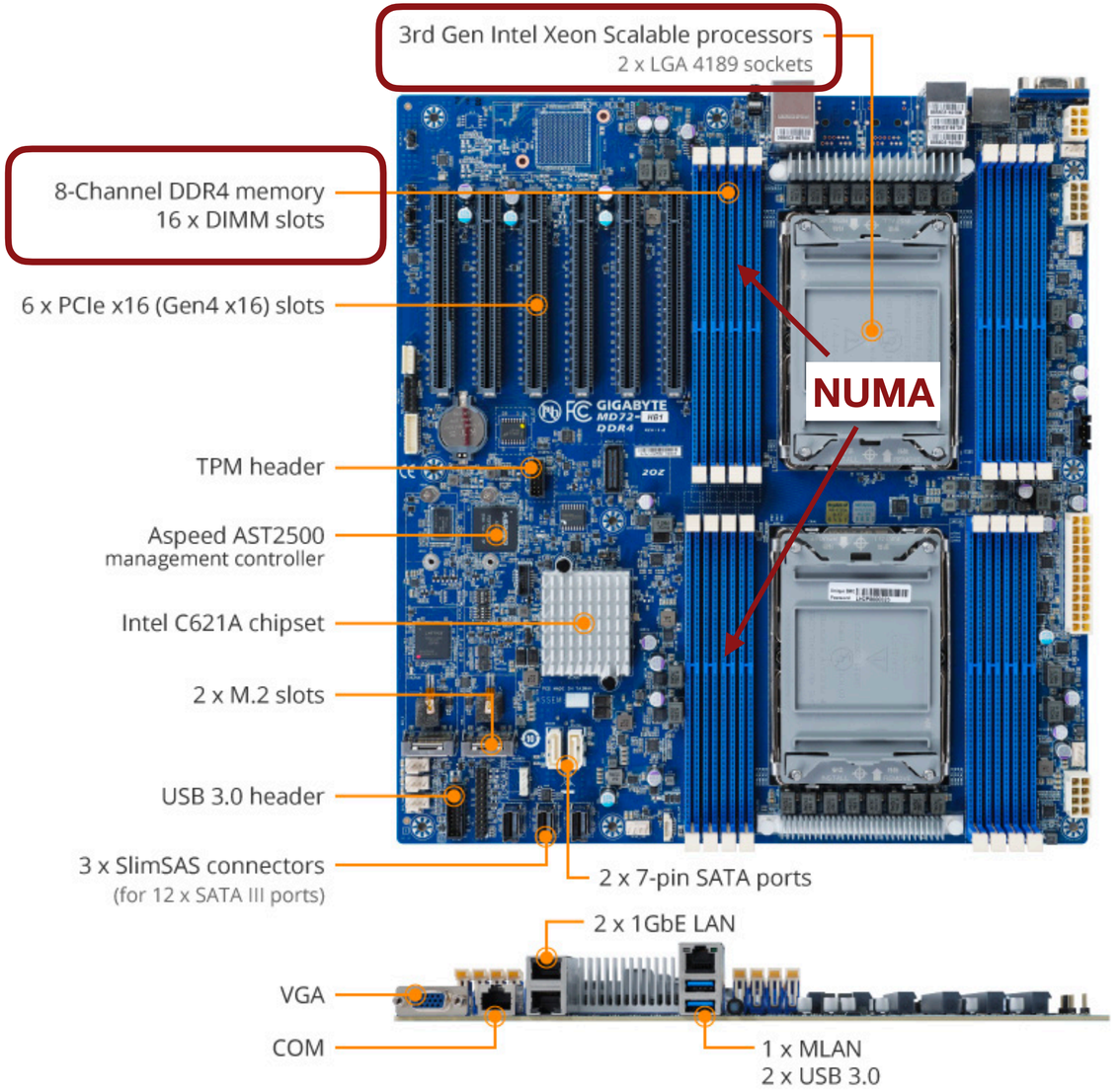
- Cache-coherent NUMA = CC-NUMA
- Most memory systems make use of **cache memory** to reduce the memory traffic.
- Cache is a **small** memory that is **much faster** than global memory.
- When a piece of data is read from global memory, it is stored in the cache. Subsequent reads use the value in the cache rather than global memory. As a result, this reduces the memory traffic significantly and speeds-up the program.



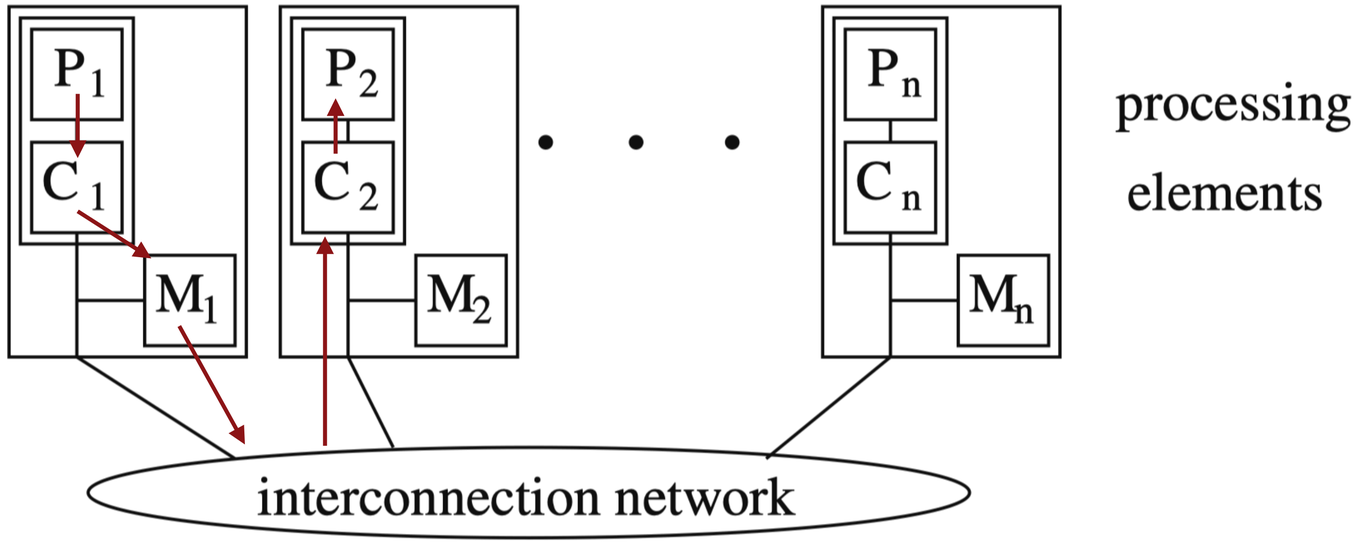
Cache-coherent NUMA or CC-NUMA

Example of CC-NUMA: Bulldozer server (AMD)

Gigabyte motherboard
3rd Gen Intel Xeon—“Ice Lake”

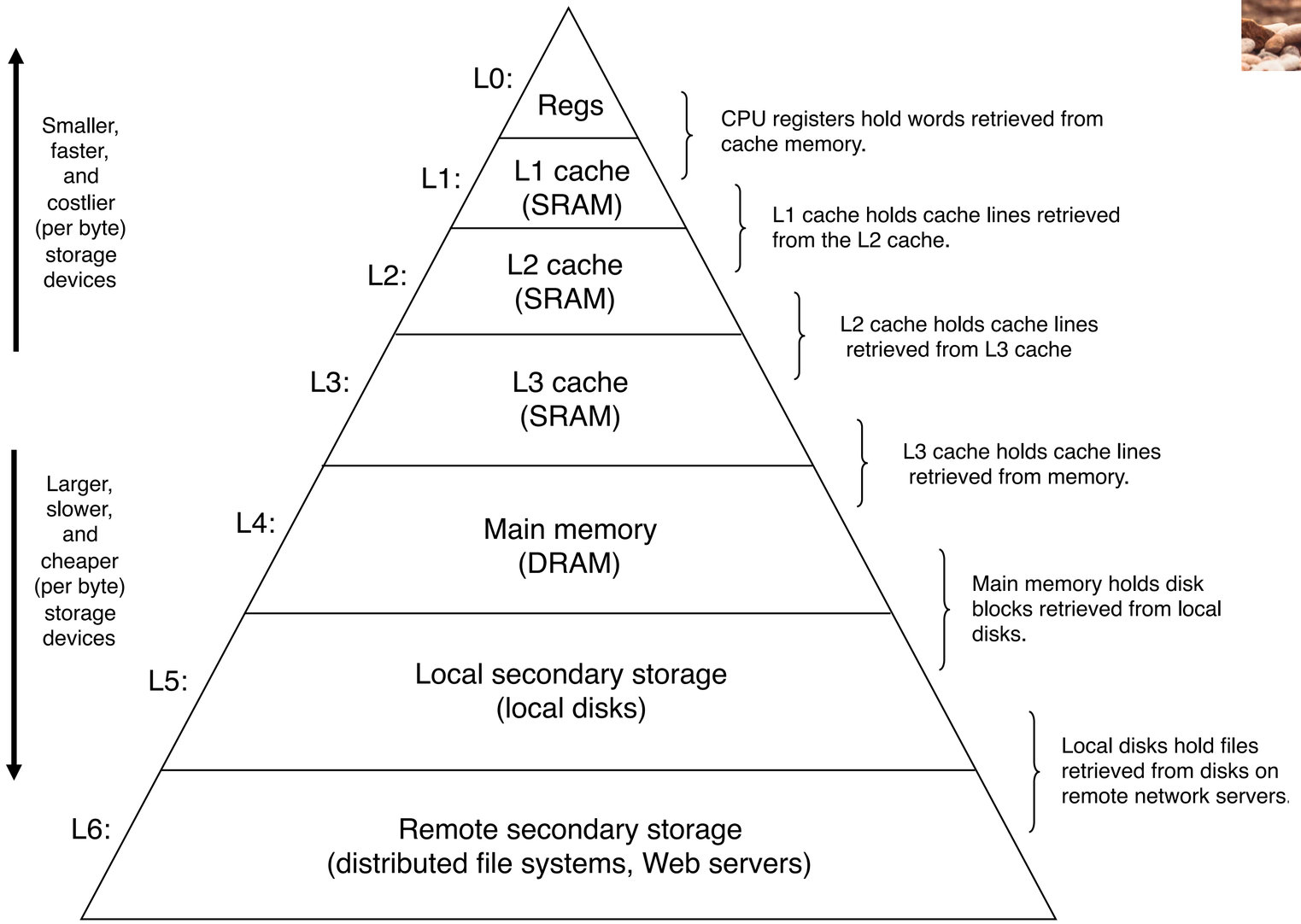


Cache coherence



- **Cache coherence** is a complex problem.
- Basic difficulty: processor P_1 writes data to memory x. Processor P_2 reads from memory x at a later instruction in the program.
- How can we make sure that P_2 gets the most recent value?
- Data x needs to be copied from cache to global memory, then from global memory to the cache of P_2.
- To ensure cache coherency, a **cache coherency protocol** must be used.
- This topic will not be further covered in this class.

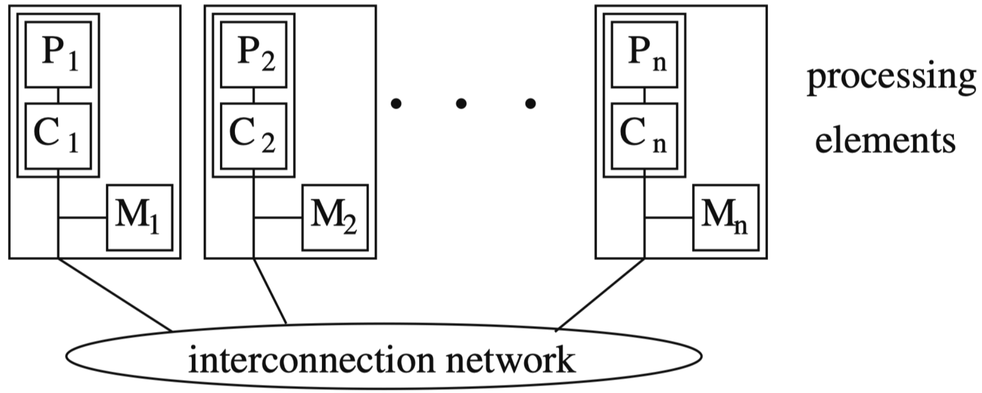
Memory hierarchy



Latency and bandwidth

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
MCDRAM		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	Similar to MCDRAM	100 GB/second
Main memory on Cornelis* Omni-Path Fabric	Limited only by cost	Depends on distance	Depends on distance and hardware
I/O devices on memory bus	6 TB	100x-1000x slower than memory	25 GB/second
I/O devices on PCIe bus	Limited only by cost	From less than milliseconds to minutes	GB-TB/hour Depends on distance and hardware

Where is my data?



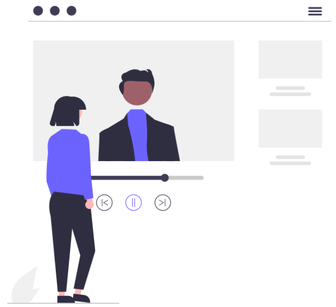
- When running a multi-threaded program, you have multiple processing units working on your computation at the same time.
- They all read from/write to memory.
- In general, it is best if a processing unit works with data that is “close,” that is access to that memory is fast.
- How can we know which memory a piece of data is in and whether it is close to the processing unit?

Memory allocation

- When you allocate dynamic memory, the memory is not immediately allocated in physical memory.
- The OS waits until the first write occurs.
- At that point, the page associated with that piece of data is allocated. By default, it is allocated closest to the processing unit that executes the write.
- **Performance tip: the processing unit working with a memory location the most should be the one touching it first.**
- We will see later how this can be used to write efficient programs.

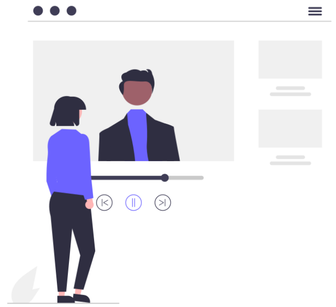
```
void first_touch() {
  const int n = 10000;
  float *A;
  A = new float[n * n];
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) A[i * n + j] = 0.0f;
}
```

Performance tips: summary



- Memory is key to developing high-performance multicore applications.
- **Memory traffic and time to access** memory are often more important than flops.
- Memory is **hierarchical and complex.**
- Memory traffic should be reduced through algorithmic changes. Goals are:
  - Increase access to **data in cache**
  - **Reduce traffic** between cache and global memory

Advanced tips



1. **Spatial locality:** the memory accesses of a program have a high spatial locality if the program often accesses memory locations with **neighboring addresses** at successive points in time during program execution.

2. **Temporal locality:** the memory accesses of a program have a high temporal locality if it often happens that the same memory location is **accessed multiple times at successive points in time** during program execution.

3. **Memory affinity:** it is best to execute instructions on cores that are closest to the memory they need to access.

Summary

- In this class, we won't have time to dive into these more advanced topics.
- They are important to write high-performance linear algebra library and for applications that need to be highly tuned for performance.
- In practice, it is difficult to optimize complex engineering programs based on all these considerations.
- **Take-home message: performance of multicore processors is a complex topic; assigning more cores does not necessarily lead to computational speed-up.**

Processes and threads

Process

- A process, in the simplest terms, is an executing program.
- Each process provides the resources needed to execute a program.
- A process typically has:
  - a unique process identifier,
  - a virtual address space,
  - executable code,
  - open handles to system objects, a security context, environment variables, a priority class, minimum and maximum working set sizes, and
  - at least one thread of execution.

Process and threads

- Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

- A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

Threads

- A thread is the entity within a process that can be scheduled for execution.
- All threads of a process share its virtual address space and system resources.
- In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled.
- The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.

Threads from the programmer's point of view

- In a C++ code, a thread is able to run a procedure in the program independently from the main process thread.

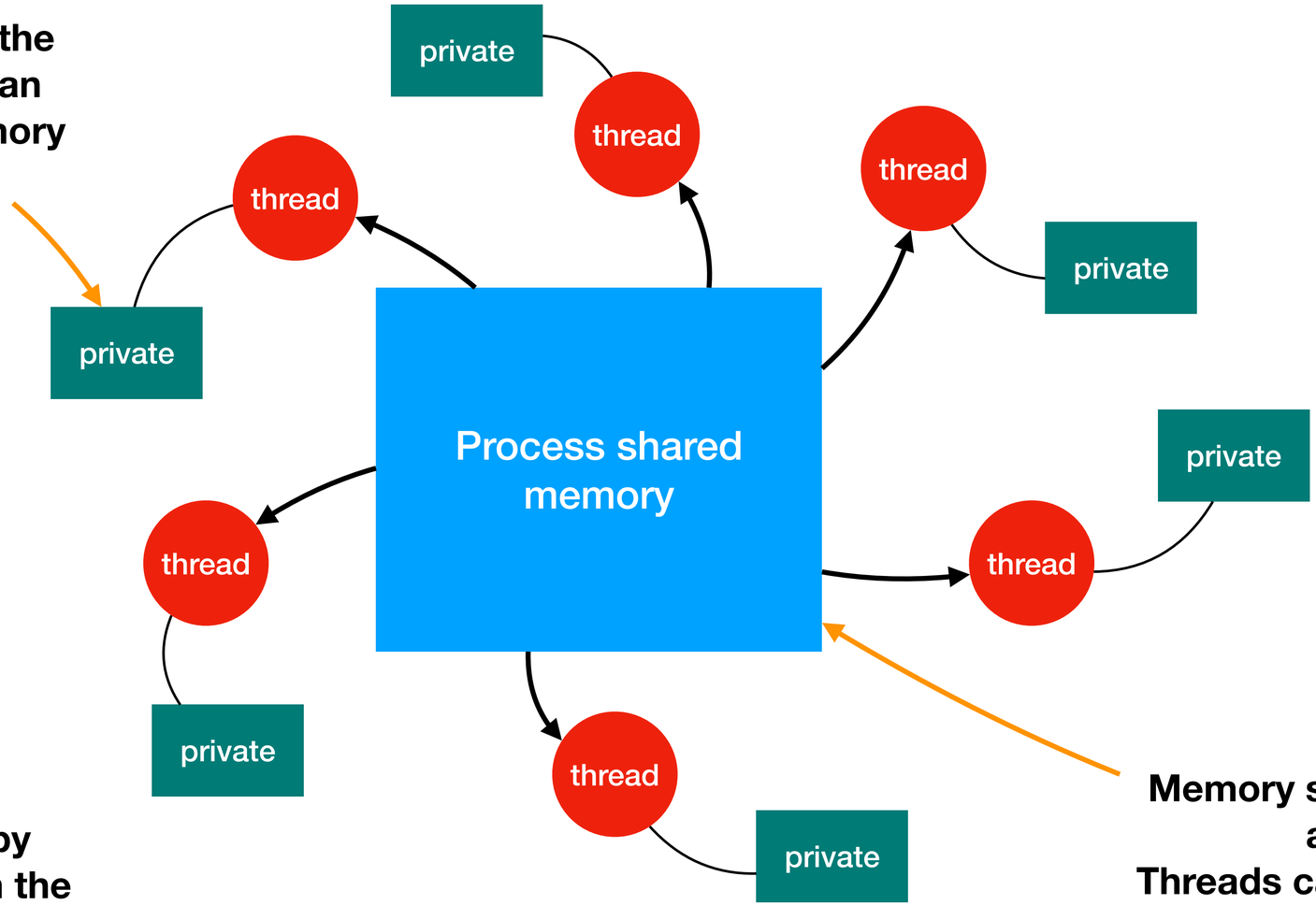
- Imagine a program that contains a number of procedures. Then imagine these **procedures** being able to be scheduled to **run simultaneously and/or independently** by the operating system. This describes **a multi-threaded program.**

Shared address space

- Multicore programming is the simplest style of parallel programming.
- This is because all threads in a process share the address space of the process, i.e., they have a common address space.
- When a thread stores a value in the shared address space, another thread of the same process can access this value.

Thread communication

This memory is private to the thread. No other thread can read and write to that memory space.



Threads exchange data by writing to and reading from the shared memory space.

**Memory space accessible to all threads.
Threads can read and write to that space.**