Parallel Hardware

All material not from online sources/textbook copyright © Travis Desell, 2012

Overview

1. Von Neumann Architecture
2. Processes, Multitasking, Threads
3. Modifications to the Von Neumann Architecture
   a. Caching
   b. Virtual Memory
   c. Instruction Level Parallelism
   d. Hardware Multithreading
4. Parallel Hardware
   e. SIMD Systems
   f. MIMD Systems
   g. Interconnection Networks
5. Performance
   a. Latency/Bandwidth
   b. Speedup and Efficiency
   c. Amdahl's Law
   d. Scalability
   e. Taking Timings

Von Neumann Architecture

Von Neumann Architecture

Your standard CPU model is the *Von Neumann Architecture*. While most CPUs don't follow this too closely any more, it is still a good model for understanding how they work at a higher level.

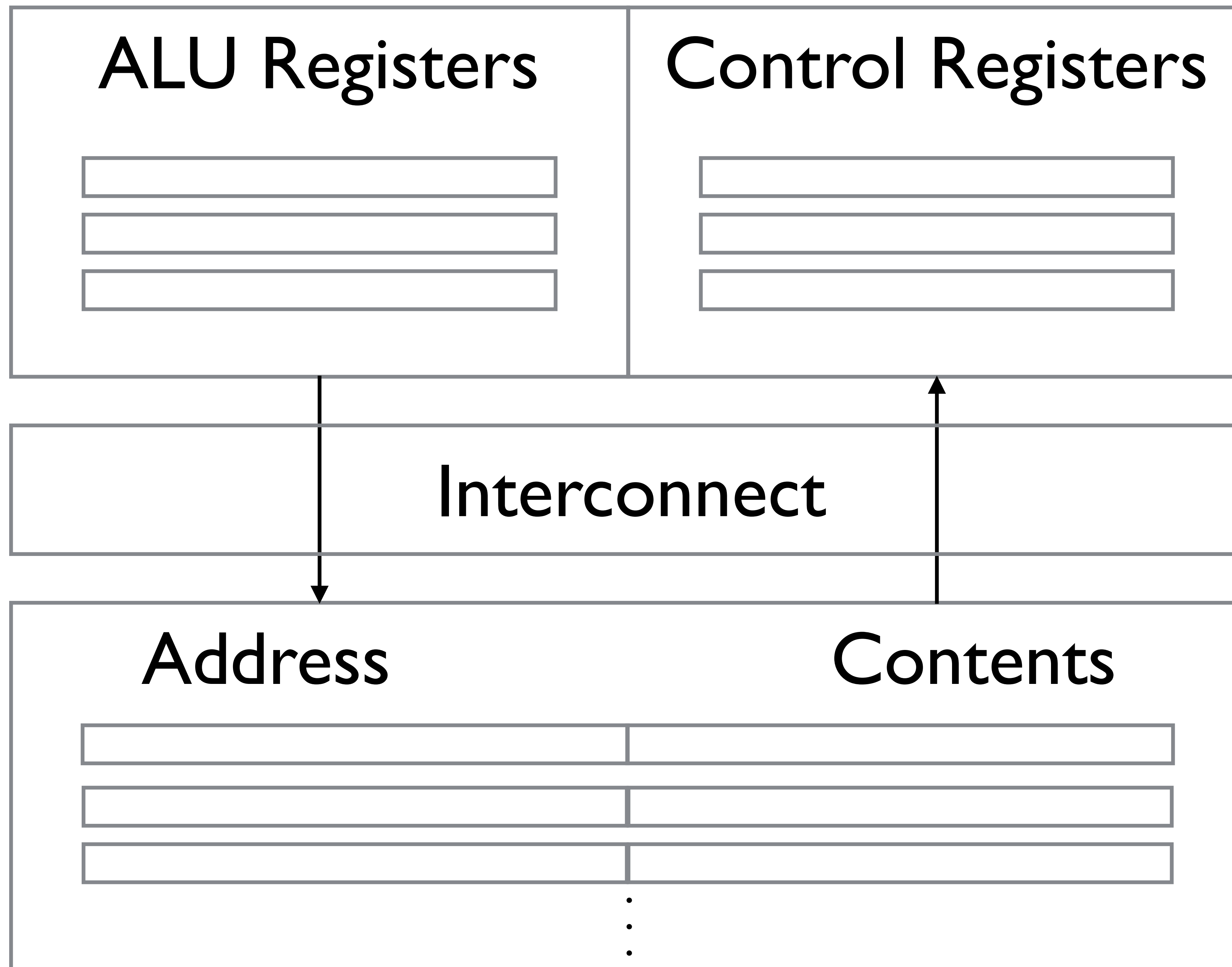
Von Neumann Architecture

The Von Neumann architecture consists of:

- Central Processing Unit (CPU)
- Main Memory
- Processsor/Core
- Interconnection between the Memory and CPU

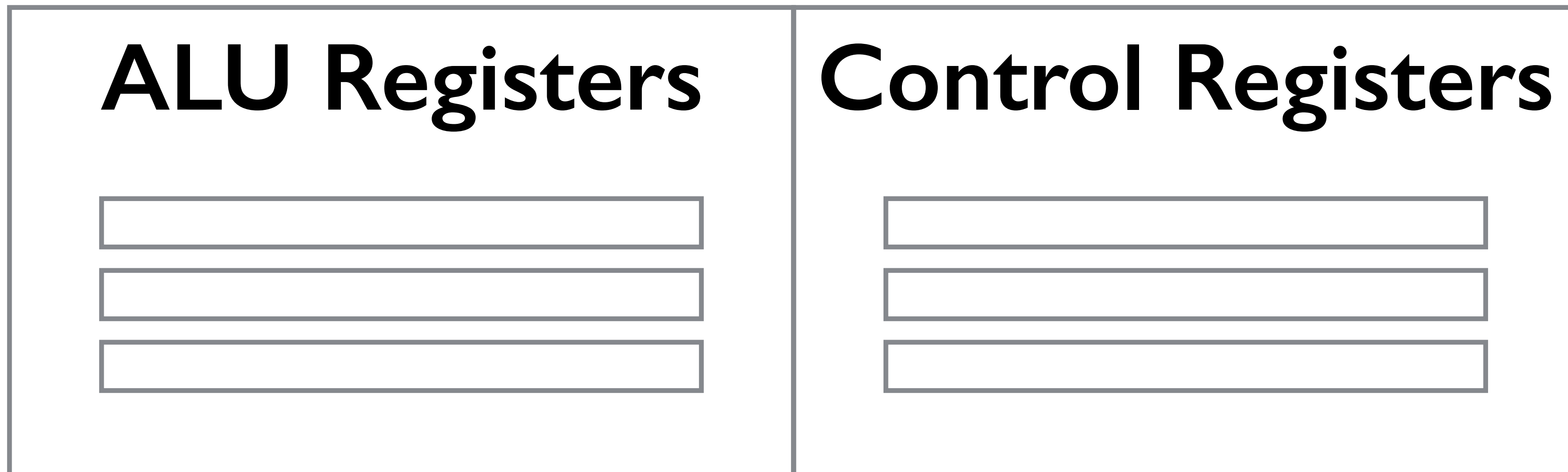
Main memory is a set of locations, which can store either instructions or data. Each location has an address (almost like a hashtable!), which is used to access the data/instruction at it.

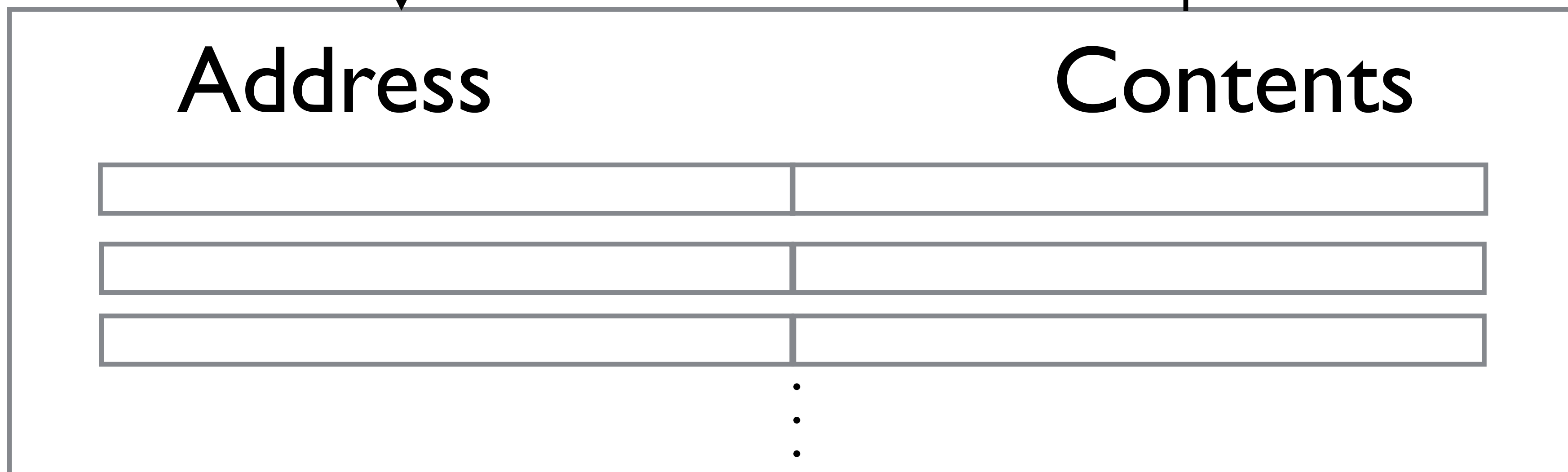
Von Neumann Architecture



Von Neumann Architecture

CPU





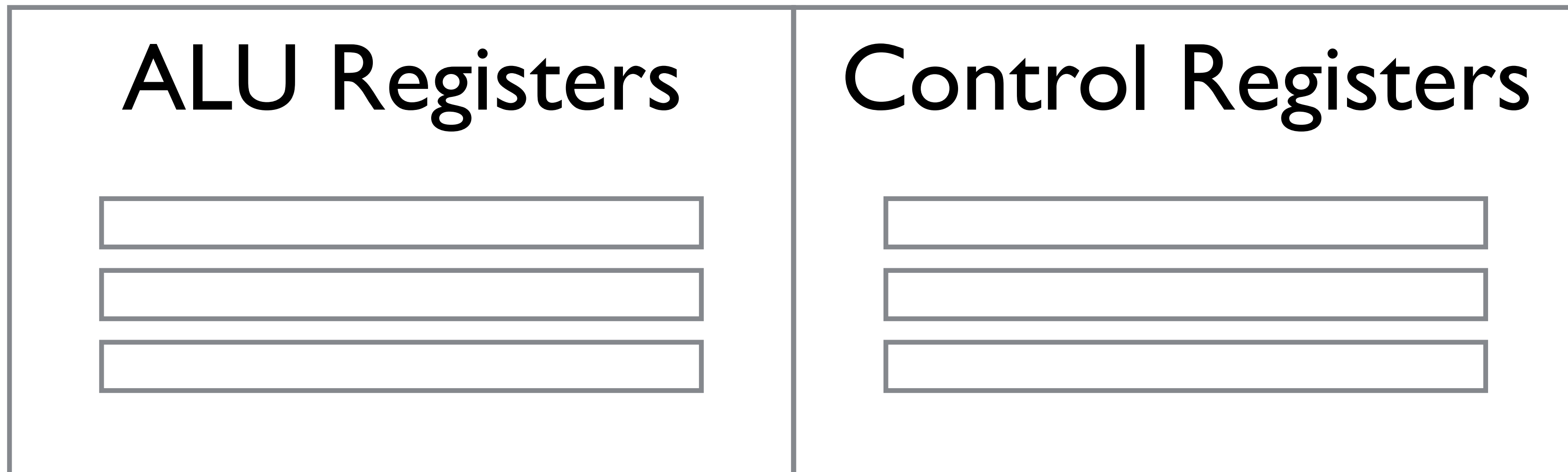
Main Memory

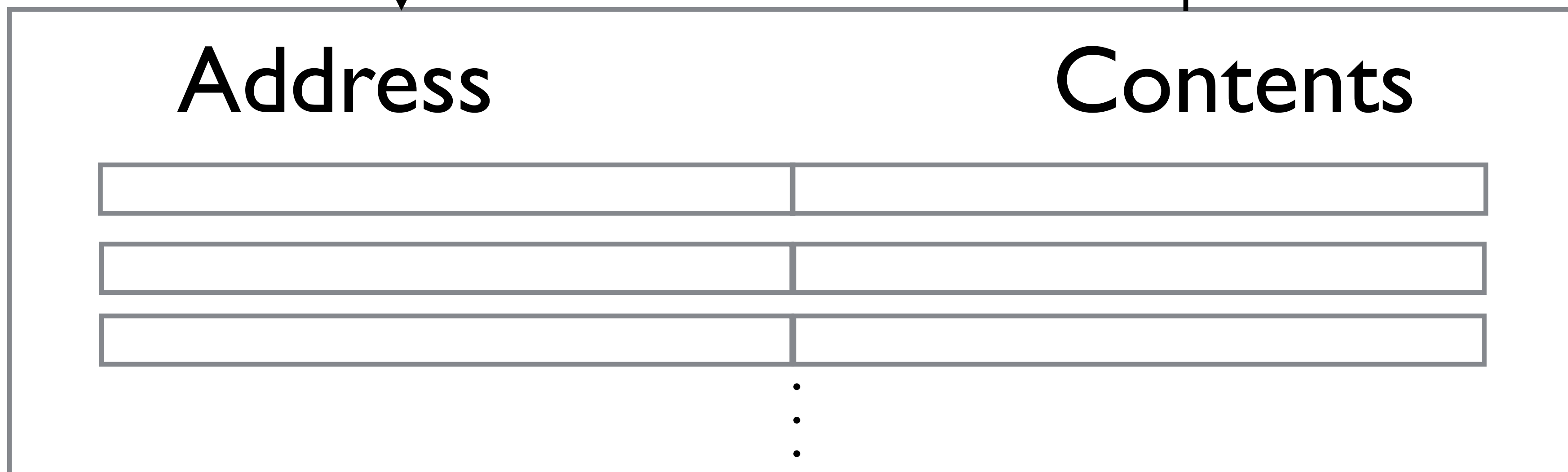
The CPU has a *control unit* and an *arithmetic and logic unit* (ALU).

The CPU stores data in registers, which it is able to access extremely fast. It also has a *program counter*, which stores the address of the next instruction.

Von Neumann Architecture

CPU





Main Memory

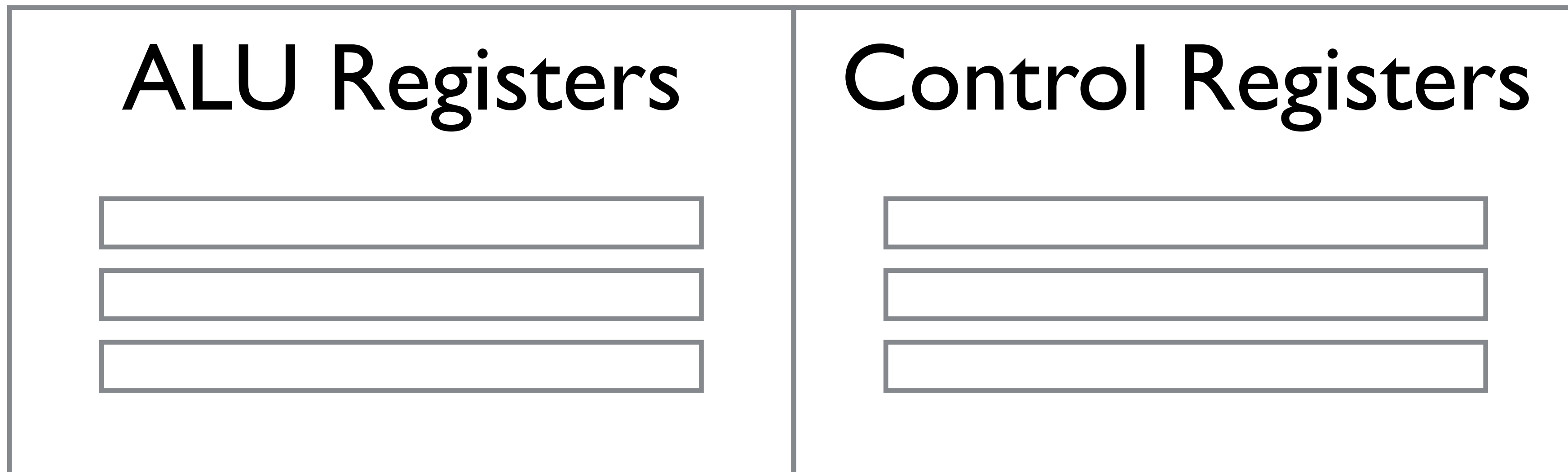
Instructions and data are transferred by the interconnect (traditionally a *bus*).

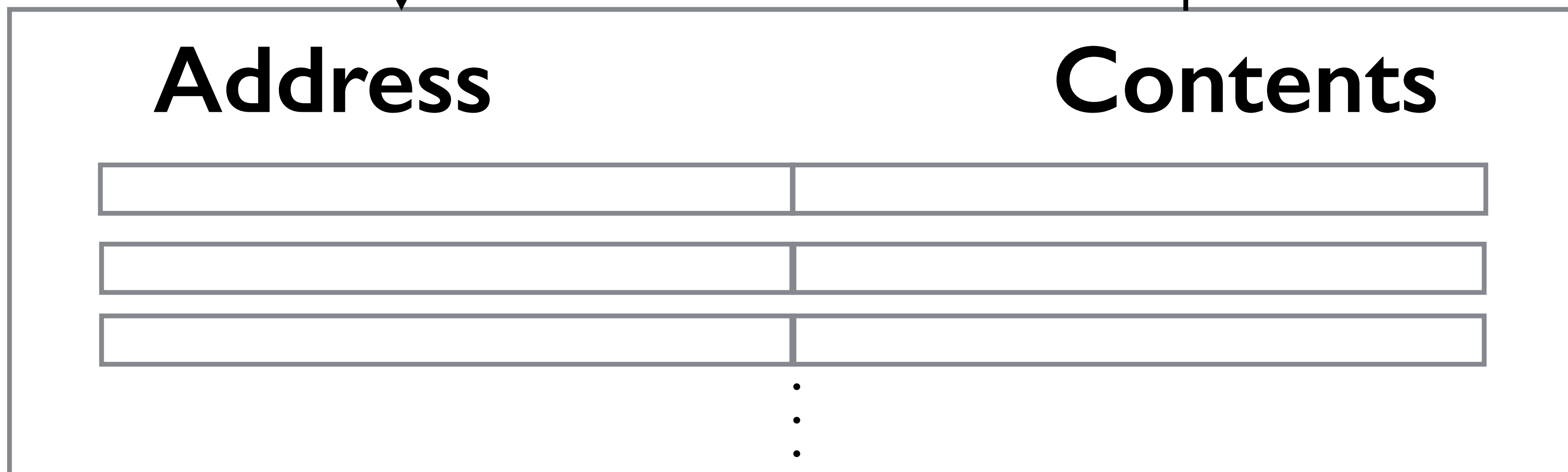
The Von Neumann architecture executes a single instruction at a time, however modern CPUs do not.

The interconnect typically transfers many instructions and data in parallel (for caching and other reasons).

Von Neumann Architecture

CPU



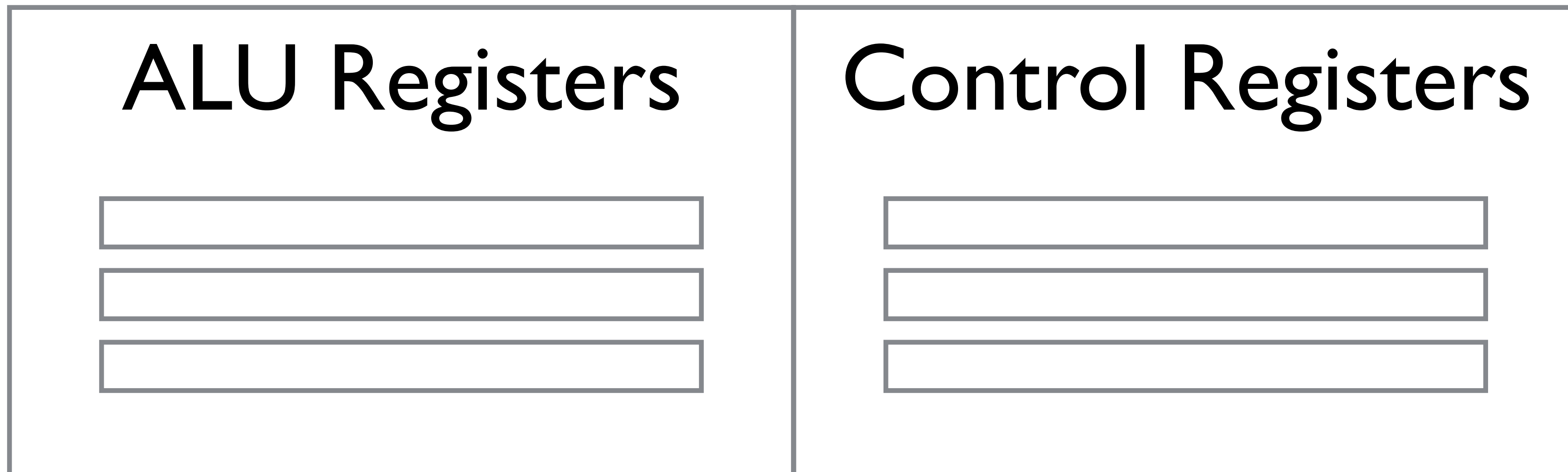


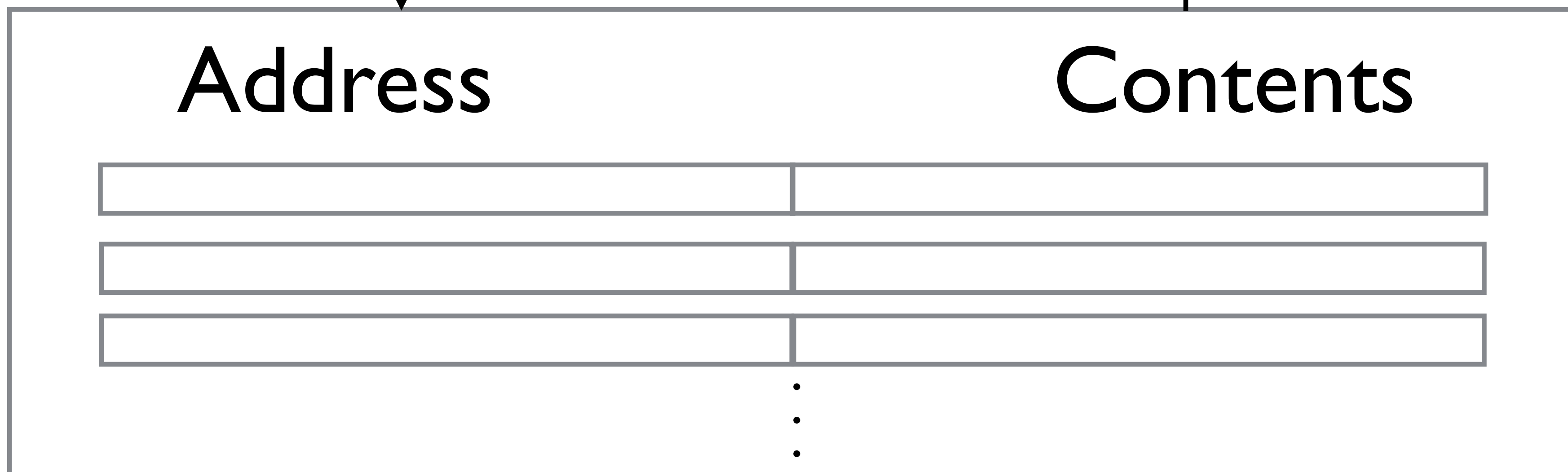
Main Memory

Again, the main memory consists of addresses (like the key in a hashtable), and their contents, which can be instructions and/or data.

Von Neumann Architecture

CPU





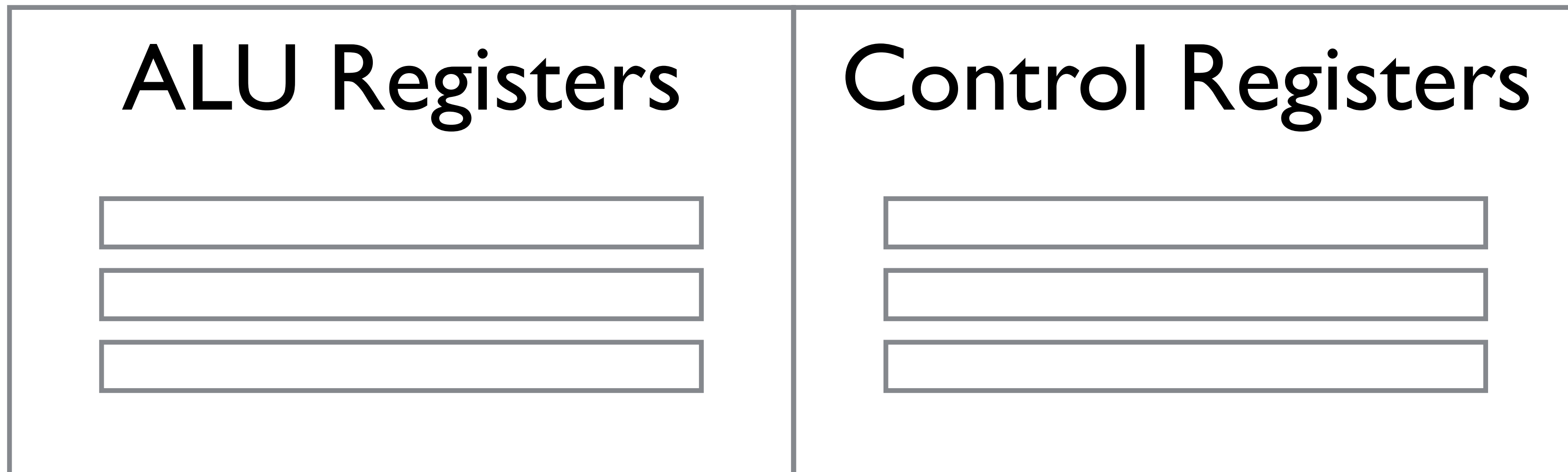
Main Memory

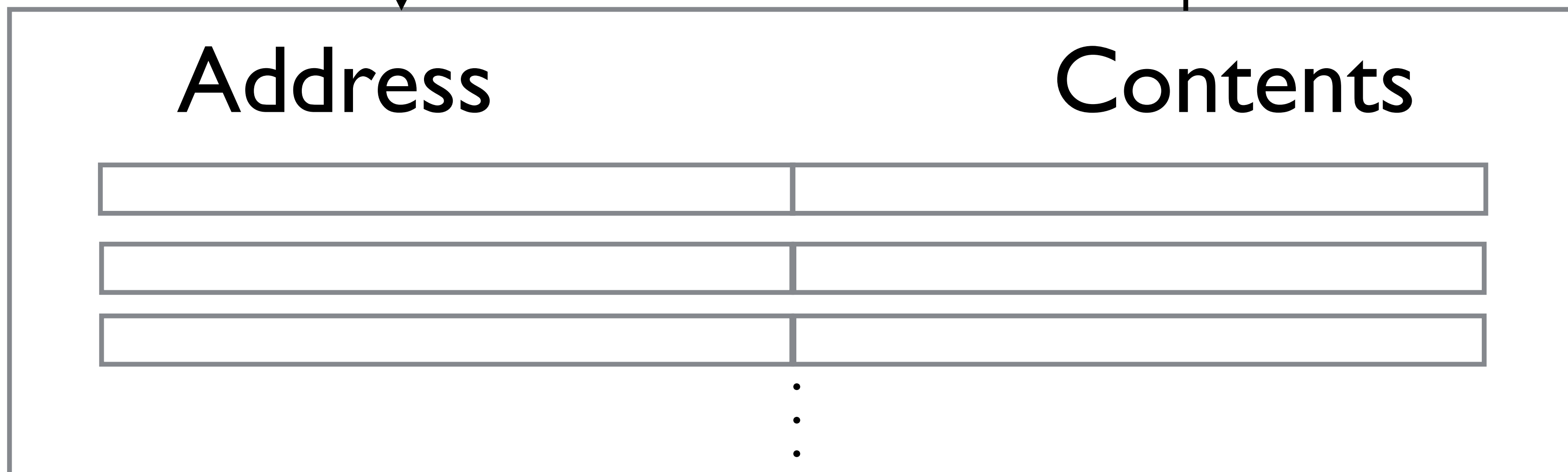
Data/instructions are *fetched* or *read*, when they transferred from memory to the CPU.

Data/instructions are *written to memory* or *stored* when they are transferred from the CPU to memory.

Von Neumann Architecture

CPU





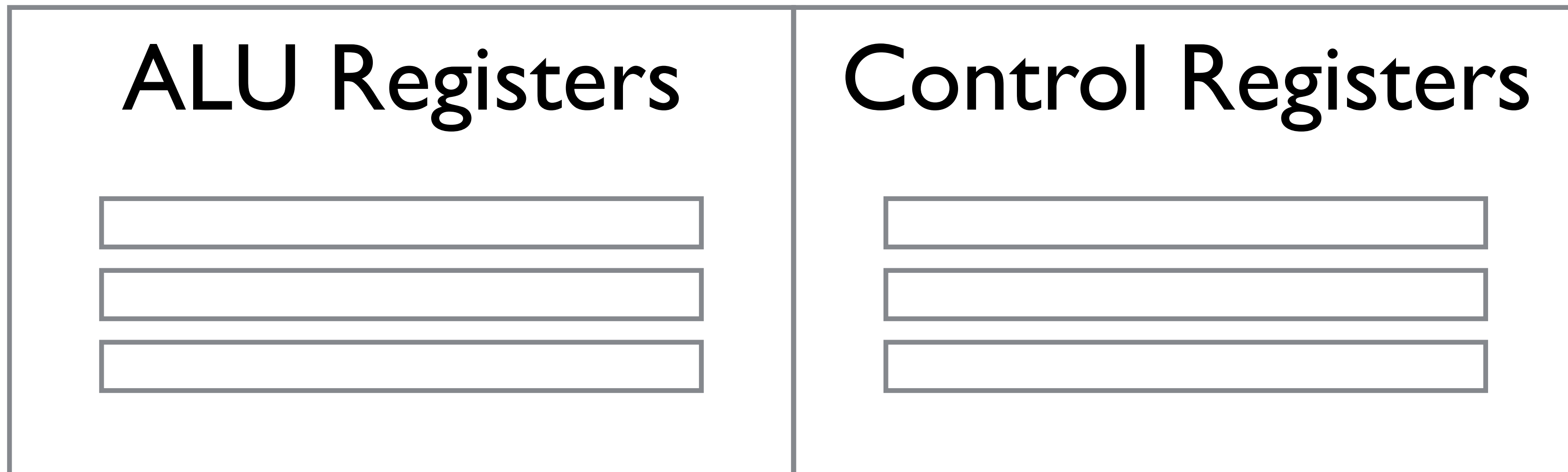
Main Memory

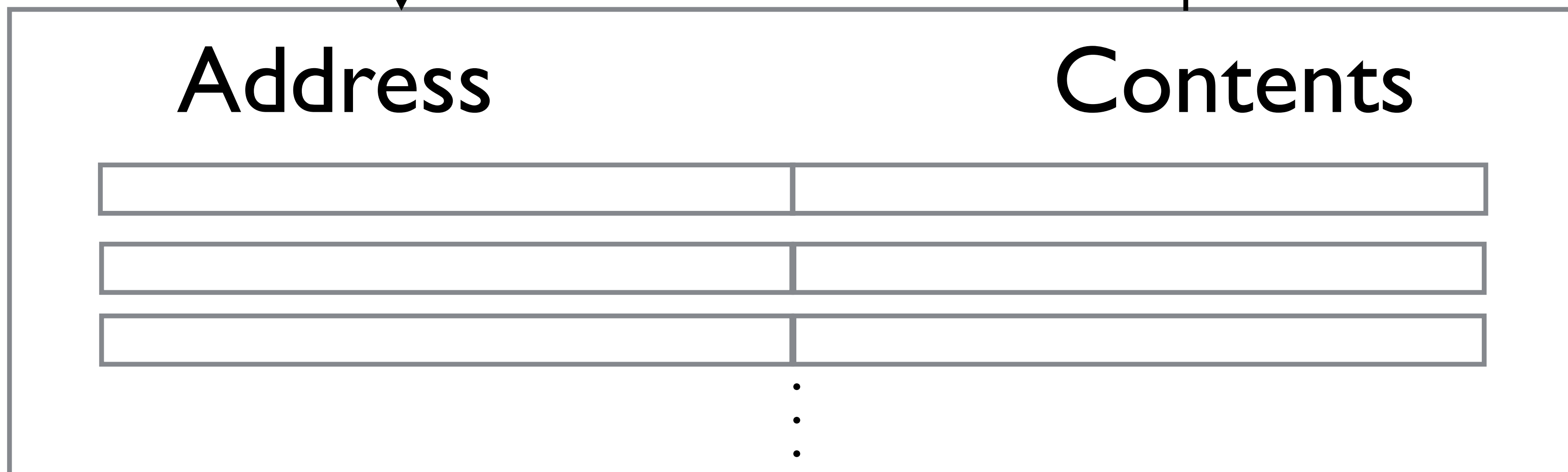
Note that a CPU must wait for all required data and instructions to be transferred before it can execute the instruction.

However, other instructions can be processed while memory is being written, so long as they don't require the updated value of that memory.

Von Neumann Architecture

CPU





Main Memory

This memory/CPU separation is referred to as the *Von Neumann Bottleneck*, as CPUs can process instructions orders of magnitude faster than items can be fetched from memory.

This issue is behind a lot of what you learn in Computer Architecture, *e.g.*, caching, branch prediction, *etc*.

Processes, Multitasking, Threads

Processes, Multitasking, Threads

Despite what Windows, OSX and all the flavors of Linux want you to think - an important (if not the most important) purpose of a computer's *Operating System* is to determine what programs run and when.

This includes controlling the allocation of memory to running programs and access to external hardware (including *network interface cards* or NICs).

Processes

Processes consist of many things:

1. The *executable* or *binary*, which is the set of instructions in machine language.
2. A block of memory, which includes the executable, a *call stack* which consists of active functions, a *heap*, which contains the data actively in use. There may be other memory locations involved as well.
3. Descriptors of resources the OS has allocated to the process (e.g., file descriptors).
4. Security information, such as what hardware and software the program can access.
5. Information about the processes state, e.g., if it is running, waiting to run, waiting on another resource, etc.

Multitasking

Modern operating systems run multiple processes simultaneously, even if there is only one CPU. This is called *multitasking*.

In a multitasking OS, processes are run for *time slices*, small intervals of time; after which the process will swap out with other waiting processes. The OS's job is to make this as seamless as possible, by minimizing process waiting times and maximizing throughput (processes completing per second).

Multitasking

In addition to being swapped out at the end of time slices, when a process is waiting on some resource (e.g., data being transferred over the network or interconnect, a file locked and in use by another process, etc.) is a good time to swap processes, as the blocked process can't do anything anyways.

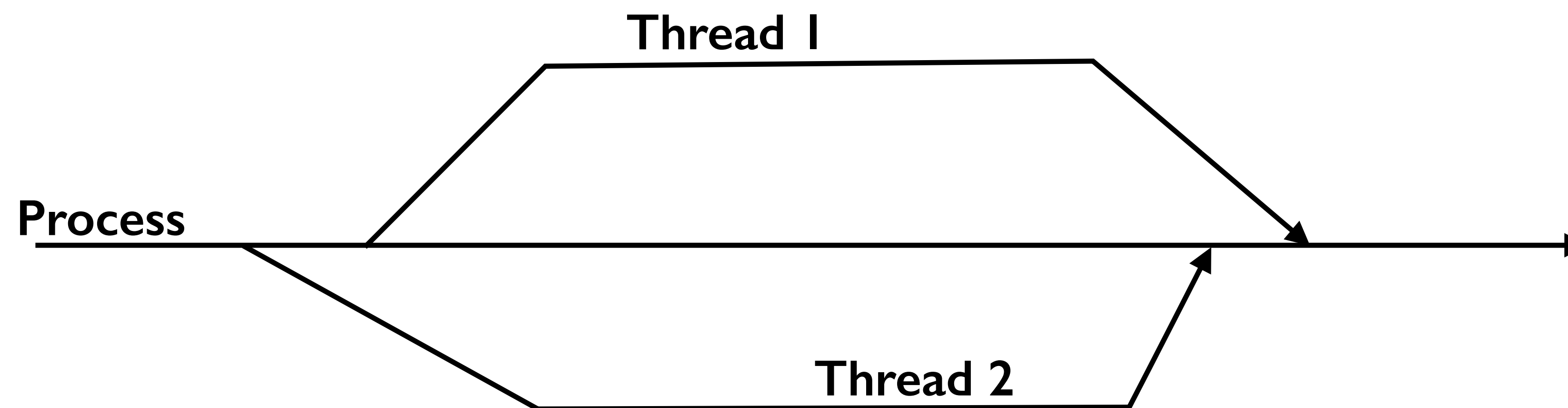
Threads

Threads allow programmers to divide their programs up into different (independent or partially independent) tasks, which can execute while others are blocked; similar to how operating systems handle processes.

Threads are more lightweight than processes, as they are contained within the same process and utilize the same memory and resources. This allows them to be swapped faster than processes. Threads still need their own program counter and call stack however.

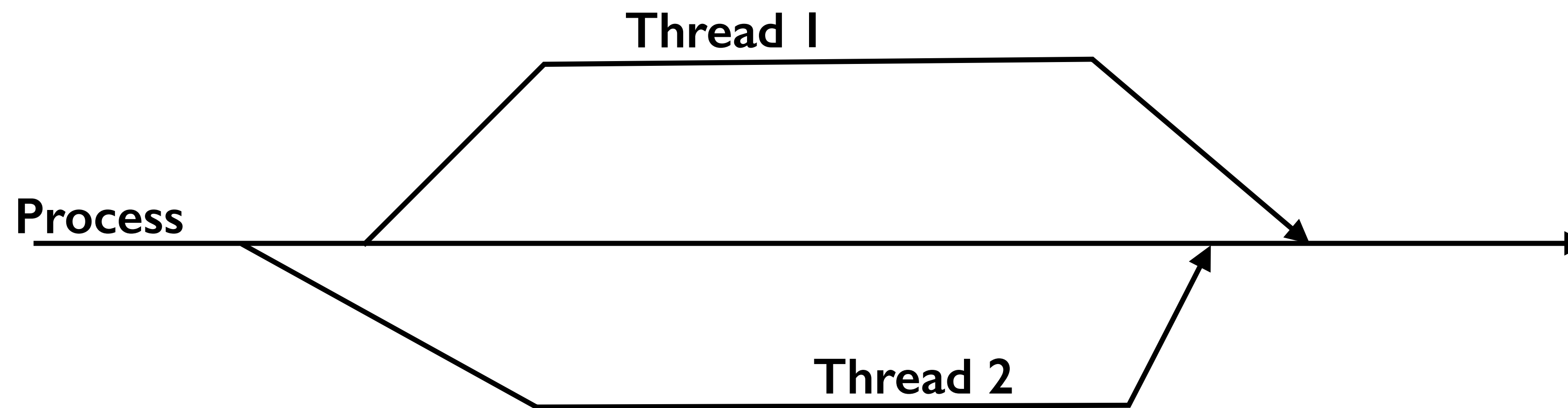
The time it takes to swap a thread or process is called *context-switching time*.

Threads



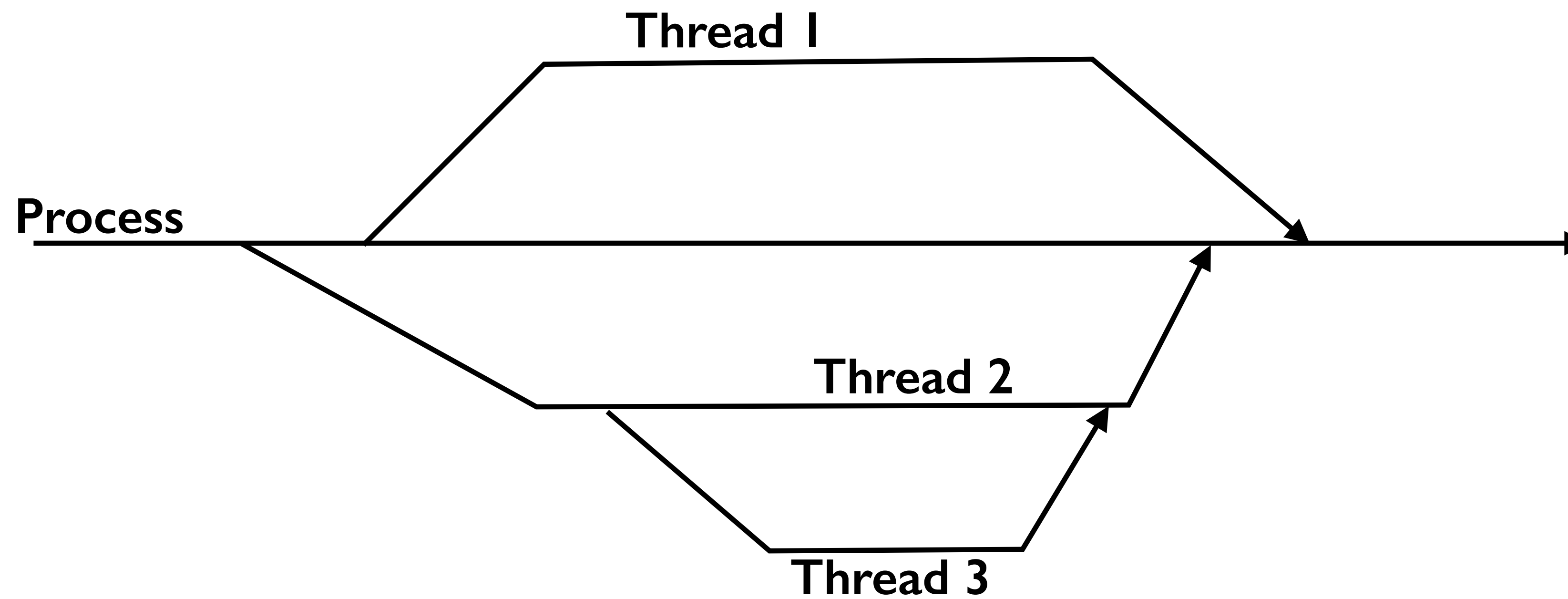
The initial process often acts as the *master thread*. It will *fork* off child threads, which later *join* back to the master process when they have completed their subtask.

Threads



As the process and its threads share the same memory, it is important to make sure that they don't try to modify the same memory at the same time, as it could lead to inconsistencies in the program execution.

Threads



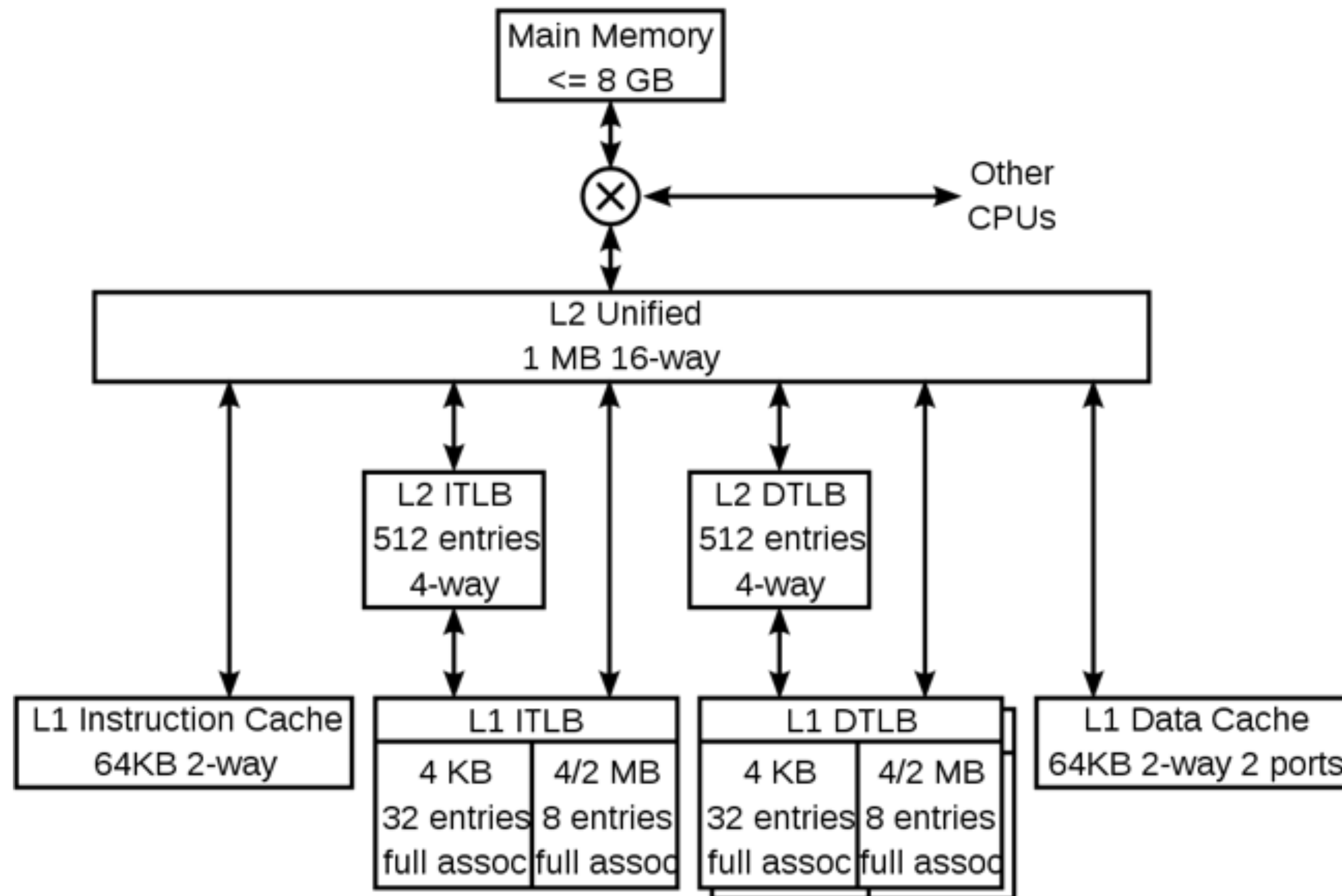
Threads can also fork off other threads, which can later join back to them (or to another thread).

Common Modifications to the Von Neumann Architecture

Caching

Caching is the main strategy used to combat the Von Neumann bottleneck. Most CPUs today a wide variety of caches, and even RAM, hard drives and other devices have their own caches.

Caching



For example, this is the cache hierarchy of the K8 core in the AMD Athlon 64 CPU.

http://en.wikipedia.org/wiki/CPU_cache

Caching

Caches are collections of memory that are stored closer to where they are used, so the memory can be accessed faster. However, speed comes at a price, so they typically arranged with the smallest fastest (and most expensive) caches being closest to the CPU, with larger slower caches (and less expensive) along the way to main memory (RAM or a hard drive).

Caching

As caches have limited sizes, typically smaller than the size of an executing program, various strategies are used to make sure that required data is in the cache when it is requested.

Generally, the strategies revolved around two ideas: *spatial locality* and *temporal locality*.

Spatial locality is the idea that when a piece of data is accessed, nearby data will probably also be needed.

Temporal locality is the idea that when a piece of data is accessed, it will probably be accessed again soon in the future.

Caching

For example, in the following loop:

```
float z[1000];
...
float sum = 0.0;
for (int i = 0; i < 1000; i++) {
   sum += z[i];
}
```

In this simple (and common) example, accessing elements of `z` demonstrates spatial locality (all the array entries will be nearby in memory); while the variables `sum` and `i` demonstrate temporal locality (they are re-used frequently).

Caching

Computing systems use *wide* interconnects to exploit locality. Memory accesses operate on blocks of datas and instructions called *cache blocks* or *cache lines* (these are usually 8-16 times larger than individual data or instructions).

Cache Mapping

When a cache line is fetched, where does it go? *Fully associative caches* allow any line to be placed anywhere in the cache; while *direct mapped caches* have a unique possible location for any cache line (determined by it's address, just like a hashtable).

n-way set associative caches provide approaches between these two extremes.

Cache Mapping

Another question is when the cache becomes full, what lines should be overwritten for new cache lines?

The most common scheme is *least recently used*, where the line in the cache least recently used is the line that is overwritten.

Cache Example

```
double A[MAX][MAX], x[MAX], y[MAX];
…
//initialize A, x; set y = 0
…
for (i = 0; i < MAX; i++) {
    for (j = 0; j < MAX; j++) {
        y[i] += A[i][j] * x[j];
    }
}
…
//set y = 0 again
…
for (j = 0; j < MAX; j++) {
    for (i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
}
```

Remember that C and C++ store multi-dimensional arrays in *row-major* order, that means they're stored in one large one dimensional array (`a[i][j]` is `a[(i * i_width) + j]`).

Suppose MAX = 4 and the array elements are stored as follows:

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Cache Example

```
double A[MAX][MAX], x[MAX], y[MAX];
…
//initialize Z, x; set y = 0
…
for (i = 0; i < MAX; i++) {
    for (j = 0; j < MAX; j++) {
        y[i] += A[i][j] * x[j];
    }
}
…
//set y = 0 again
…
for (j = 0; j < MAX; j++) {
    for (i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
}
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

In the first loop, the array A will be accessed A[0][0], A[0][1], A[0][2], A[0][3].

In our example cache, when A[0][0] is accessed, the entire line 0 will be placed in the cache. In this case, all accessing A[0][0] will be a cache miss (which is slow since it will have to go out to main memory), however the other accesses with all be hits (and very fast).

Cache Example

```
double A[MAX][MAX], x[MAX], y[MAX];
…
//initialize Z, x; set y = 0
…
for (i = 0; i < MAX; i++) {
    for (j = 0; j < MAX; j++) {
        y[i] += A[i][j] * x[j];
    }
}
…
//set y = 0 again
…
for (j = 0; j < MAX; j++) {
    for (i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
}
```

In the second loop, the array A will be accessed A[0][0], A[1][0], A[2][0], A[3][0].

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

In our example cache, when A[0][0] is accessed (a miss), the entire line 0 will be placed in the cache. A[1][0] will also be a miss, so it will have to be fetched from memory and placed in the cache, and so on.

After A[0][1], all elements will be in the cache and all hits.

Cache Example

```
double A[MAX][MAX], x[MAX], y[MAX];
…
//initialize Z, x; set y = 0
…
for (i = 0; i < MAX; i++) {
    for (j = 0; j < MAX; j++) {
        y[i] += A[i][j] * x[j];
    }
}
…
//set y = 0 again
…
for (j = 0; j < MAX; j++) {
    for (i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
}
```

However, what happens if our example cache can only hold 3 lines?

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Assuming least recently used, Loop 1 will go miss, hit, hit hit, miss, hit, hit hit, etc.

Loop 2 however, will be all misses. If a miss takes 10x as long (in modern CPUs it could be more), then loop 2, while almost identical to loop 1, will run almost 10x slower.

Virtual Memory

Virtual memory is used to handle programs that require more memory than can fit in main memory.

Even in current systems, the memory in use by all running processes can be larger than the systems main memory.

Virtual Memory

In virtual memory schemes, the active parts of memory are kept in main memory, while the others are kept in a block of secondary storage (i.e., on your hard drive) called *swap space*.

If you've installed Linux by yourself, many flavors will have you make a partition on your hard drive for this.

OSX Mavericks recently 'improved' on this by compressing the memory stored in swap (freeing up hard drive space, but potentially slowing down swapping between programs).

Virtual Memory

In many ways, virtual memory works like a cache. What's important to remember is that if your program is using large amounts of memory (or running with other programs which do), it can end up having misses when accessing main memory, causing it to have to read that memory from the disk.

Just like as different levels of cache can be orders of magnitude slower, reading from the hard drive is orders of magnitude slower than ram. If your program has a lot of *page faults* (accessing memory stored in the *swap space* on disk, instead of in main memory), it can slow down significantly.

Instruction-Level Parallelism

Instruction level parallelism (ILP) improves processor performance by having multiple processor components (or *functional units*) perform operations simultaneously.

The two main strategies are:

1. Pipelining - instructions are arranged in stages (think doing the laundry with a washer and dryer).
2. Multiple Issue - multiple instructions are done at the same time (doing the laundry with multiple washers and dryers).

Pipelining

```
double x[MAX], y[MAX], z[MAX];
…
//initialize x, y; set z = 0
…
for (i = 0; i < MAX; i++) {
    z[i] = x[i] * y[i];
    z[i] = z[i] + x[i];
    z[i] = z[i] / y[i];
}
```

Take a look at this example code. Let's say that an add takes 1 second, a multiply takes 7 seconds and a divide takes 10 seconds. Note that there are no dependencies between iterations of the loop (calculating the loop when i = 3 does not depend on i = 2).

If MAX = 100, then if this is done with no ILP, it will take (1 + 7 + 10) * 100 = 1800 seconds.

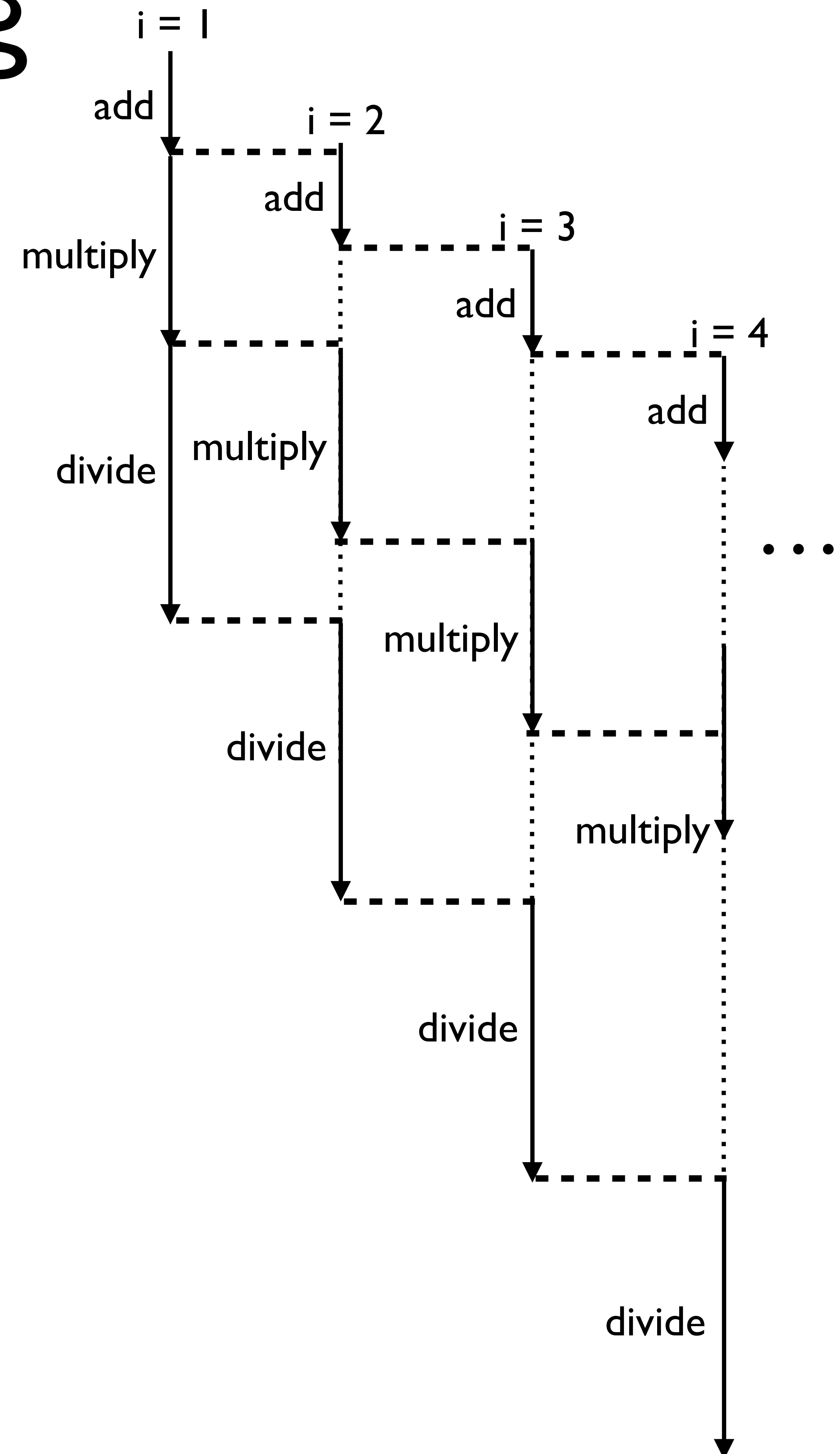
Pipelining

```
double x[MAX], y[MAX], z[MAX];
…
//initialize x, y; set z = 0
…
for (i = 0; i < MAX; i++) {
    z[i] = z[i] + x[i];
    z[i] = x[i] * y[i];
    z[i] = z[i] / y[i];
}
```

With pipelining, lets say our CPU has one adder, one multiplier and one divider unit.

The CPU can use each of these units simultaneously. With a pipeline we can do the following on the right.

Note that the performance is bounded by the long divide operation. However, we can still complete all the operations in 1 + 7 + (10 * 100) = 1008 seconds, a significant speedup.



Multiple Issue

```
double x[MAX], y[MAX], z[MAX];
…
//initialize x, y; set z = 0
…
for (i = 0; i < MAX; i++) {
    z[i] = z[i] + x[i];
    z[i] = x[i] * y[i];
    z[i] = z[i] / y[i];
}
```

With multiple issue, lets say our CPU has 10 adder, 10 multiplier and 10 divider units.

The CPU can then process 10 iterations of the loop in parallel. This means instead of taking 1800 seconds, the loop can be calculated in 180 (doing 10 at a time, simultaneously).

It is also possible to use pipelining for each of these 10 iterations, dropping the runtime down to 108 seconds - 8 for the first add and multiply, then 10 divides).

Using Pipelining/Multiple Issue

If MAX is known at compile time, the multiple issue instructions can be declared at compile time, and this is called *static multiple issue*.

If MAX is unknown at compile time (i.e., it is calculated by another part of the code based on a file, or is given as user input), in may cases it can be determined at runtime and smart hardware can do the loop using multiple issue on the fly. This is *dynamic multiple issue*. Processors which support dynamic multiple issue are often called *superscalar*.

While pipelining is done automatically the majority of the time, multiple issue cannot necessarily be done automatically. By providing information to the compiler (making loop bounds constant, etc), they can more easily perform multiple issue instructions and dramatically speed up your code. Often, you need to turn on compiler flags as well (-02 or -03 with gcc or clang will do this).

Hardware Multithreading

ILP can be hard to exploit, as many programs have long sequences of dependent instructions (meaning they cannot be parallelized easily like our example loops).

Thread-level parallelism, tries to parallelize programs at a coarser granularity.

Simultaneous multithreading, refers to combining thread level parallelism with ILP; each thread can utilizing multiple functional units via multi issue.

Hardware multithreading, allows systems to continue working when a task has stalled (waiting for disk, network, memory, etc). When this happens context will switch to another thread that is not blocked (stalled).

Hardware multithreading can be *fine-grained*, switching between threads after each instruction, skipping those that are stalled; or *coarse-grained*, only switching between threads when a thread has stalled on a time consuming operation (loading from main memory, etc).

Caching, Virtual Memory, ILP

The point is if you want to write fast software, your data and instructions need to be as close to what is executing them as possible.

Most hardware and operating systems do a good job of automating this for you, however they cannot overcome poorly designed and written code.

You need to make sure you program things so the hardware and operating system can take good advantage of caching and other performance enhancements.

All the parallelism in the world cannot make up for poorly written code.

Parallel Hardware

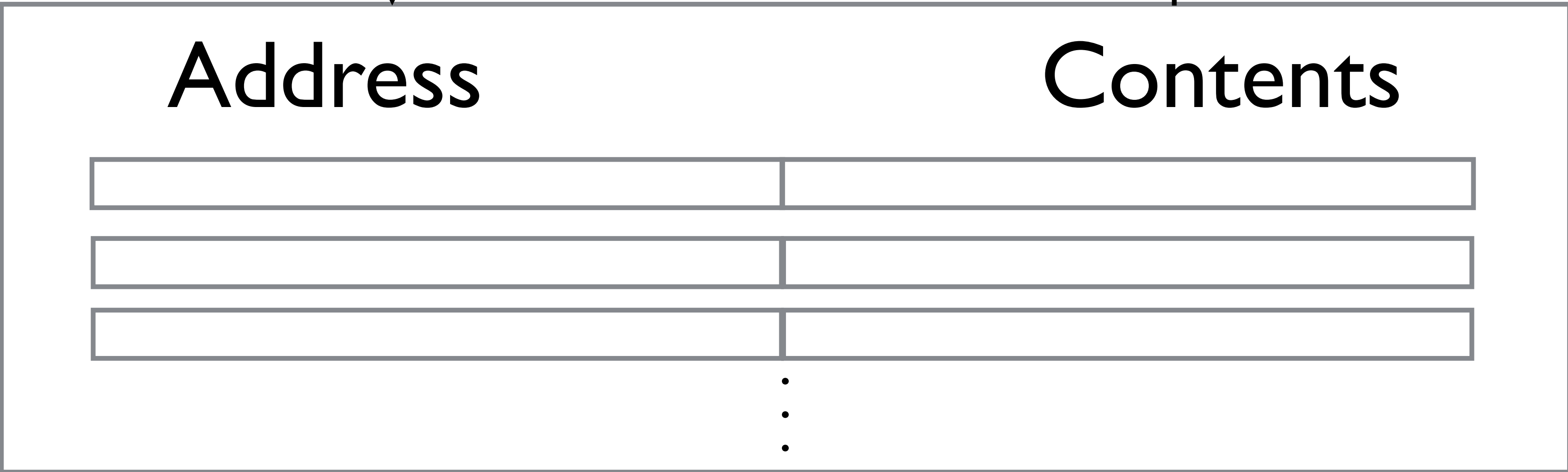
SIMD, MIMD, Interconnection Networks

There are many types of hardware used in parallel (and distributed) computing. These include:

1. Single instruction, multiple data (SIMD) systems, such as graphical processing units and vector processing units.
2. Multiple instruction, multiple data (MIMD) systems, such as multi-core CPUs.
3. Interconnection networks, both shared memory (as used inside multi-core processors), and distributed memory (as used between processors in clusters, grids and supercomputers).

SIMD Systems

Abstract SIMD systems can be thought of as having a single control unit but multiple ALUs (as in our Von Neumann architecture).



SIMD Systems

SIMD systems work by applying a single instruction over different data. The instruction is broadcast to all ALUs, which either execute the instruction on the data they have or stay idle.

```
for (int i = 0; i < n; i++) {
   x[i] += y[i];  //this does a vector add
}
```

In the above example code, a vector add is done — adding the value of the elements of one array (y) to the elements in another array (x).

SIMD Systems

```
for (int i = 0; i < n; i++) {
    x[i] += y[i];  //this does a vector add
}
```

If our system has *n* ALUs, then it is possible to load x[i] and y[i] into the ith ALU, have it perform the addition and store the result into x[i]; all in parallel.

SIMD Systems

```
for (int i = 0; i < n; i++) {
    x[i] += y[i];  //this does a vector add
}
```

If the system has *m* ALUs, where *m* < *n*, they can be loaded in blocks of m elements at a time. However, if *n* is not evenly divisible by *m*, some ALUs will remain idle in the last block. For example, if n = 13 and m = 4:

time step 1: processes i = 0…3
time step 2: processes i = 4…7
time step 3: processes i = 8…11
time step 4: processes i = 12

In the last time step, 75% of our ALUs are idle, which is not a good use of resources.

SIMD Systems

```
for (int i = 0; i < n; i++) {
    if (y[i] > 0) x[i] += y[i];  //this does a vector add
}
```

While the previous case isn’t too horrible, consider the modified code above, and half of y[i] being > 0, the other half being < 0.

In this case, the ALUs must first determine if the add should be done, and then if the add shouldn’t be done, remain idle while the other ALUs perform the add. Utilization of the system is down 25-50% (assuming how much time the conditionals take compared to the adds).

SIMD Systems

```
for (int i = 0; i < n; i++) {
    if (y[i] > 0) x[i] += y[i];  //this does a vector add
}
```

Note in a classical SIMD system, ALUs need to wait for the next instruction to be broadcast before proceeding; and they do not have instruction storage so they cannot delay executing an instruction by storing it for later (i.e., like keeping a buffer).

SIMD Systems

Despite these limitations SIMD systems are still very effective for parallelizing loops and doing the same operations over large sets of data. There is a large subset of applications (*data parallel applications*) that fall under this umbrella, such as many types of simulation and data analysis.

SIMD Systems

There have been rises and falls of SIMD systems over history. In the beginning of the 1990s, Thinking Machines was the largest manufacturer of supercomputers, and these were all SIMD systems.

At the end of the 1990s, SIMD systems were mostly limited to vector processing units on CPUs.

Recently, graphical processing units (GPUs) have become of much interest in use; however these are starting to be merged back onto the CPU; similar to what happened with vector processing units.

SIMD Systems: Vector Processors

The key characteristic of vector processors is that they operate on arrays or *vectors* of data; while traditional CPUs operate on individual pieces of data, or *scalars*.

SIMD Systems: Vector Processors

Typical vector processing units consist of:

1. *Vector registers*: registers capable of storing a vector of operands (instead of single operands) and operating simultaneously on their contents. The vector length is hardwired into the system and typically ranges from 4-128 64-bit elements.
2. *Vectorized and pipelined functional units*: the operations done on the vector(s) of operands can also be pipelined for further improved performance.
3. *Vector instructions*: as in the previous code example, the instructions can operate on vectors of data simultaneous. In the previous for loop, the *load*, *add* and *store* instructions only need to happen once for the entire vector, instead of over each individual element in the vector.

SIMD Systems: Vector Processors

4. *Interleaved Memory*: when accessing a memory system, typically there are multiple *banks* of memory. After accessing one bank, there is typically a delay before it can be accessed again, but other banks can be accessed much sooner. If the vector elements are distributed across multiple banks, the delay in loading/storing successive elements can be minimized. (Think parallel/pipelined memory access).
5. *Strided memory access and hardware scatter/gather*: this allows the vector processor to access elements across fixed intervals. (e.g, strides of 4 would be elements 0, 4, 8, 12, 16; 1, 5, 9, 13, etc). This kind of access pattern is common in nested for loops, and vector processing units provide acceleration for scattering (storing) and gathering (loading) data.

SIMD Systems: Vector Processors

Vector systems have high memory bandwidth and the data items loaded are almost always actually used (unlike cache systems); however they are limited to performing the same instructions and parallel.

They tend to be limited in their *scalability*, their ability to handle larger problems.

They are very efficient at what they do, but unfortunately that is only to the limited problem set where data structures cannot be irregular.

SIMD Systems: Graphical Processing Units

Most *graphics application programming interfaces* (graphics APIs) use points lines and triangles to represent objects.

A *graphics processing pipeline* is used to convert the representation in points, lines and triangles to the two dimensional pixels that are sent to the computer screen. Some stages in this pipeline are programmable, and specified using *shader functions*.

SIMD Systems: Graphical Processing Units

These shader functions are often very short, and are very parallel, as they are applied to multiple elements in the graphics stream.

GPUs take advantage of this by performing these shader functions in parallel, SIMD style. Recently, with CUDA and OpenCL, these previously highly specialized architecture have become more open and programmable. As such, scientists and programmers from a wide range of fields have begun using them as large scale customizable SIMD systems.

SIMD Systems: Graphical Processing Units

It should be noted that modern GPUs are not strictly SIMD systems. They're actually systems with many smaller cores, which process large batches of small (and similar) threads very quickly; each of these cores is SIMD, however the different cores can be processing separate SIMD operations.

We will go into this in a lot more detail in later lectures on CUDA and GPU programming.

MIMD Systems

The most common *Multiple-Instruction, Multiple Data* (MIMD) systems are *multicore* processors. Which nowadays are almost ubiquitous (even down to your phone's processors).

While SIMD systems are usually *synchronous*, processing their vector operations or batches of threads sequentially, MIMD systems are usually *asynchronous*; where processors or cores can operate independently at their own pace.

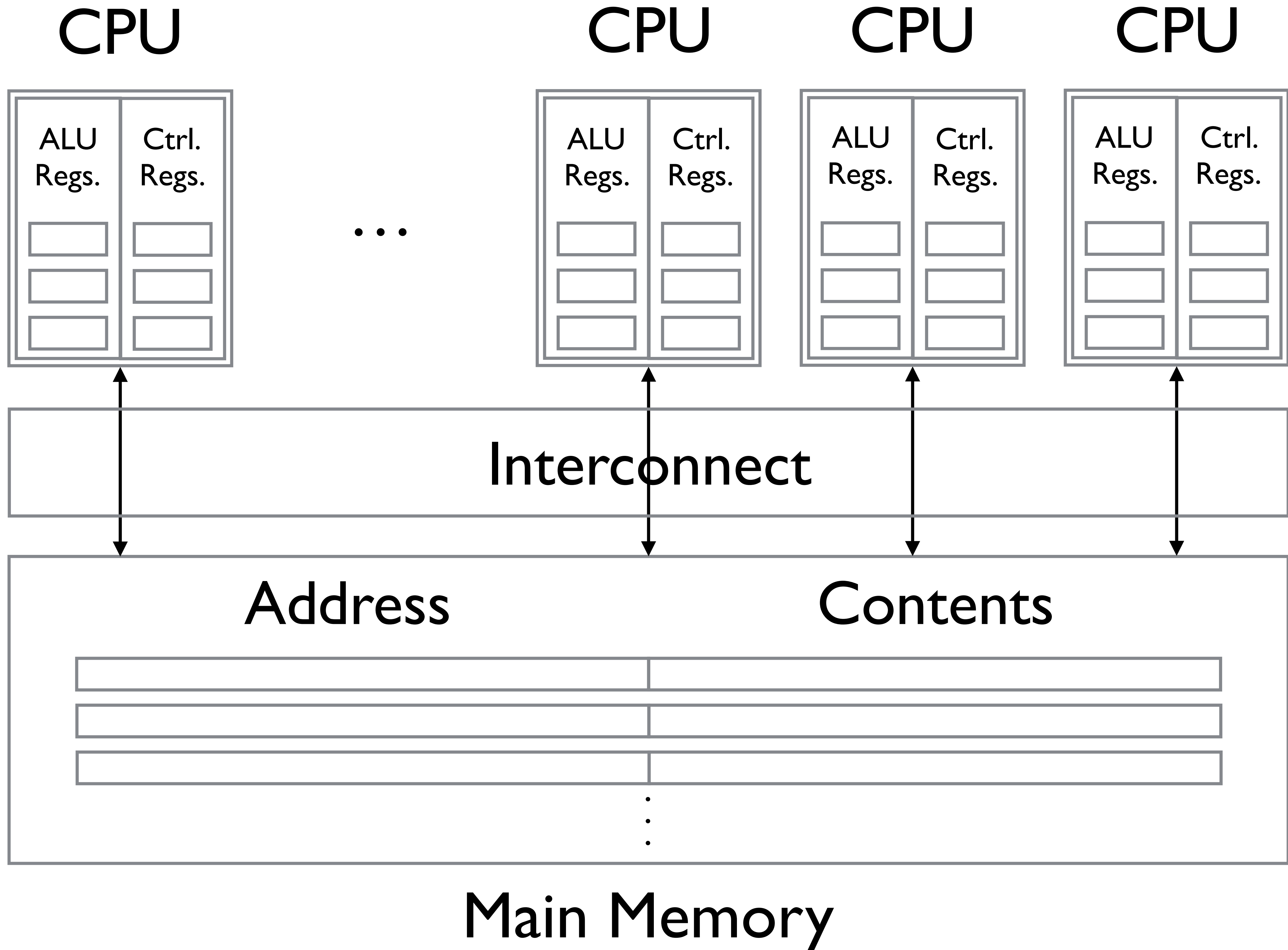
MIMD Systems

The major difference between synchronous and asynchronous systems is that synchronous systems have to wait for other parallel elements to complete operations; while asynchronous systems do not. This can make synchronous systems easier to program, however they also have inherent inefficiency.

MIMD Systems

MIMD systems are typically either *shared memory* where all processing elements connect to a single memory system; or *distributed memory*, where each processing element has its own private memory system and special functions to access the memory of other processors.

MIMD Systems: Shared Memory



MIMD Systems: Shared Memory

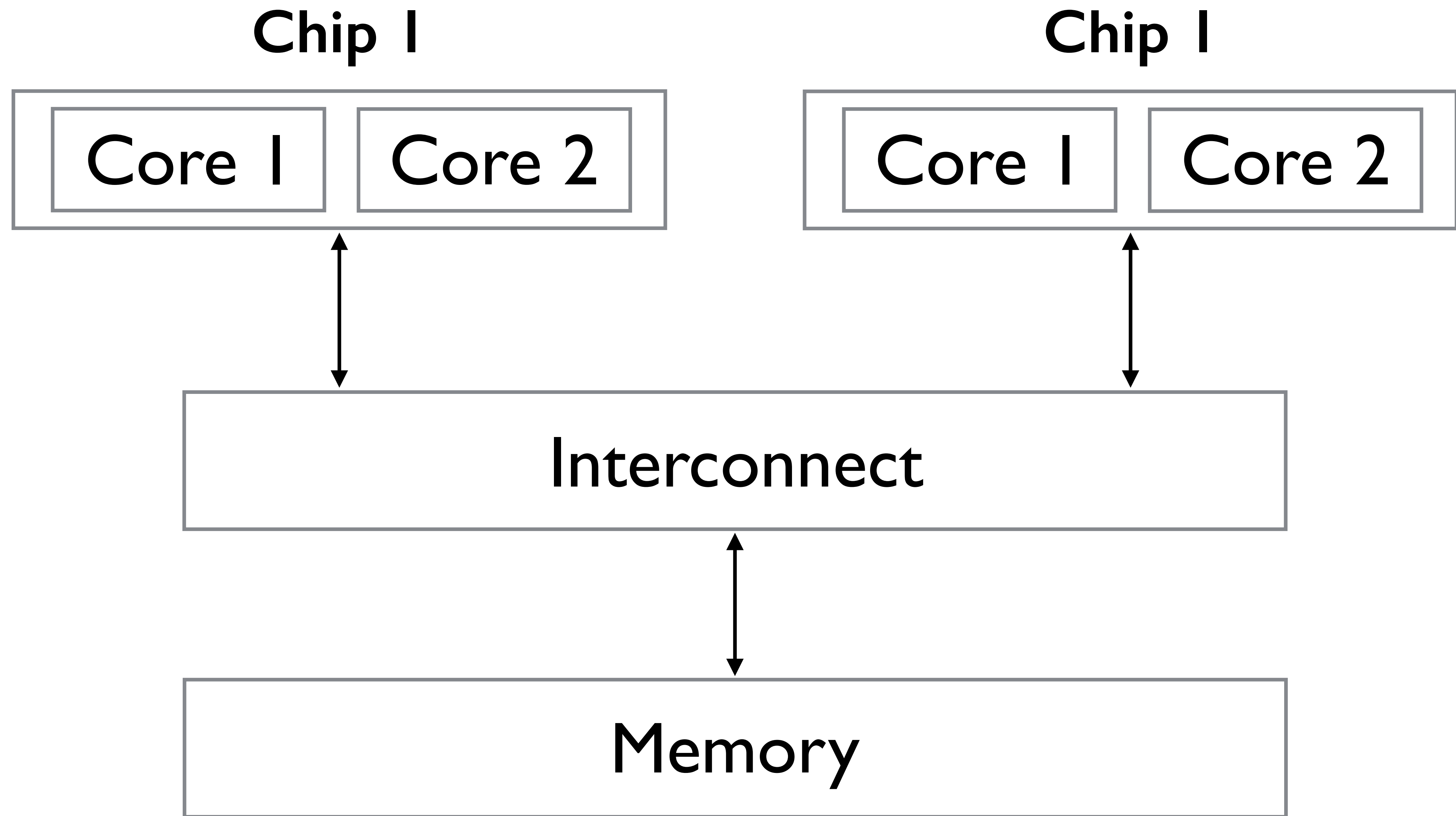
The most common shared memory MIMD systems are *multicore processors*, which can be found in almost ever CPU on the market. Each core typically has its own private L1 cache, while other caches may or may not be shared by the other cores (as we saw in the AMD cache diagram).

MIMD Systems: Shared Memory

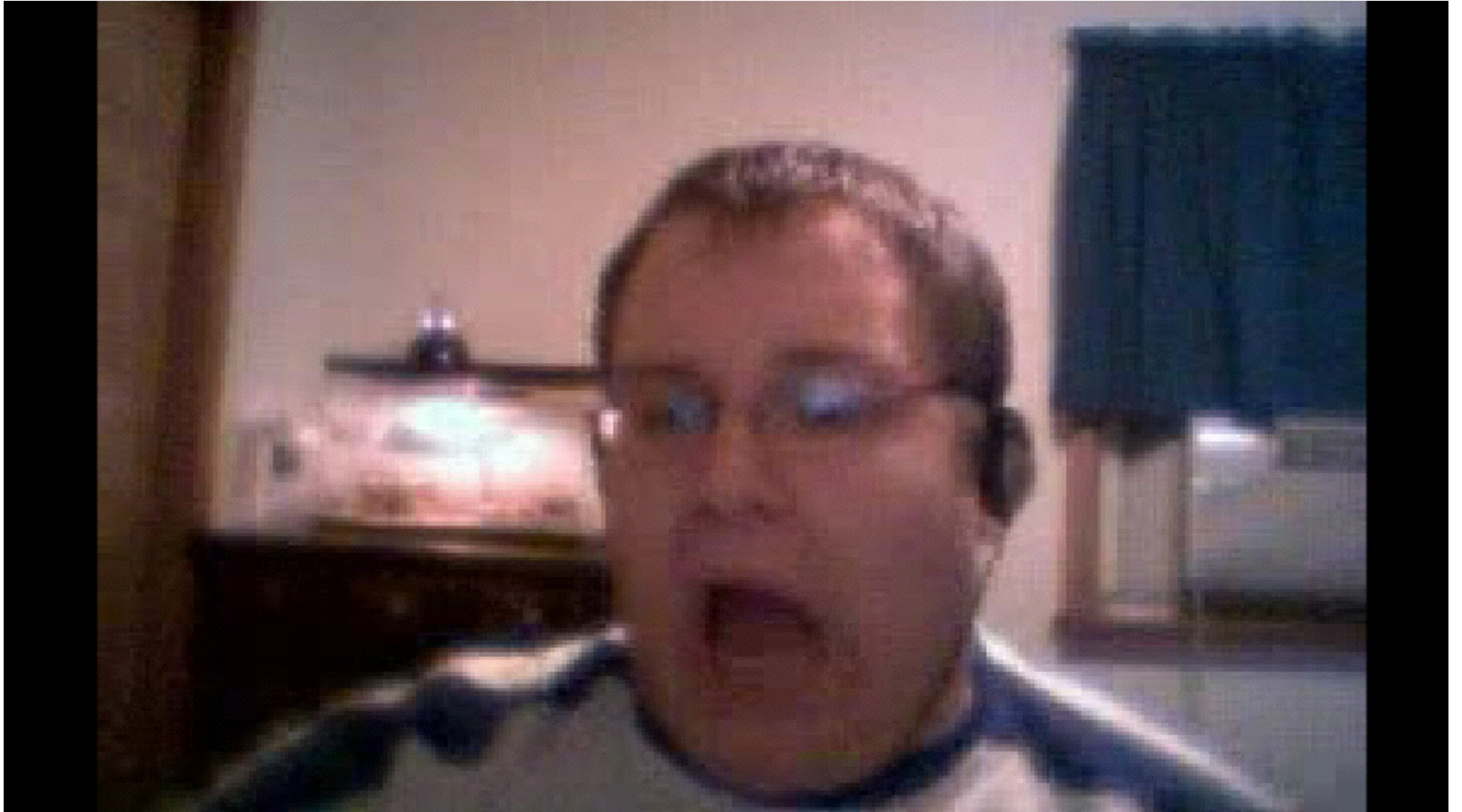
In shared memory systems, each processor either has access to the entire main memory through the interconnect; or each process has a direct connection to it's own block of memory, and uses special hardware to access the memory of other processors.

The first case is *uniform memory access* (UMA), while the second case is *nonuniform memory access* (NUMA).

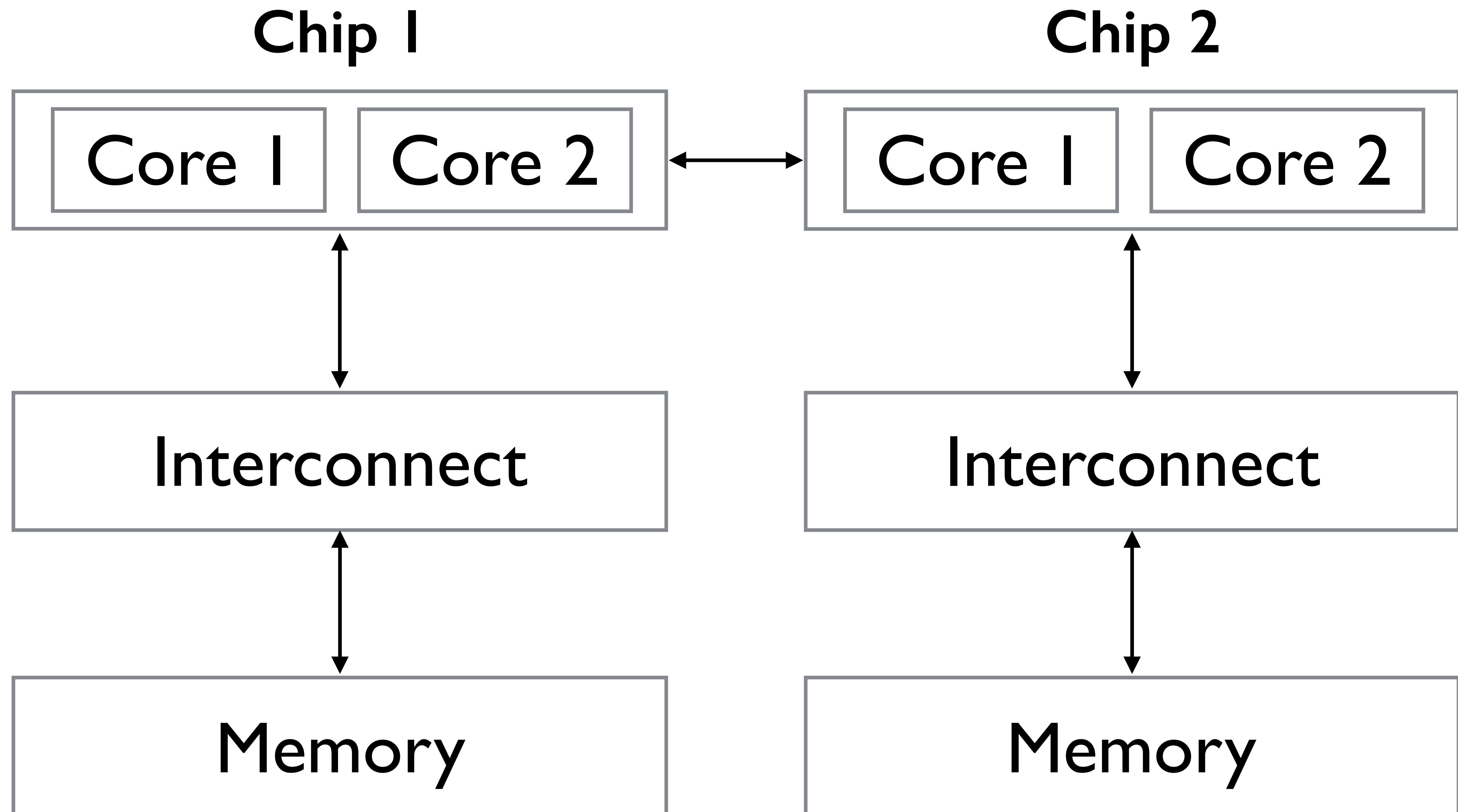
MIMD Systems: UMA



MIMD Systems: NUMA



MIMD Systems: NUMA

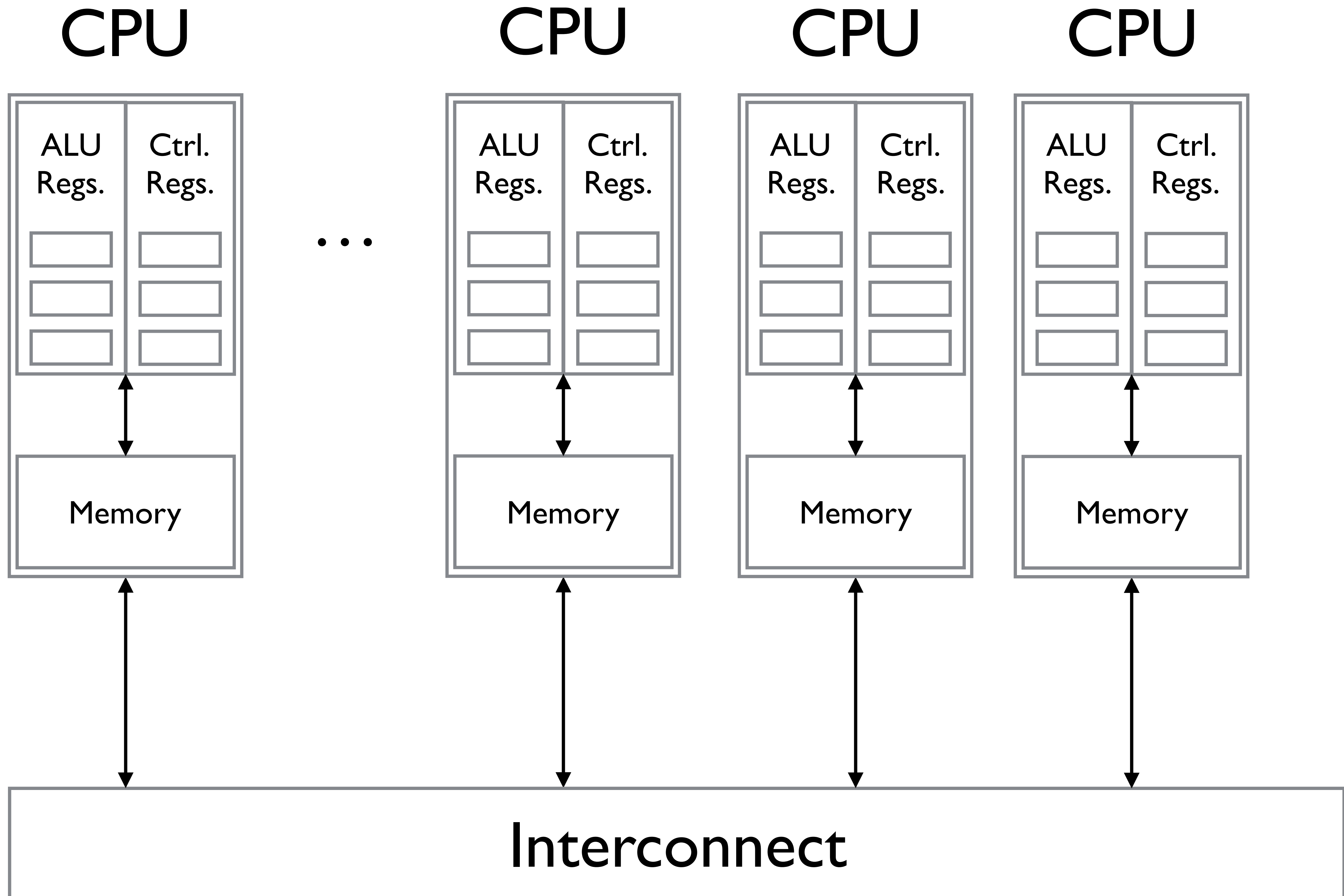


MIMD Systems: UMA vs NUMA

UMA systems tend to be easier to program, as the programmer doesn't have to worry about accessing memory used by the other chips; however this is offset by the fact that in NUMA systems local memory can be accessed quicker and NUMA systems can scale much better, allowing more total memory to be used.

In general this is a common tradeoff (ease of use vs. performance and scalability).

MIMD Systems: Distributed Memory



MIMD Systems: Distributed Memory

The most common distributed memory systems are called *clusters*. We have a couple here at UND. The computer science department has the CS cluster (which has about 20-30 dual core nodes), and there is also an older Shale cluster (which has 40 dual core nodes); and the new Hodor cluster, which has (32 eight-core nodes).

MIMD Systems: Distributed Memory

Clusters are made up of commodity CPU cores and a commodity interconnection network (such as Ethernet, Myrinet or Infiniband). Using commodity parts allows them to be easily expandable. In fact, many supercomputers are just large scale clusters using the same hardware.

Each node in a cluster typically contains a motherboard with multiple processors, each with multiple cores. The cores on a processor are shared memory; while everything else is distributed memory.

These types of systems used to be called *hybrid systems*, to distinguish them from pure distributed memory systems. However since they're essentially the only thing being used nowadays this term isn't frequently used.

MIMD Systems: Distributed Memory

Clusters are also connected into *grids*, where multiple institutions share their clusters together. In these systems, applications can potentially span multiple clusters, however the latency of communication between them can be large, they can be of different architectures and operating systems (i.e., the nodes are *heterogeneous* as opposed to *homogeneous*, where all nodes are the same), and there may be firewalls and other administrative issues to deal with.

MIMD Systems: Distributed Memory

The last common type of distributed memory systems are peer-to-peer (P2P). Bitcoin, gnutella, bit-torrent, etc. are all examples of peer-to-peer systems.

These systems tend to be highly heterogenous (more than grids); with the additional complication that the nodes can be extremely *volatile* — they can be prone to failures and disappearing randomly.

Interconnection Networks

The type of interconnect used is extremely important in the performance of a distributed system, and in many cases programming for the *topology* of the interconnect (how the nodes are connected) is extremely important as well.

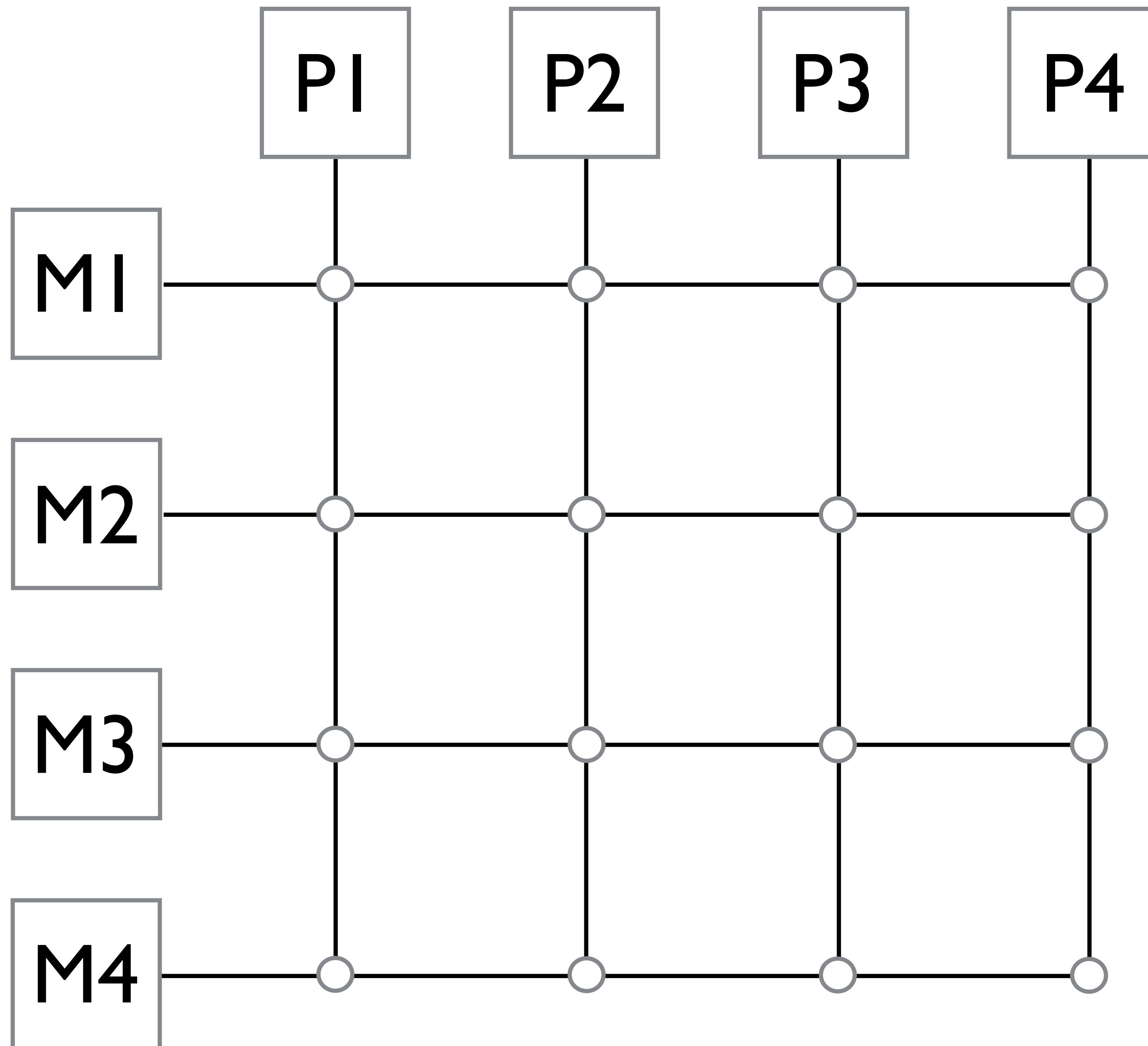
Interconnection Networks: Shared Memory

The most common shared memory interconnects are *buses* and *crossbars*.

The major difference between a bus and a crossbar, is that the nodes in a bus all share the same wires, which makes them cheaper to build, however as the number of devices connected increases it can lead to congestion.

Switched interconnects (like crossbars) use switches to control the routing of data among the connected devices to prevent congestion. These are commonly used in larger scale systems as they scale better.

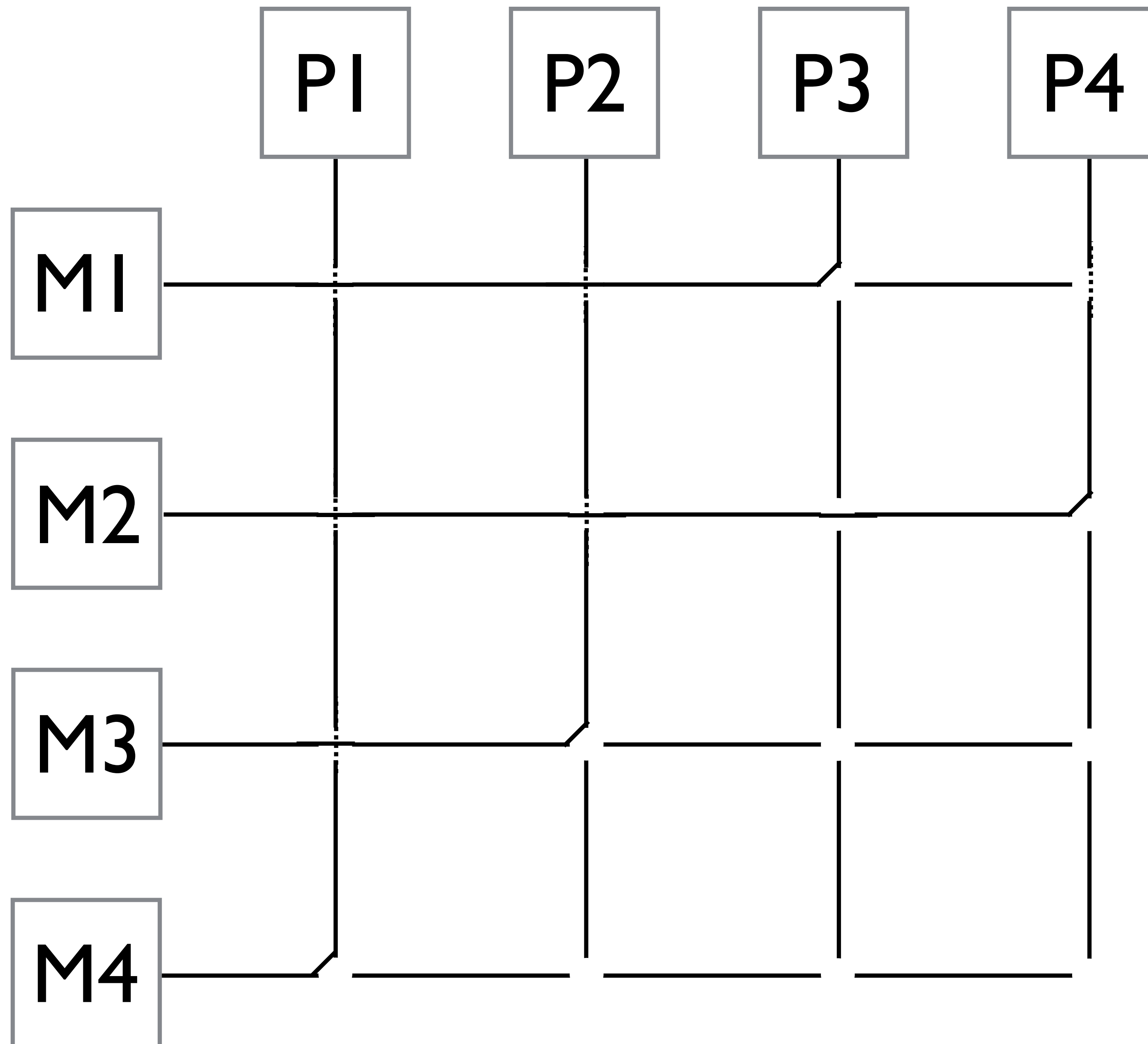
Interconnection Networks: Crossbars



Every processor (P1 … P4) has wires to every memory (M1 … M4).

The circles are switches, which control how the processors are connected.

Interconnection Networks: Crossbars



This diagram illustrates a possible setting for the switch, with the following connections:

P1-M4
P2-M3
P3-M1
P4-M2

Note it is possible for this switch to make any mapping between the four processors and four memory blocks.

Interconnection Networks: Distributed Memory

Just like shared memory interconnects, distributed memory interconnects often come in two flavors.

In a *direct interconnect*, each switch is directly connected to a processor memory pair and these switches are connected to each other.

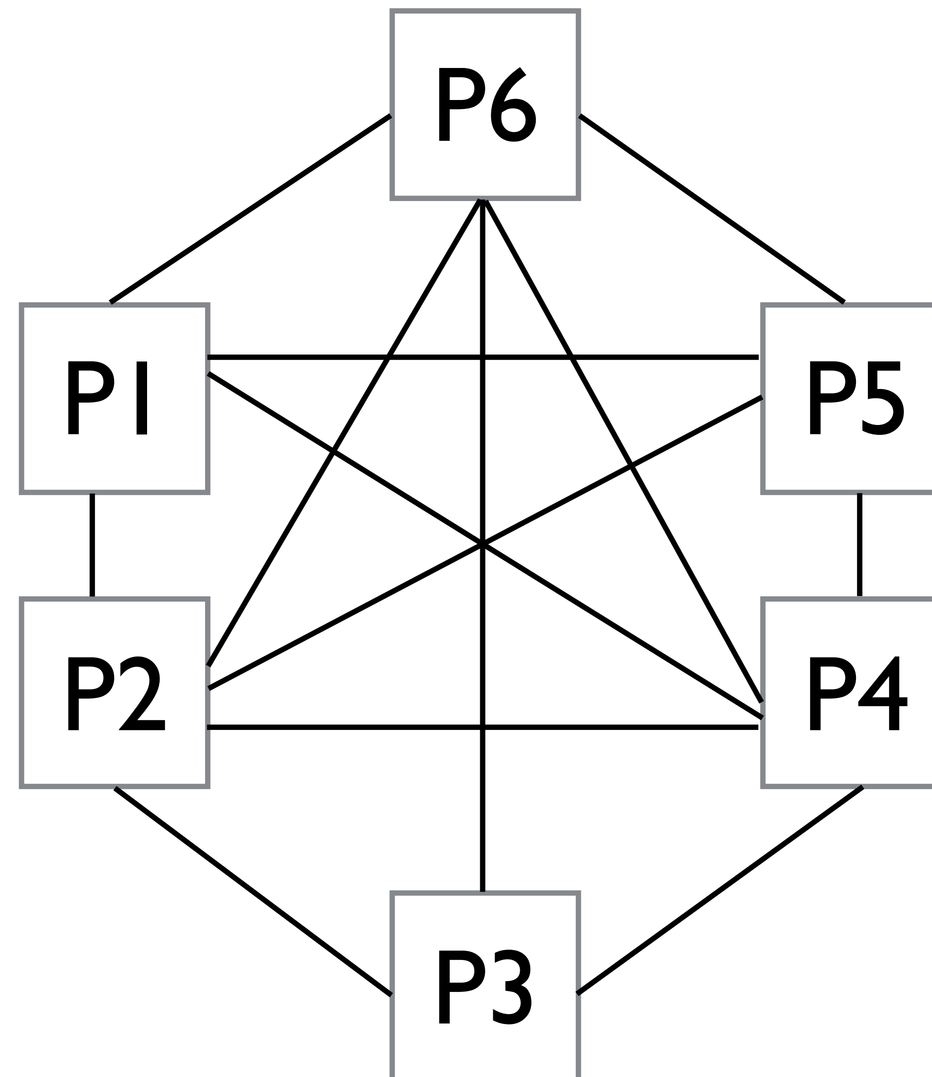
In an *indirect interconnect*, the switches are not necessarily connected to a processor.

Direct interconnects tend to be more common; however we do have an experimental indirect interconnect here at UND for research purposes.

Interconnection Networks: Direct Connects

Direct interconnects come in many types. Ideally, we would want to have a fully connected network, where every node has a connection to every other node, as in this case there will never be any congestion:

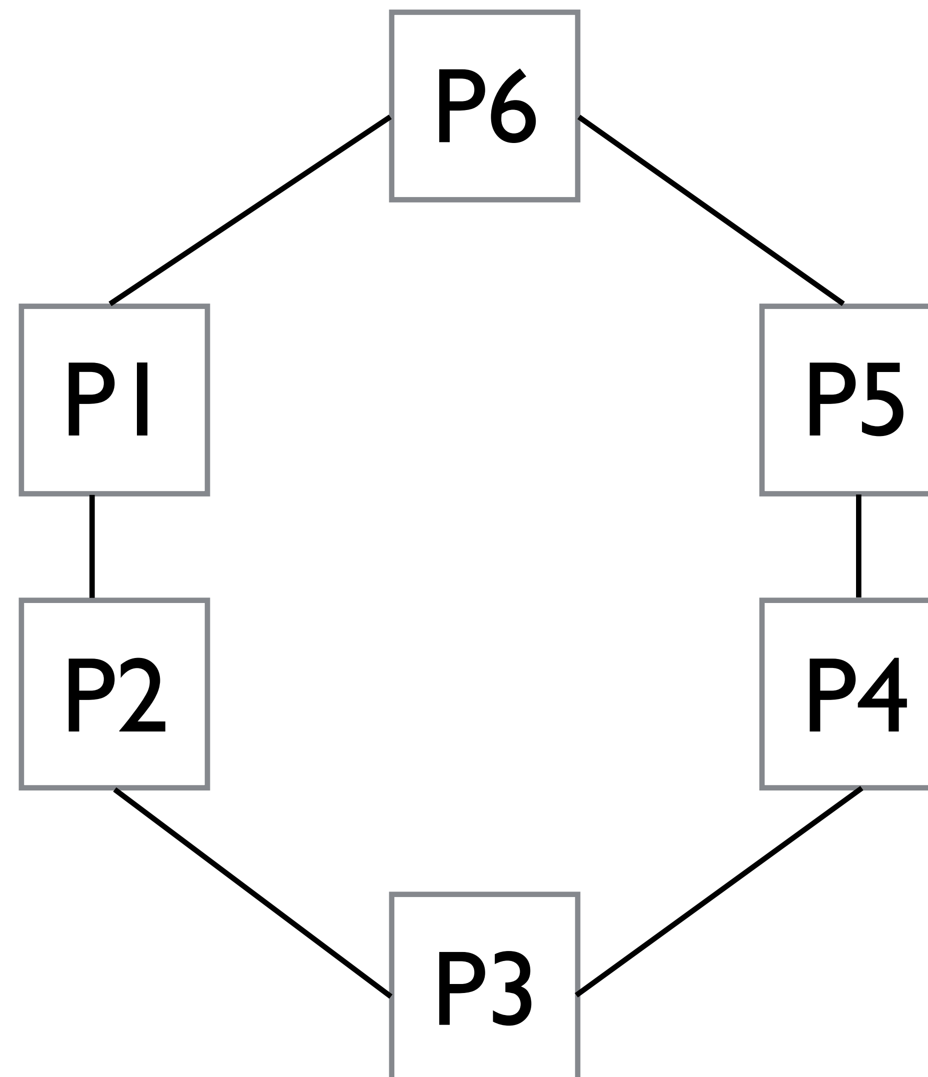
A fully connected network will have P(P-1)/2 connections, where P is the number of processors.



Interconnection Networks: Direct Connects

On the other end of the spectrum we can have a ring. However this comes at the price of potentially a lot of network congestion.

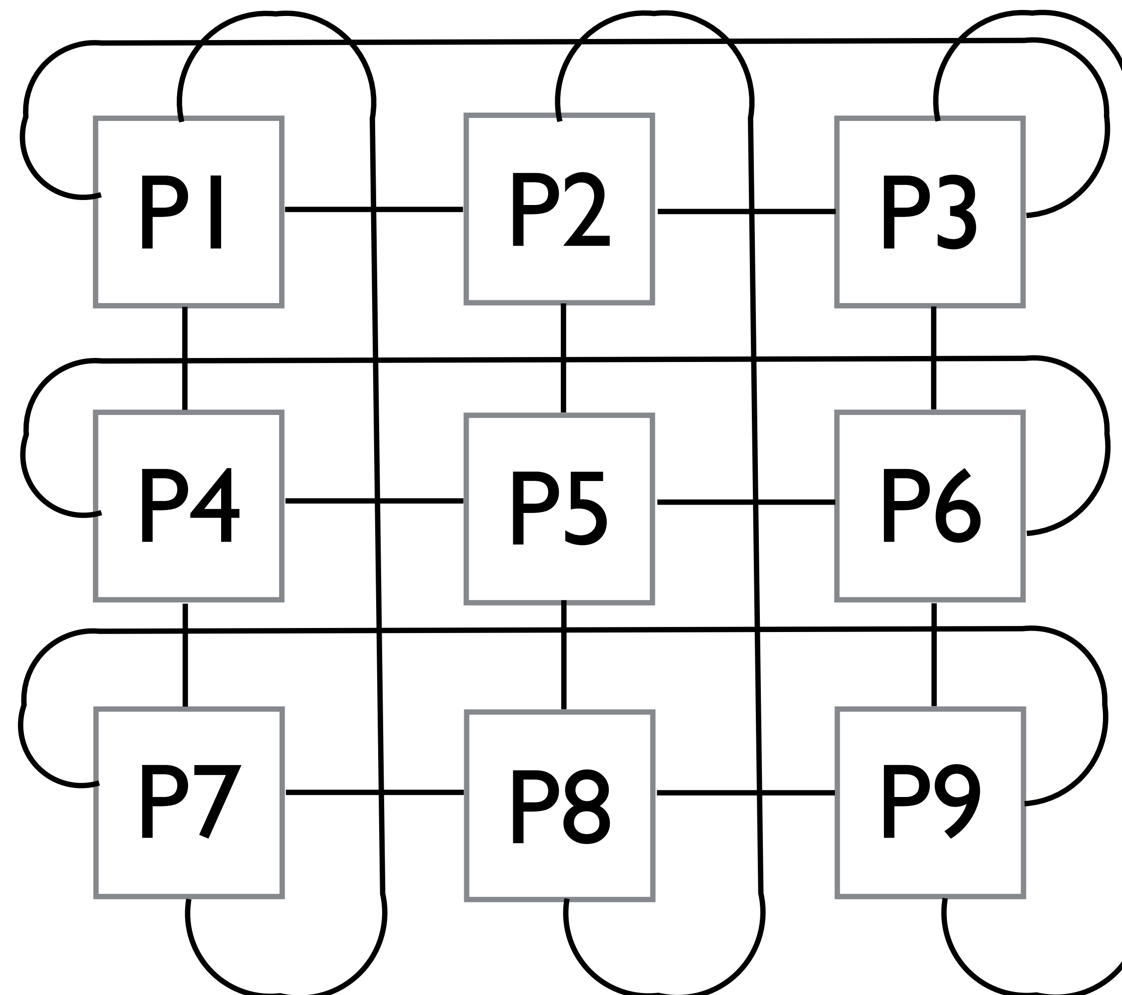
A ring will have P links, where P is the number of processors.



Interconnection Networks: Direct Connects

A common compromise between these two extremes are *toroidal meshes*. You can thing of a torus as an N dimensional cube, with connections going from each side around to the other. Here is a 2D toroidal mesh:

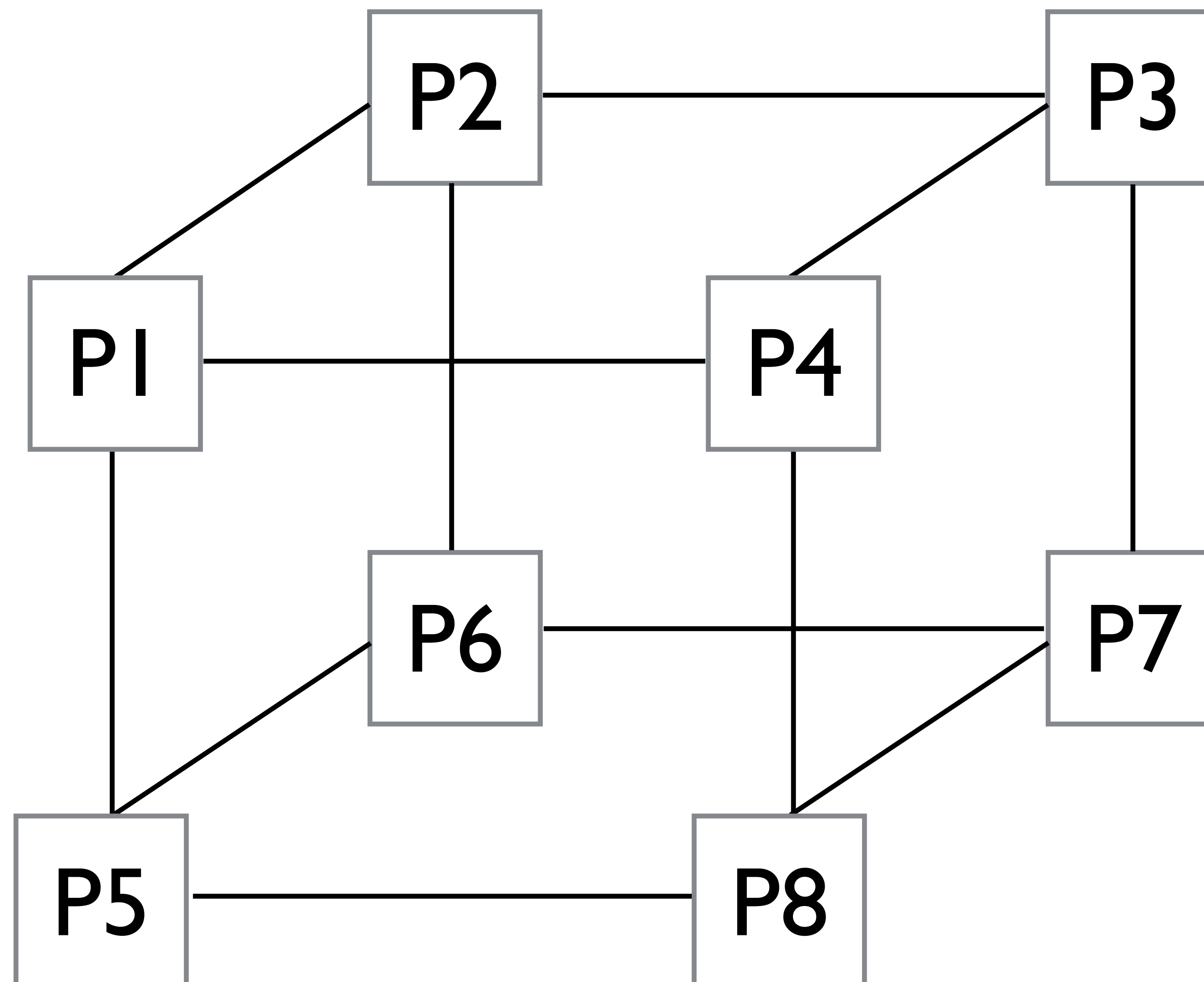
A 2D toroidal mesh will have 3P links, where P is the number of processors.



Interconnection Networks: Direct Connects

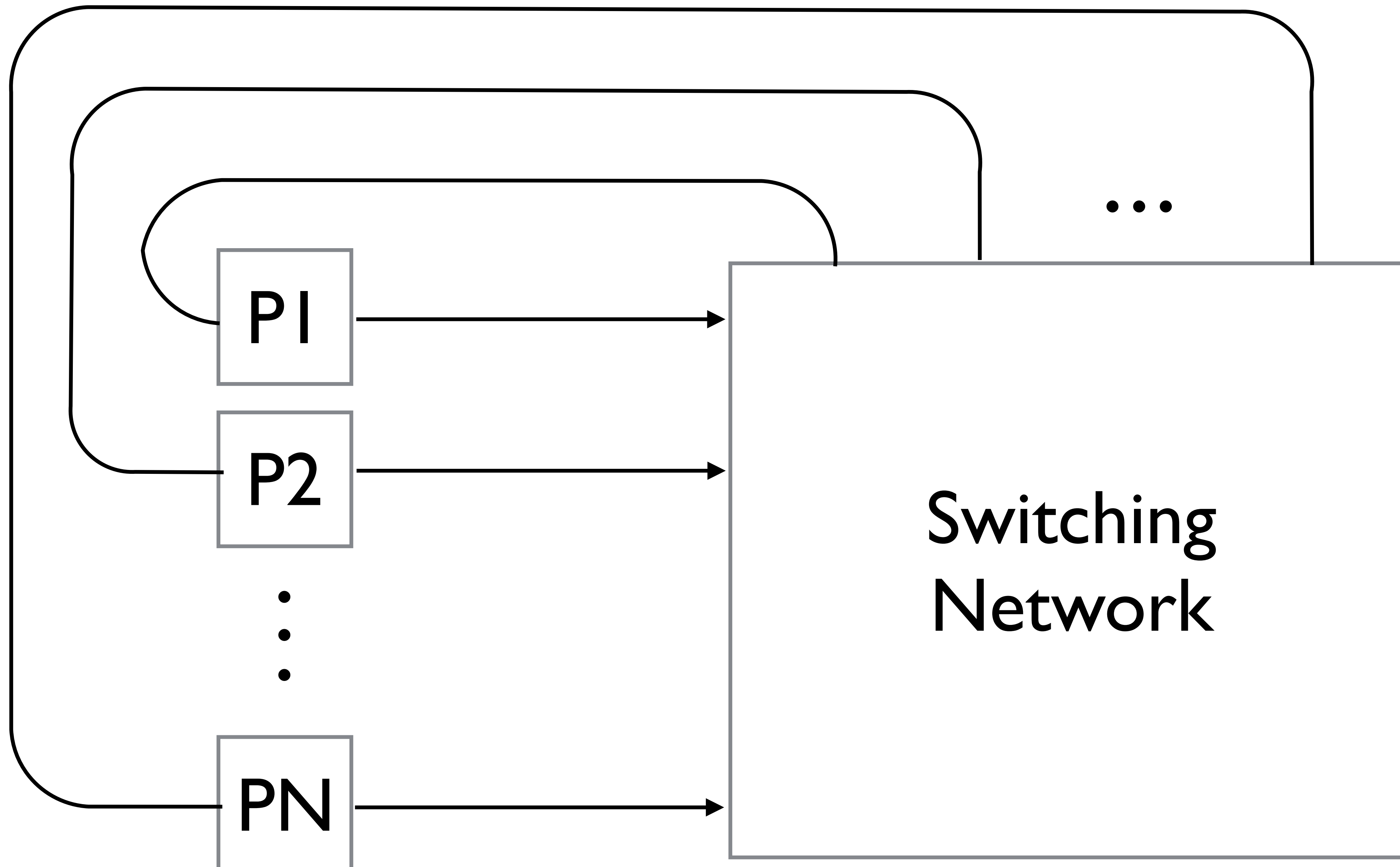
Another common compromise is a hypercube.

A hypercube of dimension d has p = 2^d nodes.



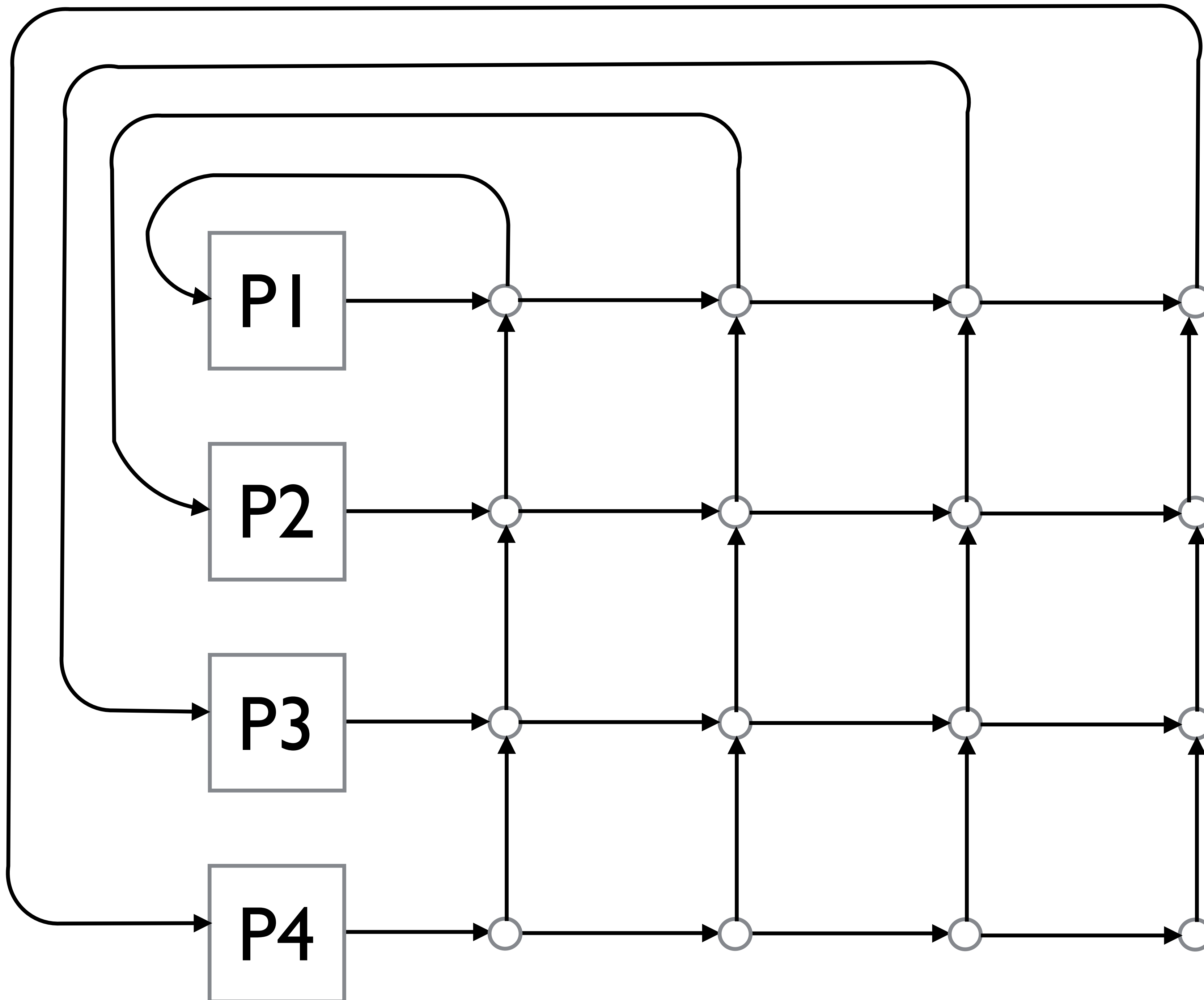
Interconnection Networks: Indirect Connects

Crossbars (as in a previous slide) are a common example of interconnects, and can be used in distributed memory systems as well. In general, indirect connects have the following form:



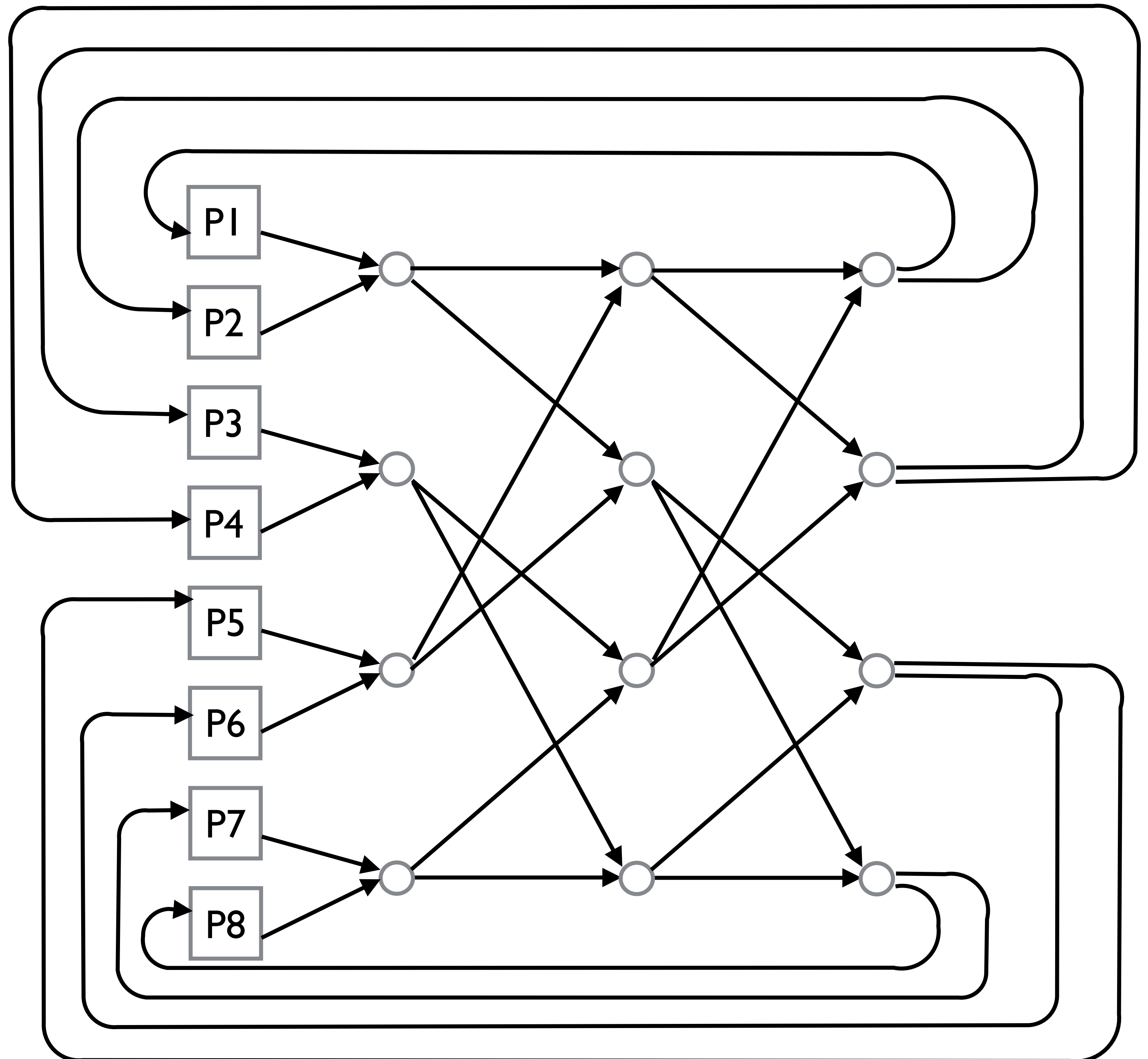
Interconnection Networks: Crossbar

For an example of a crossbar interconnect given this general form:



Interconnection Networks: Omega Network

The Omega Network is another network which uses less links, however has some congestion (in this example if P1 sends a message to P7, P2 cannot simultaneously send a message to P8).



Performance Evaluation

Latency and Bandwidth

In general, the latency of an interconnect (moving the data from processor 1 to processor 2) is L seconds, and the bandwidth is B bytes per second, then the time it takes to transmit a message of n bytes is:

$$\text{transmission time} = L + n/B$$

Given this simple formula it's easy to see that for small n, the transmission time is bounded by L; while for large n, the transmission time is bounded by B.

We need to determine how to send our data to minimize the transmission time. Common strategies involve overlapping communication with other computation to mask the transmission time.

Speedup and Efficiency

In an ideal (distributed computing) world, we can divide the work of our program equally across all cores, without making any additional work for those cores, and without any introducing any overhead from communication.

If we call the serial runtime $T_{serial,}$ the parallel runtime $T_{parallel}$, and the number of processors it's running on is p, this best case scenario is called *linear speedup*:

$$T_{parallel} = T_{serial}/p$$

Unfortunately, this scenario is not too common. On another note, if someone tells you they have a program with more than linear speedup, chances are they also probably have a bridge to sell you.

Speedup and Efficiency

We can use this function:

$$T_{parallel} = T_{serial}/p$$

To define the *speedup* S of a parallel program:

$$S = T_{serial}/T_{parallel}$$

Where linear speedup is where S = p. We can also define the *efficiency* of the parallel program:

$$E = S/p = (T_{serial}/T_{parallel})/p = T_{serial}/(p * T_{parallel})$$

Speedup and Efficiency

Most parallel programs do not have perfect speedup. Rather, there is overhead involved in communication. A more accurate function is:

$$T_{parallel} = T_{serial}/p + T_{overhead}$$

As you increase the number of processors; this overhead (often caused by latency) becomes bounding factor of how fast you can execute your program.

This leads to diminishing returns, where eventually adding more processors does not improve performance very much; and can even start adding more overhead, slowing down performance.

Amdahl's Law

In the 1960s, Gene Amdhal made an observation that has become known as *Amdahl's Law*.

It states that unless 100% of a serial program is parallelized (which is impossible), the maximum speedup is very limited. And it is limited in relation to how much of the program can be parallelized.

Amdahl's Law

Suppose we have a serial program that has a runtime of 20 seconds, that we have made 90% parallel.

In this case, the runtime will be:

$$
\begin{aligned}
& \quad 0.1 * 20s \text{ (for the non-parallel part)} \\
+ & \quad 0.9 * 20s / p \\
= & \quad 2 + 18/p
\end{aligned}
$$

This is rather obvious, but it makes a good point. We can parallelize the heck out of parts of our program, but it's still going to run as slow as the serial parts, no matter how many cores we throw at it.

Scalability

Scalability is a roughly defined term with a variety of uses. Generally, a program is said to be scalable if it can run on increasingly large sized systems.

We can talk about the limits of a programs scalability for given inputs. For example; given our previous scenario of a 90% parallel program that runs in 20s.

1 core:	20s		
2 core:	2 + (0.9 * 20) / 2	=	11s
4 core:	2 + (0.9 * 20) / 4	=	6.5s
8 core:	2 + (0.9 * 20) / 8	=	4.45s
16 core:	2 + (0.9 * 20) / 16	=	3.125s
32 core:	2 + (0.9 * 20) / 32	=	2.526s

We can see that diminishing returns means we're not getting much benefit for adding more than 8 cores.

Taking Timings

Typically we can be worried about 2 things:

1. What is the total time (wall time) that the program takes? Ultimately we want this to be as fast as possible.
2. How long do particular parts of the program take? We can use this information to eliminate bottlenecks.

Taking Timings: Considerations

Note that the time functions may have poor resolution. C's time() function returns seconds, other functions may report times in milliseconds.

However, instructions take nanoseconds (or less). Determining speedup for things that execute very quickly can be difficult if the resolution of the time function cannot effectively capture the execution time.

Taking Timings: Considerations

Another problem is that there is no good way to synchronize clocks across processors; so we can't take a start time reading on processor one, and send a message to processor two and then take the end time reading there.

What we have to do is take a start time on processor one, send a message to two, which responds back to processor one and then we can get the round trip time.

Taking Timings: Considerations

Finally, we have to be aware of the *variability* in taking time measurements.

CPUs and distributed systems are non-deterministic. Running the same thing twice may give us different runtimes (given everything else going on in the hardware, memory, caches, operating systems and interconnects). Even physical properties (like quantum physics) can effect the runtime of an operation.

Because of this, to get somewhat accurate information about the time something takes, we need to run it multiple times to get multiple readings. We can then use the min/max/average/median values to make inferences about how fast our programs is running.

Fin.

Now you have all the background information for us to start coding!