Multicore Programming

Mainack Mondal
Sandip Chakraborty

CS60203
Autumn 2024

Today's class

- Why write multicore programs?

- Because parallelism is everywhere!

- But why not (just) ILP, or SIMD, or something else?

- …and why not `n` computers?

- Comparing single and multi-core systems
  - Simultaneous Multithreading
  - SMP and NUMA

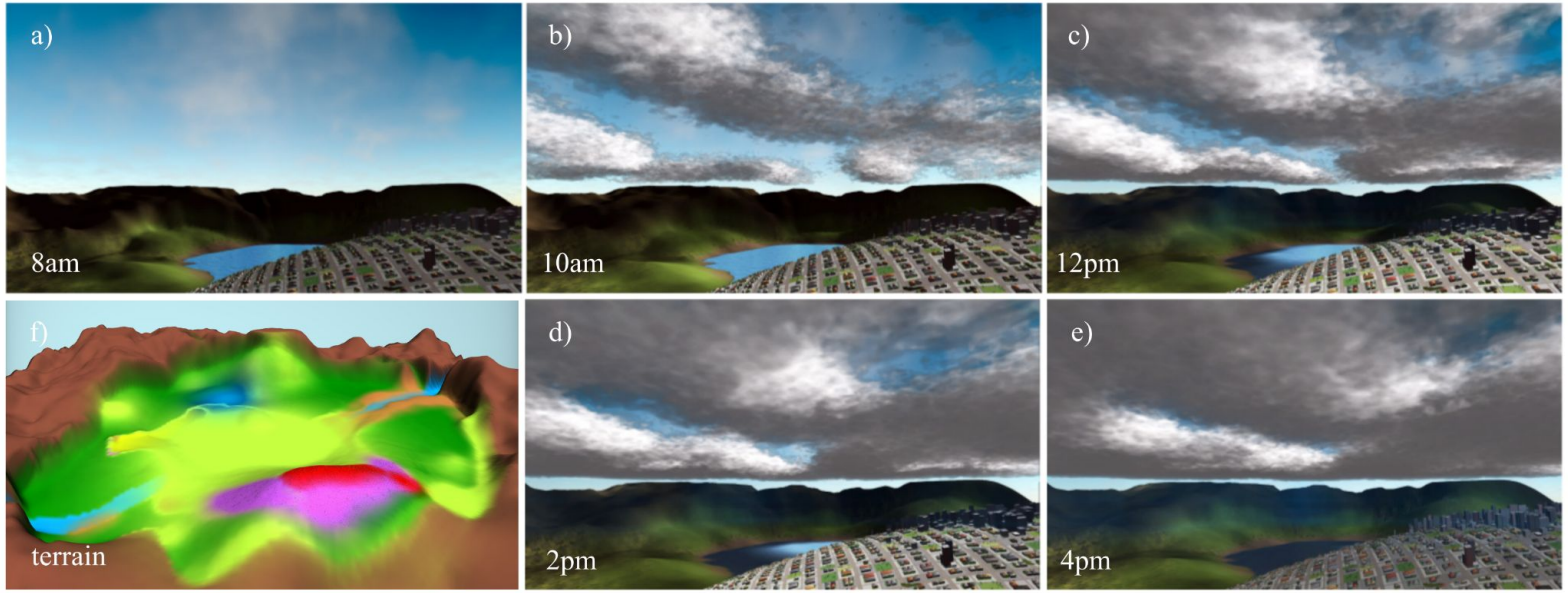
- How to write efficient multi-core programs?
  - Minimize synchronization overhead
  - Be mindful of cache coherence
  - per-core sharding, and Seastar

question…

Why write multicore programs?

Parallelism: Scientific Computing

1. Scientific computing
   - eg: weather simulation, aerodynamics, simulation of epidemics



Fast Weather Simulation for Inverse Procedural Design of 3D Urban Models, Ignacio et al., 2017

Parallelism: Computational Geometry

2. Computational geometry/computer graphics

Example-

```
// find all points with x coordinate > y coordinate
for p in points:
   if p.x > p.y: answer.append(p)
```

Clearly…all points can be processed in parallel!

Parallelism: Computational Geometry

2. Computational geometry/computer graphics

Example-

```
// for q queries find nearest neighbor among r points
for query in queries:
   ans = INF
   for point in reference:
      ans = min(ans, distance(query, point))
```

This is the nearest neighbor problem, can you parallelise this?

Parallelism is Everywhere

1. **Scientific computing**
   - eg: weather simulation, aerodynamics, simulation of epidemics

2. **Computational geometry**/computer **graphics**

   - The domain of the problem can be segmented

3. **Deep learning** (of course)

   - Tensor operations are inherently parallel

4. **Server workloads**

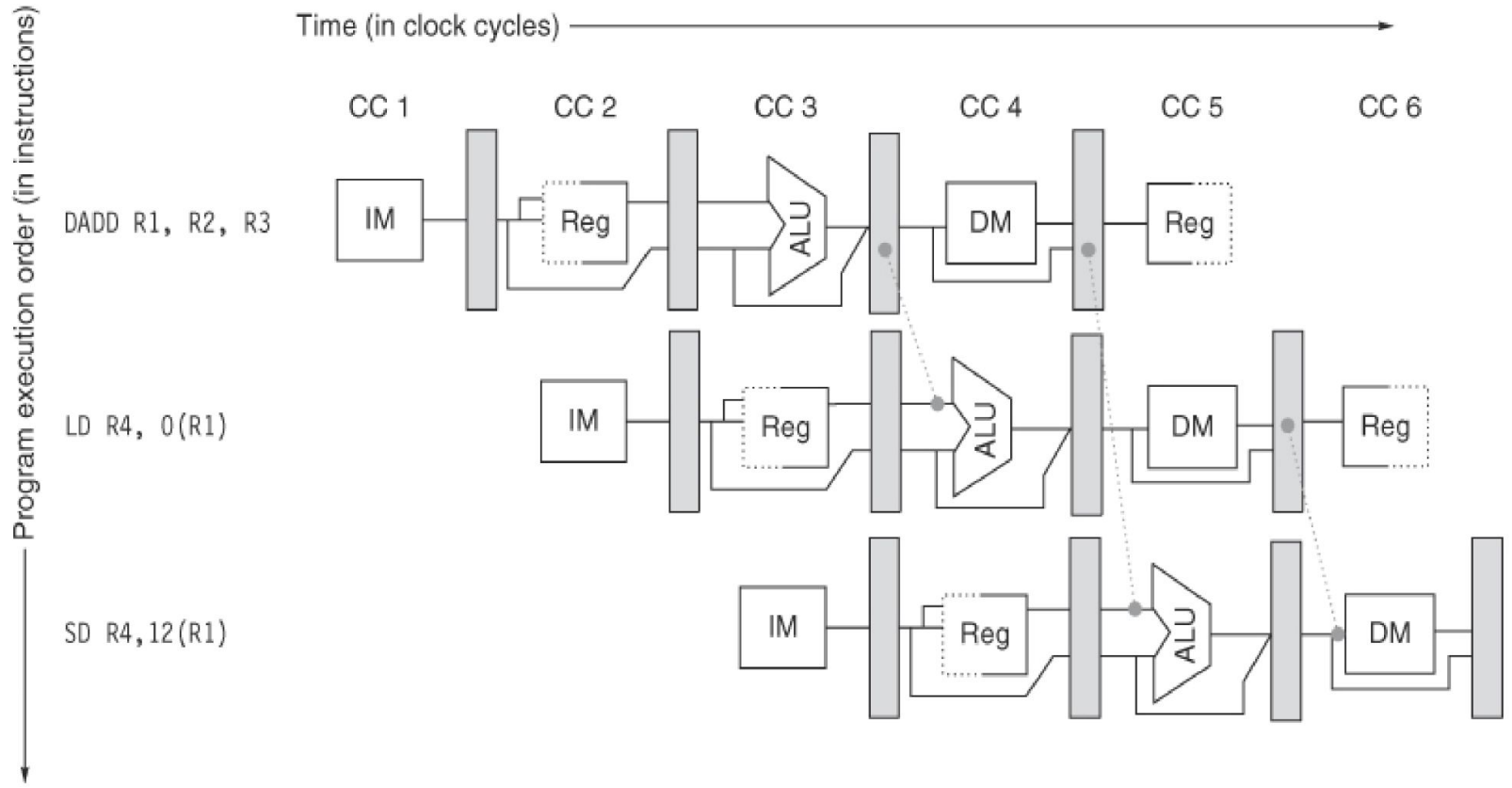
   - Handling multiple independent requests

so parallelism is everywhere…

How do we exploit it?

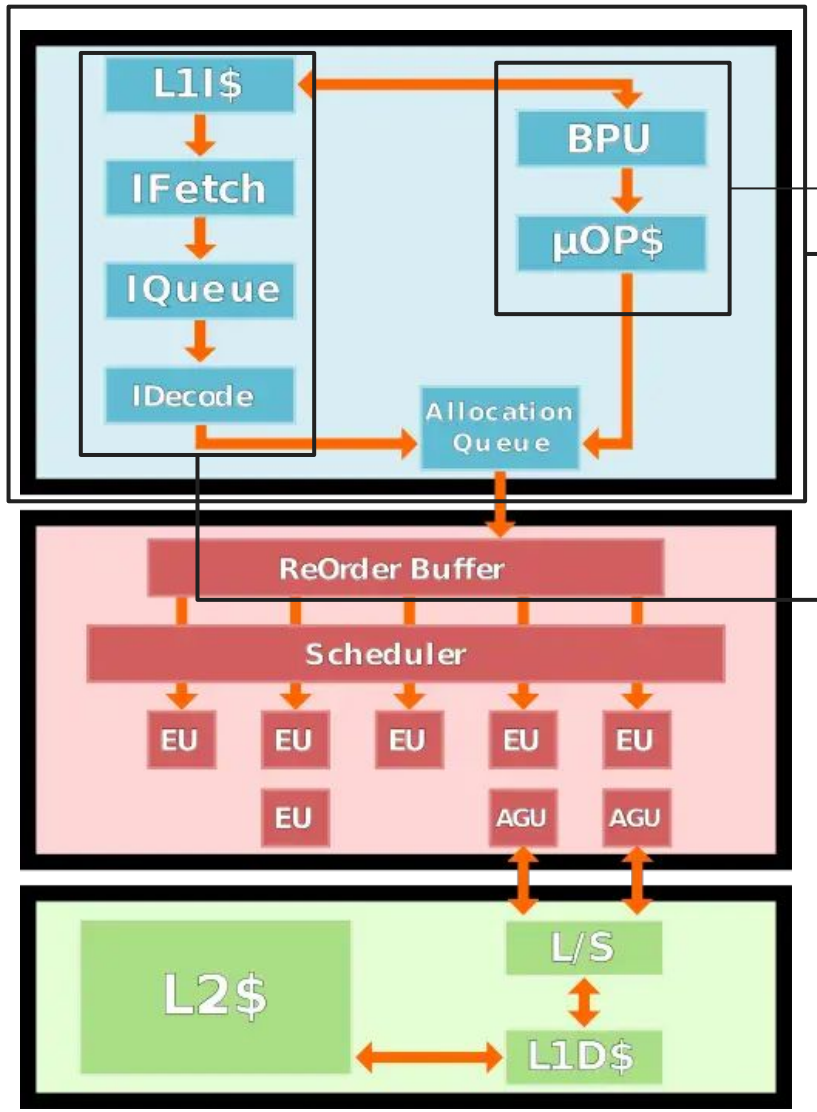
How about Instruction Level Parallelism?

Remember this?



CS4617 Computer Architecture - Lecture 20: Pipelining
Reference: Appendix C, Hennessy & Patterson Reference

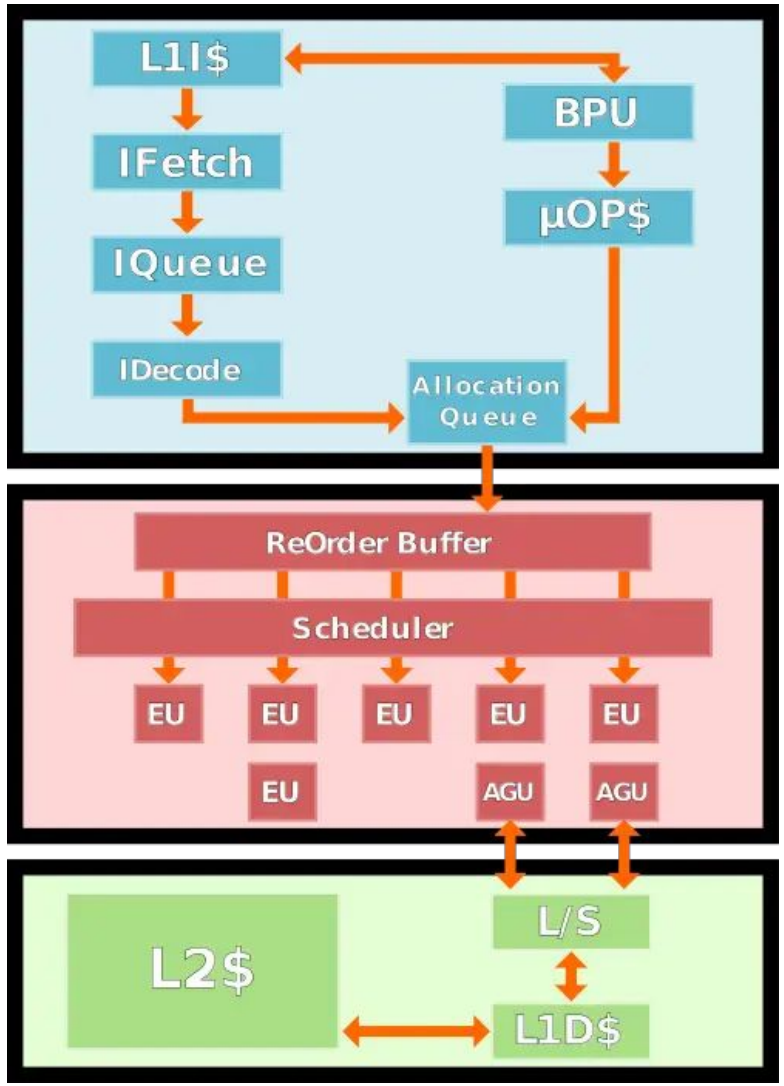
That was a fake diagram, here's the real one



The goal of the front-end is to feed the back-end with a sufficient stream of operations which it gets by decoding instructions coming from memory.

The front-end has two major pathways: the **µOPs cache path** and the **legacy path.** The legacy path is the traditional path whereby variable-length x86 instructions are fetched from the level 1 instruction cache, queued, and consequently get decoded into simpler, fixed-length µOPs.

That was a fake diagram, here's the real one



Just for reference

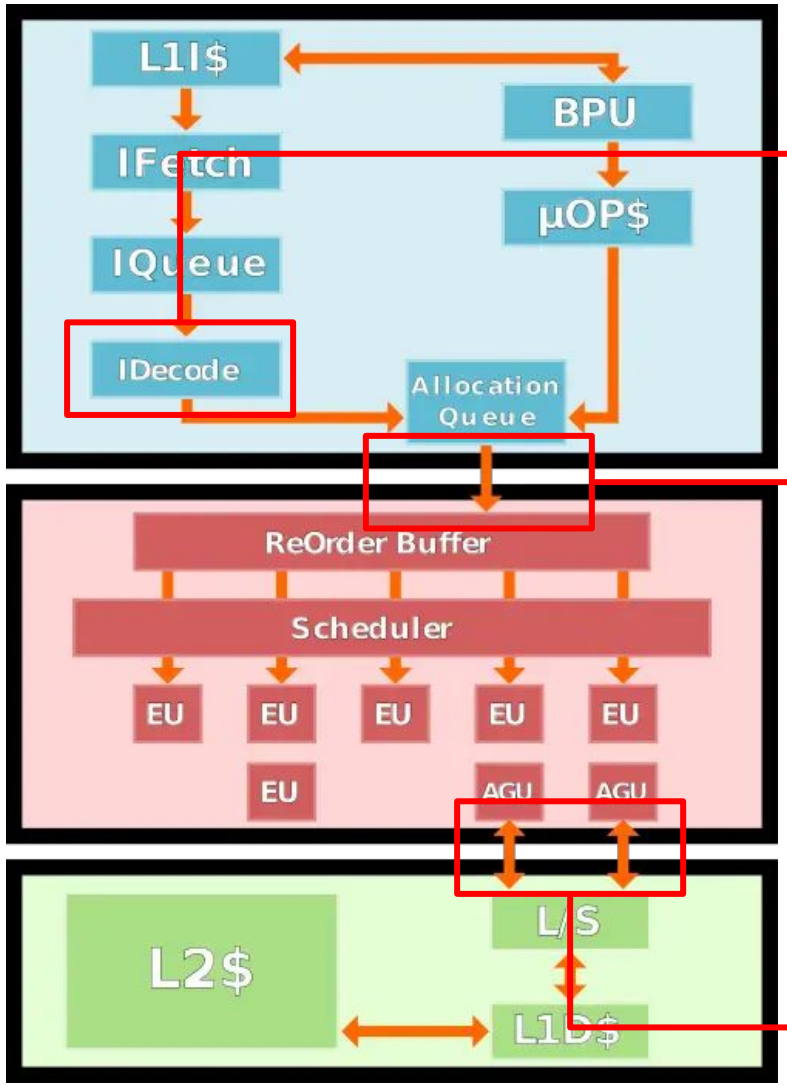
Read if you feel like it :)

We'll explain relevant parts as we go

The job of the front-end is to feed the back-end with a sufficient stream of operations which it gets by decoding instructions coming from memory.

The front-end has two major pathways: the **µOPs cache path** and the **legacy path**. The legacy path is the traditional path whereby variable-length x86 instructions are fetched from the level 1 instruction cache, queued, and consequently get decoded into simpler, fixed-length µOPs.

ILP has Architectural Bottlenecks!

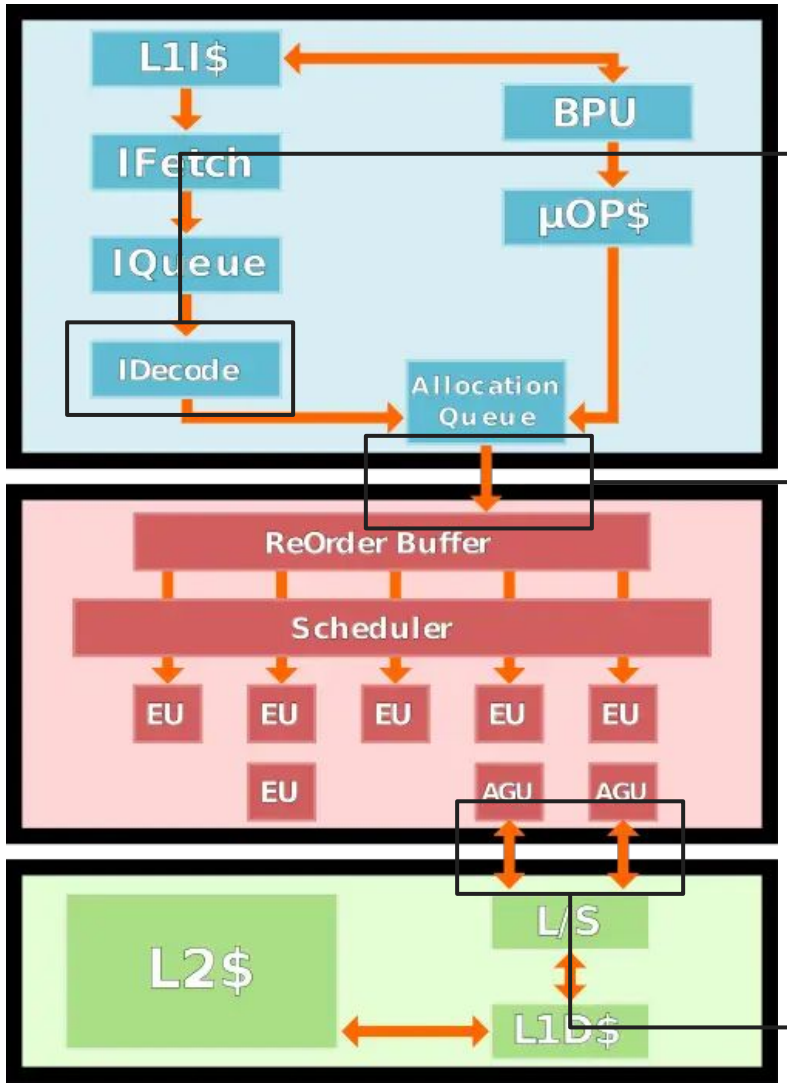


Reads instructions and “decodes” them, this circuit is like a hardware compiler and can **only decode upto “decode width” uops/cycle**

Issues (sends for execution) decoded instructions, upto “**issue width” uops/cycle**

Upto **“retire width” uops/cycle** instructions retired

ILP has Architectural Bottlenecks!



Reads instructions and "decodes" them, this circuit is like a hardware interpreter and can **only decode upto "decode width" uops/cycle**

All of these are bound by hardware circuitry, and are in 4-8 uops/cycle range

Issues decoded instructions, upto "**issue width" uops/cycle**

Any one of them being low can be a bottleneck!

Upto **"retire width" uops/cycle** instructions retired

What if there was no hardware bottleneck?

Hardware circuitry is also implementing algorithms

What if you have O(n) circuits? 64 bit computer is 4x slower than 16 bit

But for a second let us **assume there is no hardware bottleneck** (in circuit complexity or availability of functional units for parallelisation)

Would ILP be enough then?

ILP's fundamental problem

No…**we almost never see the theoretical maximum ops/cycle** even today!

1. Instruction dependencies
2. Branch mispredictions
3. Stalling on memory/cache
4. And pipeline stalls/flushes from the above

Important Note

1. We are looking at the limitations of ILP, SIMD, and distributed computing. To motivate the need for multicore programming.
2. But it should be noted that these approaches have their **own use cases** (eg, distributed computing is necessary when working on very large data)
3. In many scenarios **parallelization from some/all of these paradigms** can/should be used (eg, distributed programs with each worker is multicore)

parallelism is everywhere…

If not ILP, then…how?

How about SIMD?

1. Writing SIMD code is hard
2. Auto-vectorization support is terrible
3. Explicit SIMD (mostly) makes code unportable
4. Downclocking

How about SIMD?

1. Writing SIMD code is hard

2. Auto-vectorization support is terrible

Surely it can’t be that bad…

3. Explicit SIMD (mostly) makes code unportable

4. Downclocking

What does this code do?

```
__m256d [redacted](__m256d x) noexcept
{
    constexpr double tp_scalar = 1. / (2. * M_PI);
    __m256d tmp;

    // x = x * tp;
    const __m256d tp = _mm256_set1_pd(tp_scalar);
    x = _mm256_mul_pd(x, tp);

    // x = x - (double(.25) + std::floor(x + double(.25)));
    const __m256d v_25 = _mm256_set1_pd(0.25);
    tmp = _mm256_add_pd(x, v_25);

    tmp = _mm256_floor_pd(tmp);
    tmp = _mm256_add_pd(v_25, tmp);

    x = _mm256_sub_pd(x, tmp);

    // x = x * (double(16.) * (std::abs(x) - double(.5)));
    const __m256d v_5 = _mm256_set1_pd(0.5);
    const __m256d v_16 = _mm256_set1_pd(16.0);
    tmp = _mm256_abs_pd(x);
    tmp = _mm256_sub_pd(tmp, v_5);
    tmp = _mm256_mul_pd(v_16, tmp);
    x = _mm256_mul_pd(x, tmp);

    // x = x + (double(.225) * x * (std::abs(x) - double(1.)));
    const __m256d v_225 = _mm256_set1_pd(0.225);
    const __m256d v_1 = _mm256_set1_pd(1.0);
    tmp = _mm256_abs_pd(x);
    tmp = _mm256_sub_pd(tmp, v_1);

    tmp = _mm256_mul_pd(x, tmp);
    tmp = _mm256_mul_pd(v_225, tmp);
    x = _mm256_add_pd(x, tmp);

    return x;
}
```

What does this code do?

```
__m256d cos_vector(__m256d x) noexcept
{
    constexpr double tp_scalar = 1. / (2. * M_PI);
    __m256d tmp;

    // x = x * tp;
    const __m256d tp = _mm256_set1_pd(tp_scalar);
    x = _mm256_mul_pd(x, tp);

    // x = x - (double(.25) + std::floor(x + double(.25)));
    const __m256d v_25 = _mm256_set1_pd(0.25);
    tmp = _mm256_add_pd(x, v_25);

    tmp = _mm256_floor_pd(tmp);
    tmp = _mm256_add_pd(v_25, tmp);

    x = _mm256_sub_pd(x, tmp);

    // x = x * (double(16.) * (std::abs(x) - double(.5)));
    const __m256d v_5 = _mm256_set1_pd(0.5);
    const __m256d v_16 = _mm256_set1_pd(16.0);
    tmp = _mm256_abs_pd(x);
    tmp = _mm256_sub_pd(tmp, v_5);
    tmp = _mm256_mul_pd(v_16, tmp);
    x = _mm256_mul_pd(x, tmp);

    // x = x + (double(.225) * x * (std::abs(x) - double(1.)));
    const __m256d v_225 = _mm256_set1_pd(0.225);
    const __m256d v_1 = _mm256_set1_pd(1.0);
    tmp = _mm256_abs_pd(x);
    tmp = _mm256_sub_pd(tmp, v_1);

    tmp = _mm256_mul_pd(x, tmp);
    tmp = _mm256_mul_pd(v_225, tmp);
    x = _mm256_add_pd(x, tmp);

    return x;
}
```

calculates `cos` for four 64-bit numbers (4 element vector)

Auto-vectorization in GCC and Clang

algorithm	procedure	GCC 9		GCC 10		Clang 9		Clang 11	
		AVX2	AVX512	AVX2	AVX512	AVX2	AVX512	AVX2	AVX512
accumulate — custom	accumulate_custom_epi8	no	no	no	no	no	no	no	no
	accumulate_custom_epi32	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
accumulate — default	accumulate_epi8	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
	accumulate_epi32	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
all_of	all_of_epi8	no	no	no	no	no	no	no	no
	all_of_epi32	no	no	no	no	no	no	no	no
any_of	any_of_epi8	no	no	no	no	no	no	no	no
	any_of_epi32	no	no	no	no	no	no	no	no
copy	copy_epi8	no	no	no	no[1]	no	no	no	no
	copy_epi32	no	no	no	no[1]	no	no	no	no
copy_if	copy_if_epi8	no	no	no	no[1]	no	no	no	no
	copy_if_epi32	no	no	no	no[1]	no	no	no	no
count	count_epi8	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
	count_epi32	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
count_if	count_if_epi8	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
	count_if_epi32	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
fill	fill_epi8	no	no	no[2]	no[2]	no	no	no[2]	no[2]
	fill_epi32	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**
find	find_epi8	no	no	no	no	no	no	no	no
	find_epi32	no	no	no	no	no	no	no	no
find_if	find_if_epi8	no	no	no	no	no	no	no	no
	find_if_epi32	no	no	no	no	no	no	no	no
is_sorted	is_sorted_epi8	no	no	no	no	no	no	no	no
	is_sorted_epi32	no	no	no	no	no	no	no	no
none_of	none_of_epi8	no	no	no	no	no	no	no	no
	none_of_epi32	no	no	no	no	no	no	no	no
remove	remove_epi8	no	no	no	no	no	no	no	no

Things are bad even for stdlib algorithms!

[Autovectorization status in GCC & Clang in 2021]

Downclocking due to SIMD

Mode	Base	Turbo Frequency/Active Cores													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Normal	2,200 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,600 MHz	2,600 MHz
AVX2	1,800 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,200 MHz	2,200 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,500 MHz	2,500 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz

- CPUs have multiple (eg, 3 for Intel) levels of clock speeds (eg, L0, L1, L2)
- Executing **SIMD instructions downclocks the CPU**, and even non-SIMD instructions will execute at lower clock-cycle
- Downclocking **depends on the instructions** (eg, noticeable downclocking is observed most often x86 for FMA, and AVX-512)
- Downclocking depends on the **number of cores** on which SIMD in use (as figure illustrates for Xeon Gold 5120)

Hence it is important to benchmark and verify changes!

parallelism is everywhere…

Why not use `n` different computers?

Why not use `n` different computers?

1. **Network overhead / bottleneck between cores**
   - Each node needs data in it's memory to perform computation
   - If rate of data transfer is slower than computation (which is almost always the case), computation gets bottlenecked
2. **More power consumption for the same number of cores**
   - More operations needed to produce same results (additonal network IO, syscall overhead, and more…)
3. **Higher latency for all operations / jobs**
   - The system will first divide up the tasks, and corresponding data, and then combining results (needing network IO)

Today's class

- Why do we care about multicore programs?
  - Because parallelism is everywhere!
  - But why not (just) ILP, or SIMD, or something else?
  - …and why not `n` computers?

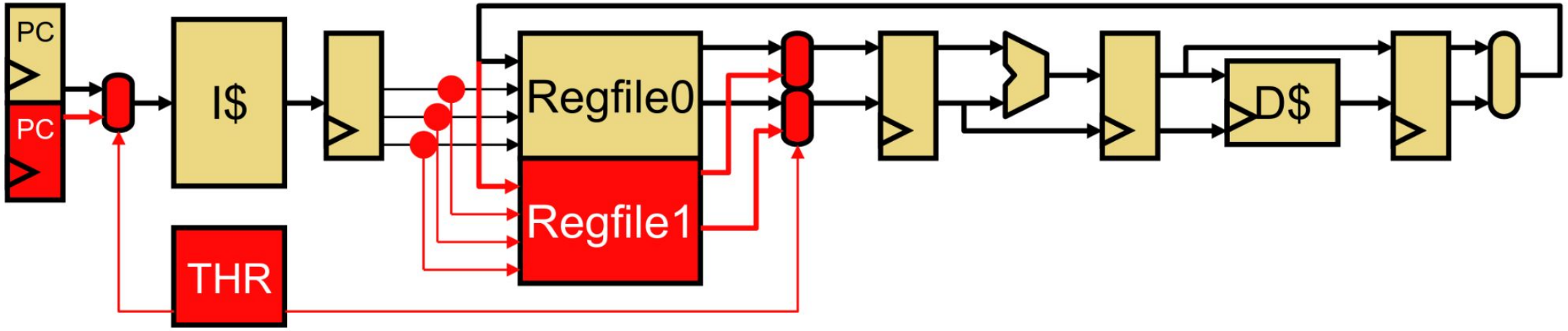
- Comparing single and multi-core systems
  - Simultaneous Multithreading
  - SMP and NUMA

- How to write efficient multi-core programs?
  - Minimize synchronization overhead
  - Be mindful of cache coherence
  - per-core sharding, and Seastar

Comparing single and multi-core systems

But how would you make a “multi”core processor?

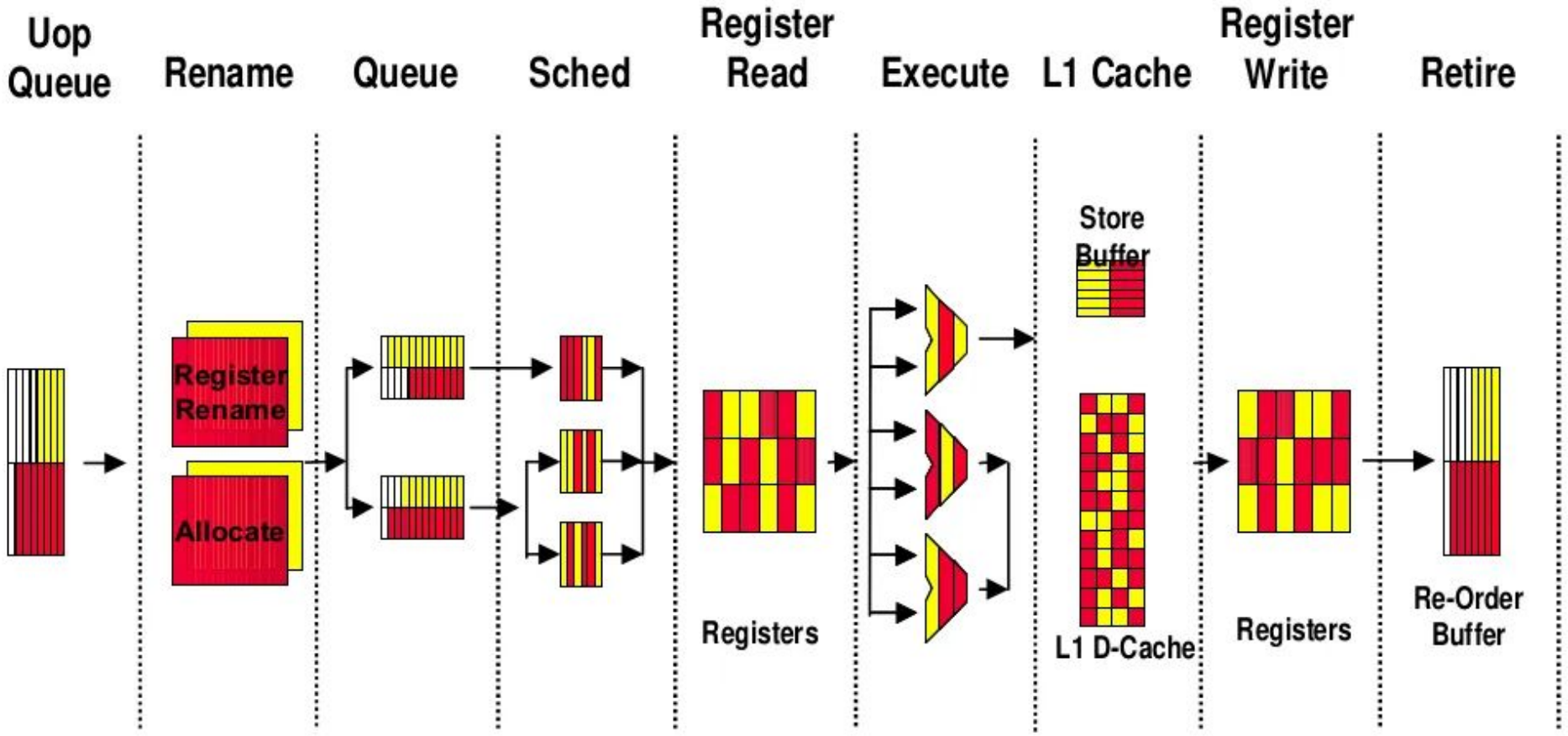
Idea 1: Simultaneous Multithreading



- Modern processors share **2 sets of registers** for one processing core
- This is done because most often one **single “thread” of execution can’t utilize all the CPU** throughput available with a core
- This is one way of making “2 cores out of 1”, and is almost universally available on computers

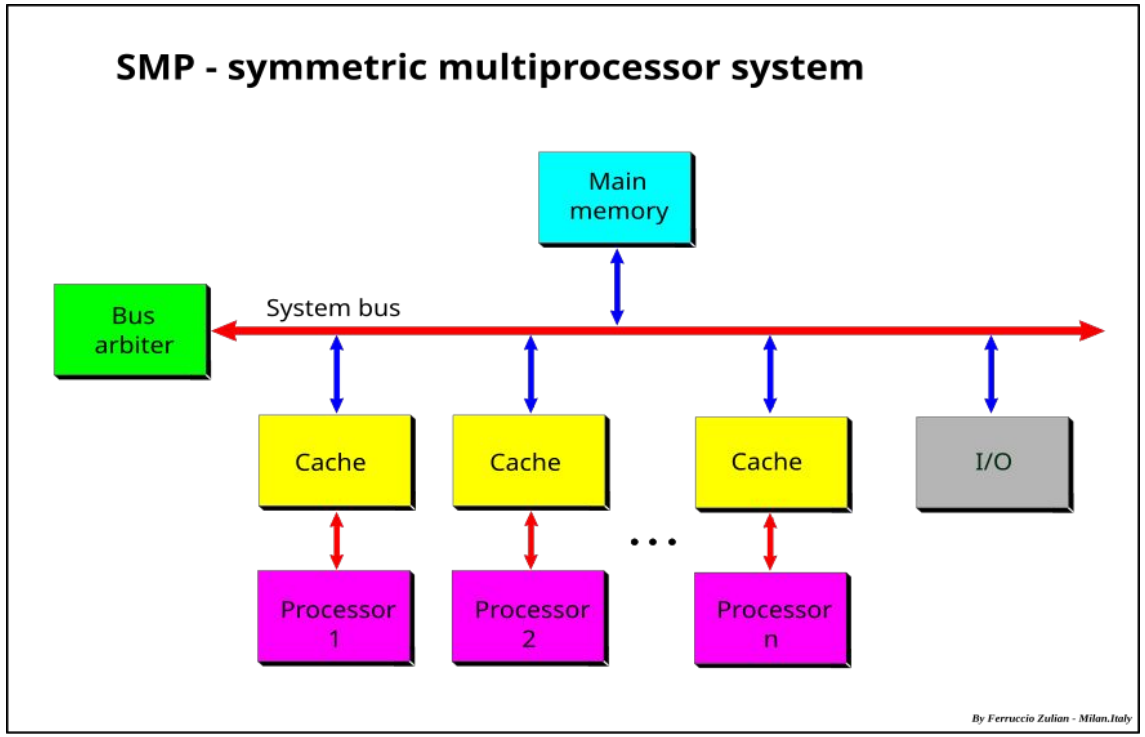
Fun fact: it’s questionable, and applications like HFTs prefer to disable it, can you guess why?

Simultaneous Multithreading: The Real Picture



Intel Technology Journal, Vol 06, Issue 01, 2002.

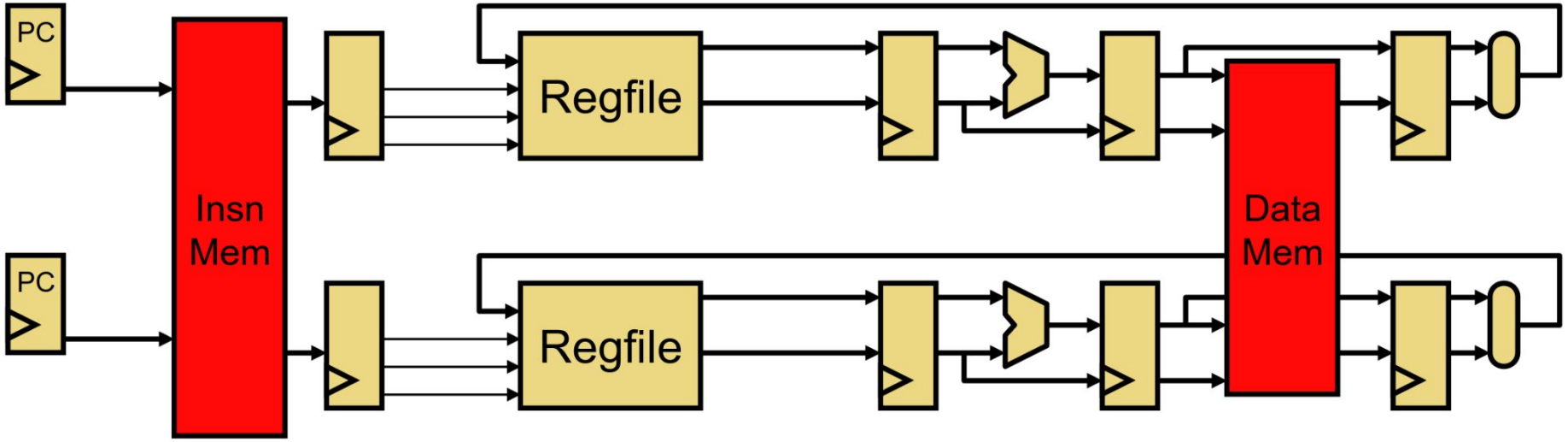
Idea 2: Symmetric Multiprocessor



Essentially, multiple cores on 1 chip, and all of them

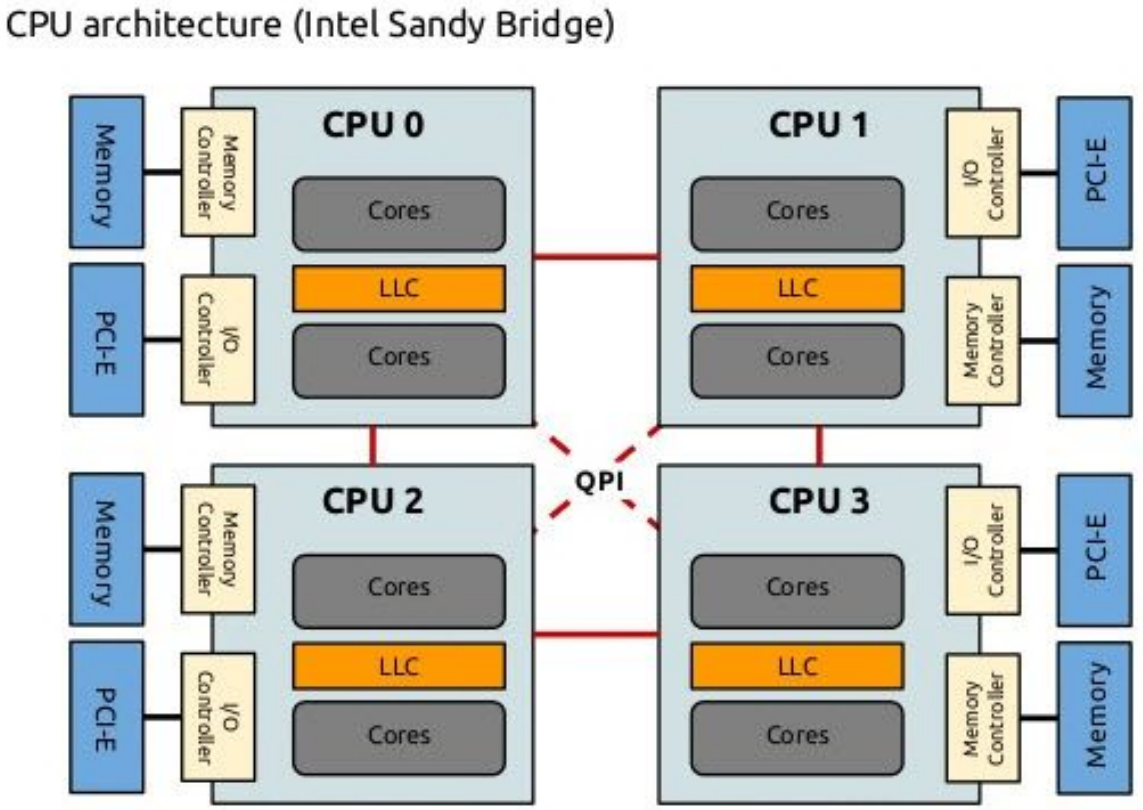
1. Have some exclusive resources (registers, functional units, …)
2. Symmetrically share the rest (lower level caches, memory…)

Symmetric Multiprocessor: Design Consideration



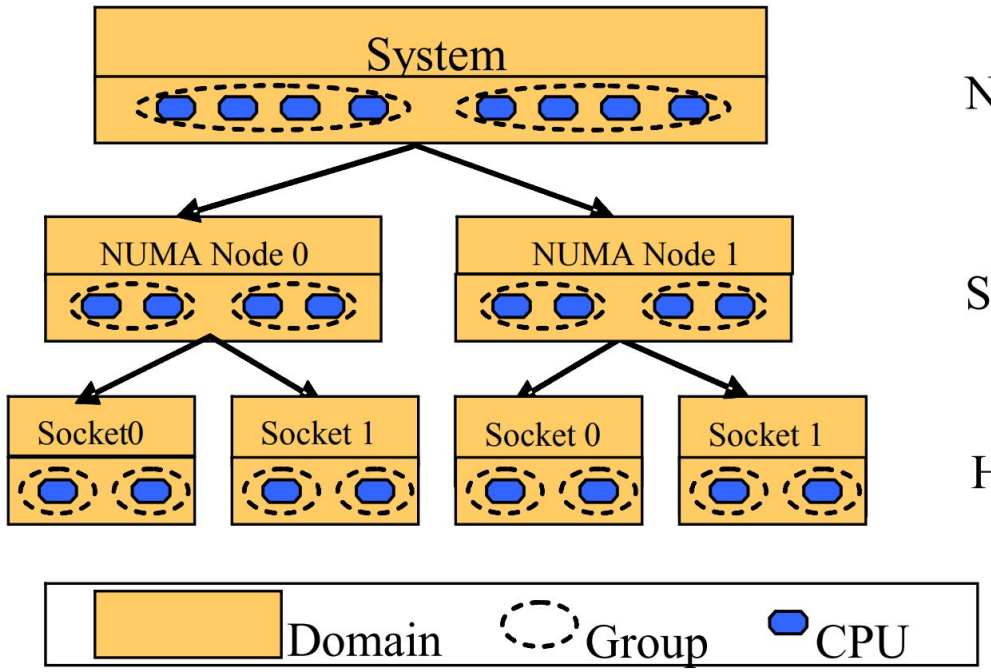
Can you tell what's wrong with this design?

Idea 3: Non-Uniform Memory Access



Non-Uniform Memory Access systems arise when the **cost of memory access is not uniform**, i.e. access-time for a word A is consistently significantly more than the access-time for a word B

NUMA Example



NUMA Domain — A NUMA system made by 2 SMP cores

SMP Domains — Can you see why this is NUMA?

HT Domains — Intel® Journal

Non-Uniform Memory Access systems arise when the **cost of memory access is not uniform**, i.e. access-time for a word A is consistently significantly more than the access-time for a word B

Today's class

- Why do we care about multicore programs?
  - Because parallelism is everywhere!
  - But why not (just) ILP, or SIMD, or something else?
  - …and why not `n` computers?

- Comparing single and multi-core systems
  - Simultaneous Multithreading
  - SMP and NUMA

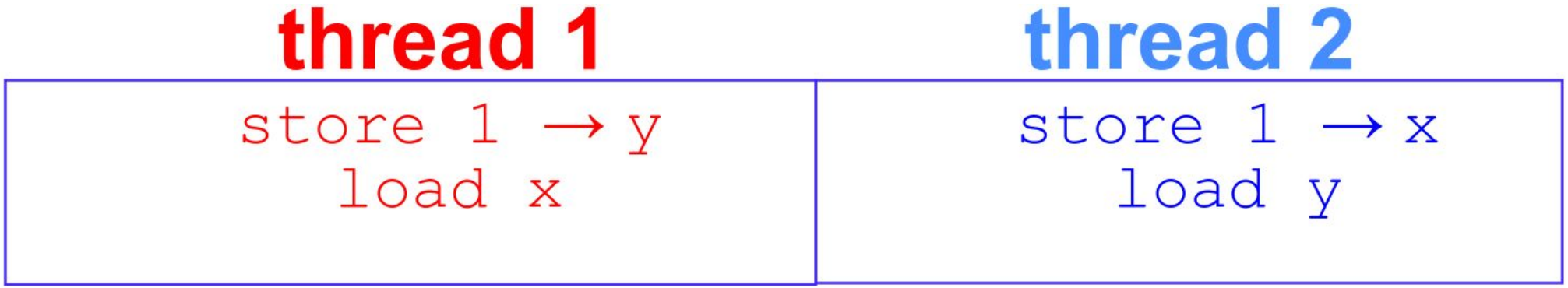
- How to write efficient multi-core programs?
  - Minimize synchronization overhead
  - Be mindful of cache coherence
  - per-core sharding, and Seastar

How to write efficient multicore programs?

Problem: Multicore programming is hard!

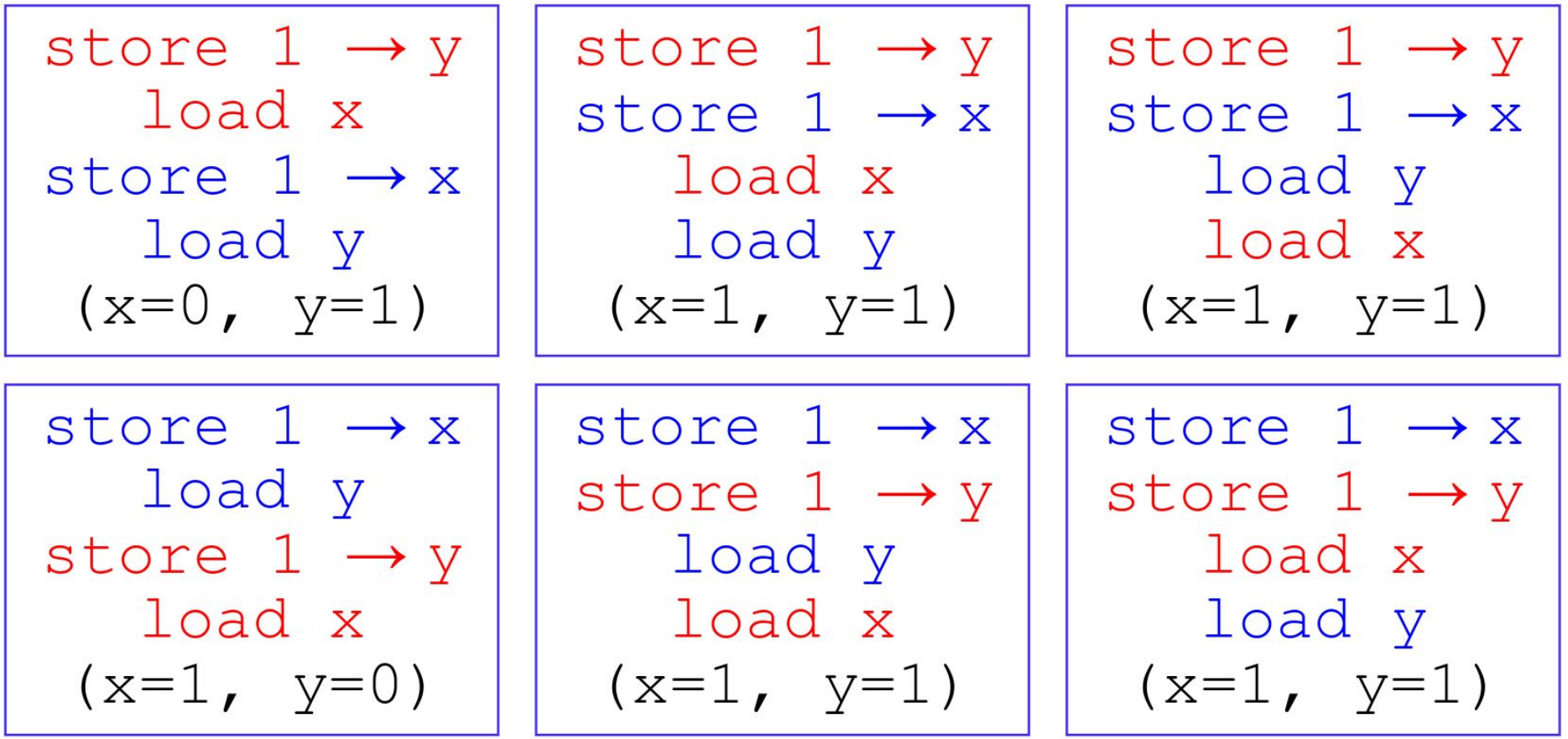
1. Behaviour depends on the instructions, and also on **scheduling of threads**
2. Leads to subtle, hard to track, and **very hard to reproduce** (non-deterministic) bugs
3. Hard to reason about correctness!

Consider 2 cores executing the following code(s). What are the possible values retrieved by the loads?



Multicore programs are hard(er) to reason about

result can be anything, anytime!



Solution: Synchronization

1. Within a thread execution is sequential (not exactly, as modern processors are out-of-order…but it behaves like it). Threads are often executing on different cores at once
2. **Across threads** there is **no guarantee on order or timing** of instructions
3. We can introduce primitives to give us **some guarantees on the ordering of the instructions** such primitives are called synchronization primitives
4. Locks are one example of this!

Synchronization Solution: Locks

We have covered locks in the OS course.

What's relevant here is-

Please avoid locks if you can 🙏

Please…

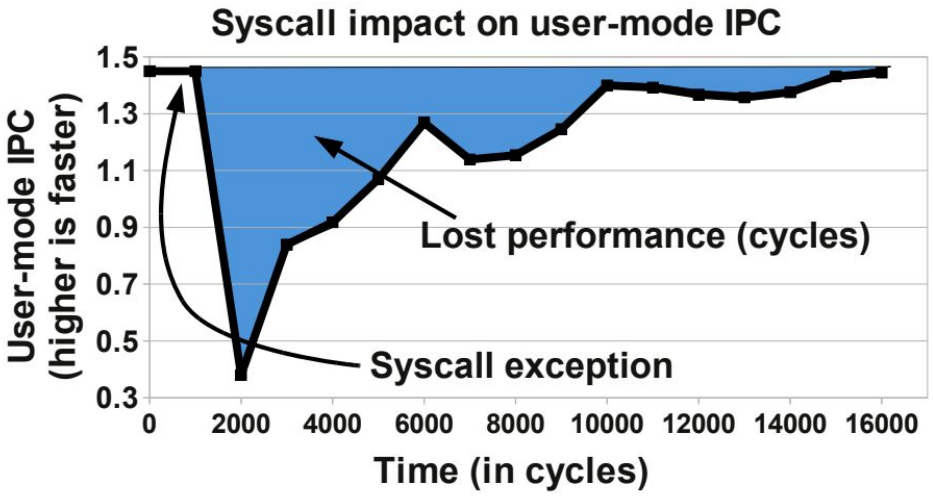
The problem with lock based synchronization

1. Locks are “reasonably” **fast when there is low/no contention**, (generally just an atomic instruction), the main overhead comes from blocking on the lock
2. Blocking on a lock is implemented using a **context switch, followed by insert on a queue** corresponding to the lock, and the thread is no longer runnable :(
3. When **lock is released, one thread wakes up** from the queue, and is runnable. When it wakes up, it is in the critical section, and has acquired the lock.

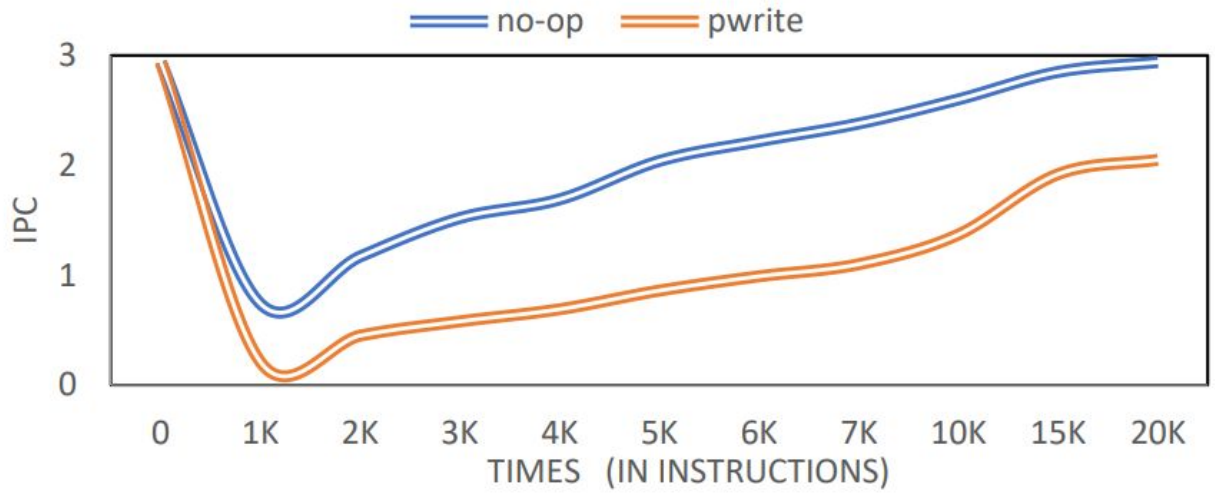
But…how bad is a context switch?

Userspace **IPC drops to around 0 right after syscall**, and takes **~20k instructions** to come back to normal

`Performance loss (Soares et al., 2010)`



`Performance loss (Zhou et al., 2023)`



People often prefer spinlocks because of this…

A naive lock-

```
int value = 0;
acquire () {
   while (test&set(value));
}

release () {
   value = 0;
}
```

So context switches (hence locks) can be pretty bad…

can we do better?

Lockless Programming! We'll cover this soon

But so far we have been taking something for granted

Cache Coherence

So far in multicore programming, we have **taken coherence of the cache for granted.**

Coherence implies that reads to a given memory location at a given time, from any core should return the same value.

1. When a core **writes to a memory location**,

   and **another core has it** in it's L1 cache, what happens?

2. Will the other core see the updated value or the incorrect one?

Cache Coherence: What is it?

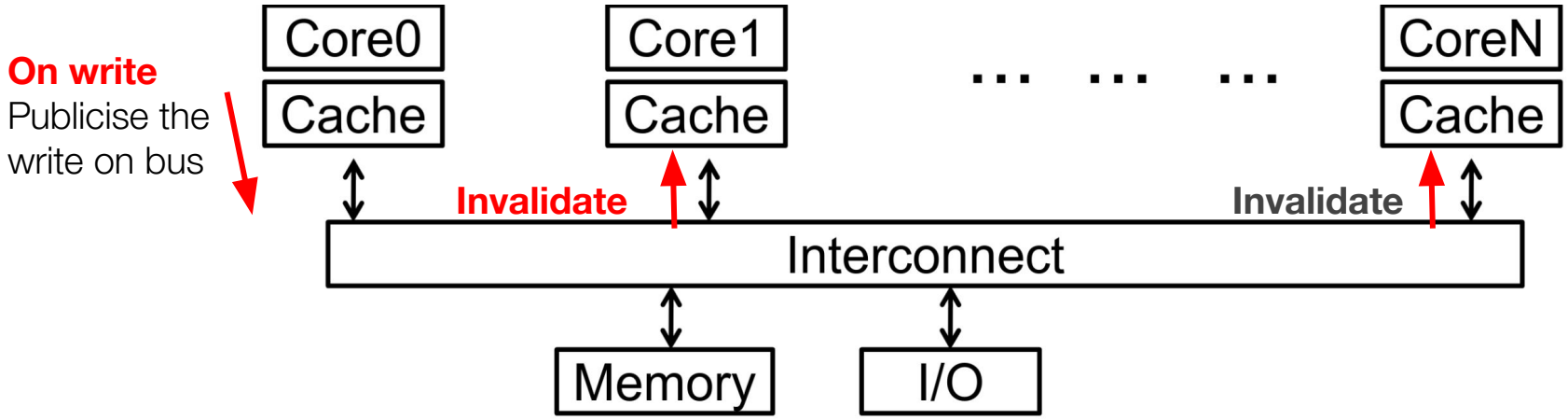
1. When a core writes to a memory location, and another core has it in it's L1 cache, what happens?

2. Will the other core see the updated value or the incorrect one?

Answer: Other core **sees the updated value**, because the hardware manages it :)

This is called **cache coherence**.

How does cache coherence work?



1. There are many protocols (VI, MSI, MESI), of different kinds (snooping based, directory based)
2. Caches maintain metadata of whether or not their entries are valid, and **on write** to a location, other **cores caching that location will mark their entry as invalid**

Cache coherence example: Busy wait

A naive busy-wait lock-

```
int value = 0;
acquire () {
   while (test&set(value));
}

release () {
   value = 0;
}
```

Can we do a better busy-wait lock for multiprocessing?

But what's the issue? Cache coherence

1. test&set instruction **always causes a write**, because of the `set` part
2. Due to the write **cache invalidation takes place** in other cores
3. Doing so in a **tight-loop floods the bus with invalidation**, this is called an invalidation storm
4. Can we do better? i.e. without doing a write every single time, can we write a busy wait lock?

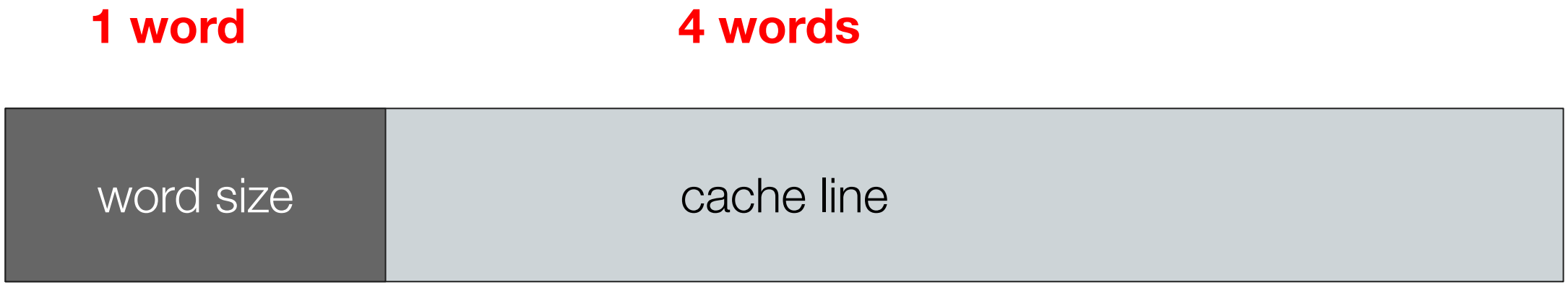
Improved code: Busy wait

```
int lock = 0; // Free
acquire() {
   do {
      while(lock); // Wait until might be free
   } while(test&set(&lock)); // exit if get lock
}
release() {
   lock = 0;
}
```

Can you see why this is better?

False sharing in cache coherence protocols

In the modern processors a **cache-line (the smallest unit in which the processor writes to cache) is larger than the word size**



False sharing in cache coherence protocols

1. In the busy-wait example, a lot of (avoidable) writes to a value take place
2. In the modern processors a **cache-line (the smallest unit in which the processor writes to cache) is larger than the word size**
3. As a consequence of this **writing to one word of the cache-line invalidates the whole cache-line**
4. Note, this invalidation is caused even when a value is not shared! Hence it is termed false sharing, and should be avoided

Avoiding synchronization/cache coherence overhead is hard…

but not impossible

Per-core sharding/shared-nothing architecture

1. Per core sharding schedules **one thread per core**
2. Communication takes place using **message passing** instead of shared memory by default
3. Because there is **no shared memory**

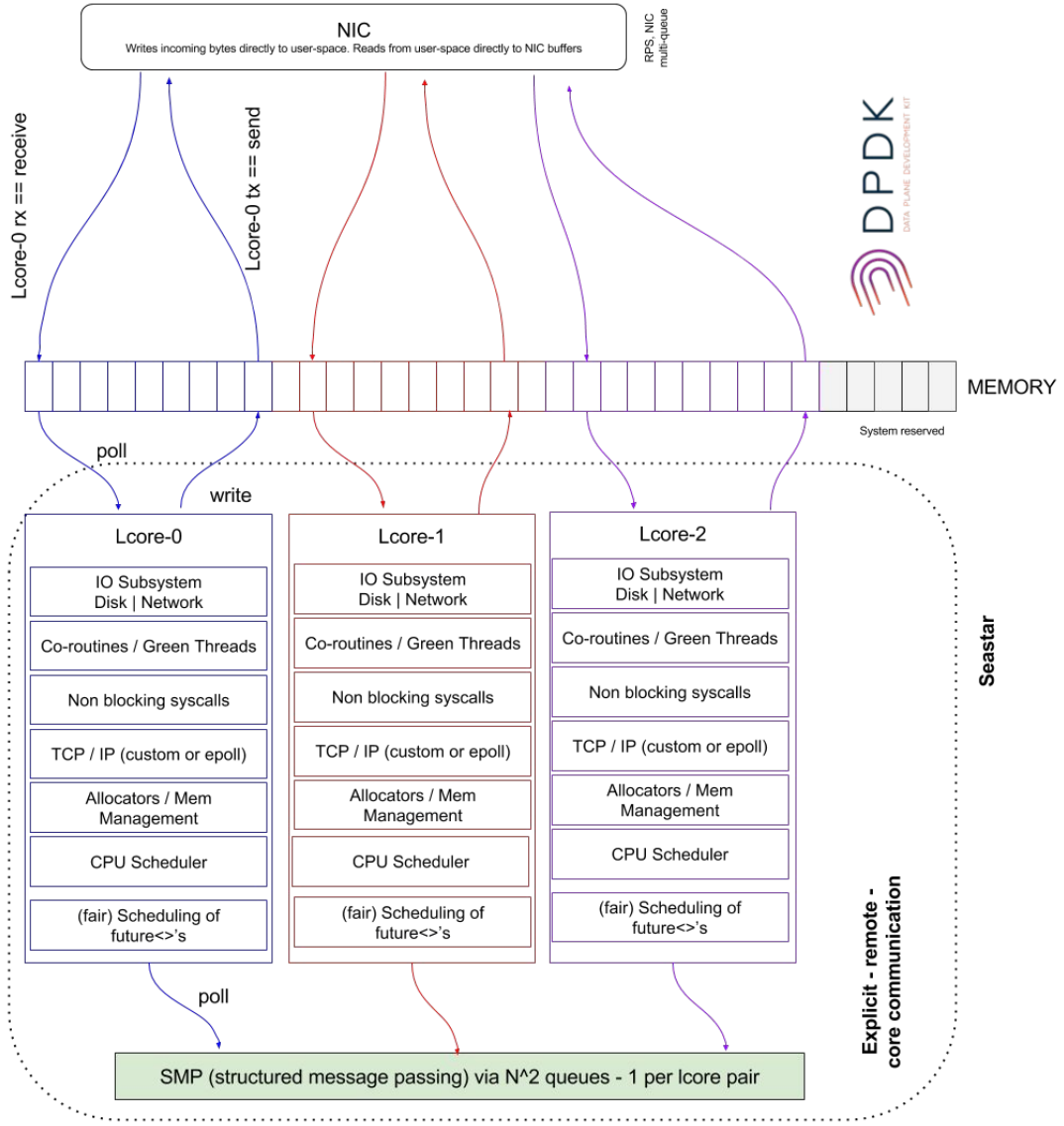
   …there is **no need for locking**, and **no cache coherence overhead**

   Yay!

Some issues to note

1. Implementation is no longer simple, and **often requires async programming**
2. Such optimizations are not necessary in most cases
3. Even when performance is important, **IO and other factors become a bottleneck**
4. To make sure that whole system is performant (no IO bottlenecks, etc) it's generally **used with techniques like kernel bypass**

A great example: Seastar (Just for reference)



Your view into any application starts with a `seastar::distributed<T>` type. This means a copy of the T is thread local. Following optimizations are done at type level:

- Small type optimizations (although `seastar::small_set<T>` and `seastar::small_map<K,V>` are missing).
- Non thread safe non-polymorphic shared pointer (local to core) via `seastar::lw_shared_ptr<T>`
- Non-thread safe polymorphic shared pointer (local to core) via `seastar::shared_ptr`
- String with small type optimizations nor atomics like the libc++
- Move only bag-o-bytes
- Circular buffers
- Linux DAIO