Nearly Unfathomable Morass of Arcana (NUMA)

Fedor G Pikus

Technical Fellow

SIEMENS

Non-Uniform Memory Architecture (NUMA)

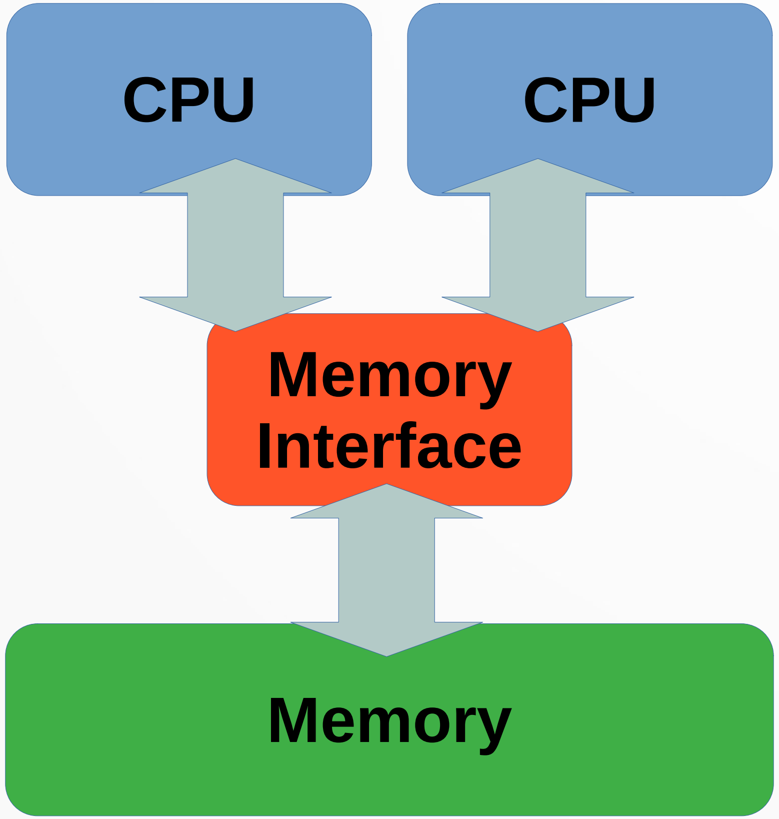
Fedor G Pikus

Technical Fellow

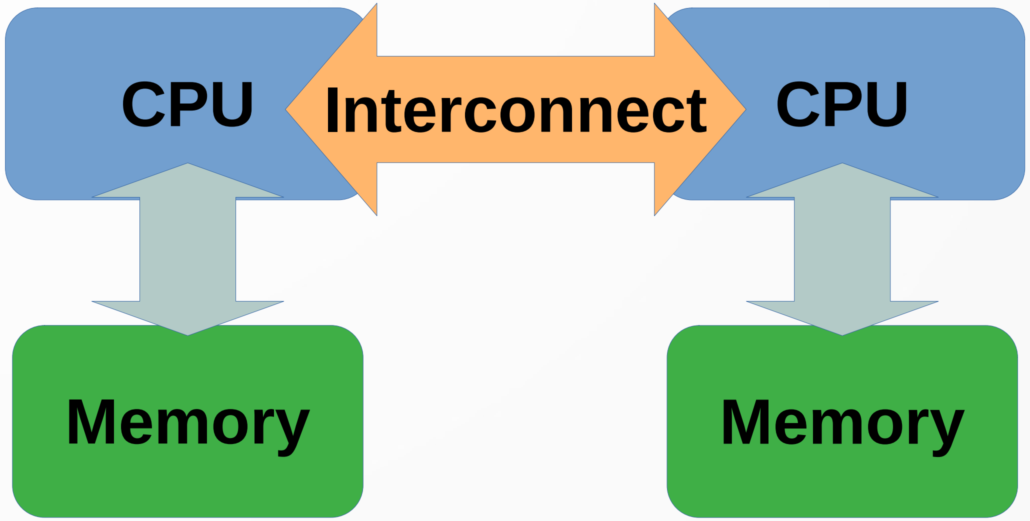
SIEMENS

Short version – slide 1:
This is NUMA

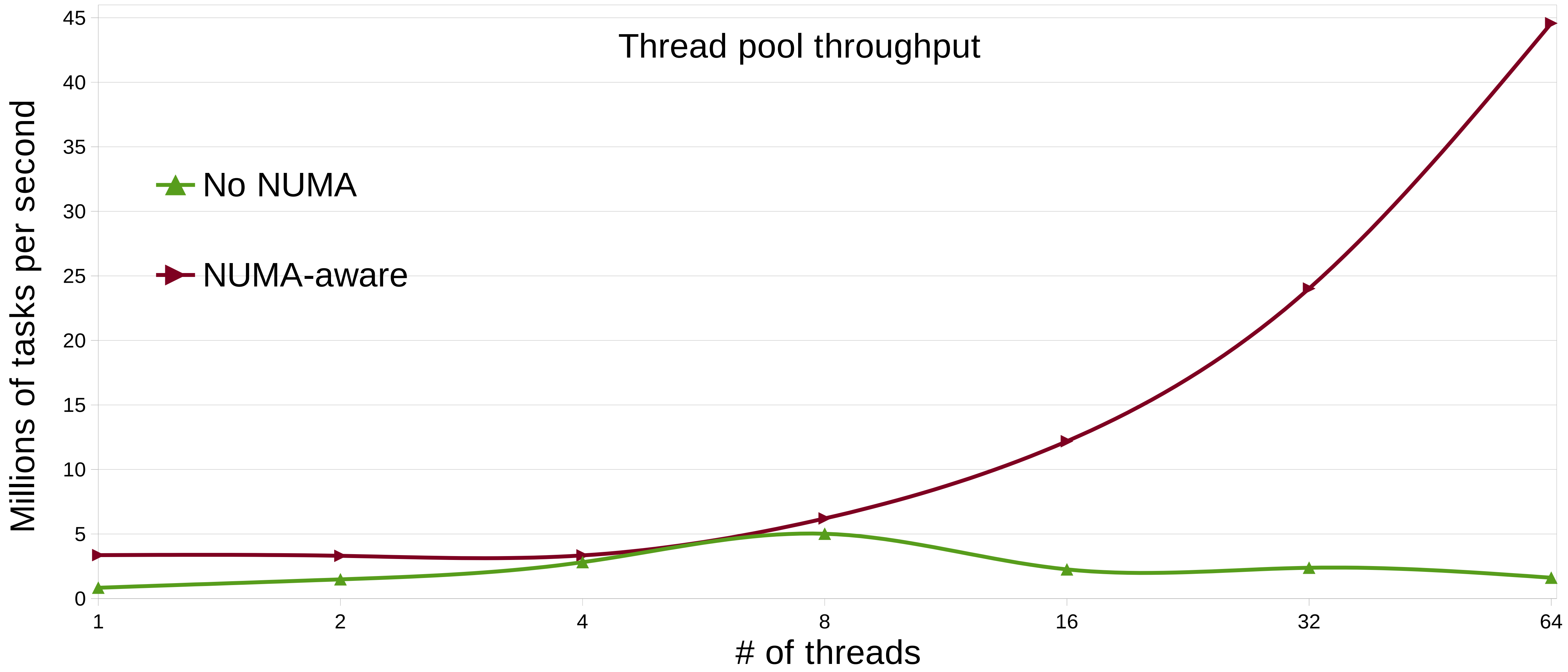
Computers went from this ...



... to this



Short version – slide 2:
This happens if you ignore NUMA



We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

We Will Learn:

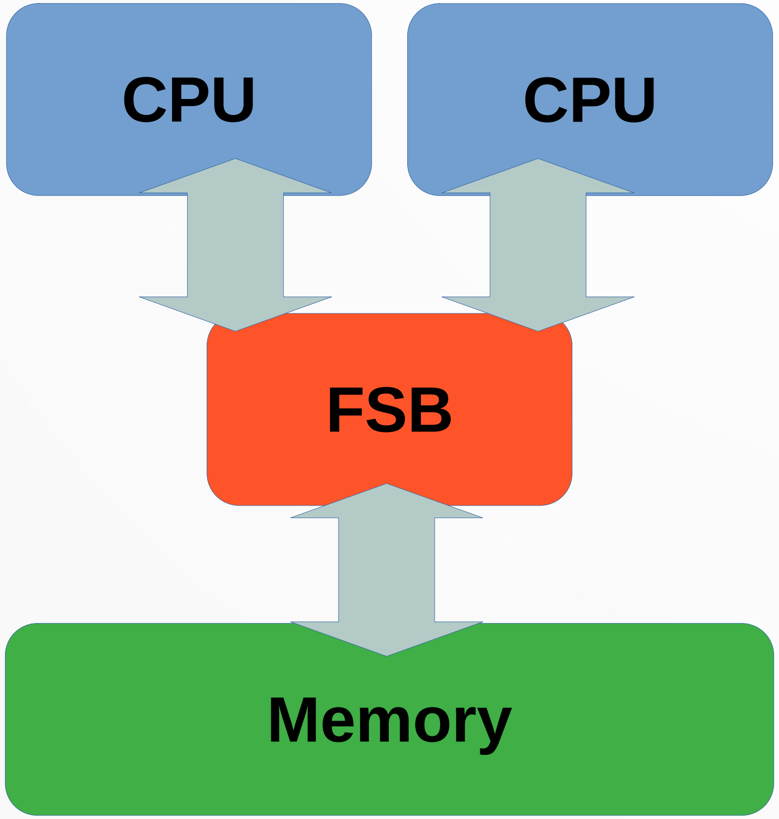
- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

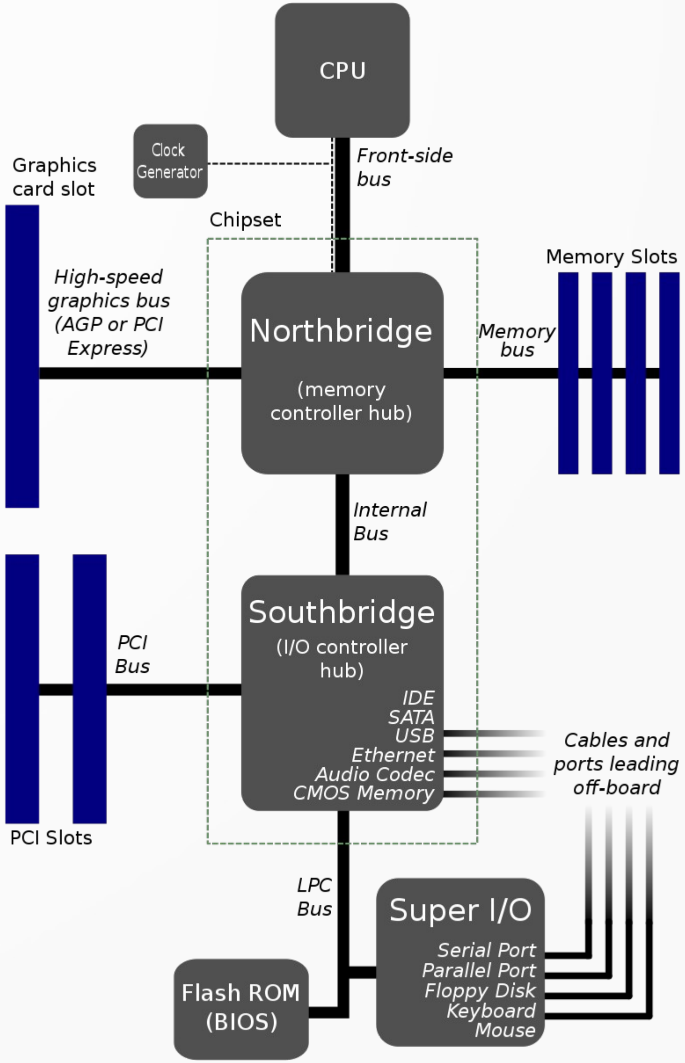
What is NUMA

- Non-Uniform Memory Architecture
- What is Uniform Memory Architecture?

Uniform Memory Architecture

- All processors have the same access to all memory
  - X86 Front-Side Bus (FSB)



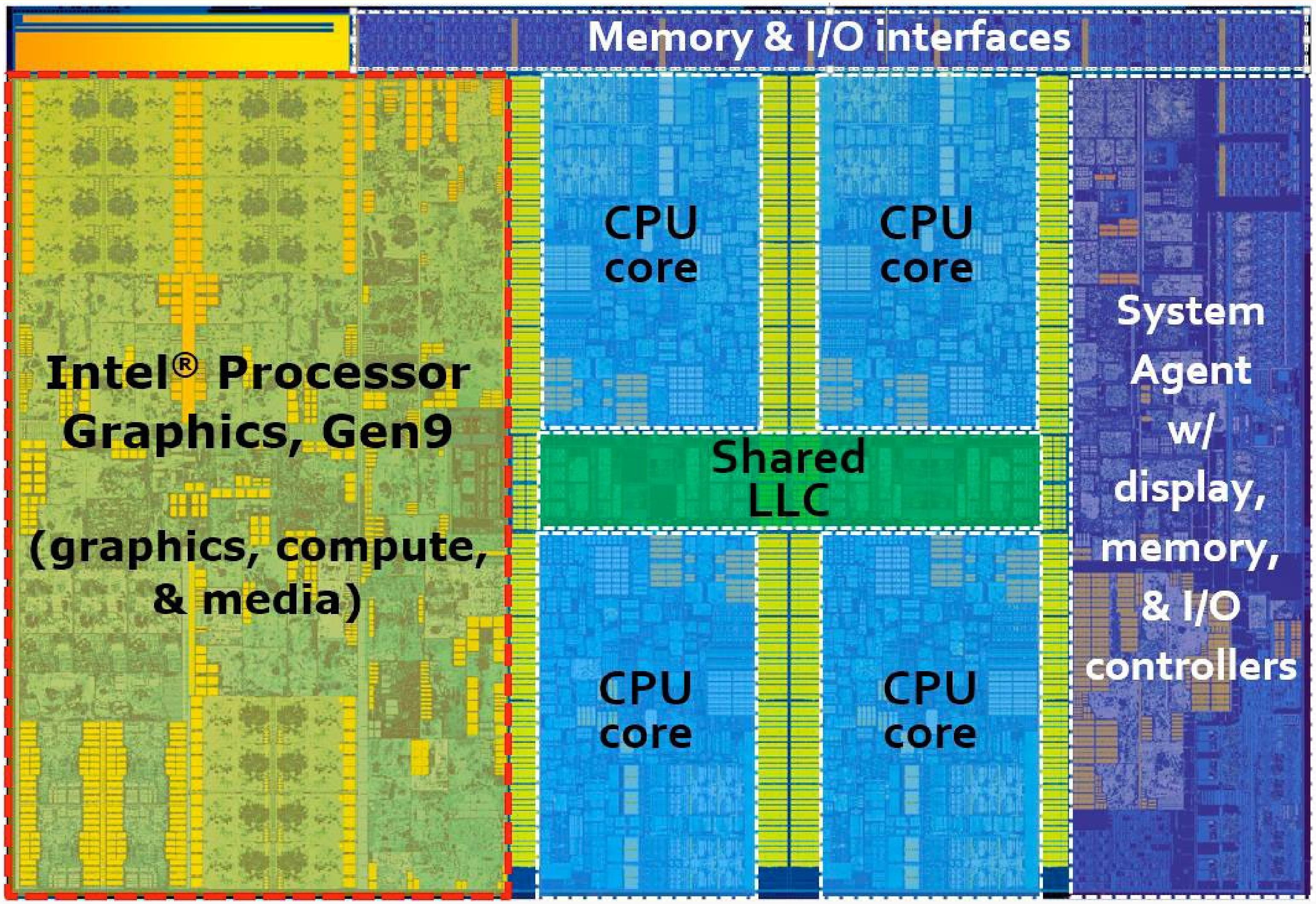


Uniform Memory Architecture

- UMA is not just for ancient hardware

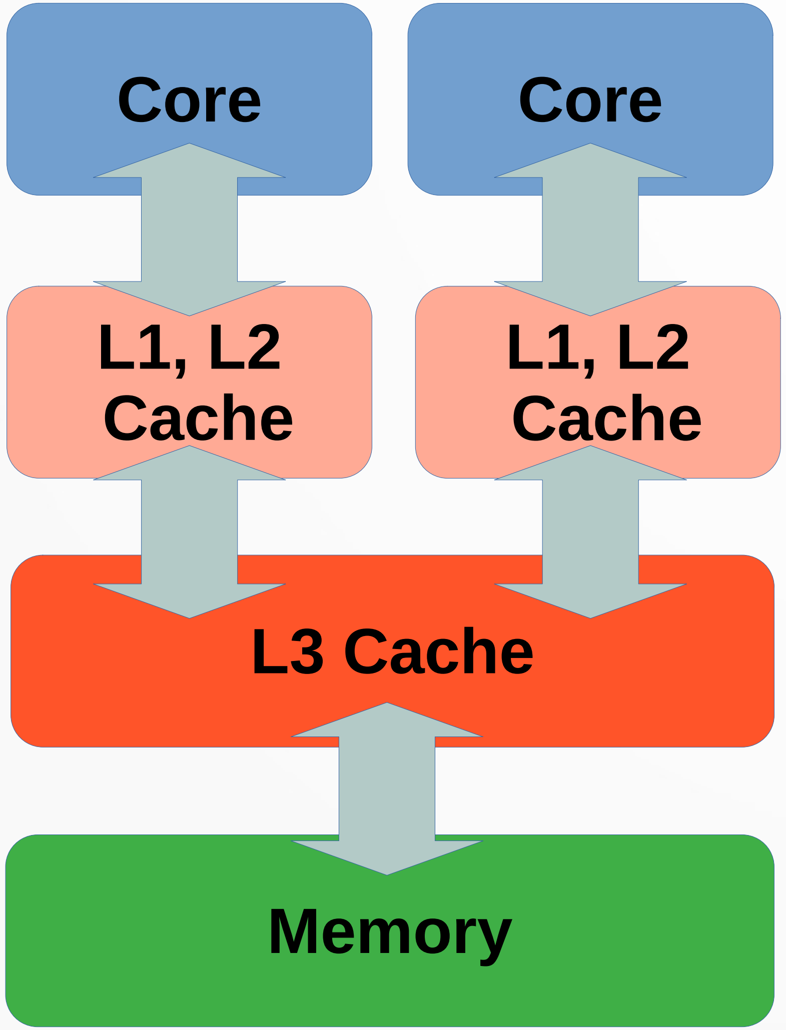
Uniform Memory Architecture

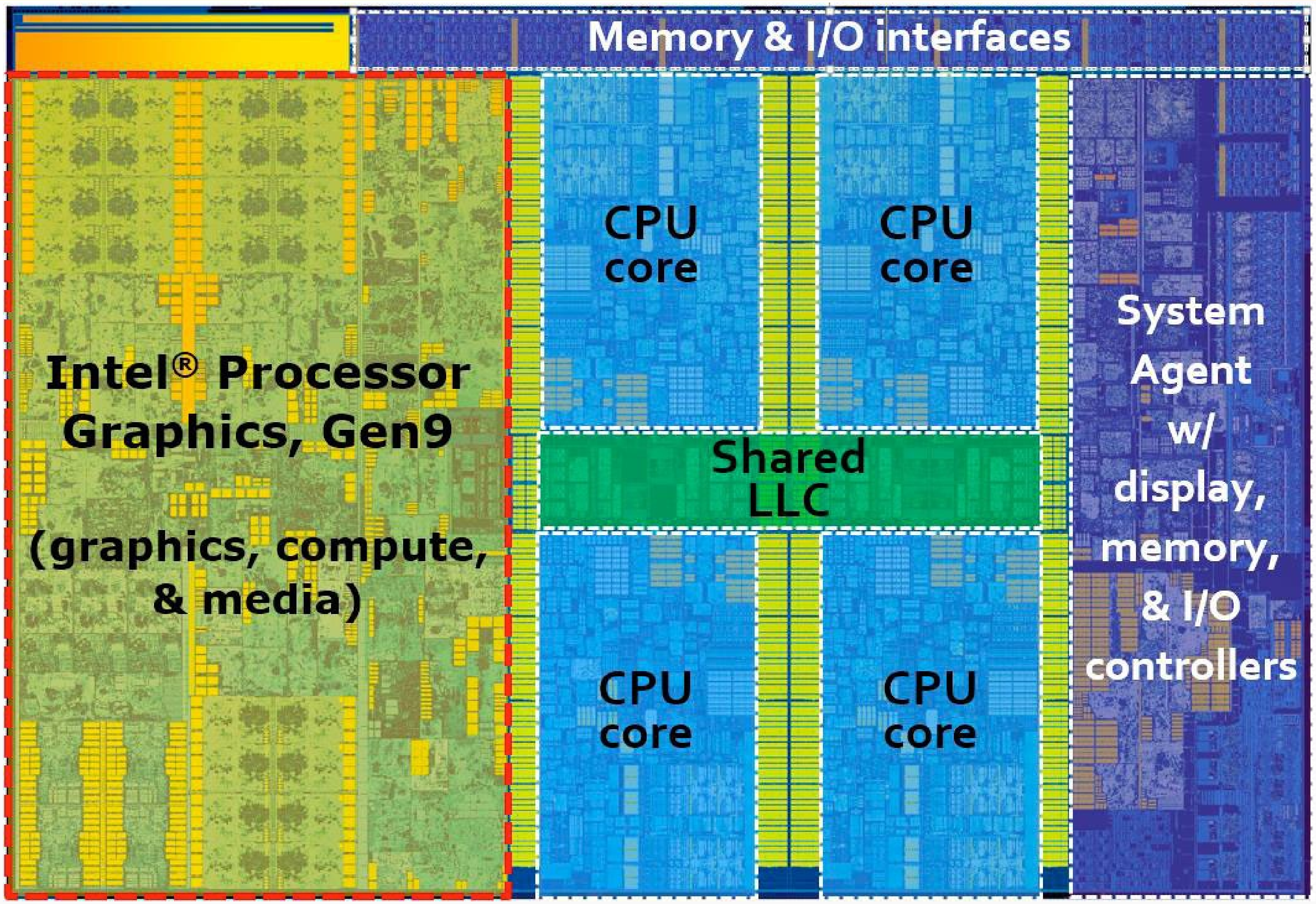
- UMA is not just for ancient hardware
- Intel SkyLake



Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

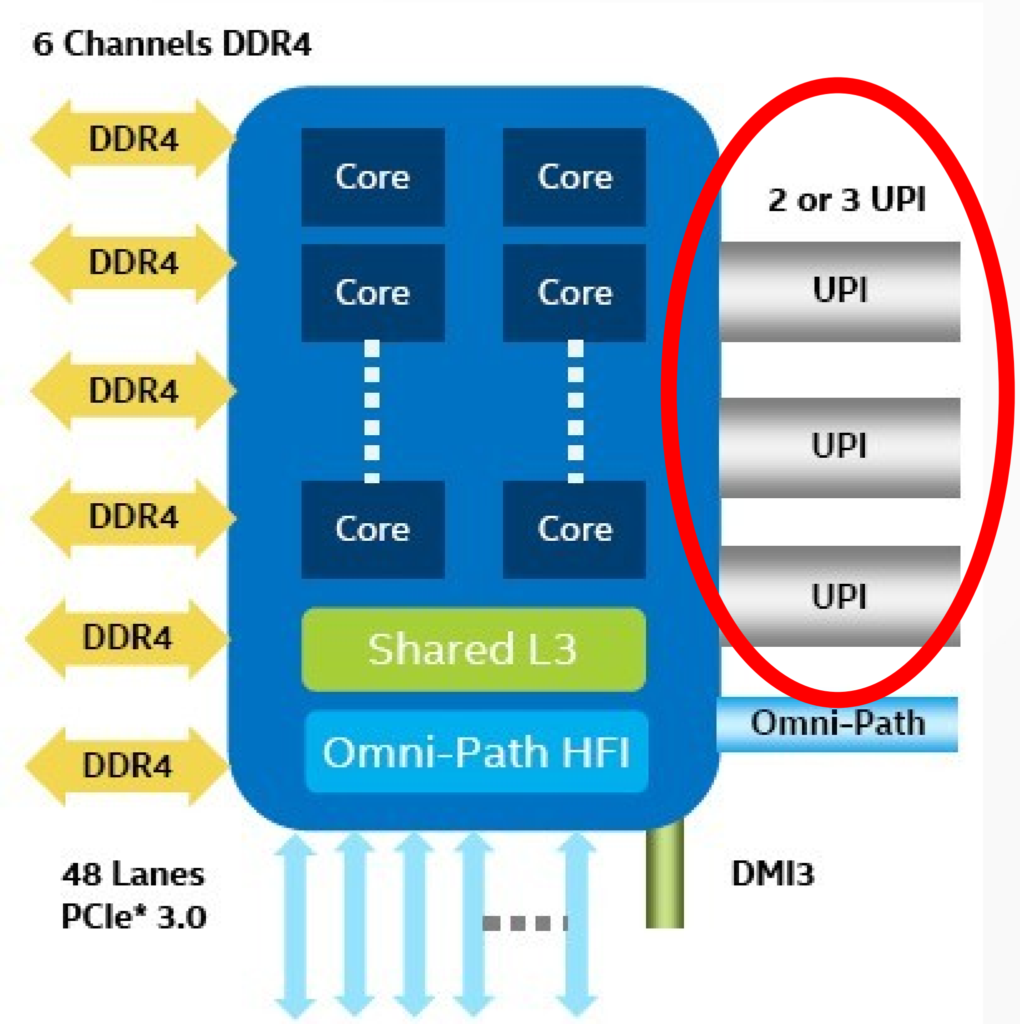
Uniform Memory Architecture




Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

Non-Uniform Memory Architecture

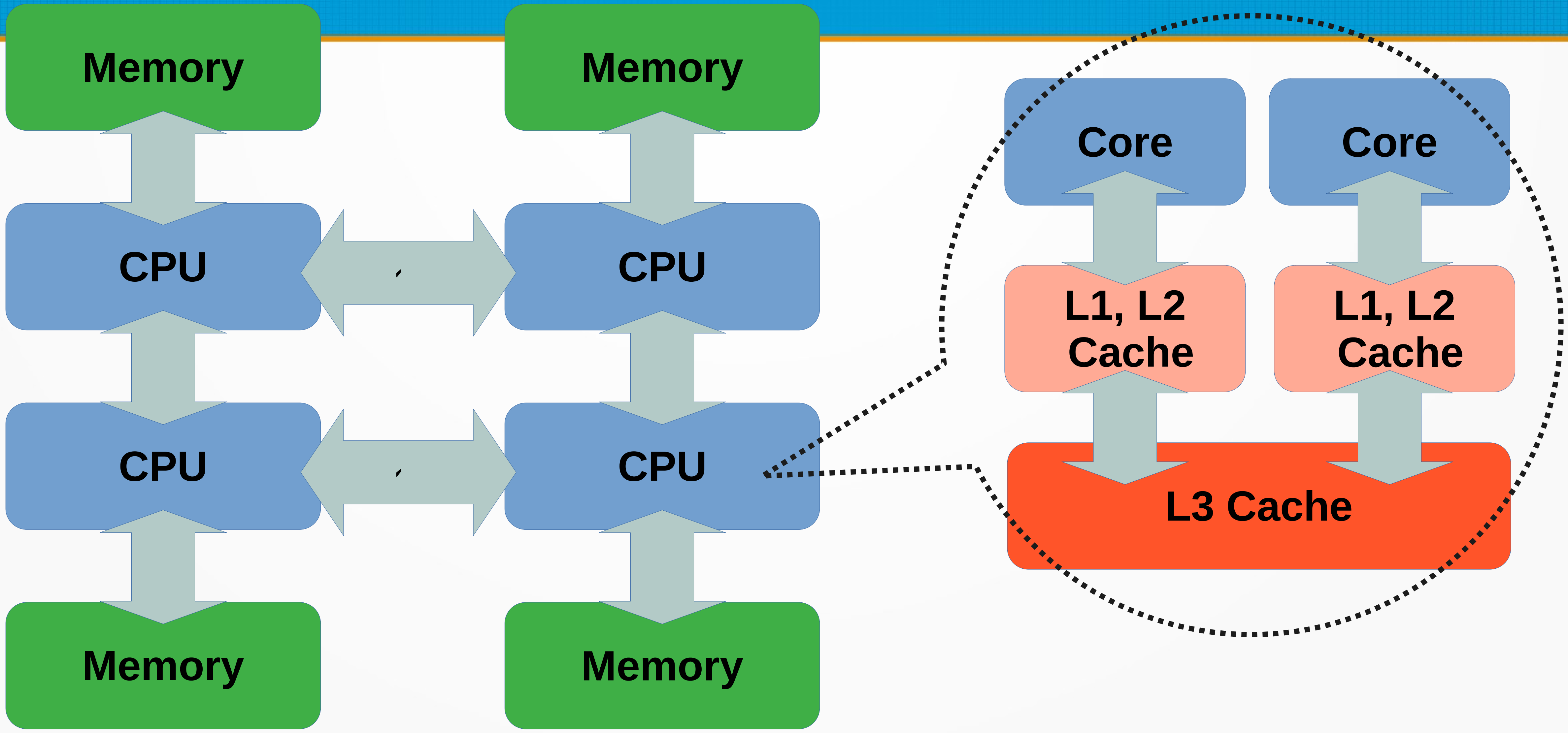
- Aren’t all systems with modern processors NUMA?
  - They can be…
- Also Intel SkyLake



What is NUMA

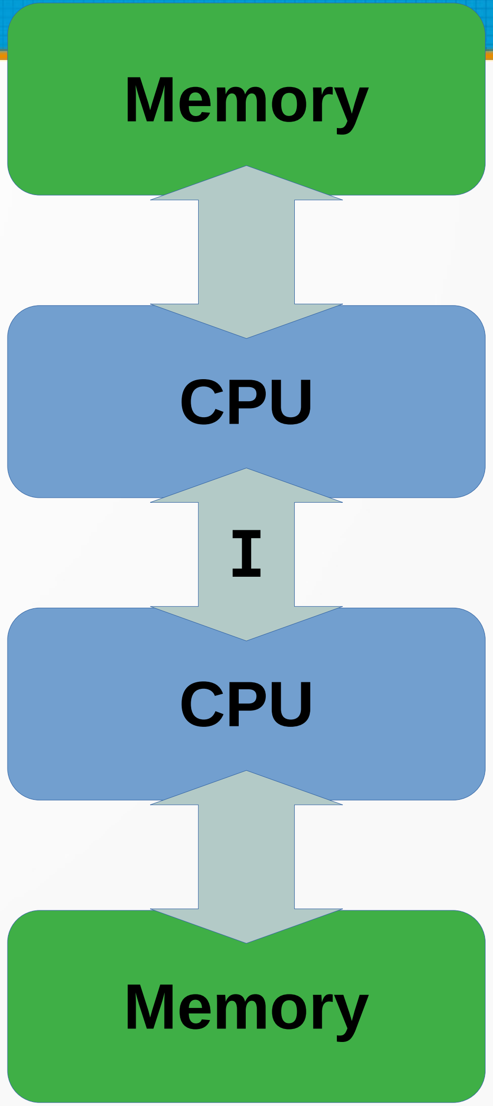
- Non-Uniform Memory Architecture
- What is Uniform Memory Architecture?
- What is Non-Uniform Memory Architecture?

Non-Uniform Memory Architecture



Brief history of NUMA

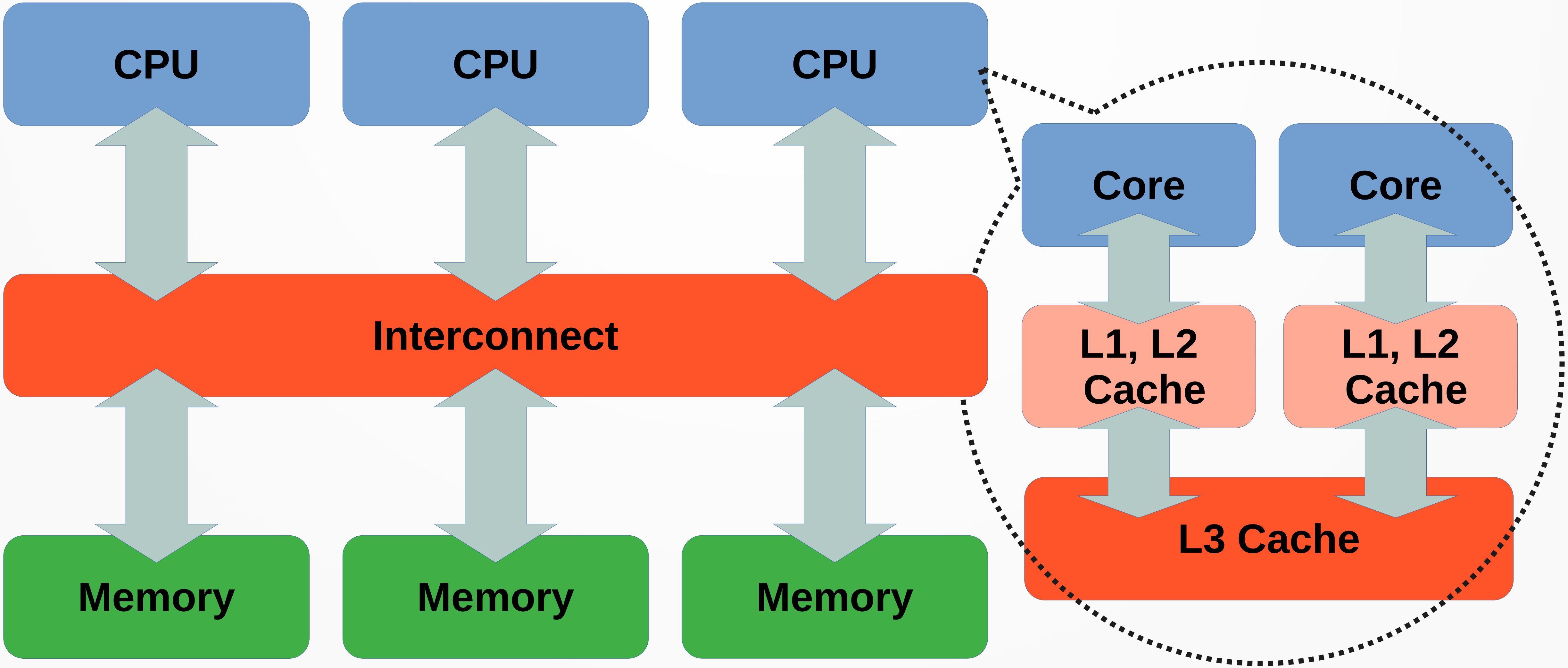
- AMD Opteron (2003) – Hyperconnect
- Intel Nehalem (2007) – Quick Path Interconnect (QPI)
  - Intel SkyLake (2017) – UltraPath Interconnect (UPI)
- X86 systems with integrated memory controllers
  - Replaces chipset controllers (North and South Bridge)



Brief history of NUMA

- 199? - Honeywell, HP, Cray/SGI, etc
- AMD Opteron (2003) – Hyperconnect
- Intel Nehalem (2007) – QPI

Not all multi-socket systems are NUMA



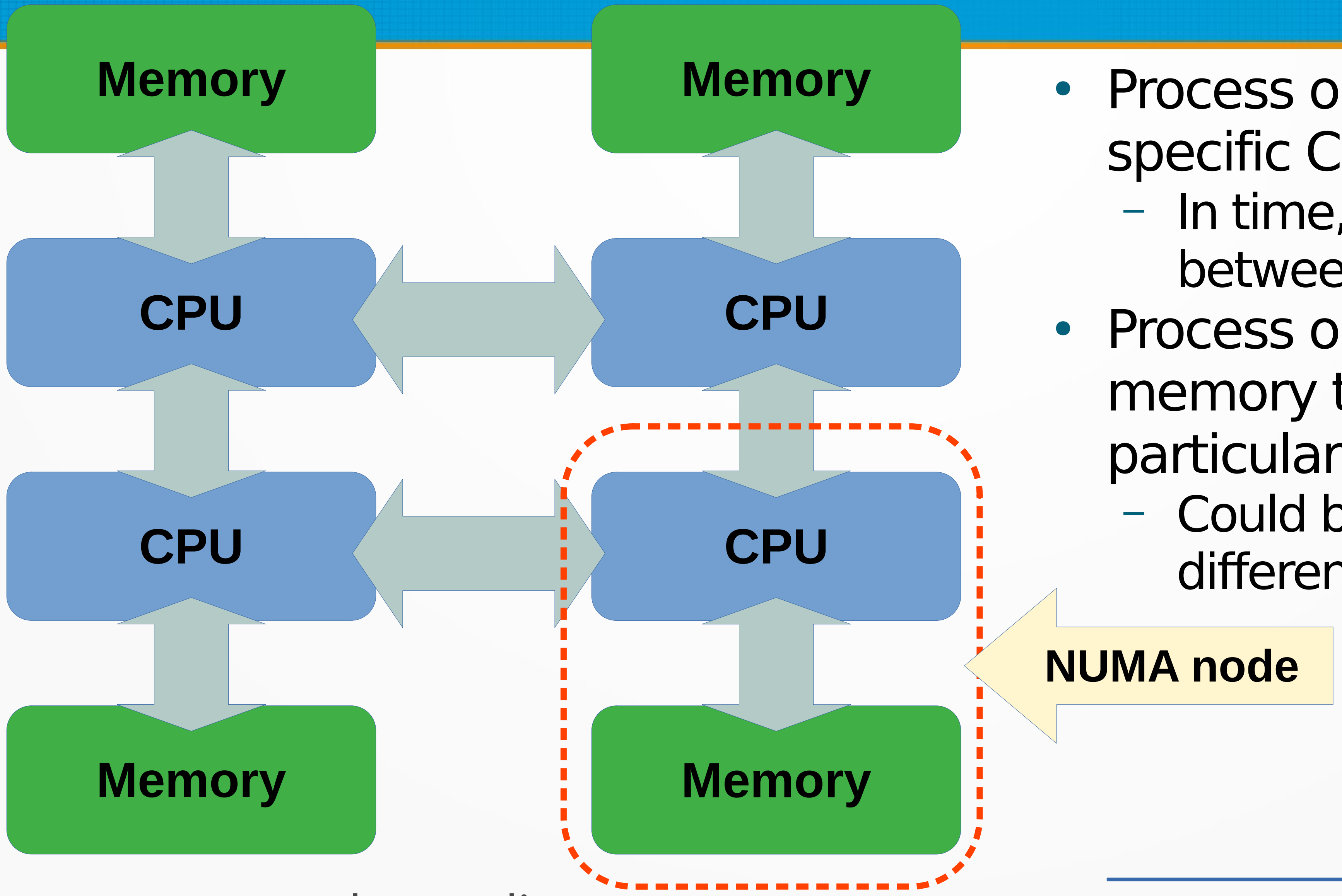
Why NUMA? Why now?

- Systems with multiple CPUs became common
  - Not just for users with unlimited $$$
- High-speed symmetric interconnect is expensive
  - Not just in $$$s but in power, heat, and complexity
- On-chip memory controllers favor NUMA
  - Each CPU accesses its own memory
- Many large systems avoided the issue by using explicit message passing
  - No global address space – no NUMA
- Not just exotic systems anymore
  - Intel E5-2640 (Haswell) 16 cores * 2 sockets – 2014, under $1000 per CPU

We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

Non-Uniform Memory Architecture



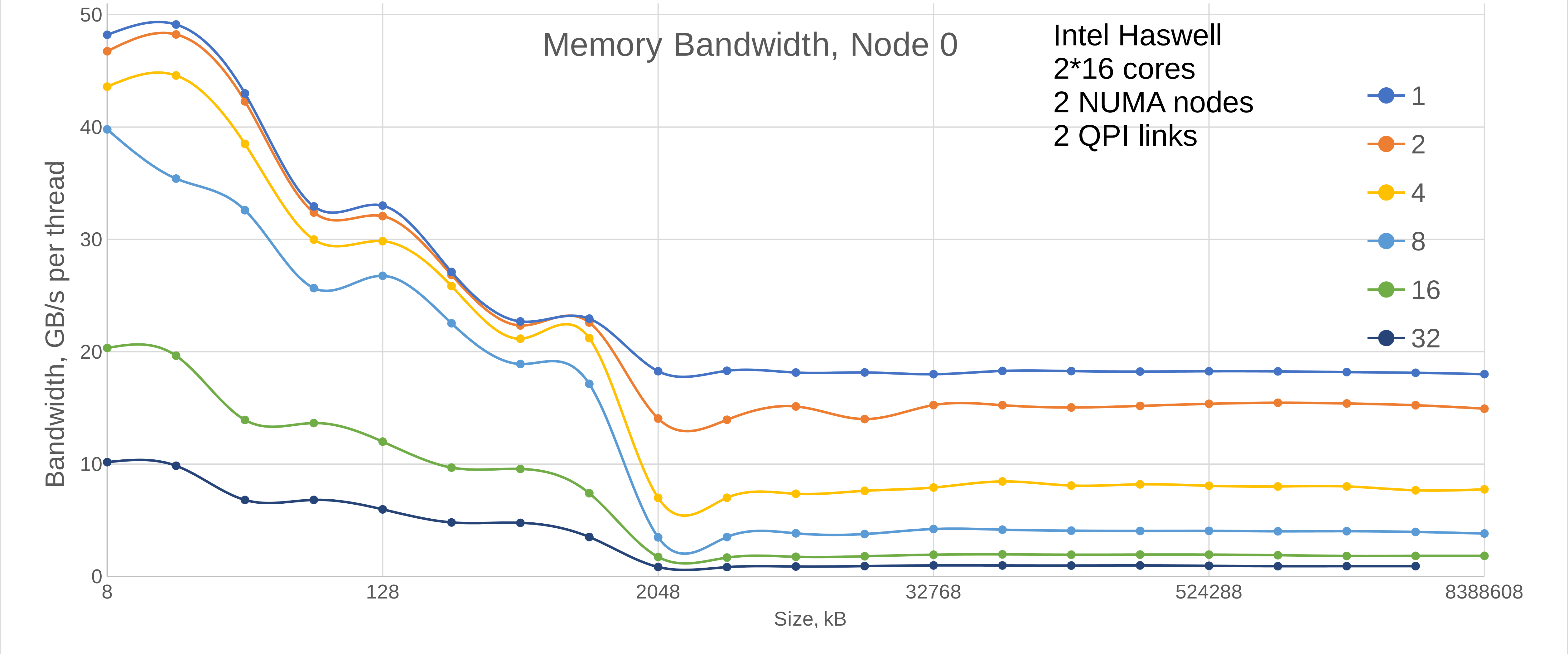
- Process or thread runs on a specific CPU at any time
  - In time, processes can move between CPUs and nodes
- Process or thread accesses memory that resides on a particular node
  - Could be the same node or different

SIEMENS

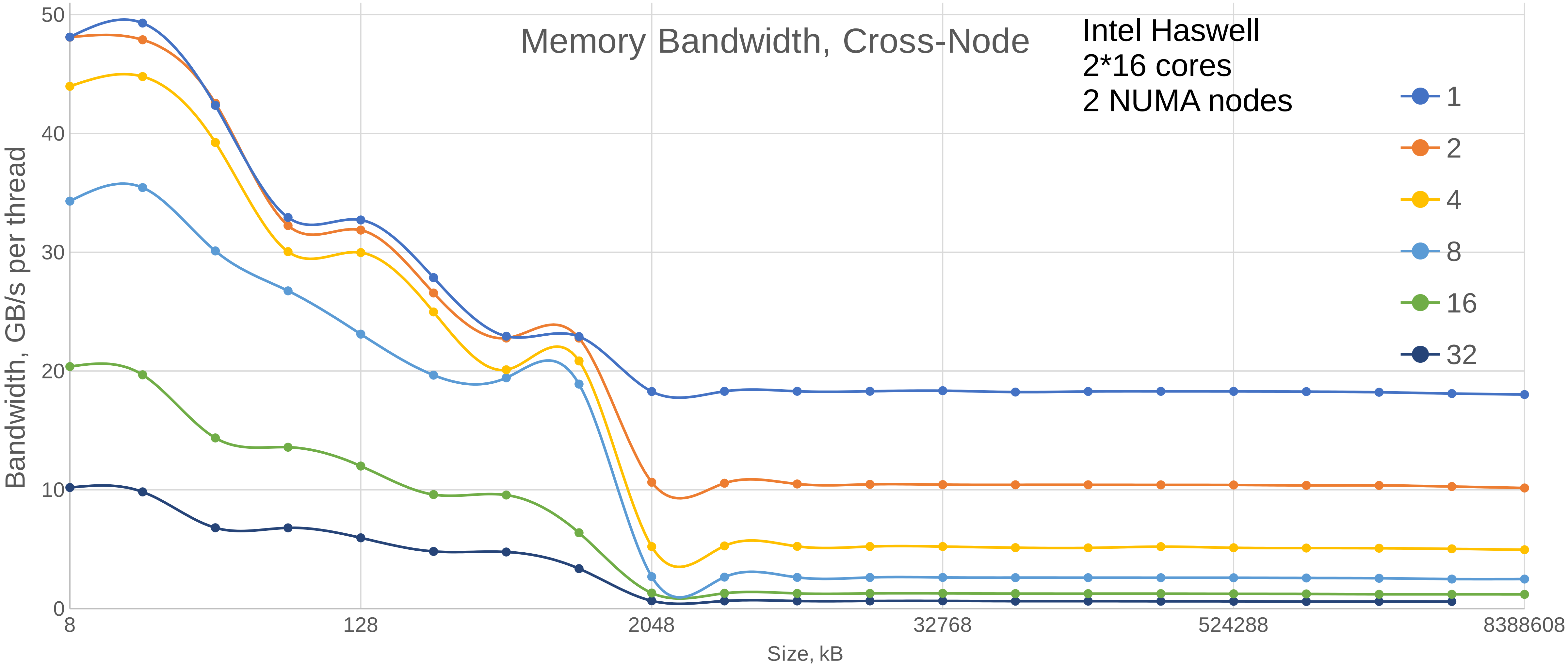
Performance implications of NUMA

- What is NUMA?
- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node’s CPU must be involved

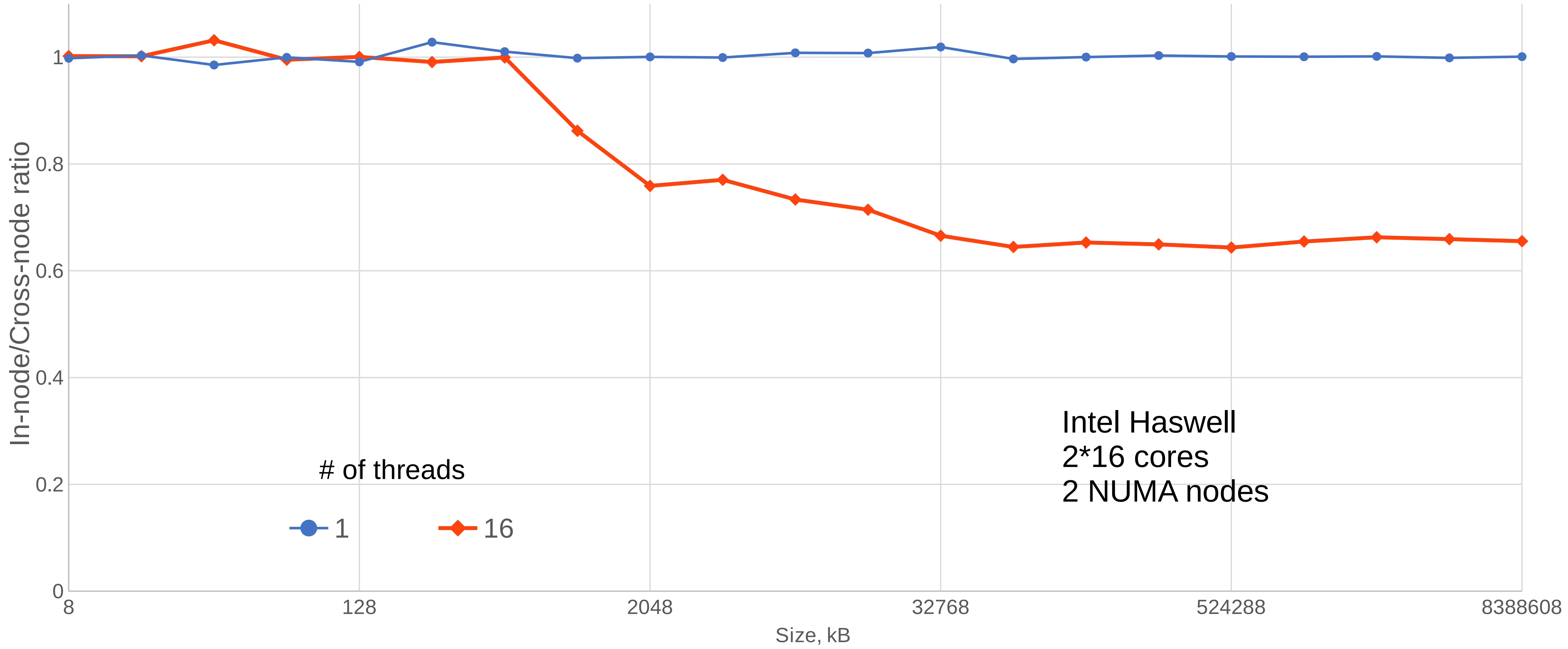
Memory Bandwidth in NUMA systems



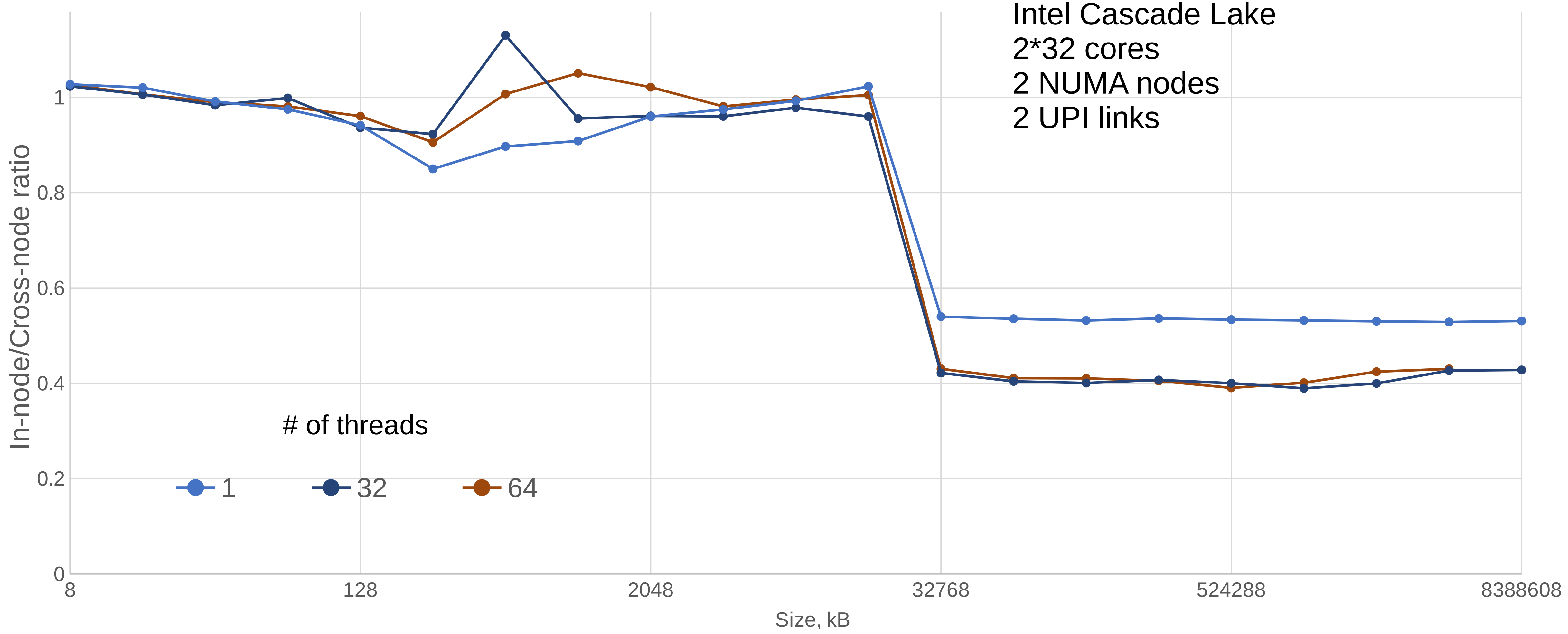
Memory Bandwidth in NUMA systems



Memory Bandwidth in NUMA systems



Memory Bandwidth in (newer) NUMA systems



Intel Cascade Lake
2*32 cores
2 NUMA nodes
2 UPI links

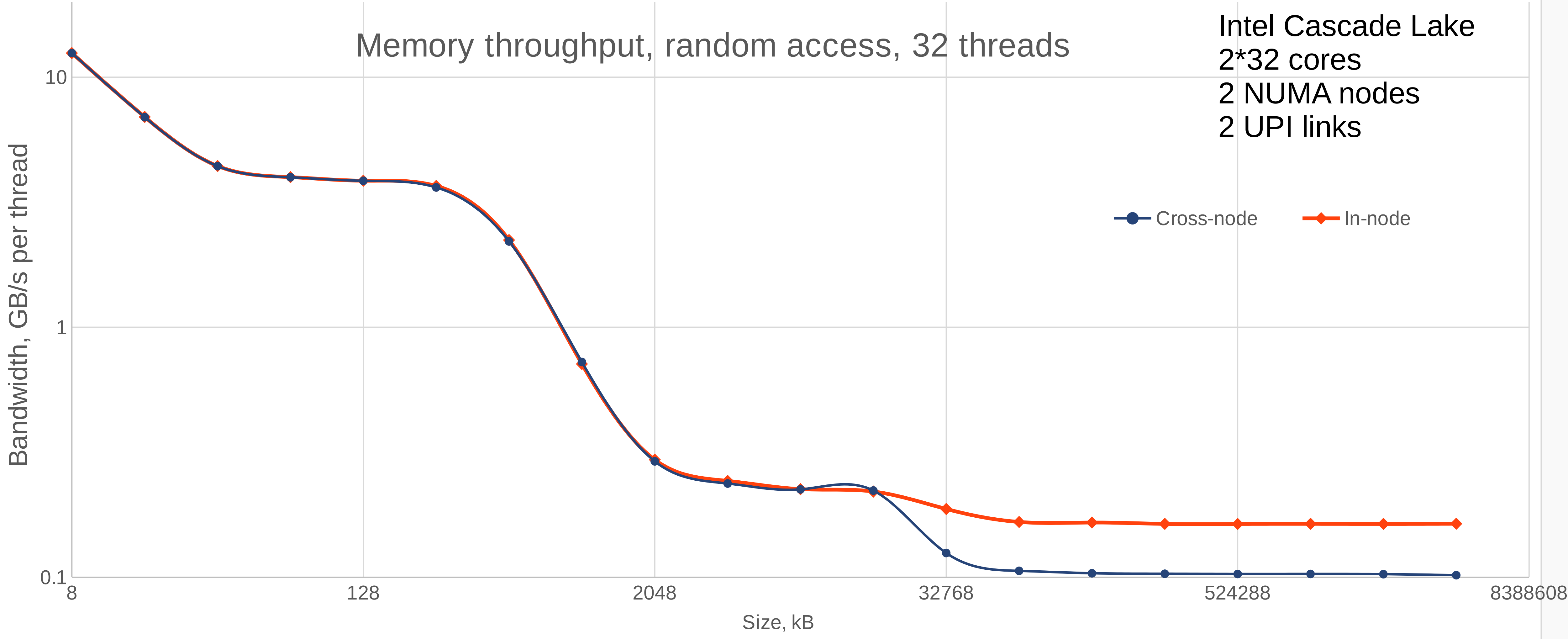
We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

Performance implications of NUMA

- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node’s CPU must be involved
- Peak bandwidth is reduced by 40% to 60% for cross-node access
  - This is remarkably universal for x86 hardware from 2014 to 2022
  - Some systems show degradation for even 1 thread, some do not
  - Newer systems have higher bandwidth but larger penalty
- Cache bandwidth is not affected (without data sharing)
- Memory bandwidth is measured by sequential data transfers

Memory Throughput in (newer) NUMA systems



Intel Cascade Lake
2*32 cores
2 NUMA nodes
2 UPI links

Performance implications of NUMA

- Accessing local memory is different from accessing non-local memory
- To access memory from another node, that node's CPU must be involved
- Peak bandwidth is reduced by 40% to 60% for cross-node access
  - This is remarkably universal for x86 hardware from 2014 to 2022
  - Some systems show degradation for even 1 thread, some do not
  - Newer systems have higher bandwidth but larger penalty
- Cache bandwidth is not affected (without data sharing)
- Memory bandwidth is measured by sequential data transfers
- Random access throughput is much smaller but cross-node penalty is similar
- All measurements are for independent memory access by each thread
  - Each thread allocates and writes its own memory

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is different from accessing non-local memory
- ~50% bandwidth penalty for remote node access (less if fits in cache)

(Real) Performance implications of NUMA

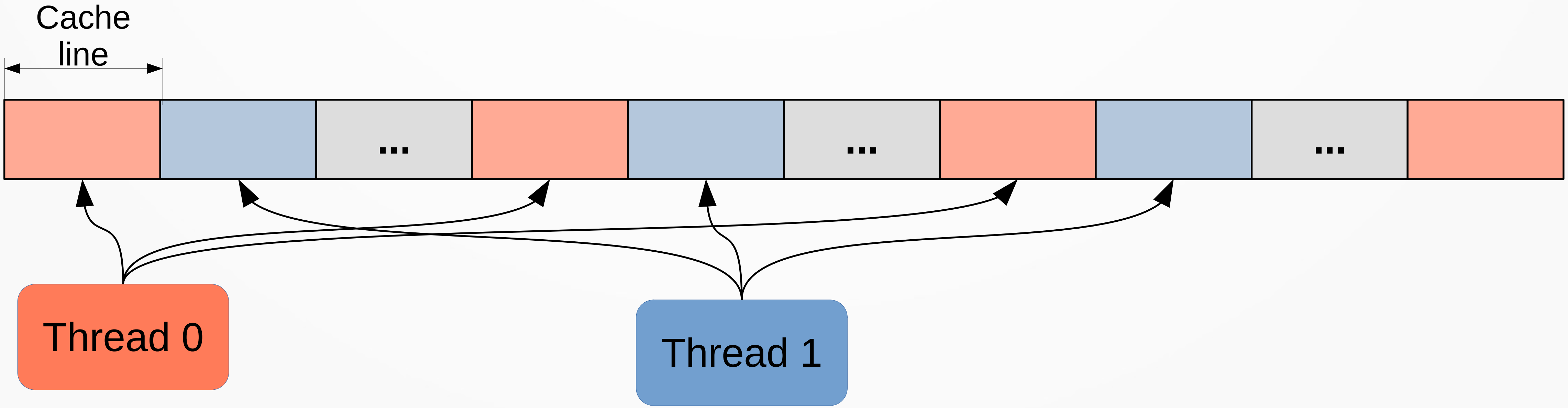
- 50% memory throughput reduction is not too bad
  - May be observable in memory-bound programs
- All measurements are for independent memory access by each thread
  - What if all threads access the same data?

(Real) Performance implications of NUMA

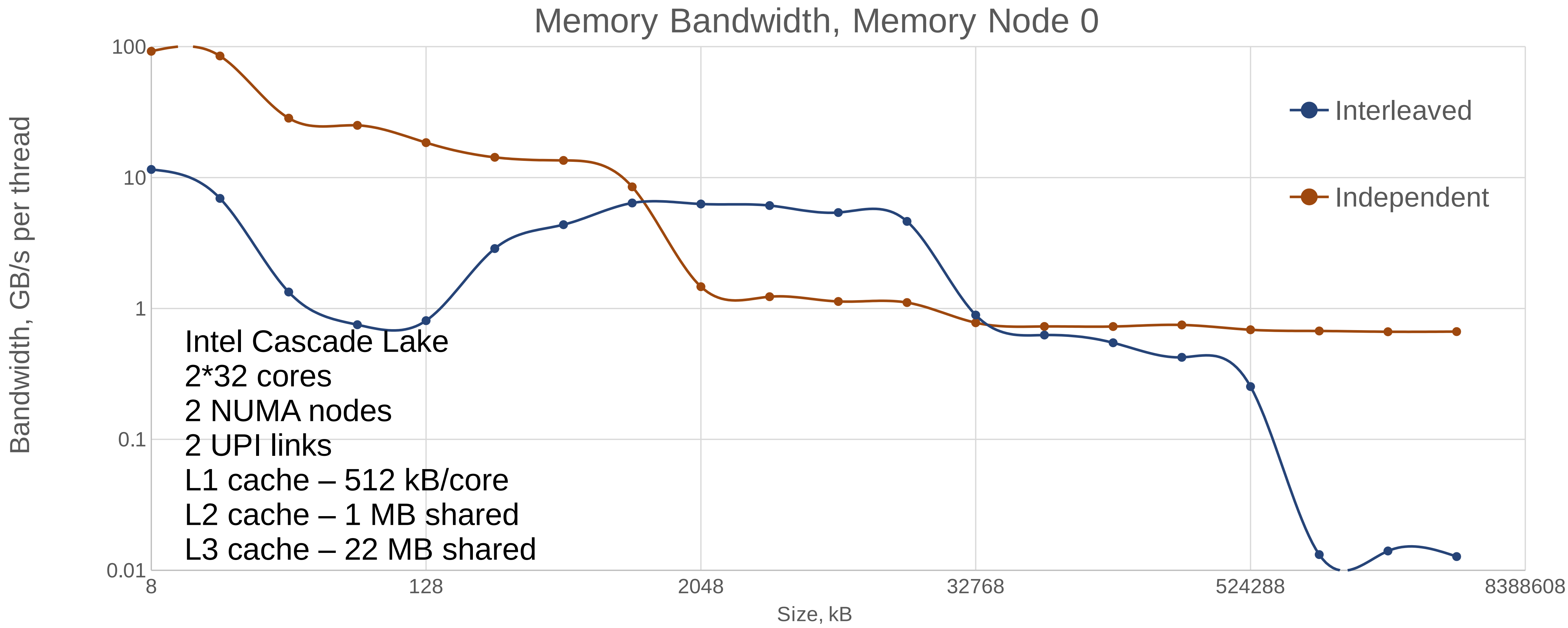
- 50% memory throughput reduction is not too bad
  - May be observable in memory-bound programs
- All measurements are for independent memory access by each thread
  - What if all threads access the same data? (for different values of “same”)
- Accessing the same address (true sharing) requires locks or atomic access
  - This is slow even between CPU cores within the same node
- Accessing different addresses within the same cache line (false sharing)
  - All the pain of sharing without the benefits (don’t do it)
- Accessing different cache lines
  - This is usually as fast as any other memory accesses

(Real) Performance implications of NUMA

- Test: all data resides on memory node 0
- 64 threads run on 64 cores (1 thread per core, all nodes)
- No cache lines shared between threads
- Thread data is interleaved and accessed sequentially (with stride)



(Practical) Memory Bandwidth in NUMA systems



Intel Cascade Lake
2*32 cores
2 NUMA nodes
2 UPI links
L1 cache – 512 kB/core
L2 cache – 1 MB shared
L3 cache – 22 MB shared

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory and cache locality are more important with NUMA
- Concurrent processing of data in close proximity may be slower

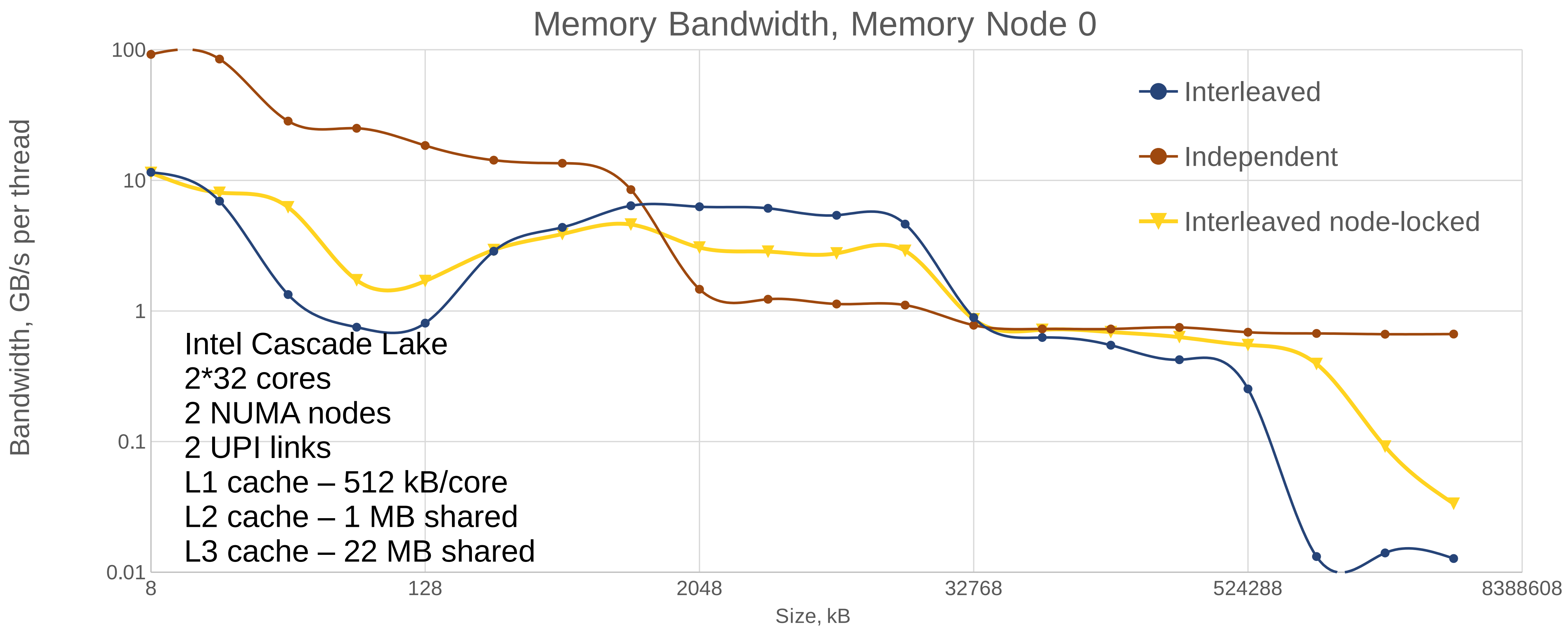
We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

(Real) Performance implications of NUMA

- Test: all data resides on memory node 0
- 64 threads run on 64 cores (1 thread per core, all nodes)
- No cache lines shared between threads
- Thread data is interleaved and accessed sequentially (with stride)
- Results depend strongly on timing of concurrent accesses

(Practical) Memory Bandwidth in NUMA systems



Intel Cascade Lake
2*32 cores
2 NUMA nodes
2 UPI links
L1 cache – 512 kB/core
L2 cache – 1 MB shared
L3 cache – 22 MB shared

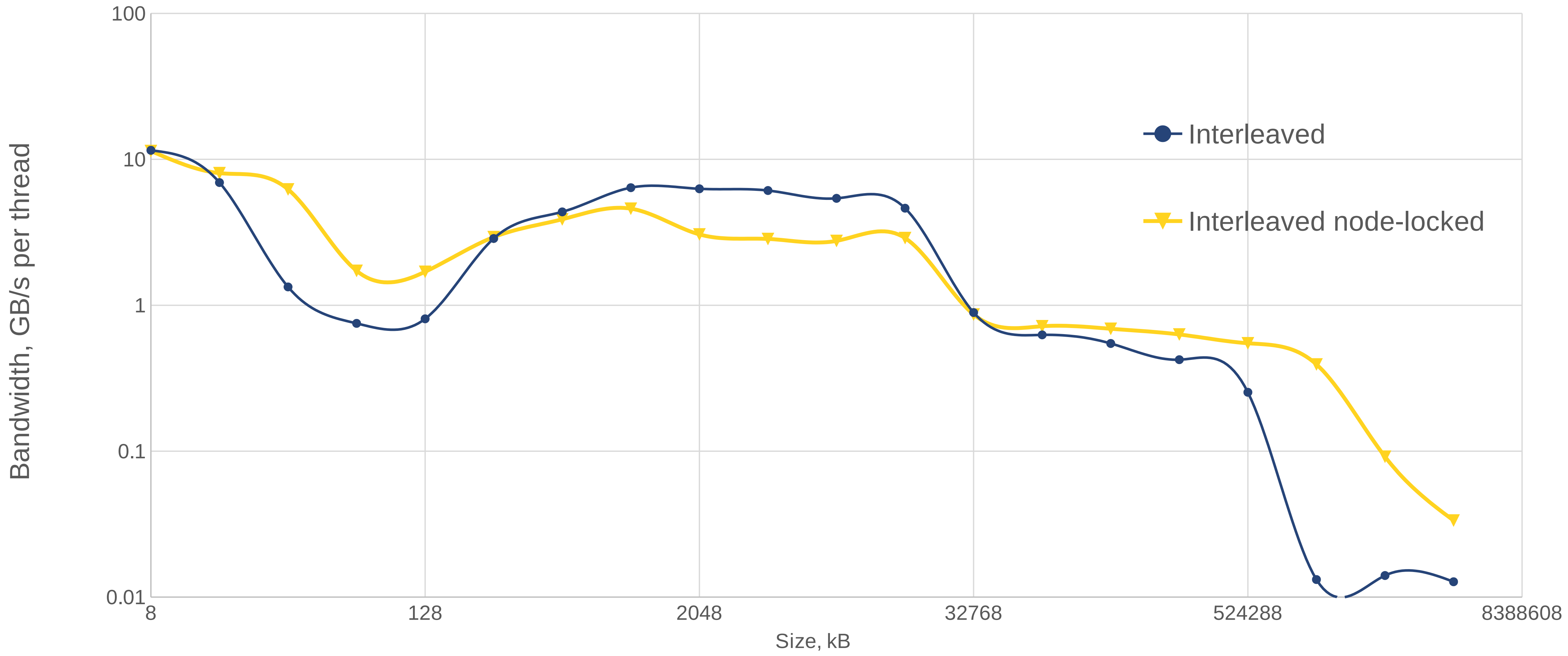
(Real) Performance implications of NUMA

- Test: all data resides on memory node 0
- 64 threads run on 64 cores (1 thread per core, all nodes)
- No cache lines shared between threads
- Thread data is interleaved and accessed sequentially (with stride)
- Results depend strongly on timing of concurrent accesses
- Running all 64 threads on the same NUMA node (only 32 cores) may be faster than using all 64 cores!

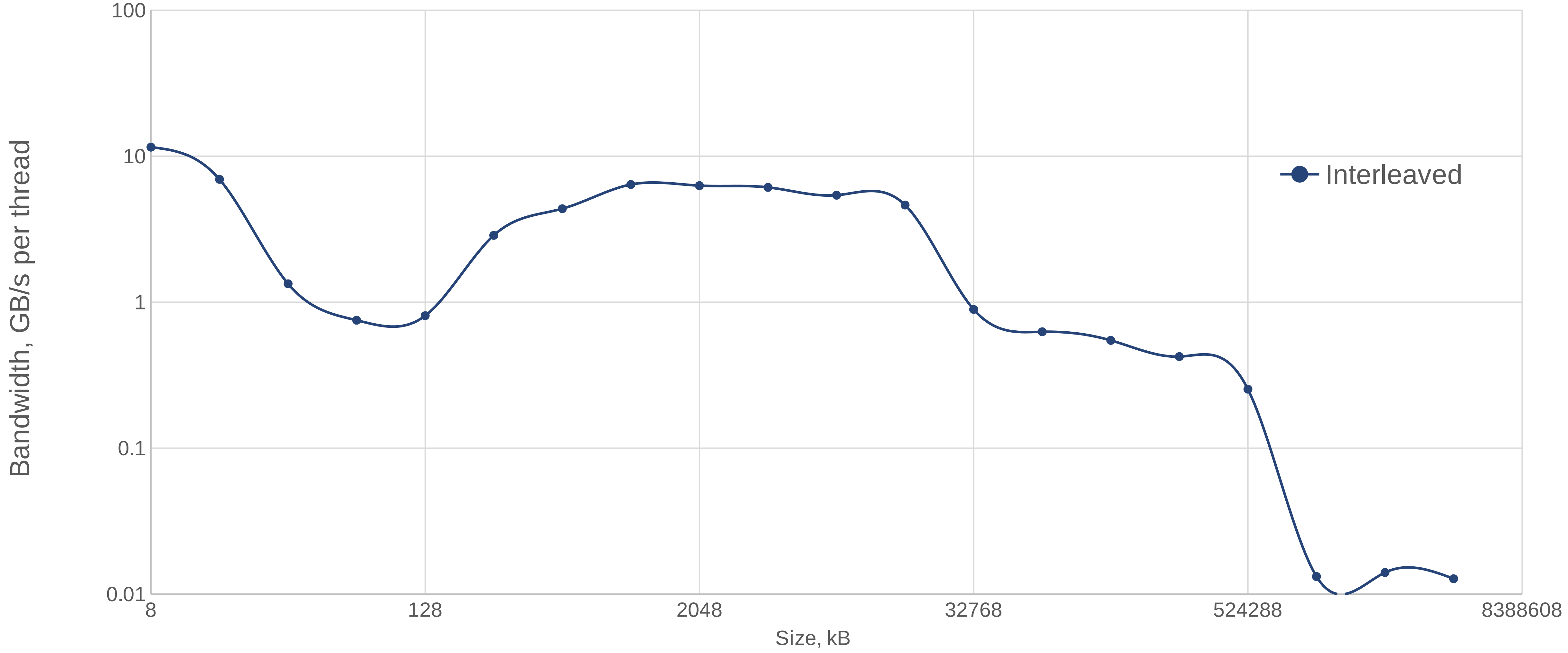
We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

Performance of a memory-bound program



Observed performance without NUMA control



If we do not control NUMA, who does?

- Unless explicitly overridden, the OS manages NUMA node loading
- CPU scheduler
  - Minimize thread migration between cores (usually)
  - Minimize thread migration between nodes (NUMA-aware scheduling)
  - Migrate threads to the node where the memory is (rarely)
- Memory allocation
  - Memory is usually allocated on the node running the thread (if available)
  - Memory may be migrated to another node if accessed from there (common)

How to find out what you have? (Linux)

```
$ lscpu
CPU(s):                120
Thread(s) per core:    2
Core(s) per socket:    15
Socket(s):             4
NUMA node(s):          4
NUMA node0 CPU(s):     0-14,60-74
NUMA node1 CPU(s):     15-29,75-89
NUMA node2 CPU(s):     30-44,90-104
NUMA node3 CPU(s):     45-59,105-119
```

How to find out what you have? (Linux)

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74
node 0 size: 386638 MB
...
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

Memory latency (relative)

How to find out what you have

- Intel memory latency checker
  - Not sure how relevant these numbers are…

```
$ mlc --latency_matrix -e -r
Measuring idle latencies for random access (in ns)...
           Numa node
Numa node      0       1       2       3
       0   130.7   187.9   193.4   200.8
       1   190.2   129.0   192.2   194.0
       2   196.0   194.4   130.1   192.9
       3   203.4   192.1   193.9   129.4
```

Simple NUMA controls (Linux)

```
numactl --cpunodebind=N program args…
```

- Run a program with all threads executing on node N (any CPU)

```
numactl --membind=M program args…
```

- Run a program with all memory allocations bound to node M
  - No memory transfer between nodes
  - Allocation will fail if the node is out of memory

Debugging NUMA problems – step 0

- Run the program with NUMA restrictions and compare the results

```
$ numactl --cpunodebind=0 –membind=0 ./membench
64 threads    4194304 kB   2.160650 GB/s
$ numactl --cpunodebind=0 –membind=1 ./membench
64 threads    4194304 kB   0.670354 GB/s
```

- This program is sensitive to NUMA allocations, need to investigate…
  - Use reason – 1 node has half or quarter of all memory and CPUs
  - Running 64 threads on 32 cores (best NUMA access) may be slower than running 64 threads on 64 cores with NUMA penalties

Debugging NUMA problems – step 0

- Run the program with/without NUMA restrictions and compare the results

```
$ ./parallel_foreach

32 threads: 4.74 million elements/second

$ numactl --cpunodebind=0 –membind=0 ./parallel_foreach

32 threads: 7.92 million elements/second
```

- This program is sensitive to NUMA allocations, need to investigate…
  - Hardware: 32 physical cores in 2 NUMA nodes (SMT – 2 logical cores per physical)

Debugging NUMA problems – step 0

- Run the program with/without NUMA restrictions and compare the results

```
$ ./parallel_foreach
32 threads: 4.74 million elements/second
$ numactl --cpunodebind=0 –membind=0 ./parallel_foreach
32 threads: 7.92 million elements/second
$ numactl --cpunodebind=0 –membind=1 ./parallel_foreach
32 threads: 7.30 million elements/second
```

- This program is sensitive to NUMA allocations, need to investigate…
  - Hardware: 32 physical cores in 2 NUMA nodes (SMT – 2 logical cores per physical)

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- Simple benchmarks with different node binding often point to potential problems but usually not reveal their origin

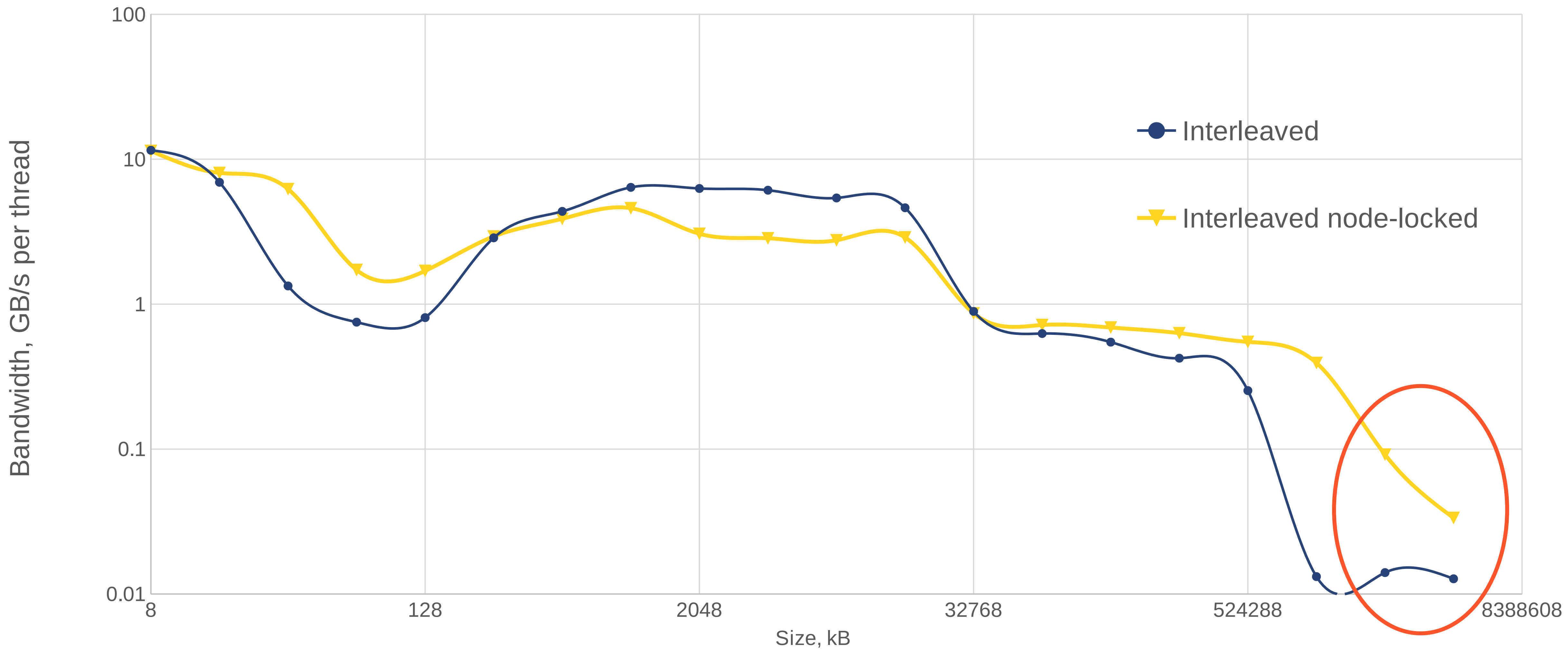
Debugging NUMA problems – step 1

- Profilers can help debug NUMA-related issues
- Indirect analysis: compare profiles for runs with different NUMA bindings
- Direct analysis: profile NUMA-related hardware counters
  - Intel VTune, Linux perf, PCM, etc
- Limited direct measurements
  - Can report QPI/UPI bandwidth per link
  - Large traffic between CPU sockets may indicate NUMA problems
- Profiling measurements are often misleading
- Linux kernel reports aggregate NUMA stats
  - Rarely accurate enough to be useful

Practical debugging of NUMA problems

- If the program (or a part of it) runs faster when restricted to subset of nodes, you likely have a NUMA-related performance issue
  - Good breakdown of execution time is essential – instrument your code
- Compare profiles with and without NUMA restrictions or with favorable vs unfavorable NUMA restrictions
- Simplify the problem – benchmark suspected code fragments
  - Sometimes hardware profiles help (sometimes they mislead)
  - Often comparing results of `numactl` runs is your best approach
- Compare results on different hardware
  - Newer systems tend to expose NUMA-related problems

Performance of a memory-bound program



Comparing VTune profiles

	Interleaved	CPU-bound to node 0	
Instructions per cycle	0.04	0.08	Low (bound is better, but we knew that)
Memory-bound	42%		Memory-bound (we know that)
L1, L2, L3, DRAM-bound	2% to 0.5%		No obvious differences
NUMA remote access	50%	0.5%	Direct metric!
UPI-bound	50%	0%	Related metrics
UPI bandwidth	70%	20%	

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling sometimes can identify the problem and the location in the code

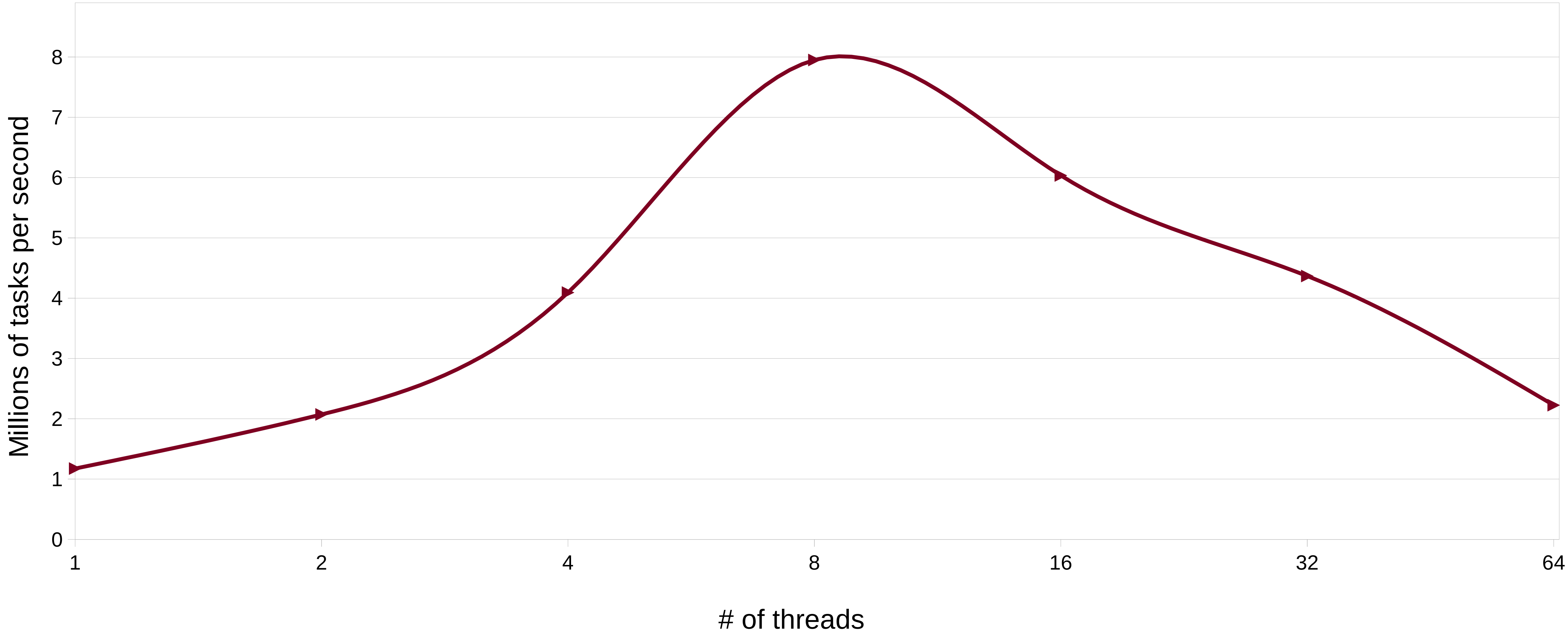
We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

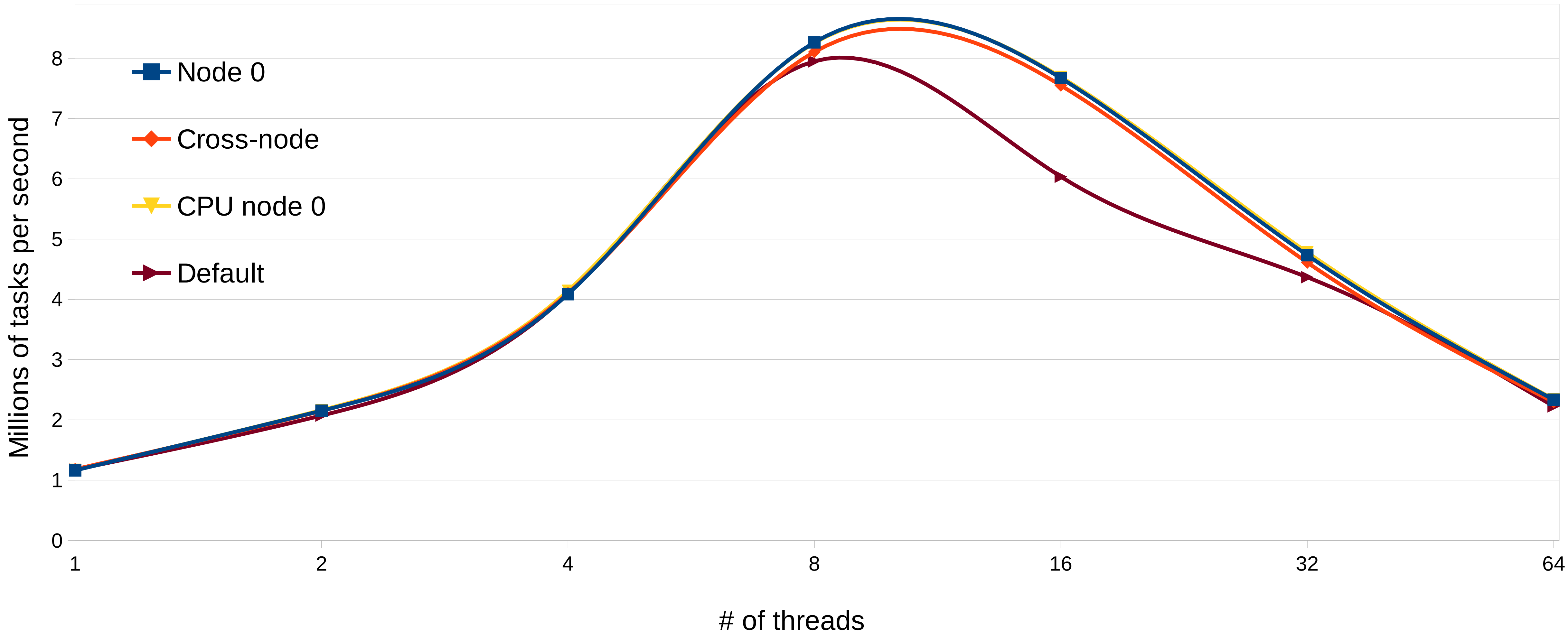
History of this case…

- Large multi-threaded program does not scale as much as expected
- Scaling is worse on newer processors
- Parts of the program run faster on one NUMA node than on two
- Problem step of the program is simplified and reduced to parallel foreach
- Per-element computations (parallel tasks) are compute-intense
  - Performance-critical code is not memory-bound
  - For simplification, memory use is further minimized

Parallel foreach (no NUMA controls)



Parallel foreach with numactl



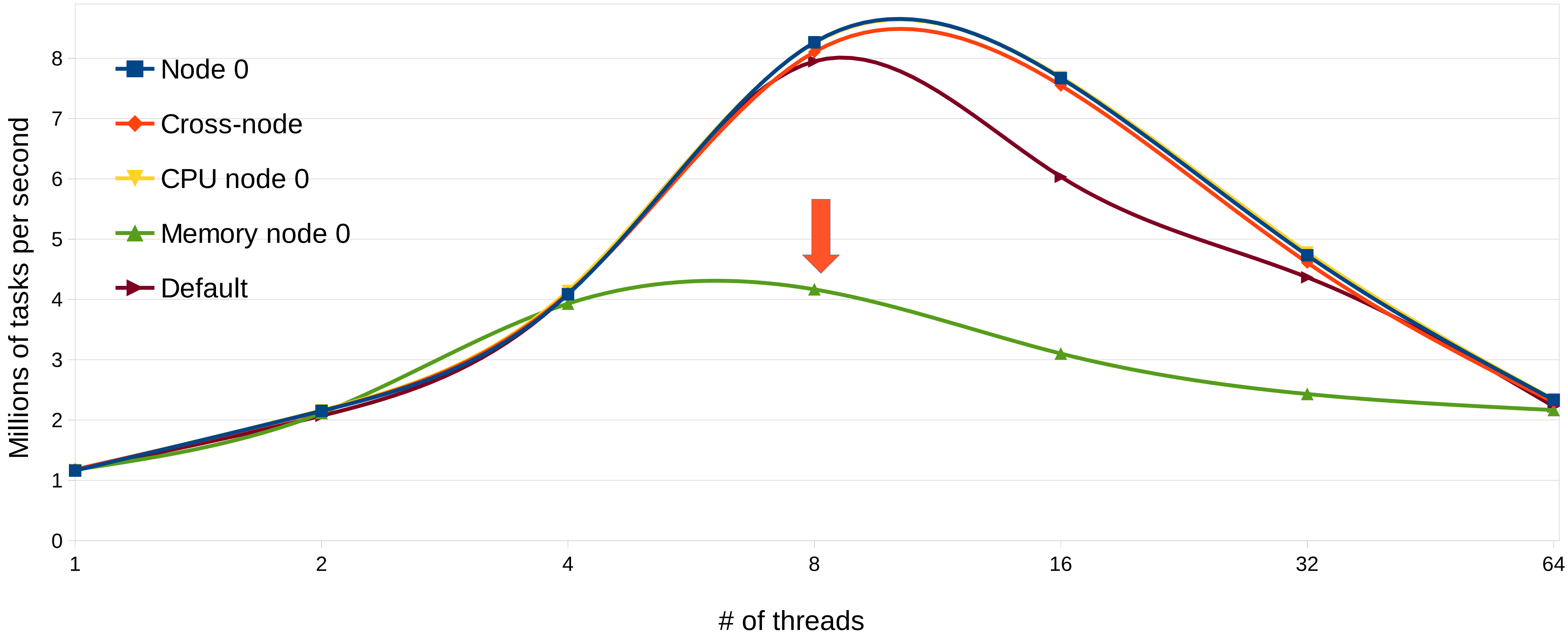
Performance analysis – first impression

- Restricting all threads to one node improves throughput
  - One node has 16 physical cores (2 SMT threads on each)
  - Running 32 threads on one node is faster than on both
- Restricting memory does not make any difference
  - Binding threads to node 0 and memory to node 1 is fast
  - Not binding threads is slower but test is unreliable, depends on the OS

Performance analysis – first impression

- Restricting all threads to one node improves throughput
  - One node has 16 physical cores (2 SMT threads on each)
  - Running 32 threads on one node is faster than on both
- Restricting memory does not make any difference
  - Binding threads to node 0 and memory to node 1 is fast
  - Not binding threads is slower but test is unreliable, depends on the OS
- If you want something done right, do it yourself
  - Bind half of all threads to node 0 and half to node 1

Parallel foreach



Performance analysis – second impression

- Memory is irrelevant
  - Makes sense – the simplified program uses very little memory
- Forcing threads to run on both nodes equally is even worse than whatever the OS does
- Restricting all threads to one node makes the program run faster
  - Makes no sense – there aren’t enough cores in one node for all threads
- Time to use the profiler

Profiling parallel foreach

- Profile the worst (strangest?) case – 32 threads on one vs two nodes
- Try different hardware counters until enlightenment achieved
- VTune’s hints may be helpful or misleading
- Profiling the problem case by itself is rarely useful
- Comparing two versions with systematic differences is often productive

Profiling with perf – original code

- 617276143376 cycles:u
  21575458142 instructions:u # 0.03 insn per cycle
  541435483399 cycle_activity.stalls_mem_any:u
- 1309864032 cache-references:u
  459547539 cache-misses:u # 35.084 % of all cache refs
- 281154229 LLC-loads:u
  85821152 LLC-load-misses:u # 30.52% of all LL-cache accesses
- 135375847 LLC-stores:u
  41651753 LLC-store-misses:u # 30.77% of all LL-cache accesses
- Also significant fraction of front-end stalls and other problems

Profiling with perf – CPU bound to node 0

-
  ```
  245669779655      cycles:u
   38546010583      instructions:u              #    0.16  insn per cycle
  114854490177      cycle_activity.stalls_mem_any:u
  ```

-
  ```
  1536581119      cache-references:u
    54666032      cache-misses:u              #    3.558 % of all cache refs
  ```

-
  ```
  232443098      LLC-loads:u
    3786766      LLC-load-misses:u            #    1.63% of all LL-cache accesses
  ```

-
  ```
  238253412      LLC-stores:u
    3184831      LLC-store-misses:u          #    1.33% of all LL-cache accesses
  ```

- Program uses very little memory, writes ~MB/s (vs ~10 GB/s bandwidth)

Comparing VTune profiles

	Unrestricted	CPU-bound to node 0	
Instructions per cycle	0.18	0.24	Low (bound is better, but we knew that)
CPU utilization	20%	30%	Low; causes: memory stalls, branch misses, instruction starvation
Front-end stalls	52%	30%	High, causes: d-cache misses, i-cache misses
Memory-bound	34%	19%	High, mostly cache-bound (why?)
NUMA remote access	80%	20%	Direct metric, but of what?

- The program is very small and uses very little memory

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling can be misleading or boil down to “something somewhere is somehow gone wrong”

What do we know? What makes sense?

- The benchmark is a parallel foreach (4M of code including C++ runtime)
- Each task is about 1us, uses 8B of memory
- On NUMA systems, scaling is poor when thread count exceeds number of CPUs in a single node
- Binding all threads to one NUMA node improved throughput slightly
  - Even if the node doesn't have enough CPUs
- Binding memory to one NUMA node makes no difference
- Profilers report large percentage of LLC misses
  - The entire data set fits into LLC
- Profilers report front-end problems usually seen in much larger code

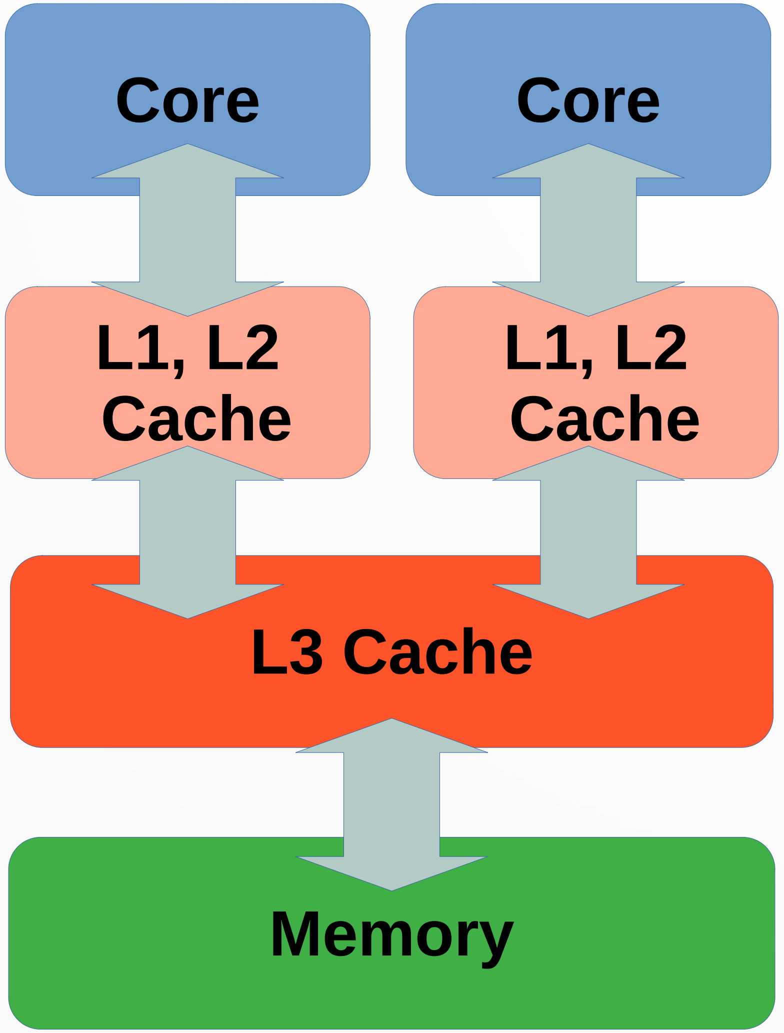
What do we know? What makes sense?

- The problem is worse in newer systems (UPI scales worse than QPI)
  - UPI has higher bandwidth
- Time to ask for help from the people who designed the processor
  - Latest version of UPI has even higher bandwidth (and runs even slower)
  - The program is latency-bound
- QPI cross-node latency: 120-140 ns (40% over in-node)
  UPI cross-node latency: 140-160 ns (80% over in-node)
- Newer CPUs have higher interconnect bandwidth and higher latency
  - Both relative and absolute latency

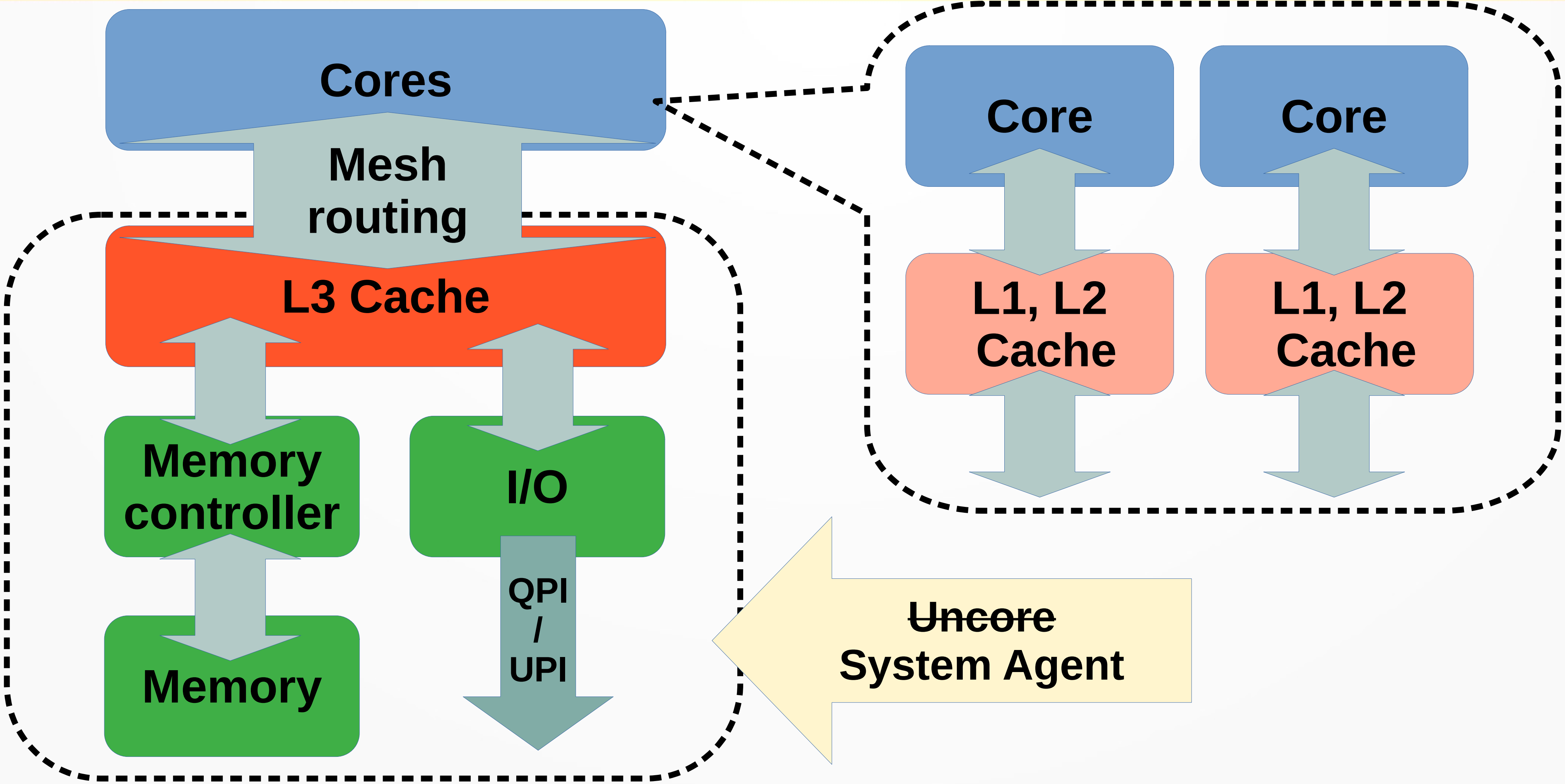
Why (when) latency is important?

- Random access of small amounts of data
  - Not relevant for our problem
- Concurrent access of the same data – data sharing
  - Throughput is limited by the latency of atomic accesses
- Foreach is perfectly parallel, no data sharing at all, but...
- Any thread scheduler is going to have some shared data
  - Task queue (queue lock or atomics)
  - Task count or whatever you wait on
  - Other data, depending on the thread scheduler/pool implementation

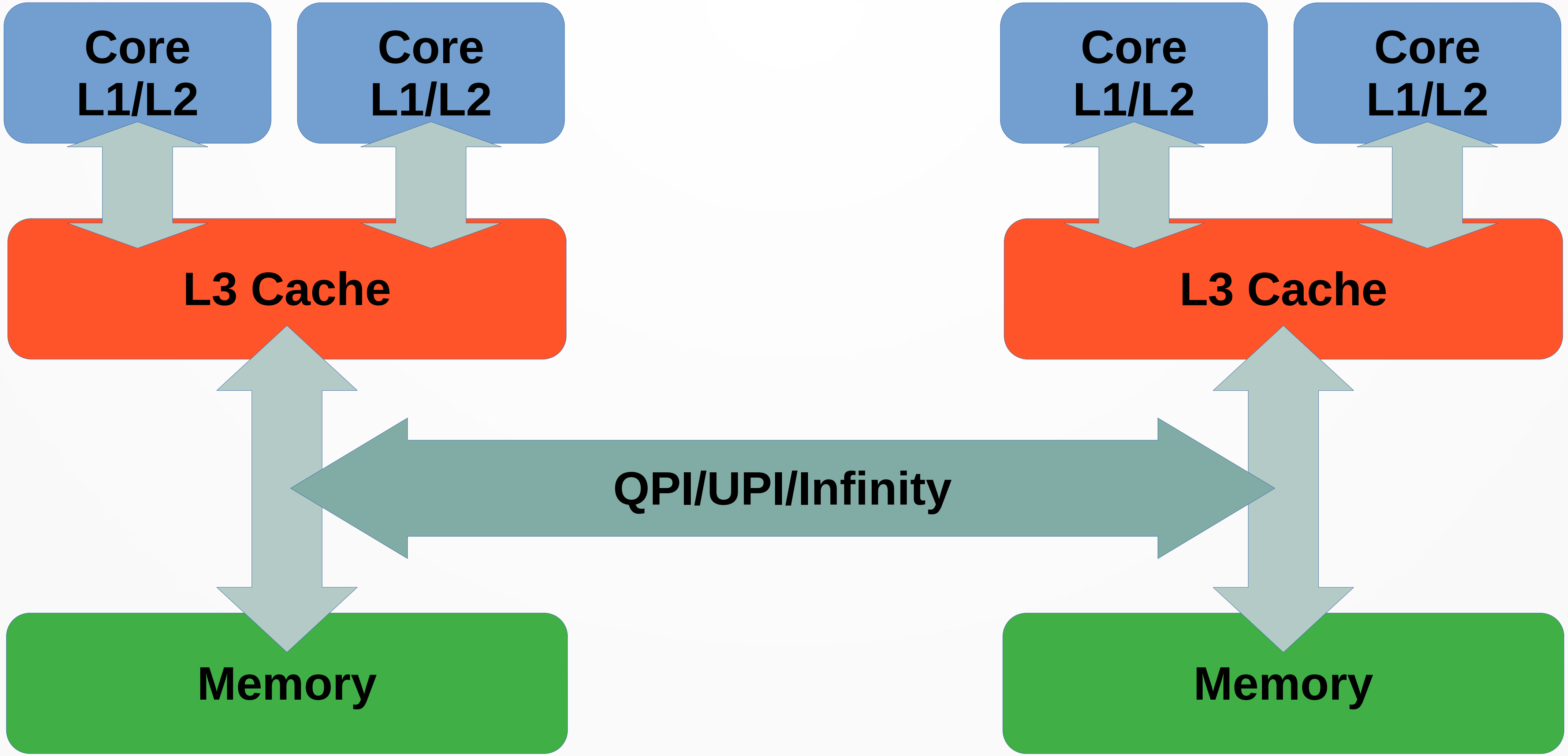
Multi-threading and latency



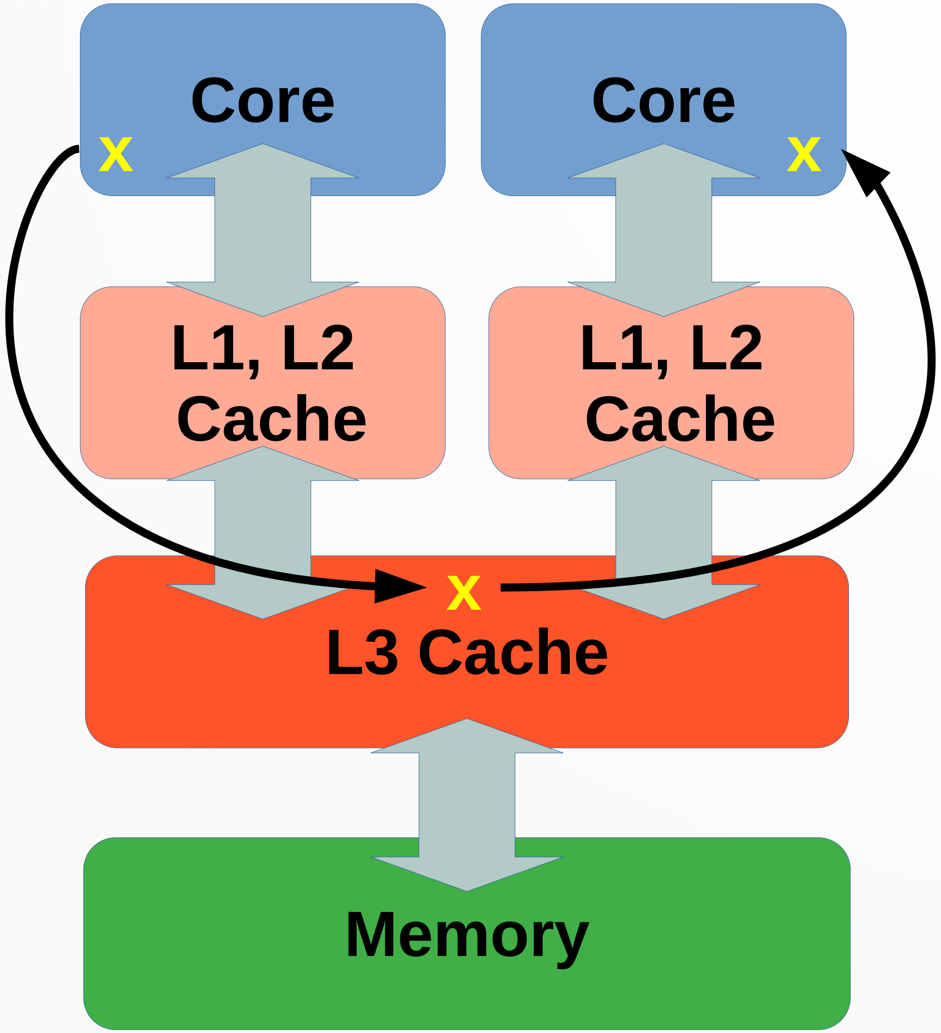
Multi-threading and latency



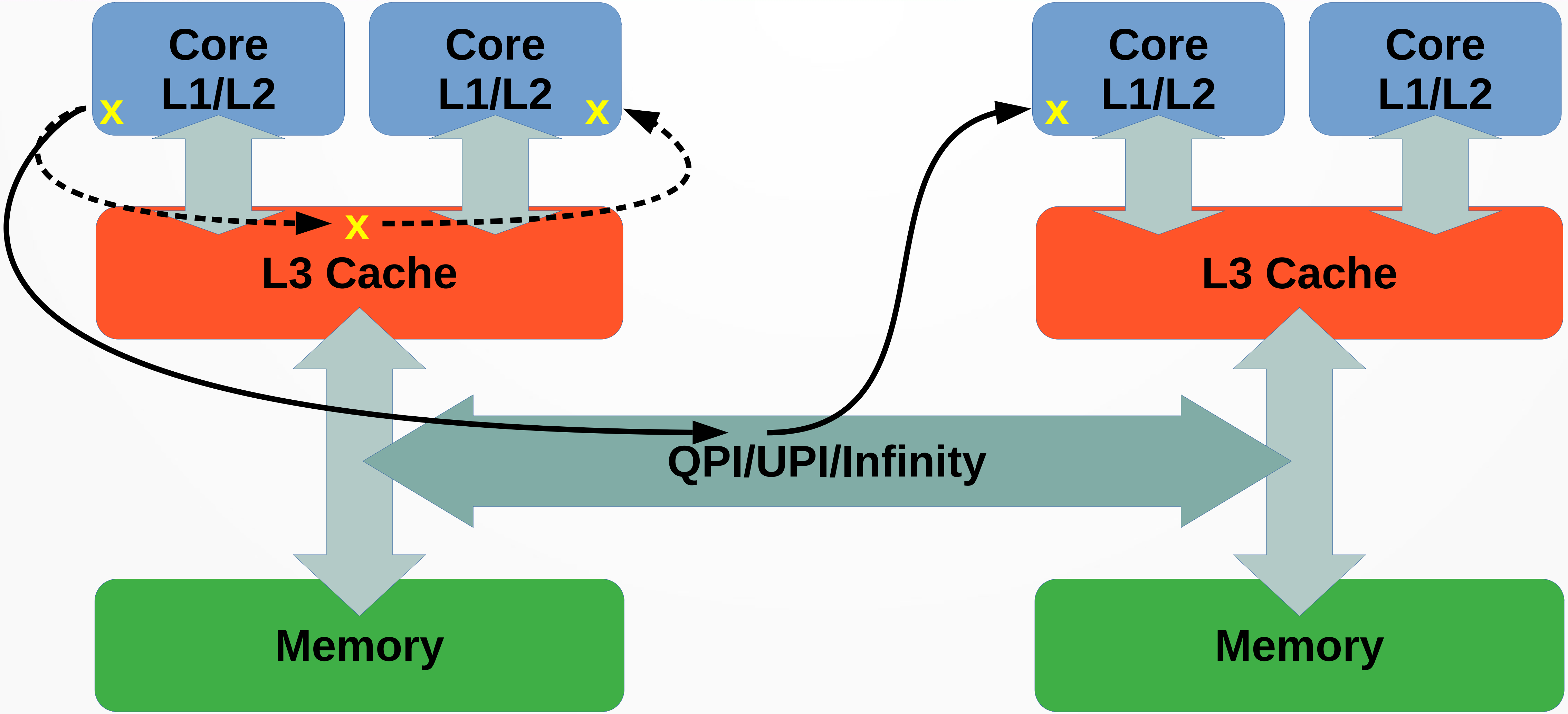
Multi-threading and latency



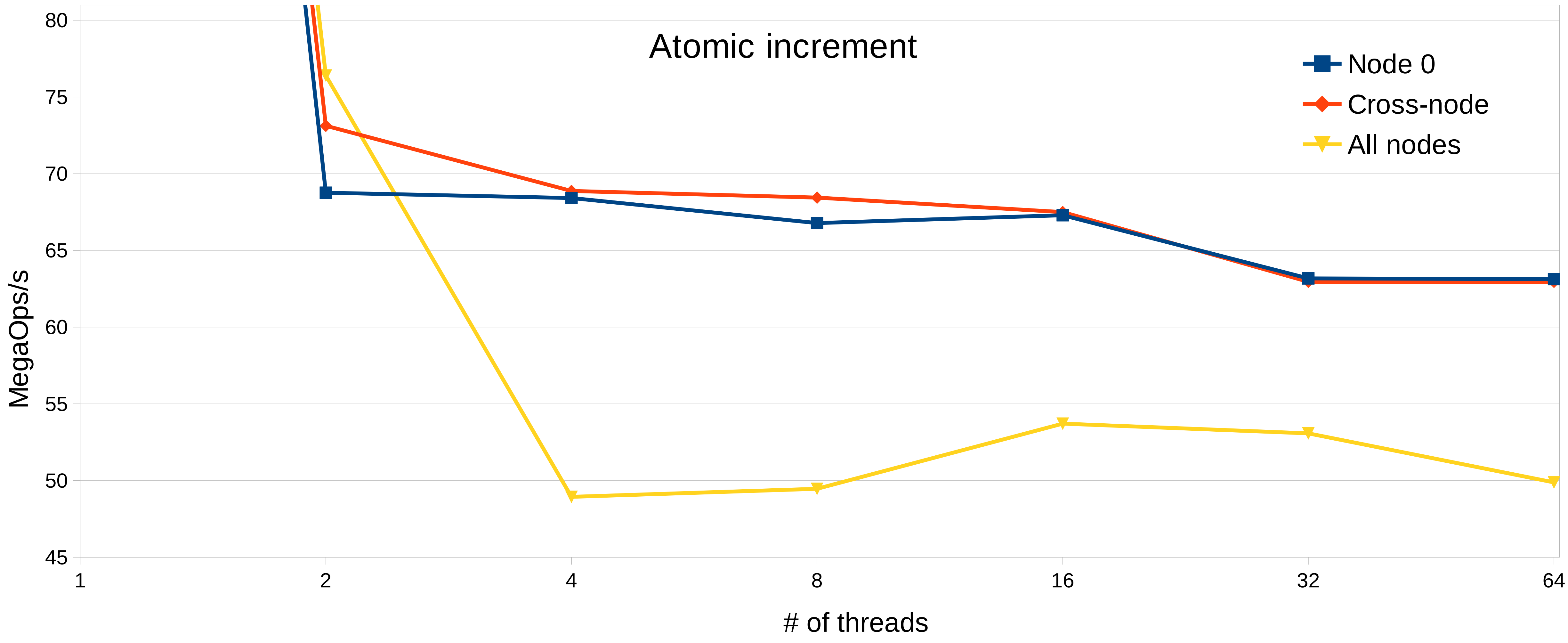
This is your data sharing



This is your data sharing on NUMA



Is NUMA performance hurt by data sharing?



Is NUMA performance hurt by data sharing?

- Performance of a single atomic operation often shows some slowdown on NUMA systems
  - Especially for Read-Modify-Write operations
  - Usually not for atomic load or store (so RCU algorithms may have advantage)
  - Intel QPI systems may not show any slowdown at all
  - AMD Epyc systems show little to none NUMA penalty
- Most concurrent data structures and executors use shared variables and atomic operations
  - Locks too (usually atomic exchange)
- Maintaining consistent global shared state on NUMA systems is expensive

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for

We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

NUMA control beyond numactl

- NUMA programming API allows very fine control of thread and memory interactions with NUMA hardware

```
#include <numa.h>              // Link with -lnuma
#include <numaif.h>
```

- Restrict the calling thread to a subset of NUMA nodes or CPUs:

```
numa_run_on_node(), numa_sched_set_affinity()
```

also `pthread_setaffinity_np()` from pthreads library

- Restrict memory allocations by the calling thread to specific NUMA nodes:

```
numa_set_preferred(), set_mempolicy()
```

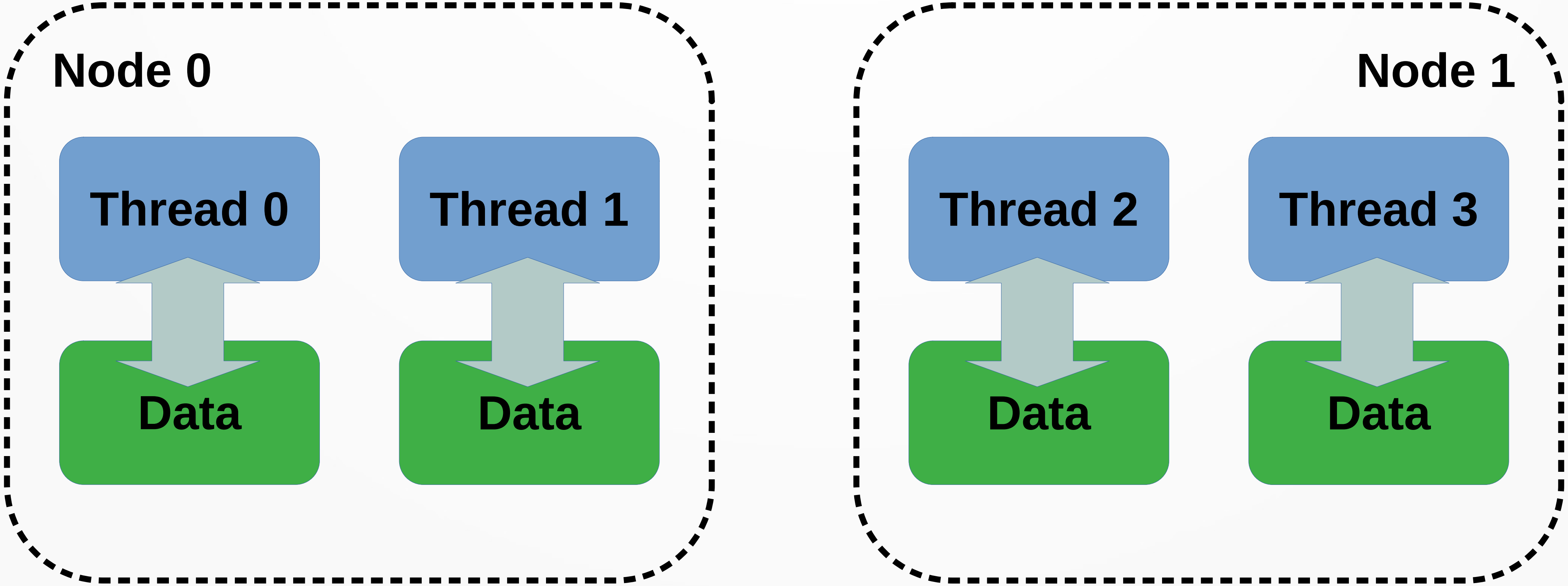
- Functions for direct allocations on the specified node, moving memory between nodes, querying current NUMA policies and hardware capabilities
- Caution: cpu-to-node mapping is SLOOOW (cache cpu maps and masks)

When to use NUMA controls?

- In many applications, never (it just works)
- Areas of concern:
  - High-throughput or low-latency concurrent data structures
  - Thread pools, executors, schedulers, etc
  - Memory-bound programs
- Try to bind the program to one NUMA node (or bind CPU and memory to different nodes) to see if performance is affected
- If you must use NUMA API, encapsulate it (NUMA pool, NUMA counter)
  - Exception: NUMA-friendly memory management may impact the way you do concurrency in your algorithm

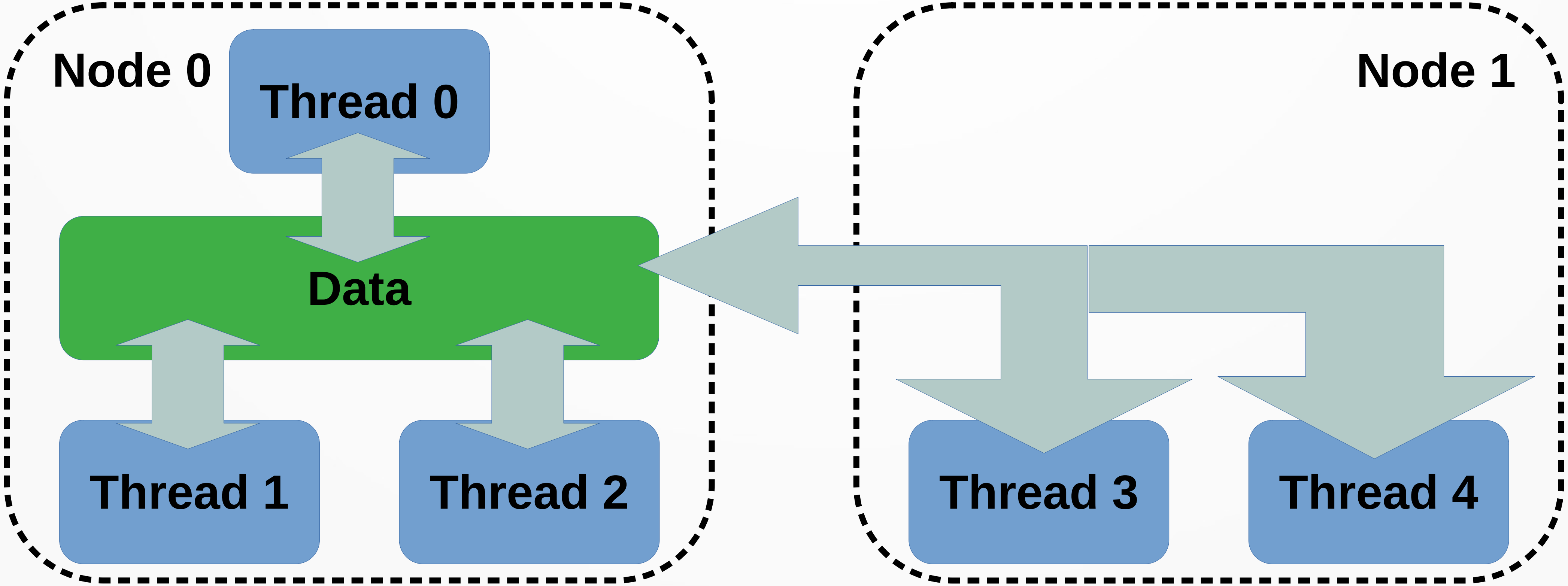
Memory-bound programs

- If each thread mostly uses its own memory, you can bind threads to NUMA nodes (but be mindful of load balancing)
  - Verify that it is needed – OS scheduler usually does good enough job



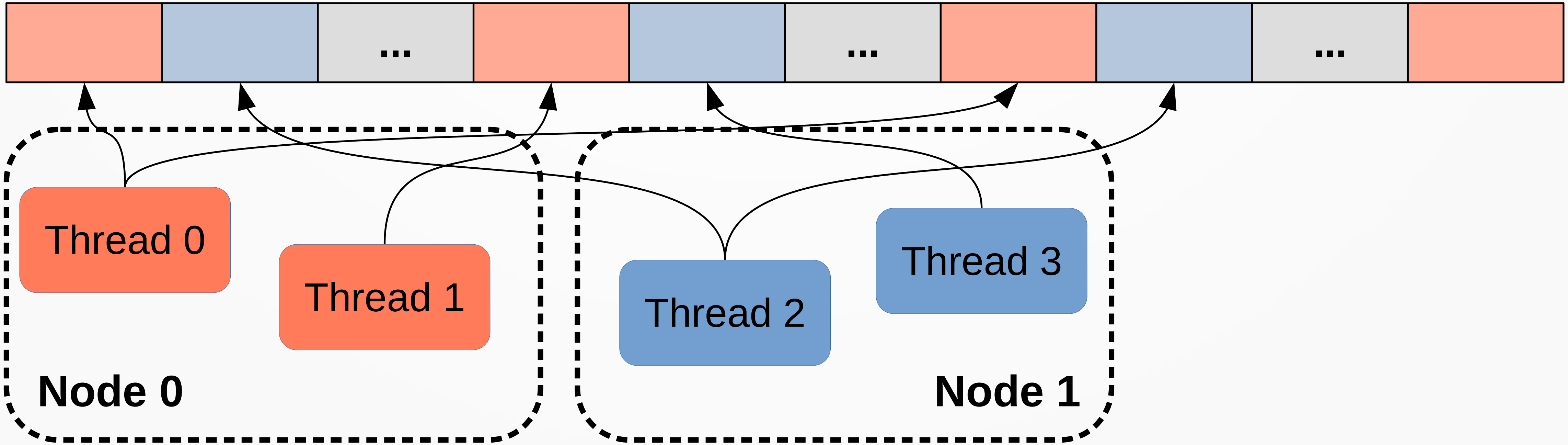
Memory-bound programs

- Memory is usually allocated on the calling thread’s node by default
  - But if the parent thread allocates memory, you may want to explicitly move it
  - Recent Linux kernels will eventually do it for you (with some caveats)



Memory-bound programs

- If multiple threads update the same data structure (but not the same memory locations) the working sets of these threads must be separated
  - Read-only data is OK
- Algorithm changes may be required to increase per-thread work size
  - Separated by how much? One page (4K) may be enough



Memory-bound programs

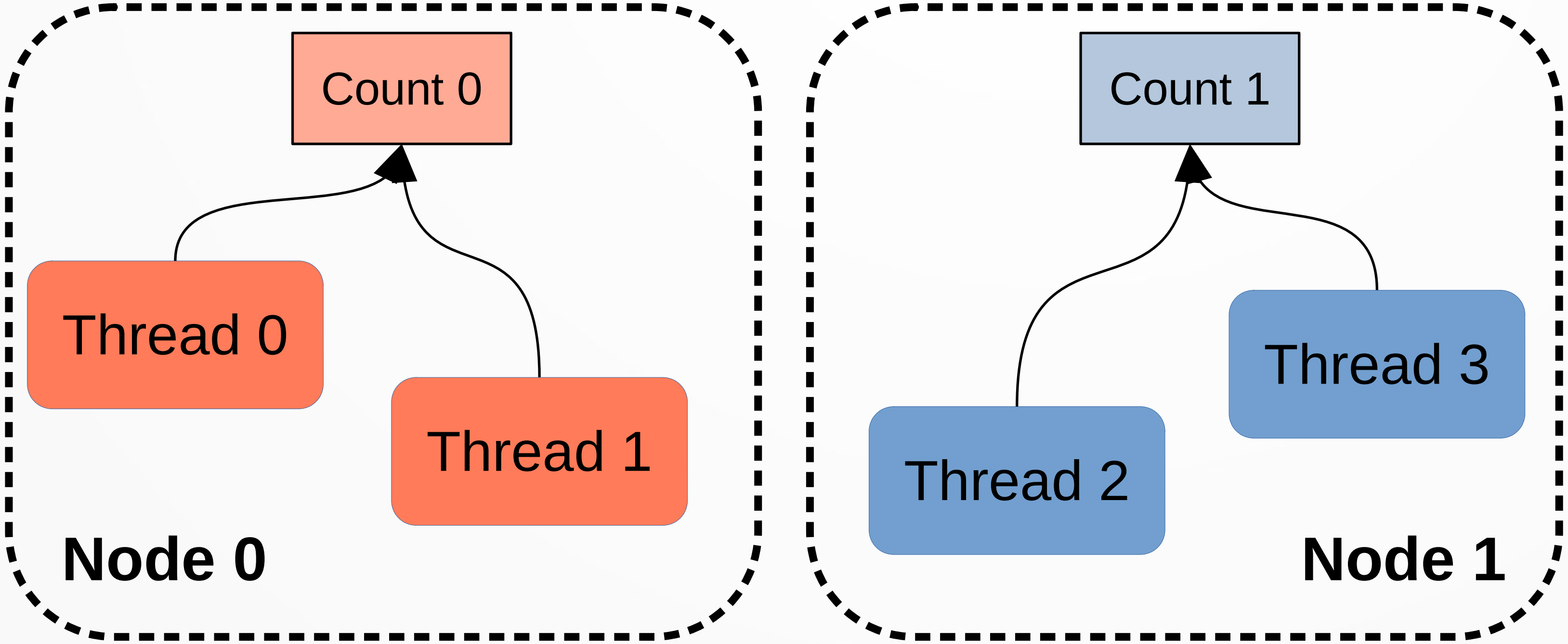
- If each thread mostly uses its own memory, you can bind threads to NUMA nodes (but be mindful of load balancing)
  - Verify that it is needed – OS scheduler usually does good enough job
  - Memory is usually allocated on the same node by default
  - But if the parent thread allocates memory, you may want to explicitly move it
  - Recent Linux kernels will eventually do it for you (with some caveats…)
- If multiple threads update the same data structure (but not the same memory locations) the working sets of these threads must be separated
  - Read-only data is OK
- Algorithm changes may be required to increase per-thread work size
  - Separated by how much? One page (4K) may be enough (but there are complications…)

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for
- Good practices for memory-bound programs are more important
- Consider binding threads and/or memory if OS defaults are insufficient

Latency-bound programs

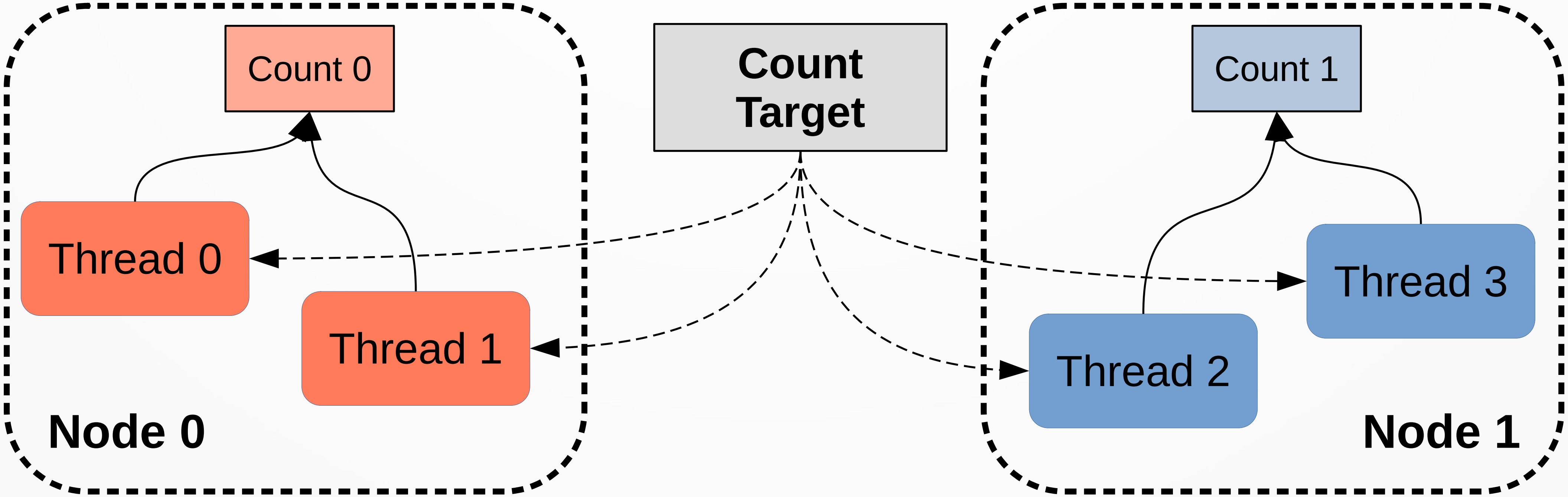
- If cross-node traffic caused by some shared data is significant, the implementation has to be redesigned to use per-node state



- But we still need a global state...

NUMA-aware counter

- If cross-node traffic caused by some shared data is significant, the implementation has to be redesigned to use per-node state
- Synchronization is likely to be more complex



NUMA-aware counter

```
class NUMA_task_count {
  static constexpr size_t max_node_count = 64;
  task_count_t done_task_counts_[max_node_count];
  std::atomic<size_t> task_count_;
  public:
  size_t Count() const {  // This is what we wait on
    size_t res = task_count_.Acquire_Load();
    for (size_t i = 0; i != num_nodes; ++i)
      res -= done_task_counts_[i].value.Acquire_Load();
    return res;
  }
}; // NUMA_task_count
```

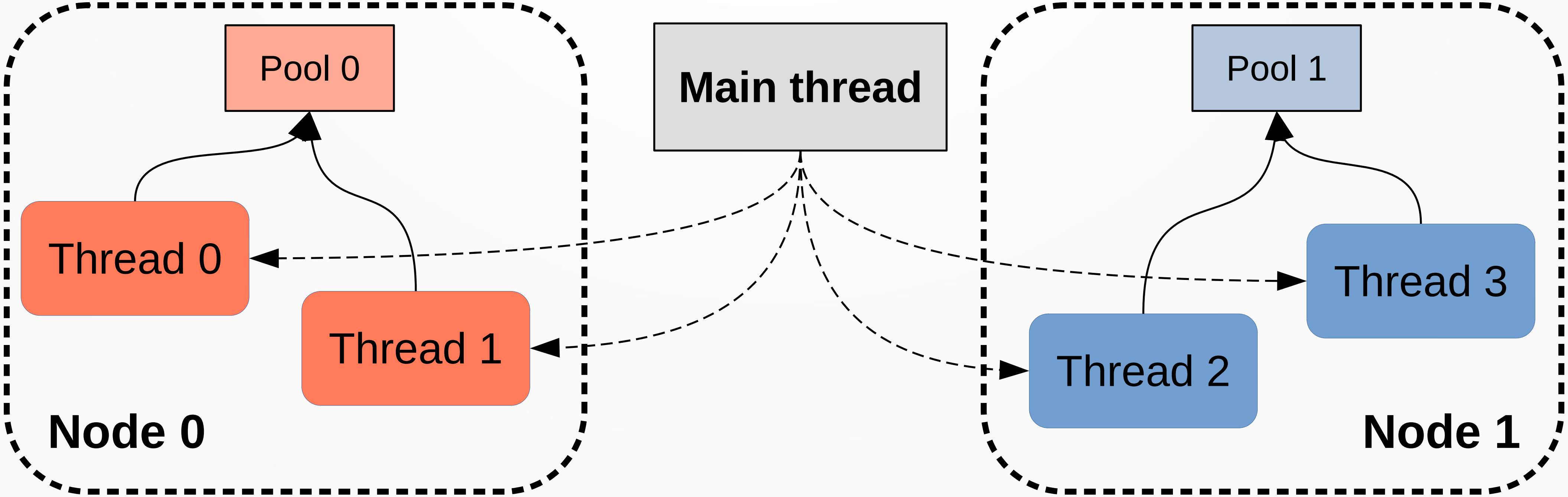
`size_t alignas(64)`

read-only during wait

updated during wait

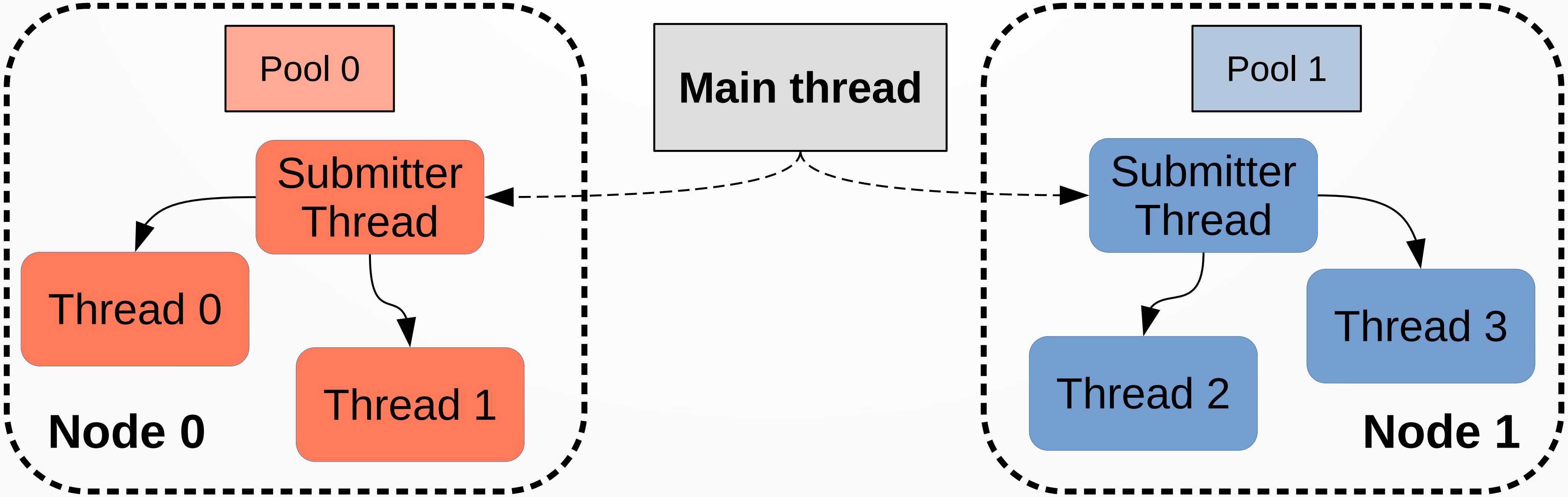
NUMA-aware thread pool

- Per-node pools, cross-node work stealing (only when one node is empty)
- Task submission from the main thread is the bottleneck



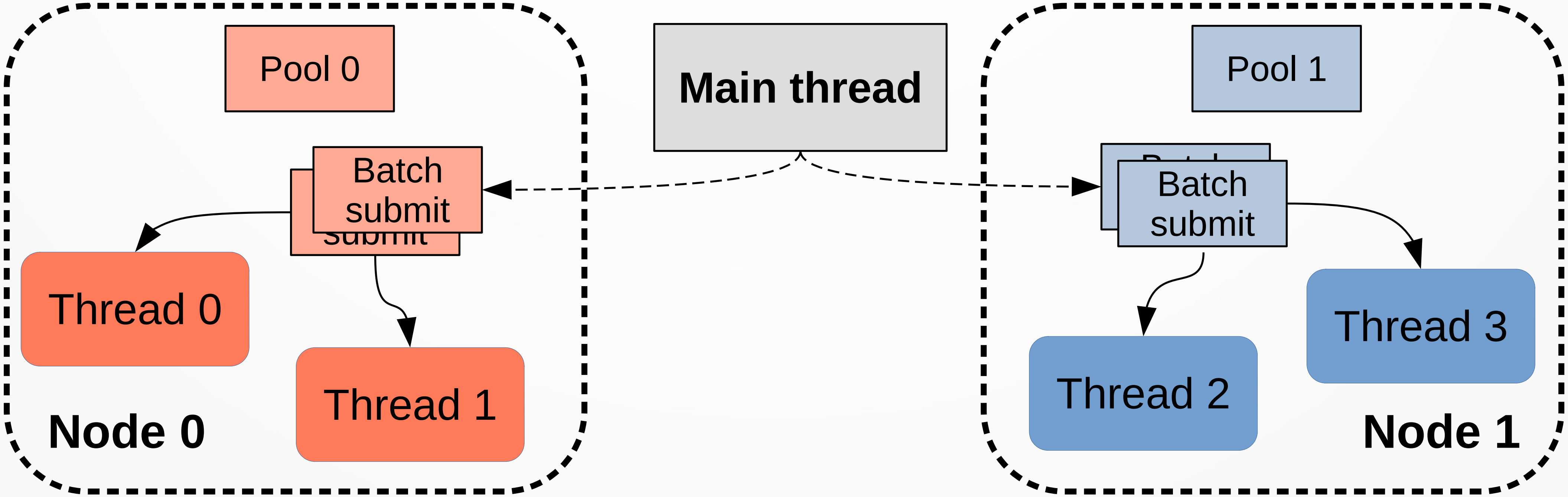
NUMA-aware thread pool

- Per-node pools, cross-node work stealing (only when one node is empty)
- Task submission from the main thread is the bottleneck
- Can be solved with a two-stage task submitter

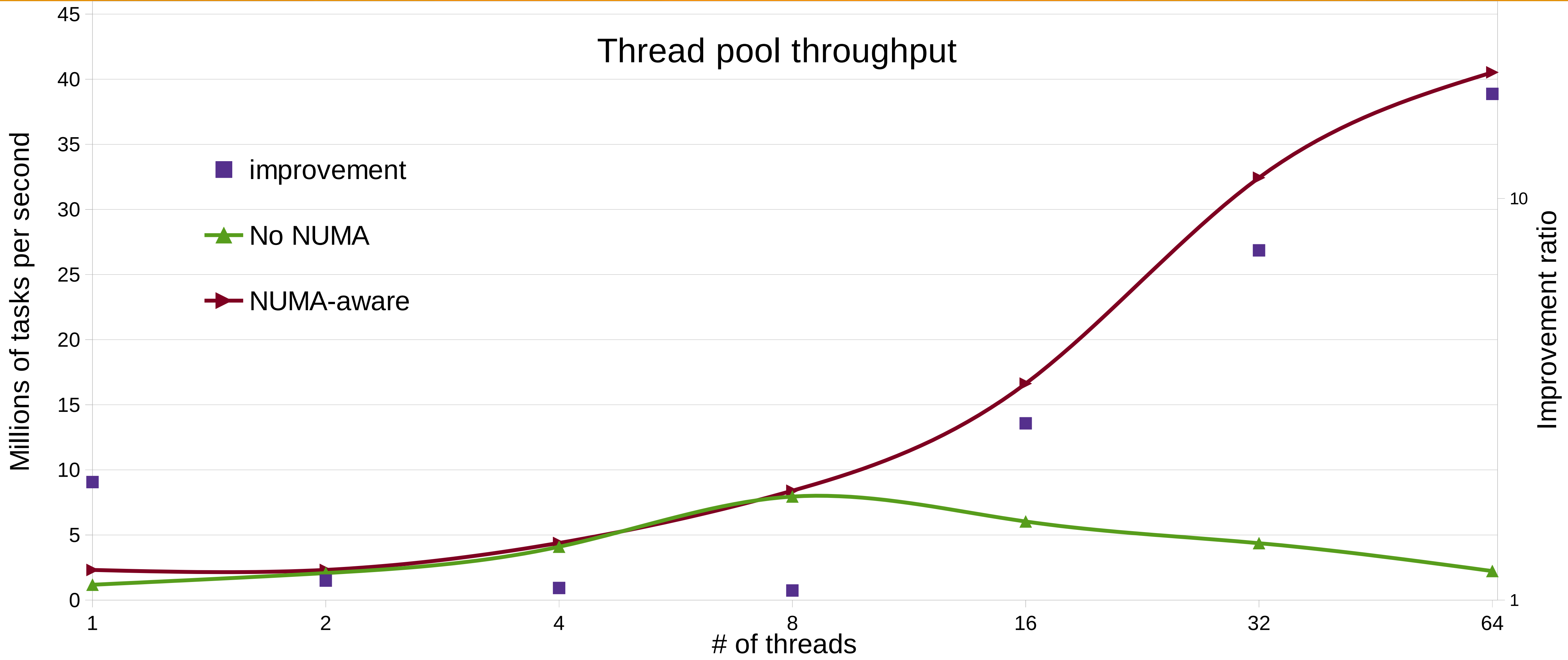


NUMA-aware thread pool

- Per-node pools, cross-node work stealing (only when one node is empty)
- Reduce cross-node data contention – fewer interacting threads or less frequent interaction



NUMA-aware thread pool



Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for
- Use good practices for memory access and node binding only if needed
- [Some] Concurrent data structures need to be redesigned for NUMA

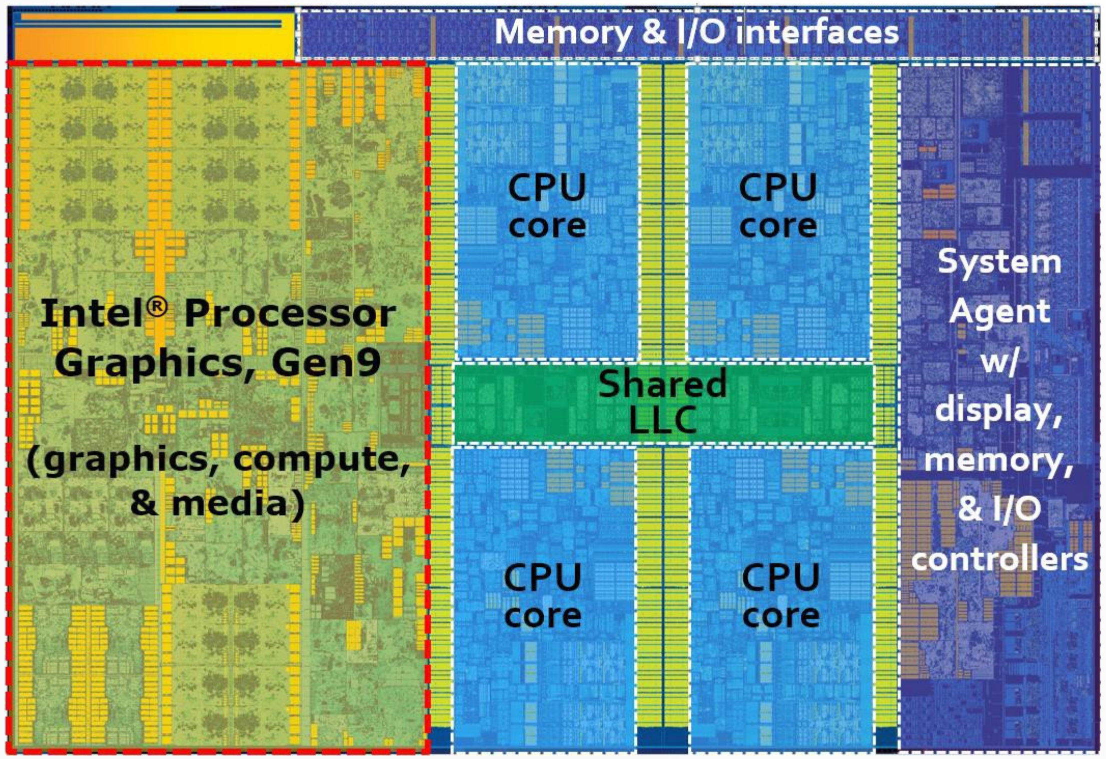
We Will Learn:

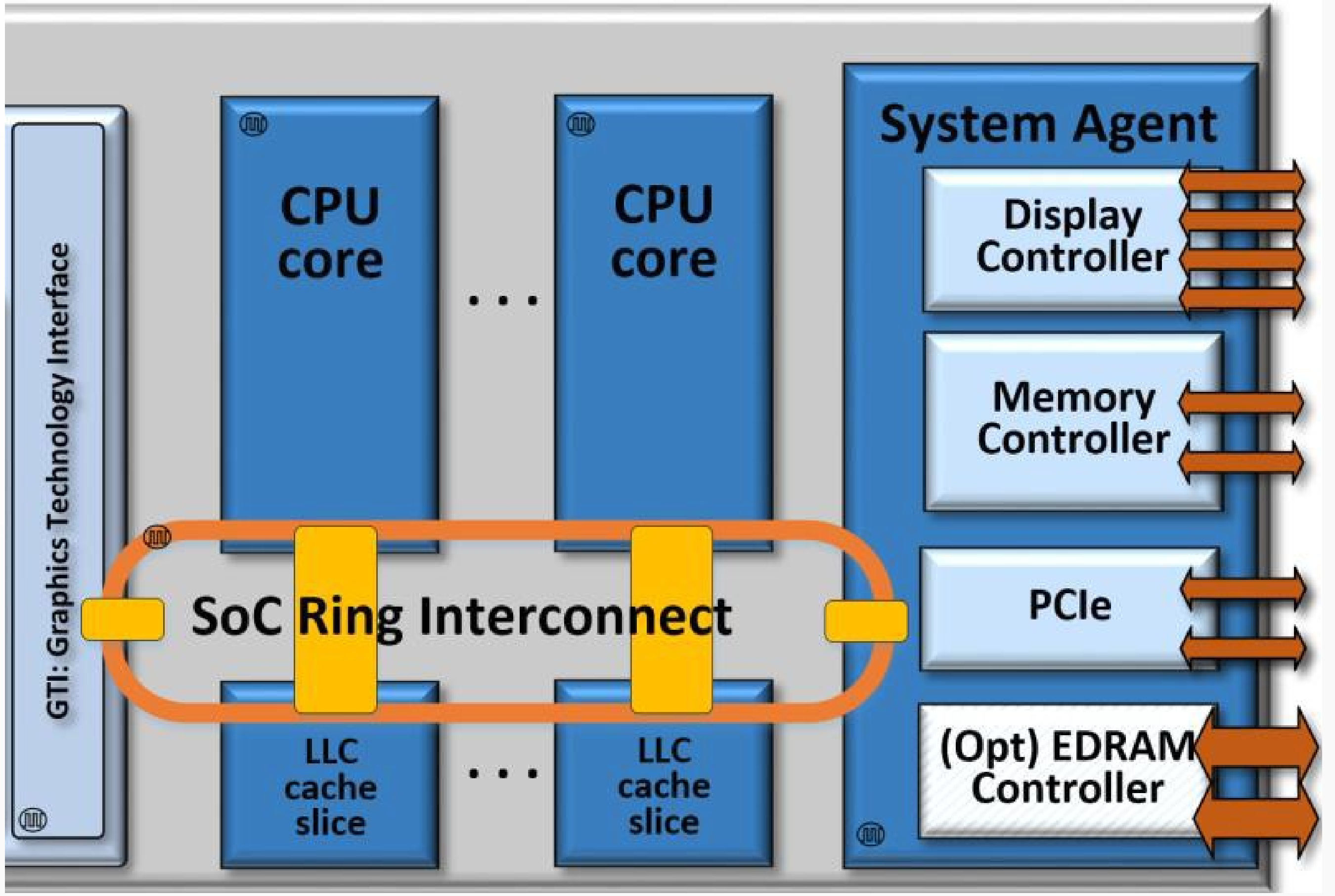
- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

NUMA concerns for distributed applications

NUMA concerns for distributed applications

- Distributed systems often send a lot of traffic across the network


Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers



NUMA concerns for distributed applications

- Distributed systems often send a lot of traffic across the network
- Which NUMA node is the network controller connected to?

```
# /sbin/lspci
04:00.0 Ethernet controller: Intel Corporation 82599ES
# cat /sys/bus/pci/devices/0000\:04\:00.0/numa_node
0
```

- Does it matter? Sometimes.
- Case study: distributed application (1000s of remote CPUs)
  - Use numactl to bind to node 0 vs node 1 – 10% run time difference
  - Built-in throughput tester: 20% difference between nodes 0 and 1

NUMA concerns for I/O-bound applications

- High-speed PCI-E devices are attached to a specific NUMA node
- NUMA may be important if the device is very fast
    - Simple test: bind the program to node 0 or node 1 and compare
        - Comparing with unbound run mostly tests the OS scheduler
- No case study – none of our programs are sufficiently I/O-bound

NUMA concerns for GPUs and accelerators

- GPUs are a special case of I/O
  - GPU interfaces are usually faster than any other device
  - Data transfer is often the bottleneck of GPU acceleration
- CUDA Bandwidth test for Tesla V100: 10GB/s (node 0) vs 7GB/s (node 1)
- BUT…
- If GPU-CPU transfer is a concern, pinned memory should be used
- Pinned memory is on the GPU node
  - At least by default?
  - Transfer rate 13GB/s

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for
- Use good practices for memory access and node binding only if needed
- [Some] Concurrent data structures need to be redesigned for NUMA
- For I/O-bound programs, use the right node for the data

We Will Learn:

- What is NUMA
- Why NUMA matters for programmers
- Performance implications of NUMA
  - Straightforward
  - Non-intuitive
  - Weird
- How to debug NUMA issues
- How to program for NUMA systems

Real-life case

- Program runs slower on (faster) NUMA hardware than on old machines
- The slowdown is limited to certain places in the code
- For these functions, slowdown is large (2x to 10x)
- The code is not limited by bandwidth or latency but there is one common trait: they touch large amounts of memory
  - Like updating links in a list of large elements
- The slowdown is OS-dependent:
  - Not present on old versions of Linux (overall, newer OS → faster runs)

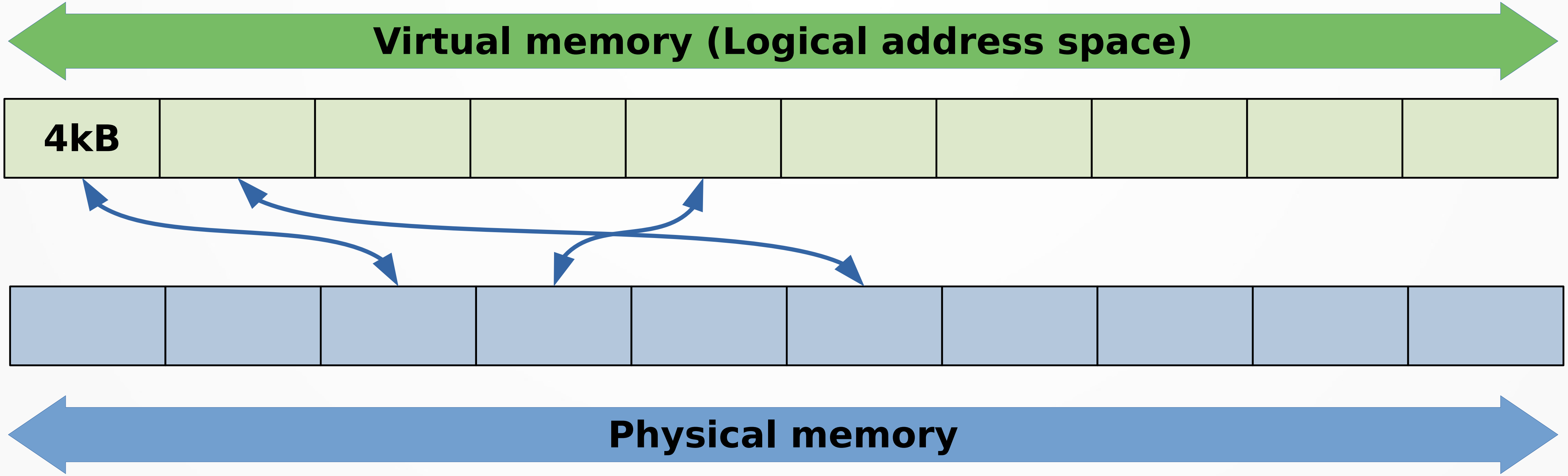
Debugging of the slowdown

- NUMA-related: the execution steps in question run faster when bound to one NUMA node
- Not easy to reproduce: specific memory access patterns are essential, hard to reproduce artificially
  - Plan B: modify the real program to run the problem step repeatedly or on larger data volume
- Profile with hardware counts: cache misses typical for semi-random memory access but no worse than elsewhere
- Built-in time reporting: real time is longer than CPU time
  - Even with multi-threading
  - There is no I/O or network traffic
  - Detailed time reporting: mostly system CPU time, not user time

Debugging of the slowdown

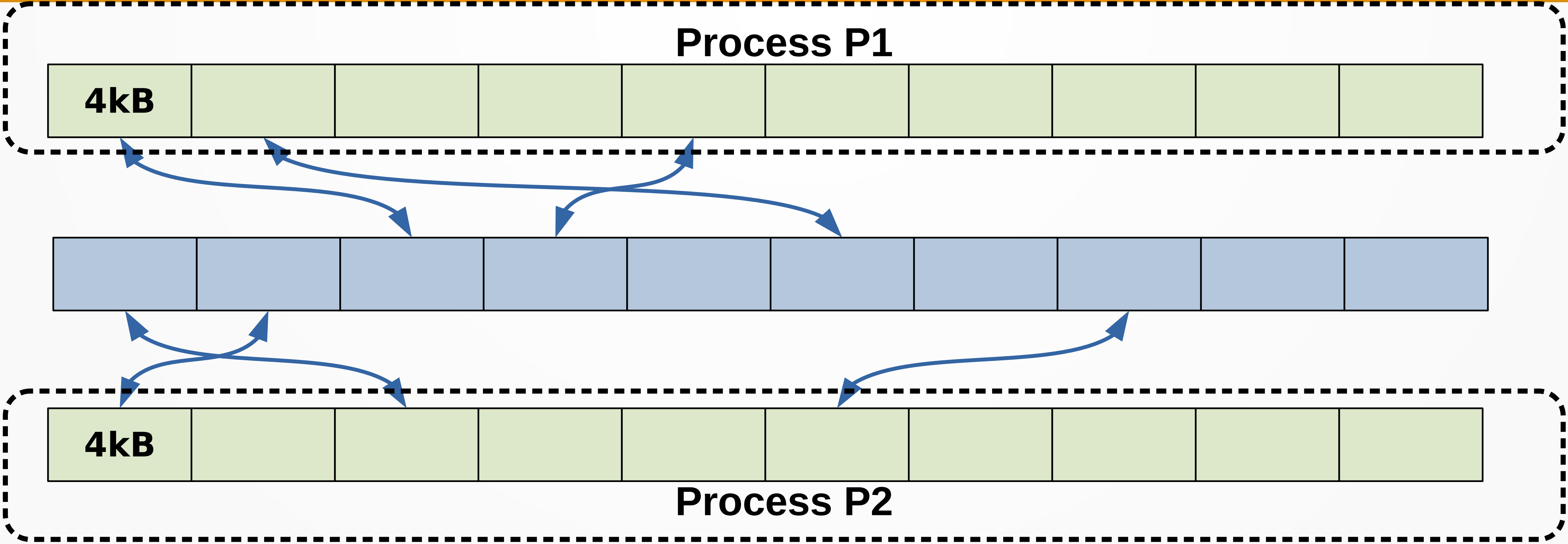
- Debug setup: modify the real program to run the problem step repeatedly or on larger data volume
- Detailed time reporting: mostly system CPU time, not user time
- Profiling shows that the time is spent in the Linux kernel
- Profiling with kernel symbols shows that the time is spent in two sections:
  - NUMA management (memory migration, etc)
  - TLB updates
- What is TLB?

Virtual memory management



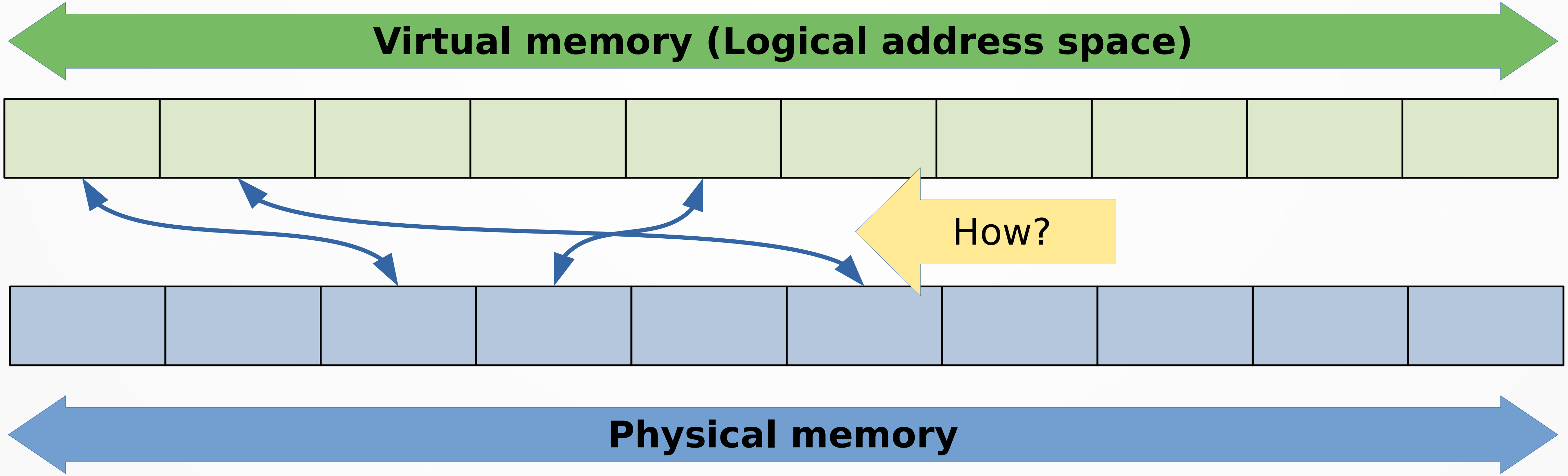
- Only pages that are used are mapped to physical pages
- Physical addresses are arbitrary and not visible to the program

Virtual memory management



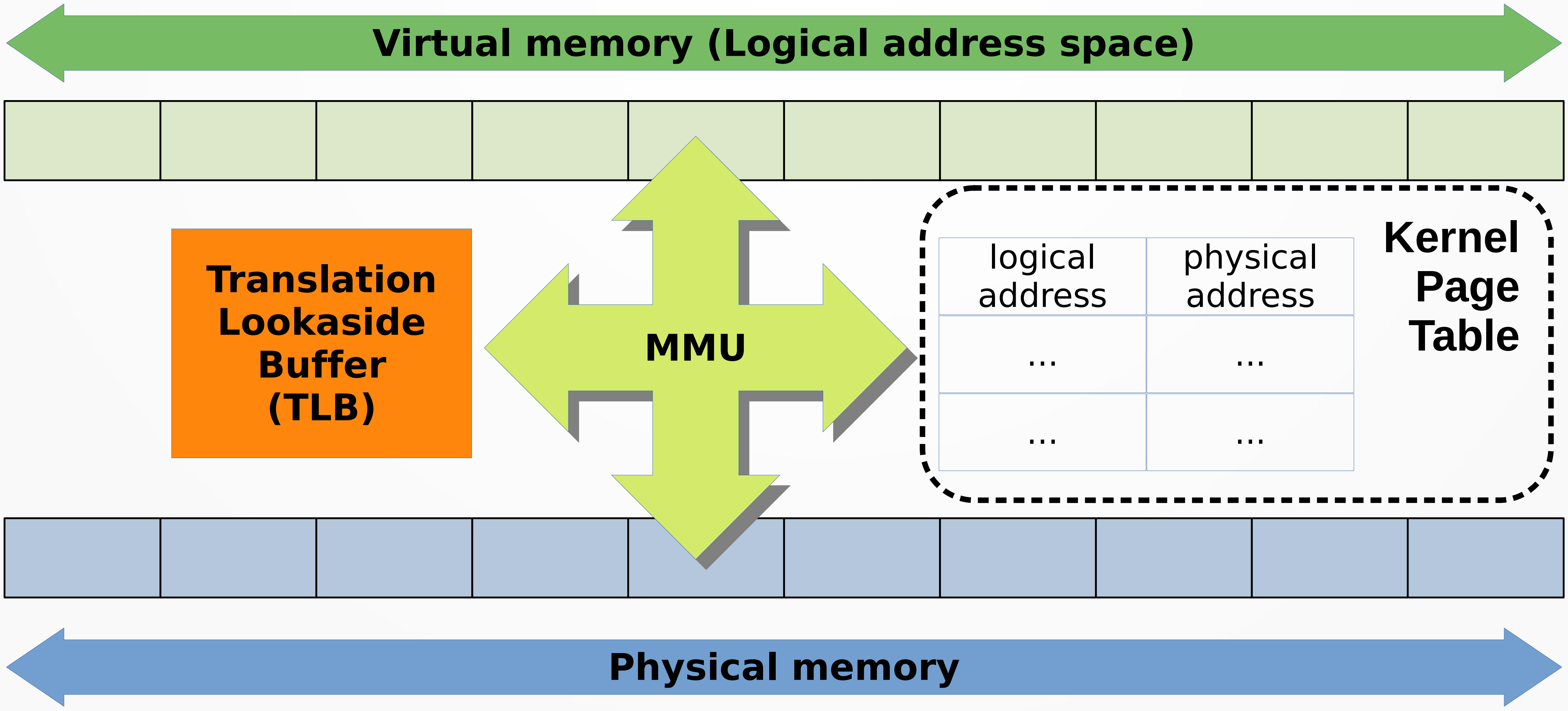
- Each process has its own virtual address space
- All virtual addresses are mapped to physical memory

Virtual memory management



- Mapping of logical to physical addresses is handled by the Memory Management Unit (MMU, part of the processor)
- MMU looks up the address map in the kernel page table

Virtual memory management



Virtual memory (Logical address space)

Translation Lookaside Buffer (TLB)

MMU

Kernel Page Table

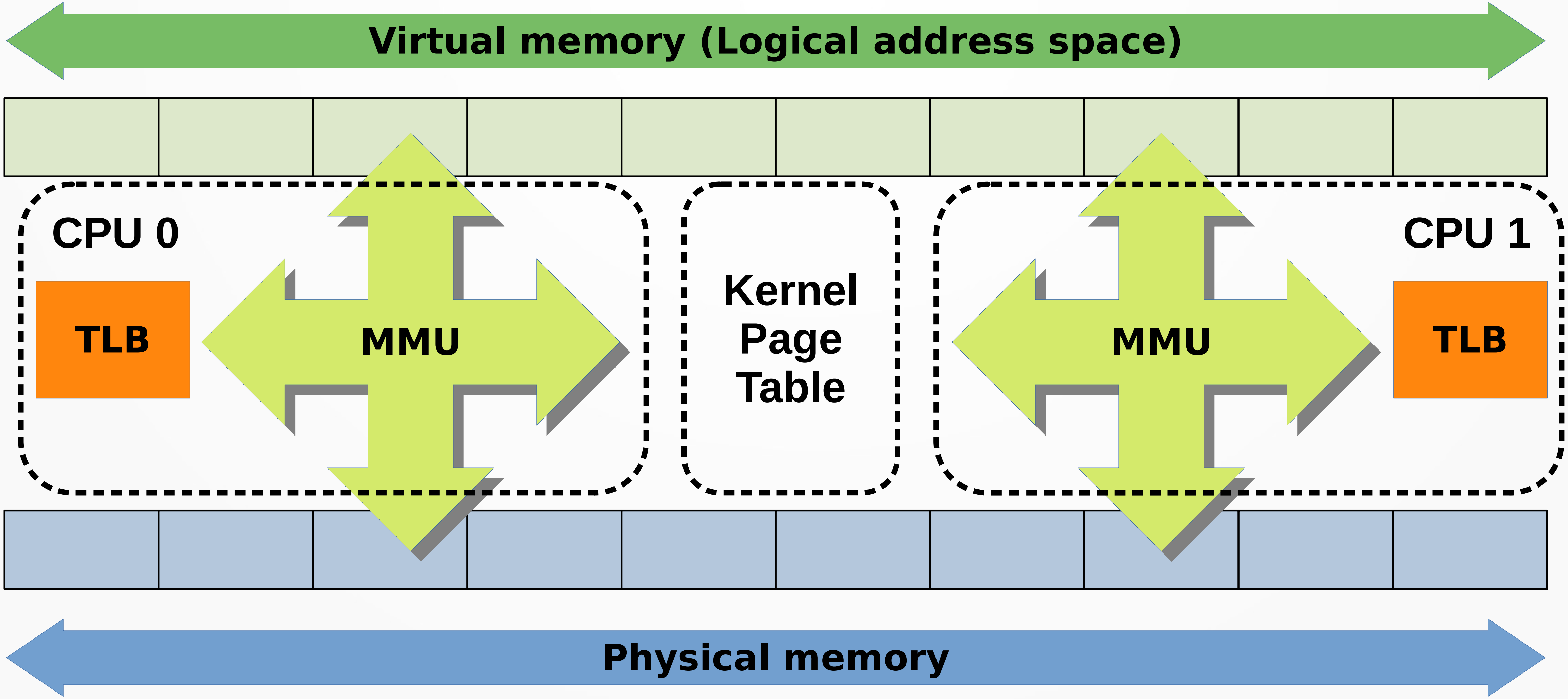
logical address	physical address
…	…
…	…

Physical memory

Virtual memory management

- The OS maintains the complete map of logical to physical addresses for all processes in the page table (per 4K page)
- The CPU has a hardware cache for recently used addresses
  - Translation lookaside buffer (TLB)
  - [Set-]Associative cache, similar to the regular CPU caches
    - Also known as Content-Addressable Memory
- If the address is found in the TLB (TLB hit), the MMU uses it
- If the address is not found in the TLB (TLB miss), the kernel has to walk the page table to find the map
- If the address is not found in the page table, page fault is triggered

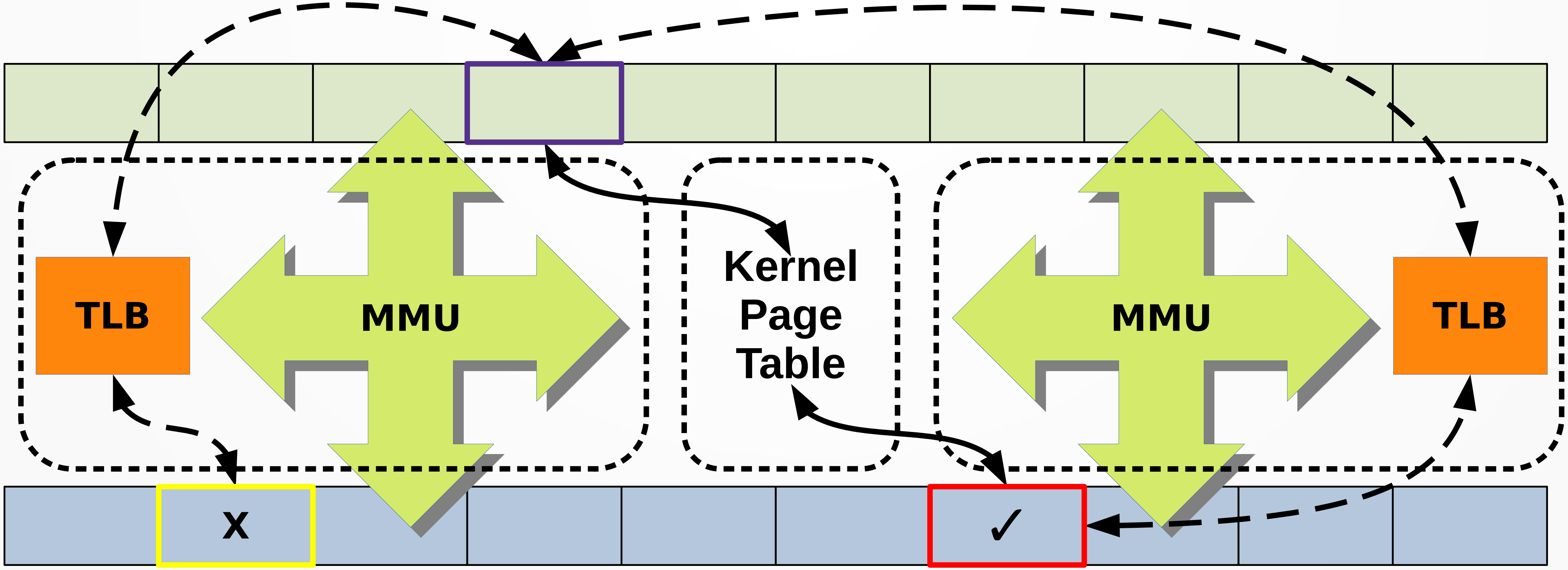
Virtual memory management in NUMA systems



Virtual memory management in NUMA systems

- Logical address space is per process
  - Spans all NUMA nodes
- Physical address space is global
  - Each address belongs to a particular node but accessible by any CPU
- Logical to physical address map (page table) is global
- MMU and TLB are CPU circuits, generally one per node
  - Some architectures have several TLBs in one CPU – not important now
- TLB is similar to other caches: local cache of the global state
- Unlike CPU caches, there is no hardware TLB coherency
- TLB content can differ between CPUs

Virtual memory management in NUMA systems



- If TLB content becomes outdated, the kernel must update it
  - There is no hardware assist for TLB updates

Virtual memory management in NUMA systems

- TLB on different processors can get out of date
  - Unlike caches, there is no hardware TLB coherency
- The kernel updates the TLB using a “TLB shootdown”
  - *“Translation Lookaside Buffer Consistency: A Software Approach,” David L. Black, R. Rashid, D. Golub, C. R. Hill, R. Baron, ASPLOS 1989*
  - Update may affect just one address or many (possibly all)
  - TLB shootdown is an inter-processor interrupt
  - Kernel code runs on the CPU when a shootdown occurs
- TLB shootdown handling can be seen in the profiler (kernel calls) and is accounted as “system time”
- Why do shootdowns happen when that particular code is running?

TLB shootdowns and their causes

- A TLB shootdown is processed by the kernel and interrupts the CPU
- TLB shootdown happens when a TLB is out of date, usually caused by
  - ~~Changes in memory mapping~~
  - ~~Memory access modes and protection~~
  - ~~Explicit changes to page table~~
  - NUMA migrations
- The profiler shows that the problem section spends its time in the kernel calls for TLB management and NUMA load balancing
- Migrating pages from one NUMA node to another changes the physical address and requires TLB updates

Debugging NUMA and TLB shootdowns

- TLB shootdowns cannot be disabled but the profiler can show the time spent in the kernel processing them
  - Recent Intel CPUs have hardware counts for TLB flushes
- NUMA migration can be disabled
  - Bind the program to a single NUMA node
  - Turn off automatic NUMA balancing

    ```
    echo 0 > /proc/sys/kernel/numa_balancing
    ```
- Disabling NUMA migrations “solves” the performance slowdown
  - The overall program generally becomes slower
  - Auto balancing is a global state (affects everyone, needs root)

Reducing the impact of TLB shootdowns

- TLB shootdowns cannot be disabled
- NUMA balancing shouldn’t be disabled
- Why this particular code?
  - Happens in code that allocates and deallocates a lot of memory rapidly
    - The solution is to pre-allocate and reuse memory – we already do that
  - Happens in multi-threaded code that quickly walks over large memory
    - Changing this behavior may be possible but may incur costs elsewhere
- Can we make TLB shootdowns cheaper?
  - It’s mainly the cost of the page table walk (one entry per 4kB)
  - Does it have to be 4kB?

Page table optimization

- Page table problems are not new to NUMA
  - But can be worse in NUMA systems because of TLB inconsistency
- Often the solution is to enable “huge pages” – 2MB page
  - Must be enabled in the OS (usually on by default)

    ```
    # cat /sys/kernel/mm/transparent_hugepage/enabled
    always [madvise] never
    ```

  -
    ```
    echo always > /sys/kernel/mm/transparent_hugepage/enabled
    echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
    ```
  - https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html
- “Madvise” means requested by the program
  - “Always” means the kernel can allocate huge pages as it sees fit
  - In practice, you cannot rely on that and must request huge pages

Using huge pages in your program

- Converting a virtual address range to huge pages:
  `madvise(address, size, MADV_HUGEPAGE);`
  - address and size must be 4kB-aligned (size must be n*2M to be useful)
- The memory must come from `mmap()` private anonymous map
  - `mmap()` is used to reserve large chunks of virtial address space
    - No physical memory is reserved immediately
    - Page table may be optimized for large unmapped regions
    - Physical pages are reserved as page faults are processed
- Using huge pages reduces the number of page table/TLB entries
- Switching to huge pages resolved the observed slowdown

Performance implications of NUMA

- What is NUMA?
- Accessing local memory is faster than accessing non-local memory
- Per-thead memory, data sharing, and cache locality are more important
- True data sharing can be significantly more expensive
- Simple benchmarks with different node binding often point to problems
- Hardware-assisted profiling is useful if you know what to look for
- Use good practices for memory access and node binding only if needed
- [Some] Concurrent data structures need to be redesigned for NUMA
- For I/O-bound programs, use the right node for the data
- There’s going to be something you don’t know about…

Nearly Unfathomable Morass of Arcana (NUMA)

Questions?

Possibly answers too...