HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 9

Fabian Wermelinger
Harvard University
CS205

Tuesday, February 21st 2023

LAST TIME

- Combined constructs and lexical extent of parallel regions
- Data environment in OpenMP: shared and private data attributes?
- Synchronization primitives in OpenMP
- OpenMP library routines and environment variables

TODAY

Main topic: ***Different types of shared memory architectures, thread affinity, cache coherency and false sharing***

Disclaimer: all references to the OpenMP specification are for version 5.1

Details:

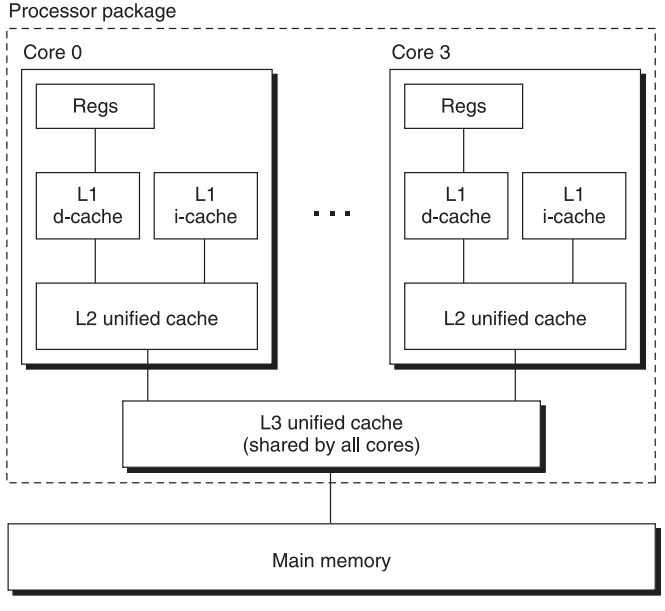
- Uniform and non-uniform memory access (UMA and NUMA)
- Thread affinity and the NUMA first-touch policy
- Cache coherency protocols in shared memory (the missing piece in the memory consistency model)
- False sharing

AGENDA CHECK:

- Project milestone 2 due tonight → https://harvard-iacs.github.io/2023-CS205/pages/project.html#M2
- Reading assignment for Thursday is about the Roofline performance analysis model by Williams et al. → https://code.harvard.edu/CS205/main/blob/master/reading/03_williams2009a.pdf

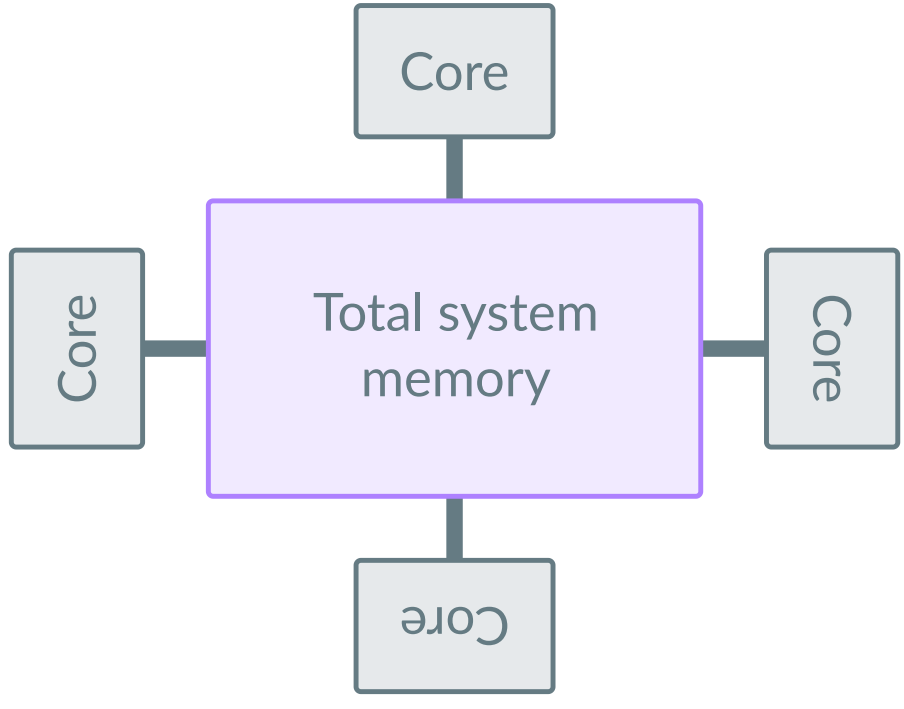
UNIFORM MEMORY ACCESS

Recall: this sketch from lecture 4:



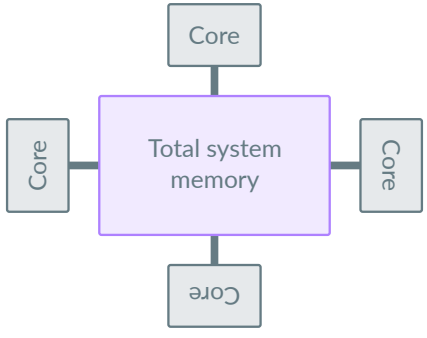
- Multiple cores with one memory management unit
- All cores in that architecture share the system bus on the chip
- The cores have uniform memory access

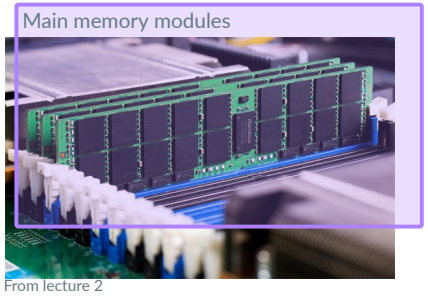
Uniform memory access (UMA)



- Also known as *symmetric multi-processing* (SMP)
- The *cost* to access the memory is the same for each core

UNIFORM MEMORY ACCESS

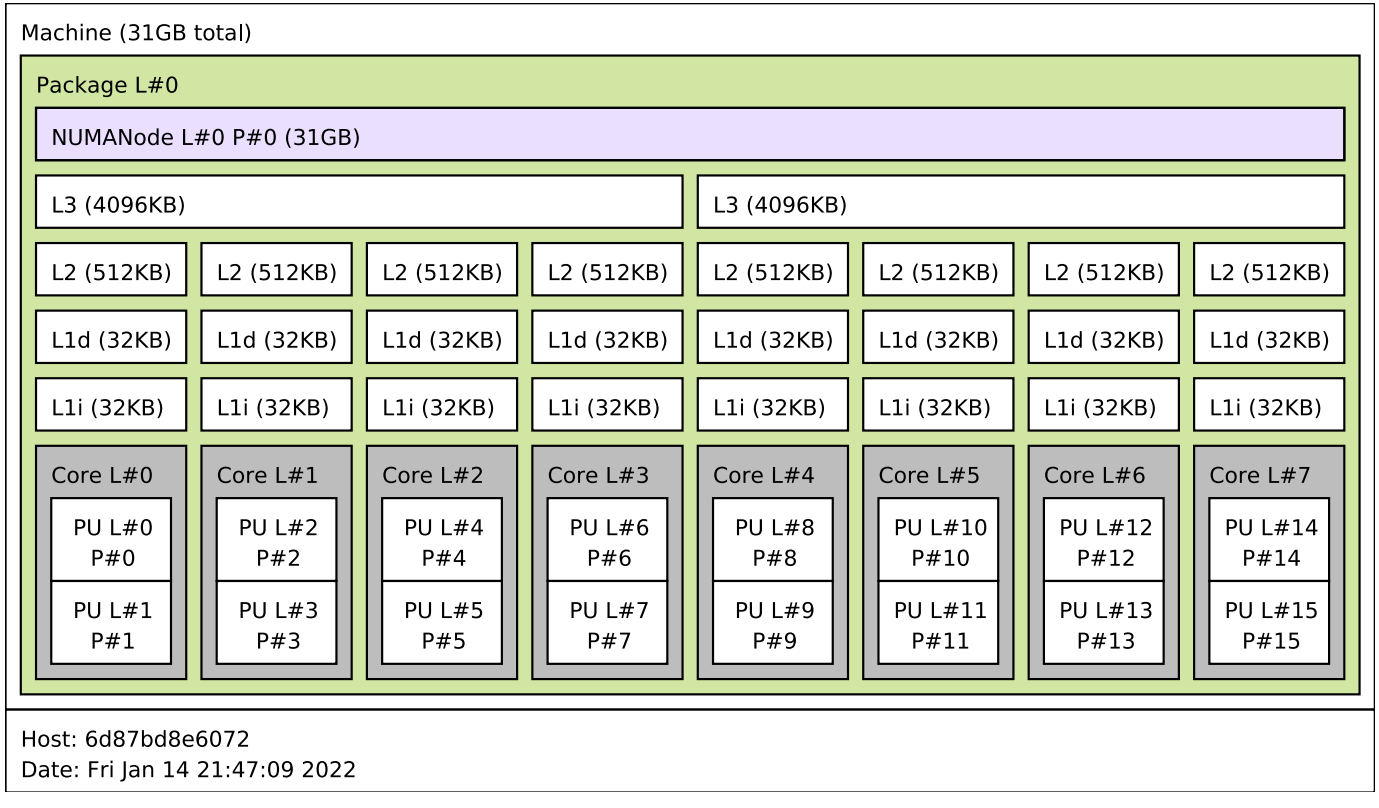




From lecture 2

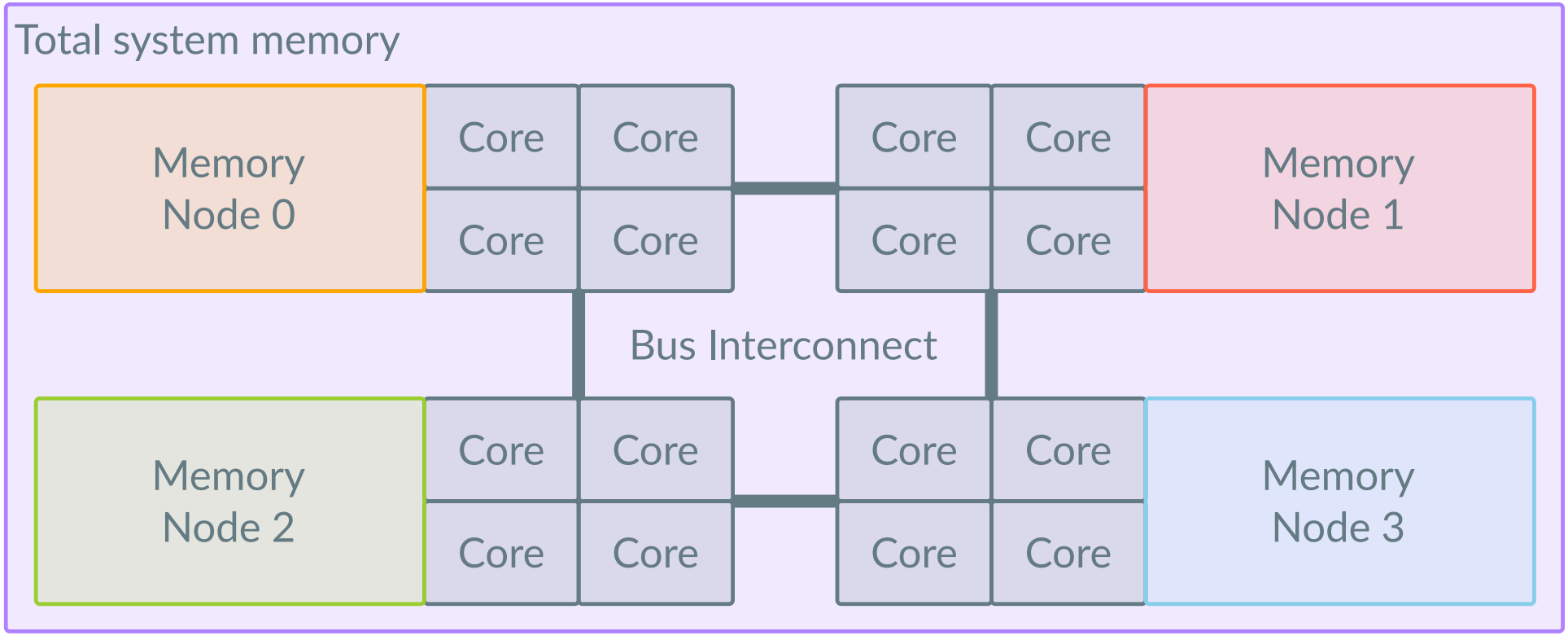
Main memory modules in a standard system associated with a (N)UMA node.

UMA Example: most laptops and consumer grade desktops



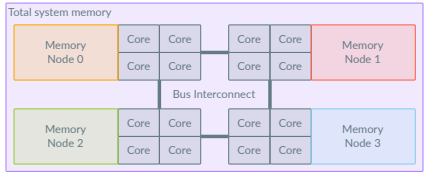
- AMD Ryzen 7 PRO 4750U on my laptop is an UMA system
- There is *one* (N)UMA node with direct access to the total system memory (32GB)
- The span of the (N)UMA node block across *all* cores in the figure above indicates that these cores have *equal* access to that particular memory node

NON-UNIFORM MEMORY ACCESS



- In the image above, the architecture could be a motherboard with 4 sockets, each hosts a 4-core CPU with *its own* memory management unit (MMU). It could also be a 16-core CPU architecture with 4 internal memory nodes, each exclusively assigned to 4 cores.
- In either case, *all* cores have access to all of the system memory.
- The *access cost* is *non-uniform*. For example, a core attached to memory node 0 can directly access its memory node but cannot directly access memory node 3.

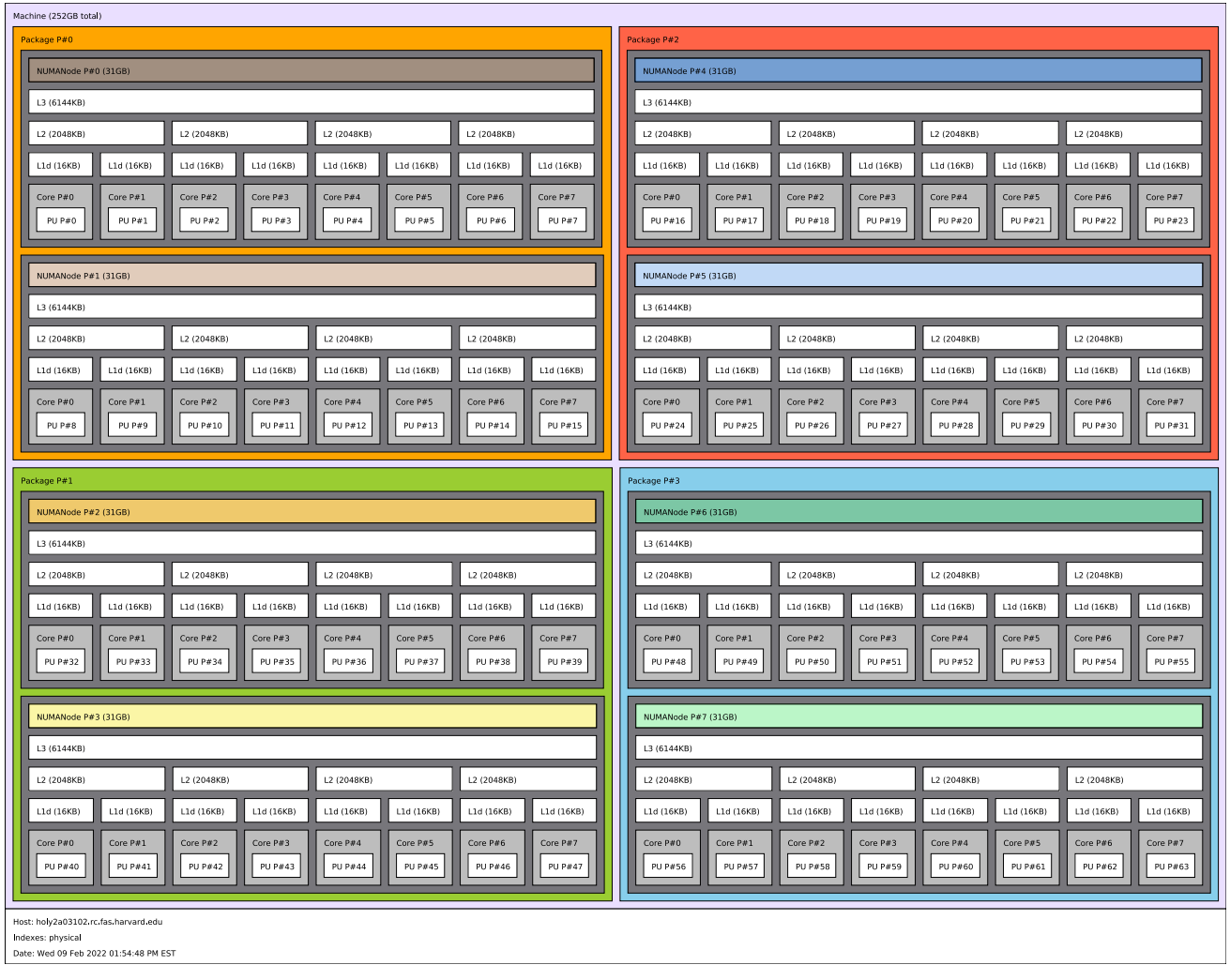
NON-UNIFORM MEMORY ACCESS



Recall: lab 1

- *Compute node:* 2 Intel Xeon E5-2683 v4 CPUs on one motherboard (2 sockets)
- One CPU has 1 NUMA node, each with *direct* access to 64 GB DRAM
- Total node memory is 128 GB managed by 2 NUMA nodes

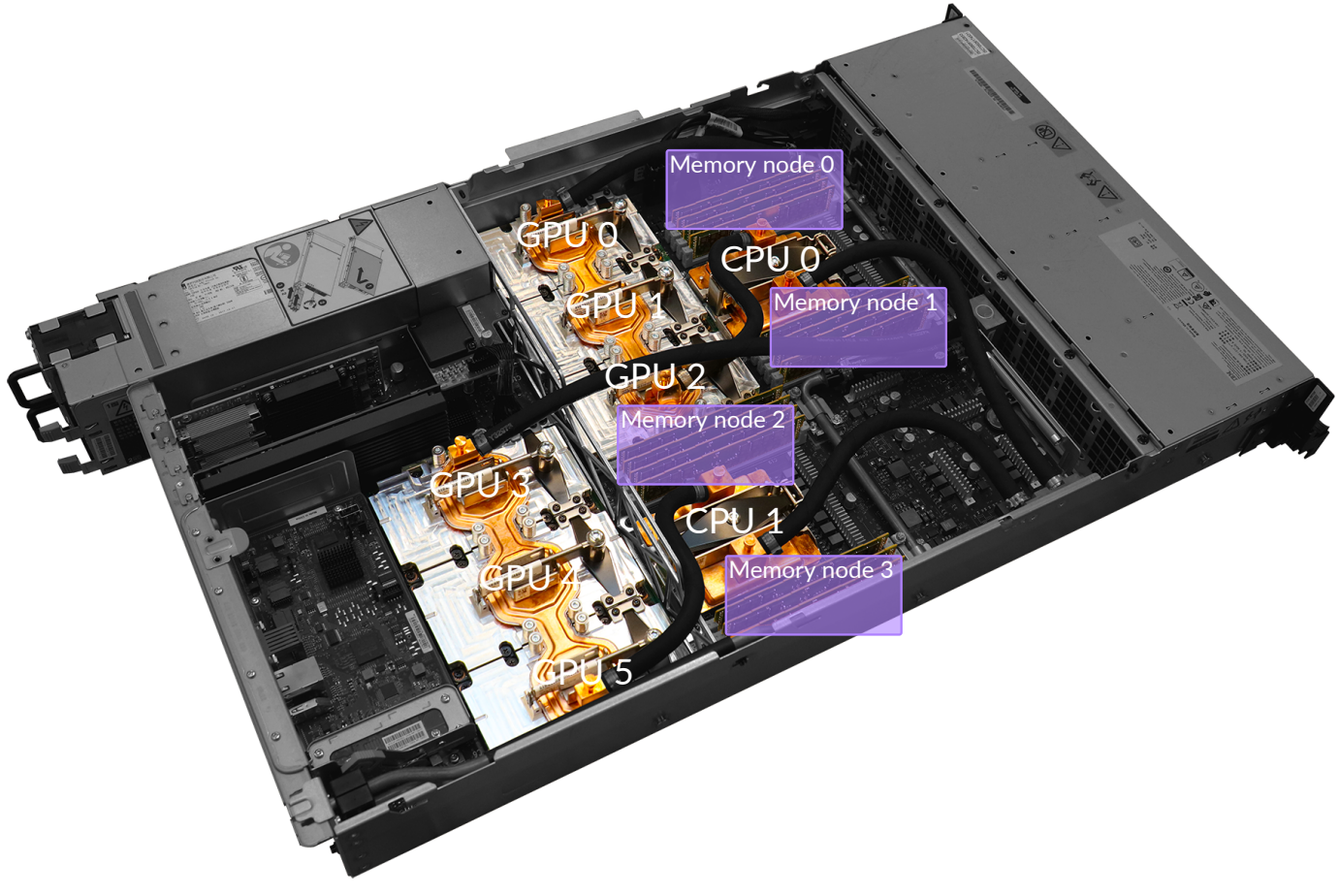
NUMA Example: old compute nodes (last year) on academic cluster: 4 AMD Opteron 6376 CPUs on one motherboard (4 sockets!)



NON-UNIFORM MEMORY ACCESS

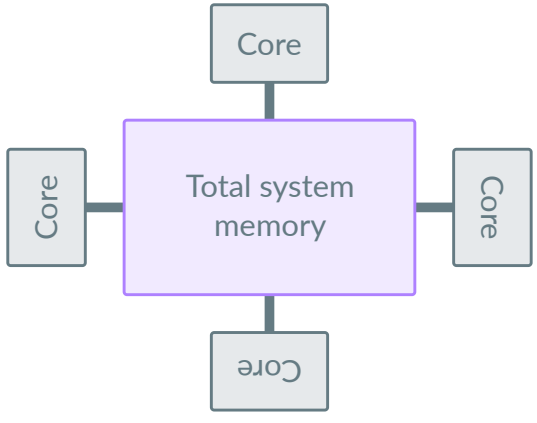
- Summit supercomputer at Oak Ridge National Labs
- 4'608 compute nodes
- 2 IBM Power9 CPUs and 6 Nvidia V100 GPUs per node
- 512 GB host memory per node

- Nodes are NUMA with 4 memory nodes
- Each memory node hosts 128 GB of memory



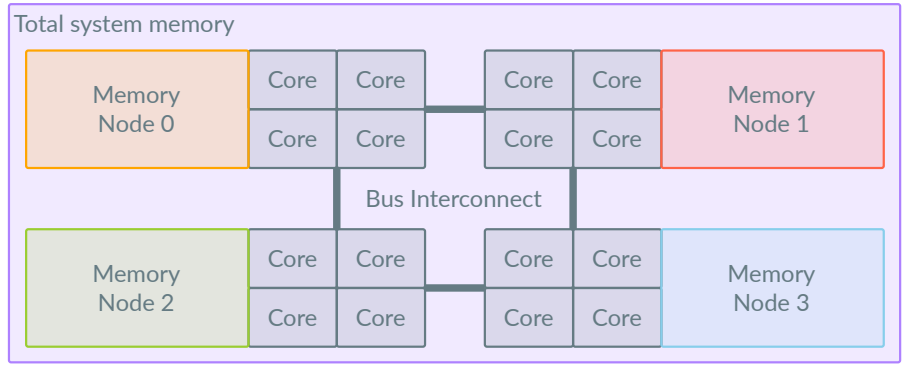
UMA/NUMA DIFFERENCES

Uniform Memory Access (UMA):



- UMA typically supports *a few cores* only (e.g. up to 8 cores)
- Your laptop or desktop at home are UMA systems
- It can only be a few cores because the system bus must be able to service the memory requests for *all cores*
- UMA is also referred to as *symmetric multi-processing* (SMP)

Non-Uniform Memory Access (NUMA):



- Supporting *a large number of processors* requires distributed shared memory (distributed locally on the hardware → *not the same as MPI!*)
- Otherwise extreme pressure on the bus resulting in high latencies
- NUMA architectures usually have multiple sockets with separate memory modules
- NUMA is also referred to as *distributed shared memory* (DSM)

NUMA PLATFORMS

- Both, UMA and NUMA are *shared memory* architectures. Cores on these platforms share the same memory space.
- The total memory space *may be distributed* locally on the motherboard (physical separation). This physical separation is *transparent* from the software point of view.
- Distributing the memory *increases the bandwidth* (why?) and *reduces latency* to local memory.
- Memory requests to a *remote* NUMA node are associated with *higher latency* (they are more expensive) than requests to the local NUMA node.
  → *therefore, efficient use of NUMA platforms requires additional effort in your software.*

Note: distributed programming with MPI is very different. It is not possible to access the remote memory of the distributed process without a *protocol* that runs on both processes. The specification of such a protocol is essentially what MPI is. On a NUMA platform you can access a remote memory node (on the same motherboard, of course). This is a nice read if you are further interested: https://queue.acm.org/detail.cfm?id=2513149

NUMA PLATFORMS

A common code pattern:

```
1  int main(void)
2  {
3      // 1. get dynamic memory from allocator
4      double *A = new double[1000];
5
6      // 2. initialize data
7      for (int i = 0; i < 1000; ++i) {
8          A[i] = 0.0;
9      }
10     // other code...
11
12     // 3. perform work in parallel
13 #pragma omp parallel for
14     for (int i = 0; i < 1000; ++i) {
15         A[i] = do_work(i, A);
16     }
17
18     // clean up
19     delete[] A;
20     return 0;
21 }
```

The code on the left shows common steps performed when working with plain old data (POD). It could be data that you manage in some C++ class or any other user defined types where you control how data is *initialized*. The steps are:

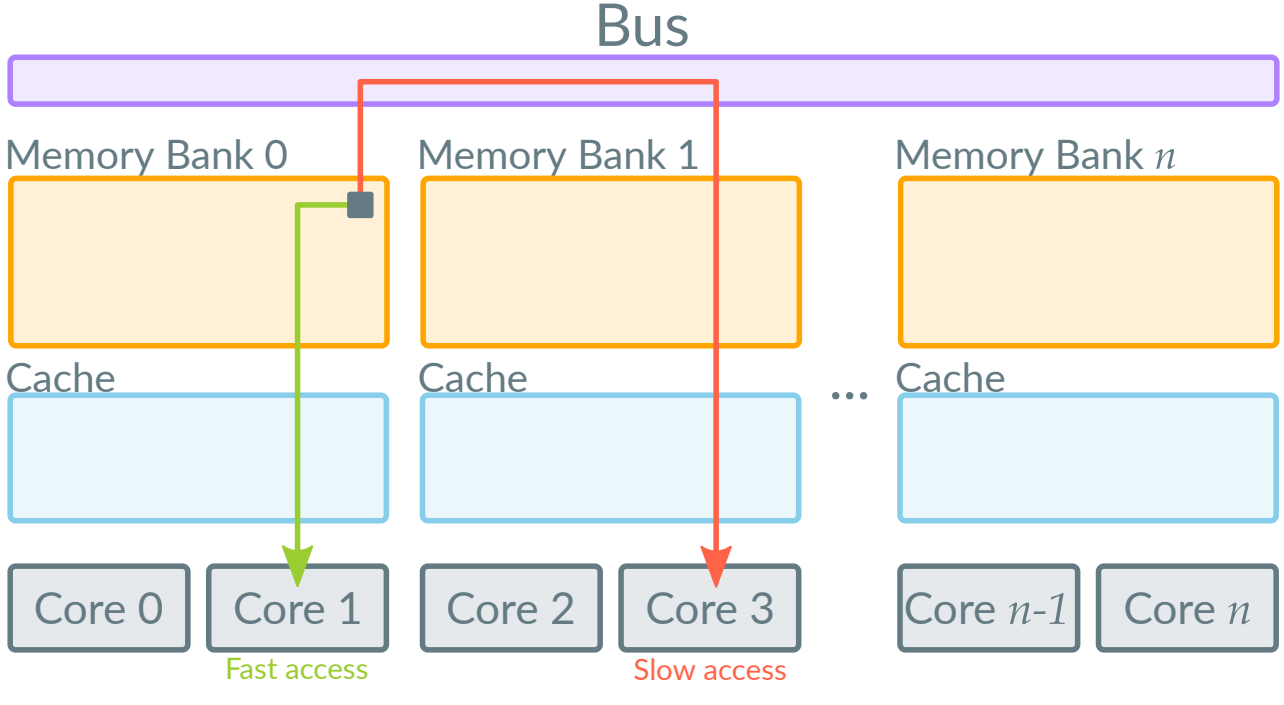
1. Allocate a chunk of *dynamic* memory (memory on the heap)
2. Initialize the data structures to some values
3. Partition the data among threads and perform work in parallel

Assume we are working on a NUMA platform. In which of the memory modules will the chunk of memory allocated in step 1 be placed?

NUMA FIRST TOUCH POLICY

Memory affinity is not decided by the memory allocation *but by initialization!*
First touch policy: memory is mapped to the NUMA domain that first touches it.

- If core 1 first touched the data element shown in the figure, it will be placed in a memory bank of the NUMA domain associated with core 1
- If another core not in this NUMA domain accesses the data element, it will be a *slower access* (higher latency since the memory you access is physically further away)



NUMA FIRST TOUCH POLICY

NUMA first touch with OpenMP:

```
1  int main(void)
2  {
3      // 1. get dynamic memory from allocator
4      double *A = new double[1000];
5
6      // 2. initialize data: first touch policy
7  #pragma omp parallel for
8      for (int i = 0; i < 1000; ++i) {
9          A[i] = 0.0;
10     }
11     // other code...
12
13     // 3. perform work in parallel
14 #pragma omp parallel for
15     for (int i = 0; i < 1000; ++i) {
16         A[i] = do_work(i, A);
17     }
18
19     // clean up
20     delete[] A;
21     return 0;
22 }
```

1. Allocate a chunk of *dynamic* memory (memory on the heap). This chunk of memory has not been touched yet (not initialized).
2. Initialize the data in parallel using a specific team of threads.
3. Partition the data among threads and perform work in parallel. The same team of threads should be used here (same configuration, be aware of `OMP_DYNAMIC`).

Note: depending on how the data `A` is accessed inside `do_work`, you might access memory in different memory banks. A different data representation that is better aligned with the algorithm you use may be more beneficial.

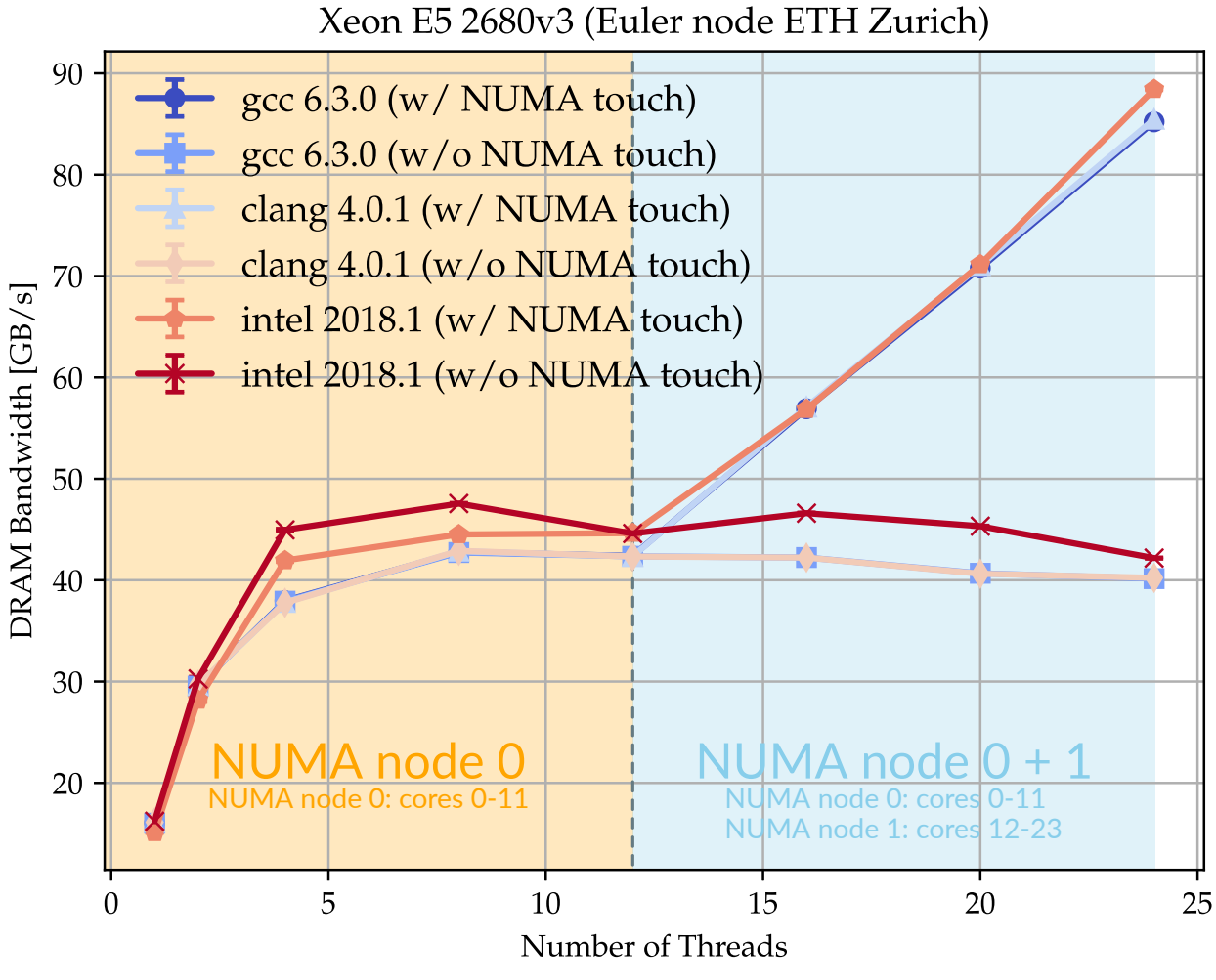
NUMA FIRST TOUCH POLICY

Be careful with C++ container types:

```
1  #include <vector>
2
3  int main(void)
4  {
5      // this constructor will initialize the data to 0.0
6      std::vector<double> A(1000, 0.0);
7
8      // here it depends on what the default constructor does
9      std::vector<double> A(1000);
10
11     // might not be first touch!
12 #pragma omp parallel for
13     for (int i = 0; i < 1000; ++i) {
14         A[i] = 0.0;
15     }
16     // other code...
17
18     // perform work in parallel
19 #pragma omp parallel for
20     for (int i = 0; i < 1000; ++i) {
21         A[i] = do_work(i, A);
22     }
23
24     return 0;
25 }
```

- C++ uses RAII which means that container types like `std::vector` initialize the data.
- If the container initializes the data, then the first touch policy will be violated.
- It may further depend on the constructor you use.
- The constructor used in line 6 on the left will definitely violate the first touch policy.
- The constructor used in line 9 on the left *may or may not* violate first touch, depending on its implementation → *do not rely on it*.

NUMA FIRST TOUCH POLICY ON ETHZ CLUSTER



- Two Intel Xeon E5 2680v3 CPUs with 12 cores each
- Without the first touch policy you waste half of your bandwidth because accesses to the distant NUMA node cost about twice as much on this architecture!
- The degraded bandwidth will not be enough to service the data requests on all 24 cores on this node.

Data obtained with STREAM benchmark: https://www.cs.virginia.edu/stream/ref.html

THREAD AFFINITY

- On NUMA architectures (and in general) it is important that we can control on which cores threads will run.
- Thread affinity refers to the thread–core mapping at runtime.
- OpenMP supports processor binding (thread affinity) since the v3.0 specification.
- By default, the OpenMP runtime decides on which cores threads run.

The easiest way to control affinity is via environment variables: see Section 2.6.2

```
OMP_PROC_BIND='true'  # OpenMP threads are bound to their initial place. Thread affinity is enabled
OMP_PROC_BIND='false' # The runtime environment may move threads to different places. Thread affinity is disabled
```

Specific places:

```
OMP_PROC_BIND='master'  # At the same place as the primary thread (deprecated)
OMP_PROC_BIND='primary' # Replaces `master` in v5.1
OMP_PROC_BIND='close'   # On different places close to each other
OMP_PROC_BIND='spread'  # On different places spread across possible places
```

Nested parallelism:

```
OMP_PROC_BIND='close, spread, spread' # Defines affinity on 3 levels of nested parallelism
```

THREAD AFFINITY

You can also be explicit in your code by using an OpenMP clause:

```
1 #pragma omp parallel proc_bind(master|primary|close|spread) [other clauses] new-line
2     structured block
```

primary
: This thread affinity policy instructs the execution environment to assign every thread in the team to the *same place* as the `primary` thread.

master
: Same as `primary`. Deprecated since OpenMP v5.1.

close
: This thread affinity policy instructs the execution environment to assign the threads in the team to places *close* to the place of the parent thread (may not necessarily be the `primary` thread in nested parallelism).

spread
: This thread affinity policy creates a *sparse* distribution of the threads in the available places.

If

```
1 OMP_PROC_BIND='false'
```

then `proc_bind` clauses on `parallel` constructs *will be ignored*. See Section 6.4.

Use this environment variable to display the thread affinity setting at runtime:

```
1 OMP_DISPLAY_AFFINITY='true'
```

See Section 6.13

PLACES

You can control the granularity of thread affinity via OpenMP places:

```
1 OMP_PLACES='list of places'
```

There are *many* possibilities to define places in OpenMP, see Section 6.5 for all details.

There are a few abstract names available that are useful. They are listed below with *increasing granularity* (from fine grained control to coarse grained control):

`threads`
: Each place corresponds to a single *hardware thread* on the device.

`cores`
: Each place corresponds to a single core (having one or more hardware threads) on the device.

`ll_caches` (v5.1+ only)
: Each place corresponds to a set of cores that share the last level cache on the device.

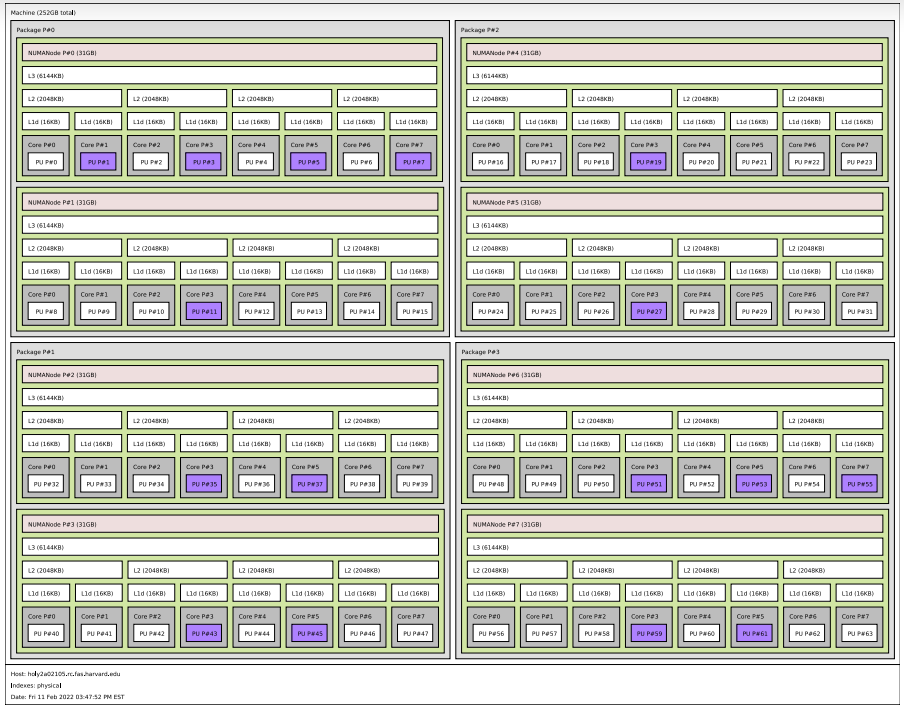
`numa_domains` (v5.1+ only)
: Each place corresponds to a set of cores for which their closest memory on the device is: *the same memory and at a similar distance from the cores. OpenMP v5.1 and beyond only.*

`sockets`
: Each place corresponds to a single socket (consisting of one or more cores) on the device.

THREAD AFFINITY: EXAMPLES

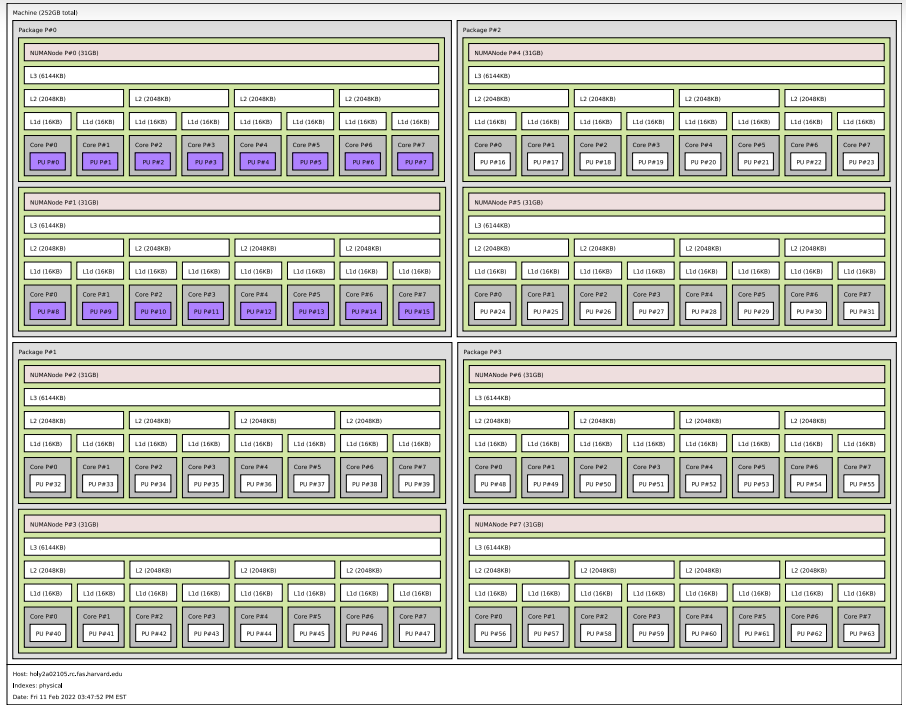
OMP_PROC_BIND – compute node on the academic cluster:

```
1  $ OMP_NUM_THREADS=16 OMP_PROC_BIND='false' ./affinity
```



Running this command again generates a *different* mapping every time.

```
1  $ OMP_NUM_THREADS=16 OMP_PROC_BIND='true' ./affinity
```



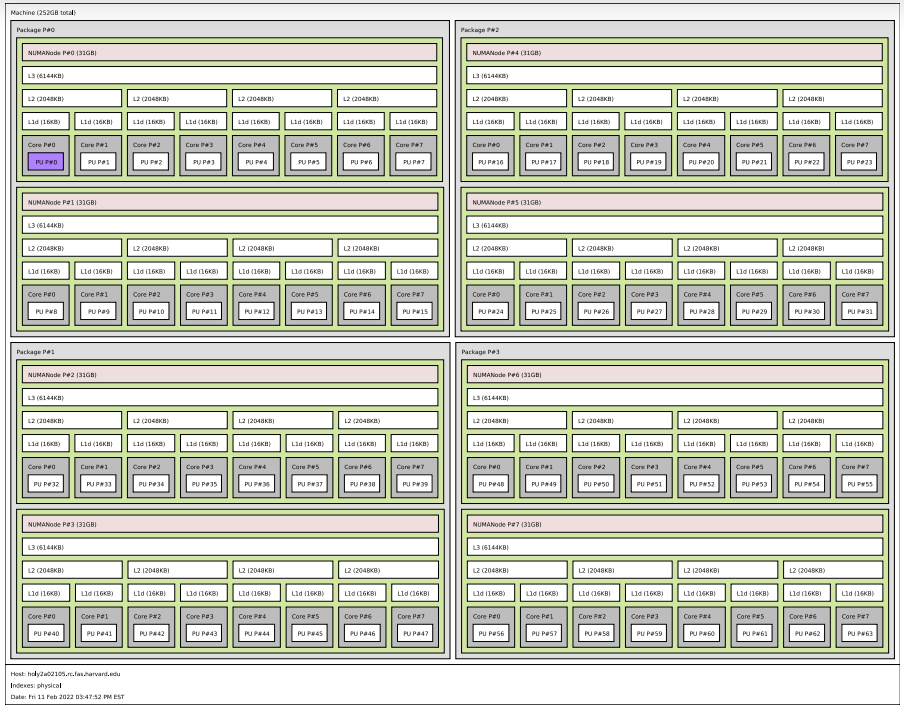
Running this command again generates the *same* mapping. ***Note:*** the following generates the same mapping:

```
1  $ OMP_NUM_THREADS=16 OMP_PROC_BIND='close' ./affinity
```

THREAD AFFINITY: EXAMPLES

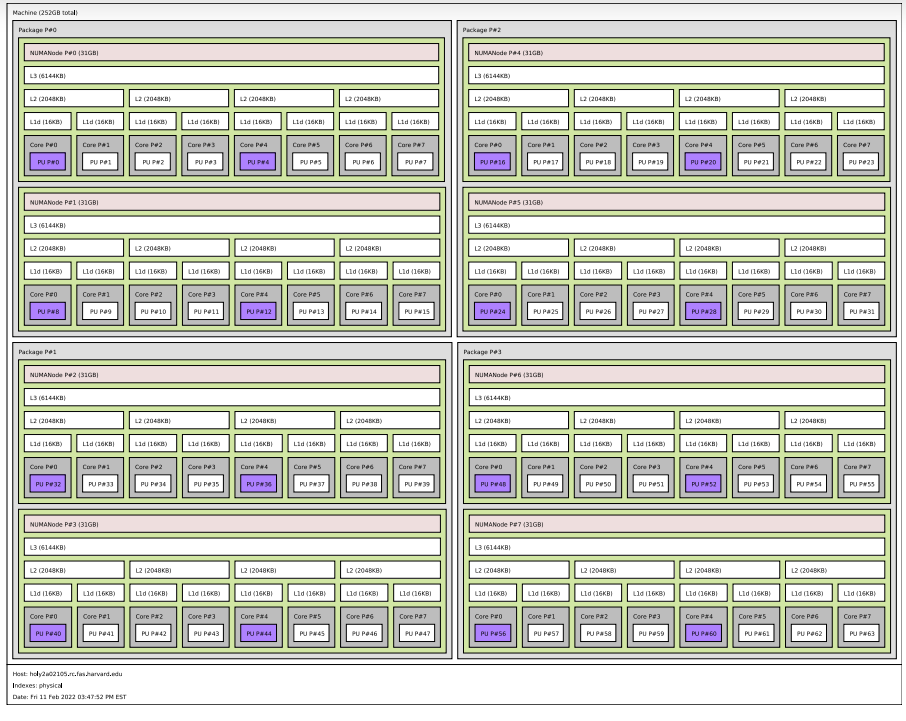
OMP_PROC_BIND – compute node on the academic cluster:

```
1  $ OMP_NUM_THREADS=16 OMP_PROC_BIND='master' ./affinity
```



Over-subscription: All 16 threads map to *the same* physical core!

```
1  $ OMP_NUM_THREADS=16 OMP_PROC_BIND='spread' ./affinity
```

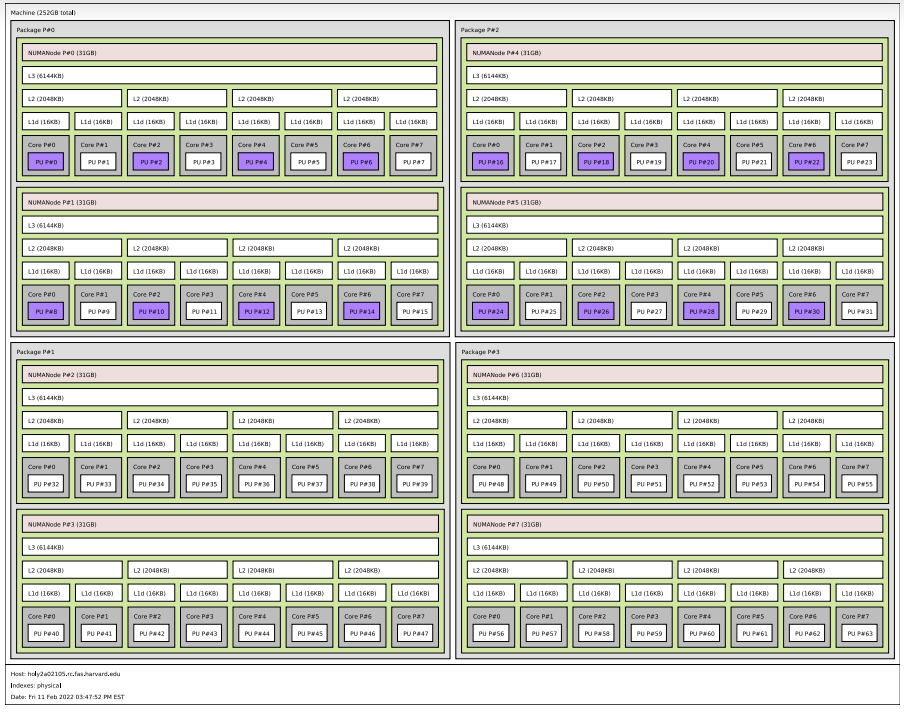


Running this command again generates the *same* mapping every time.

THREAD AFFINITY: EXAMPLES

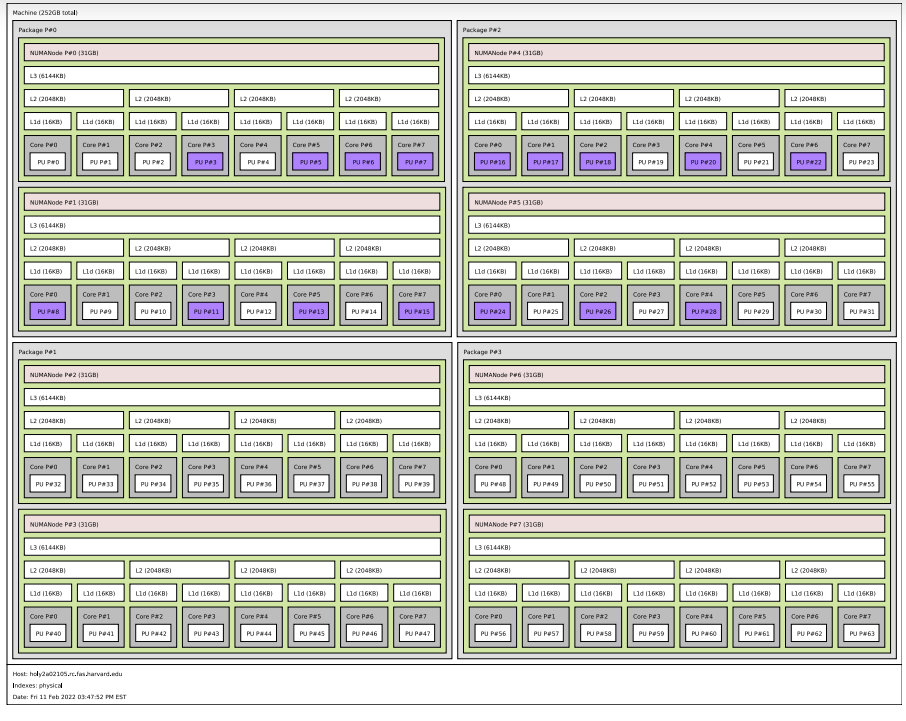
OMP_PLACES – compute node on the academic cluster:

```
$ OMP_NUM_THREADS=16 OMP_PLACES='cores' ./affinity
```



Running this command again may generate a *different* mapping. Only the places will be respected. Thread ID and core ID order may also not be monotonic.

```
$ OMP_NUM_THREADS=16 OMP_PLACES='sockets(2)' ./affinity
```



Running this command again generates a *different* mapping every time on the same two sockets. `OMP_PLACES='sockets'` is allowed to use all sockets.

CACHE COHERENCY

- The memory consistency model we discussed ensures a correct order of read/write operations such that the program is data-race free. (If it is a relaxed consistency model you are responsible!)
- We still miss a piece in that model because our system uses *cache memories*.
- What if multiple threads need access to data that is located in the same cache line? The memory consistency model we discussed so far is not enough!
- Threads run on different cores which have their exclusive L1 data caches. They [threads] will have their own copies of the same cache line which may become *incoherent* as data is modified with multiple threads.
- A *cache coherence protocol* is required that ensures the following properties:
  1. ***Write propagation:*** changes to data in a cache line must be propagated to *anyone* who holds a copy of the same cache line.
  2. ***Write serialization:*** writes to a *single memory location* must be seen by all threads in *the same order*. If thread 0 writes to a memory location, thread 1 must see that write in its cache line first before it can write to same memory location.

CACHE COHERENCE PROTOCOLS

- There are a number of different cache coherency protocols. Examples are:
  - ***MESI:*** **M**odified, **E**xclusive, **S**hared, **I**nvalid (most common, a.k.a. Illinois protocol)
  - ***MOESI:*** **M**odified (exclusive), **O**wner (shared), **E**xclusive, **S**hared, **I**nvalid (AMD)
  - ***MESIF:*** **M**odified, **E**xclusive, **S**hared, **I**nvalid, **F**orward (Intel)
- Mostly *two* different policies are used for a coherence protocol (see bus snooping):
  - ***Invalidation-based:*** only write misses hit the bus, subsequent writes to the same cache line are local (works with write-back caches). Good for multiple writes to the same cache line in the same cache.
  - ***Update-based:*** all sharers continue to hit cache line after one core writes (assumes that shared lines are accessed often). Many local writes may waste bandwidth.
- MESI is an ***invalidation-based*** protocol

MESI

Terminology:

Clean line
: Content of cache line and main memory is identical, i.e., main memory is *up to date* (such a line can be evicted without write-back).

Dirty line
: Content of cache line and main memory differ, i.e., main memory is *stale* (the line needs to be written back eventually, *when* that happens depends on the protocol details).

Bus transaction
: A signal on the bus that can be observed by all caches. Usually a *blocking* operation, i.e., not parallel.

Local read/write
: A load/store operation that originates from a core connected to the cache.

Cache line states in MESI:

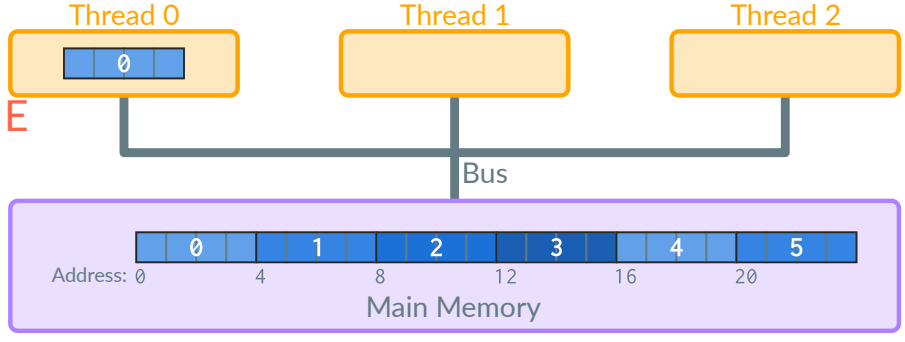
Modified (*M*)
: The *local cache line copy* has been modified and *no copies* exist in other caches → *Main memory is stale.*

Exclusive (*E*)
: No copies exist in other caches → *The main memory is up to date.*

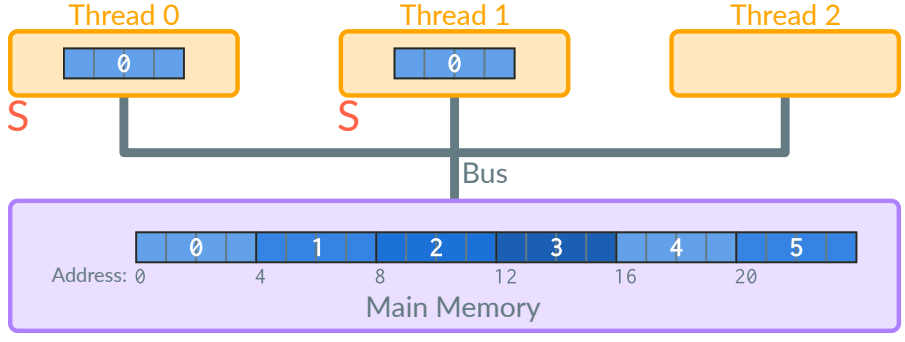
Shared (*S*)
: *Unmodified* copies *may* exist in other caches → *Main memory is up to date.*

Invalid (*I*)
: Cache line is incoherent and different from what is currently in the cache.

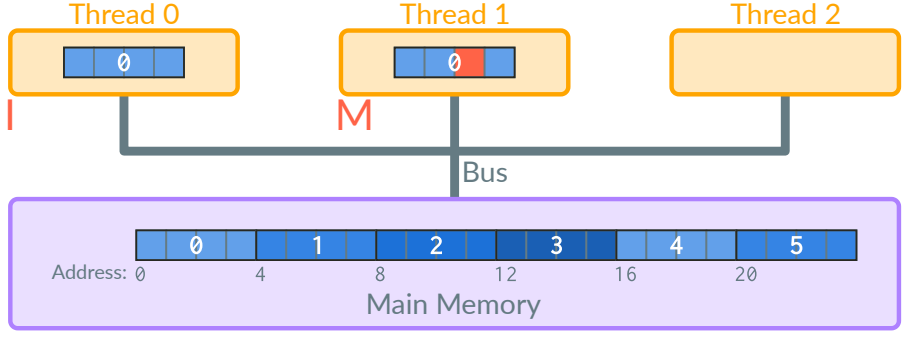
MESI EXAMPLE



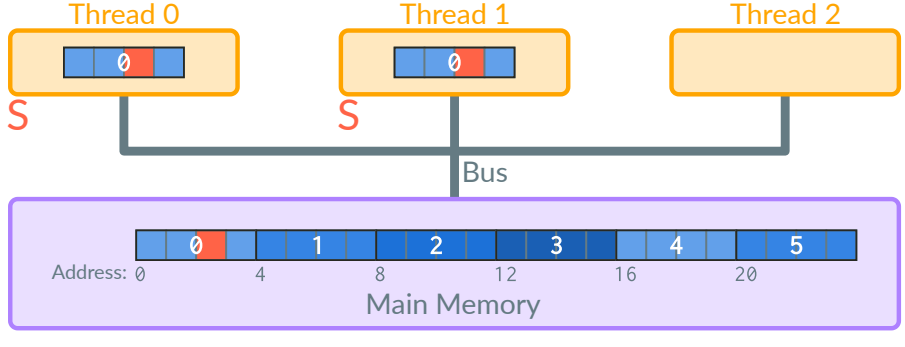
- Thread 0 requests read access to some data contained in cache line 0.
- The cache line will be marked exclusive for that thread (**E**).



- Thread 1 requests read access to some data contained in cache line 0.
- The cache line is now marked shared (**S**). Both threads have a *copy* of the same cache line.



- Thread 1 modifies its local cache line copy (**M**).
- The coherency protocol signals this by marking the cache line of thread 0 invalid (**I**).



- The shared state is restored by propagating the change (**S**). Depending on the protocol:
  - The change may be written to main memory and invalid cache lines are updated (MESI).
  - Other protocols may broadcast changes and not write to main memory immediately.

PARALLEL π

Suppose we are given the following sequential code that approximates

$$\pi = 4 \int_0^1 \frac{1}{1 + x^2} \, \mathrm{d}x$$

```cpp
#include <iostream>

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum = 0.0;

    for (int i = 0; i < nsteps; ++i) {
        const double x = (i + 0.5) * step;
        sum += 1.0 / (1.0 + x * x);
    }

    std::cout << "pi = "
              << 4.0 * sum * step << '\n';
    return 0;
}
```

How would you go about parallelizing this code with OpenMP?

PARALLEL π

A proposed solution:

```cpp
#include <iostream>
#include <omp.h>
#define NTHREADS 8

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum[NTHREADS] = {0};

#pragma omp parallel num_threads(NTHREADS)
    {
        const int tid = omp_get_thread_num();

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            sum[tid] += 1.0 / (1.0 + x * x);
        }
    }

    // reduce per thread sum
    for (int i = 1; i < NTHREADS; ++i) {
        sum[0] += sum[i];
    }
    std::cout << "pi = "
              << 4.0 * sum[0] * step << '\n';
    return 0;
}
```

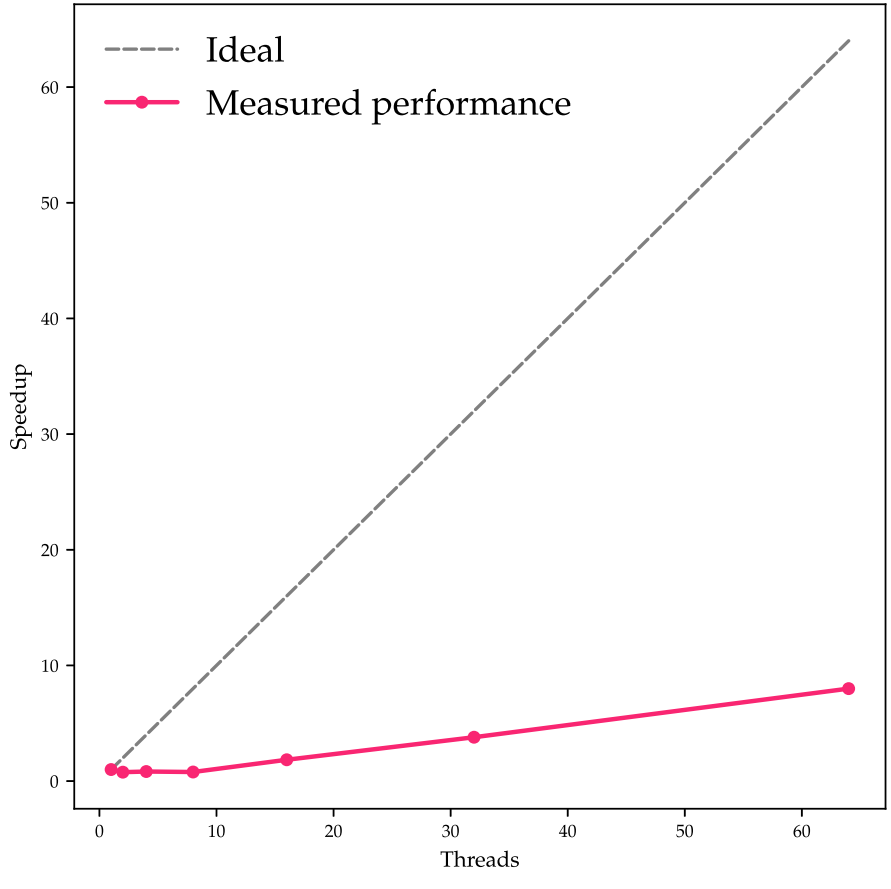
1. Initialize storage for per thread partial sums.
2. Parallelize the loop with an OpenMP worksharing construct. There is no need to wait at the implicit barrier.
3. Each thread updates its partial sum in the allocated storage *(this is not a race condition!)*.
4. Primary thread reduces the partial sums to the final result after the parallel region.

PARALLEL π

A proposed solution:

```cpp
#include <iostream>
#include <omp.h>
#define NTHREADS 8

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum[NTHREADS] = {0};

#pragma omp parallel num_threads(NTHREADS)
    {
        const int tid = omp_get_thread_num();

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            sum[tid] += 1.0 / (1.0 + x * x);
        }
    }

    // reduce per thread sum
    for (int i = 1; i < NTHREADS; ++i) {
        sum[0] += sum[i];
    }
    std::cout << "pi = "
              << 4.0 * sum[0] * step << '\n';
    return 0;
}
```

Performance benchmark:



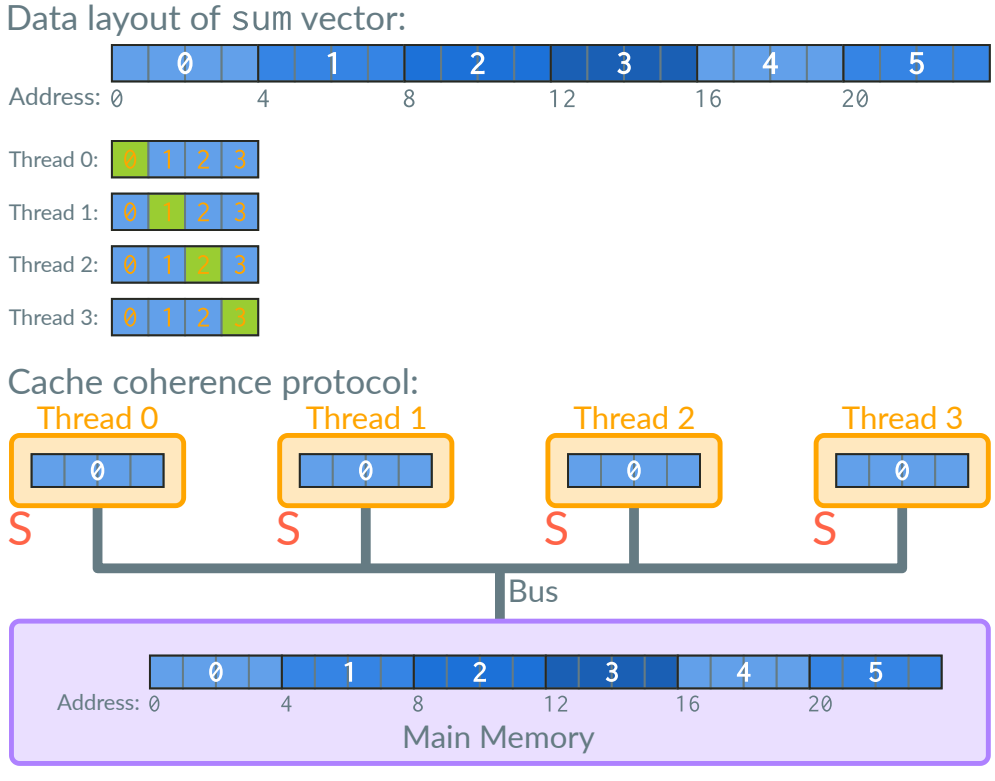
What is wrong?

FALSE SHARING

A proposed solution:

```cpp
#include <iostream>
#include <omp.h>
#define NTHREADS 8

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum[NTHREADS] = {0};

#pragma omp parallel num_threads(NTHREADS)
    {
        const int tid = omp_get_thread_num();

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            sum[tid] += 1.0 / (1.0 + x * x);
        }
    }

    // reduce per thread sum
    for (int i = 1; i < NTHREADS; ++i) {
        sum[0] += sum[i];
    }
    std::cout << "pi = "
              << 4.0 * sum[0] * step << '\n';
    return 0;
}
```

Recall the cache coherence protocol:



- The choice of storing the partial sums in a contiguous array is bad because cache lines will be shared among threads.
- *It creates a lot of pressure on the cache coherence protocol!*

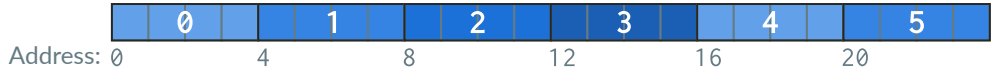
FALSE SHARING

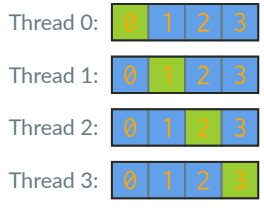
A proposed solution:

```cpp
#include <iostream>
#include <omp.h>
#define NTHREADS 8

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum[NTHREADS] = {0};

#pragma omp parallel num_threads(NTHREADS)
    {
        const int tid = omp_get_thread_num();

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            sum[tid] += 1.0 / (1.0 + x * x);
        }
    }

    // reduce per thread sum
    for (int i = 1; i < NTHREADS; ++i) {
        sum[0] += sum[i];
    }
    std::cout << "pi = "
              << 4.0 * sum[0] * step << '\n';
    return 0;
}
```

Recall the cache coherence protocol:

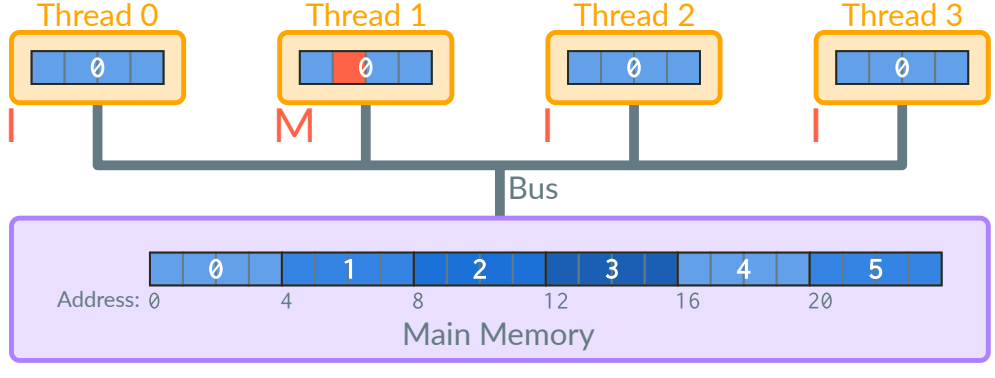
Data layout of sum vector:





False sharing leads to a form of cache thrashing similar to conflict misses. They are not the same! False sharing is frequent replacements of *the same* cache line due to the cache coherence protocol.

Cache coherence protocol:



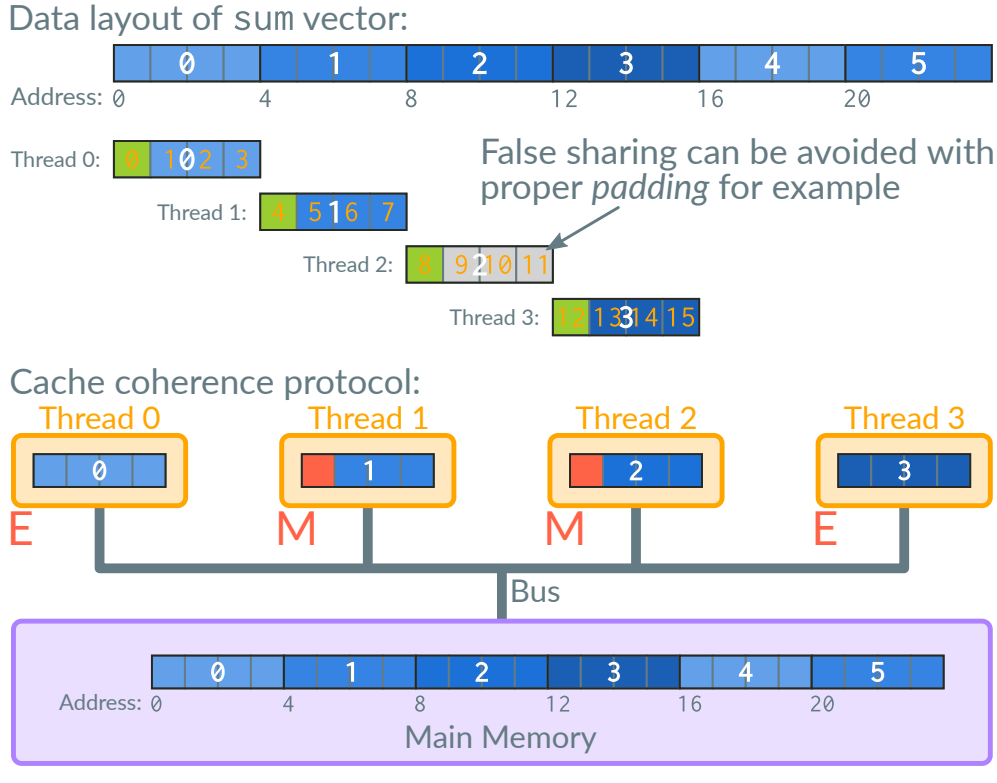
- The choice of storing the partial sums in a contiguous array is bad because cache lines will be shared among threads.
  → *creates a lot of pressure on cache coherence protocols!*

FALSE SHARING

A proposed solution (with padding):

```
1  #include <iostream>
2  #include <omp.h>
3  #define NTHREADS 8
4
5  int main(void)
6  {
7      const int nsteps = 100000;
8      const double step = 1.0 / nsteps;
9      double sum[4 * NTHREADS] = {0};
10
11 #pragma omp parallel num_threads(NTHREADS)
12     {
13         const int tid = omp_get_thread_num();
14
15 #pragma omp for nowait
16         for (int i = 0; i < nsteps; ++i) {
17             const double x = (i + 0.5) * step;
18             sum[4 * tid] += 1.0 / (1.0 + x * x);
19         }
20     }
21
22     // reduce per thread sum
23     for (int i = 1; i < NTHREADS; ++i) {
24         sum[0] += sum[4 * i];
25     }
26     std::cout << "pi = "
27               << 4.0 * sum[0] * step << '\n';
28     return 0;
29 }
```

Recall the cache coherence protocol:



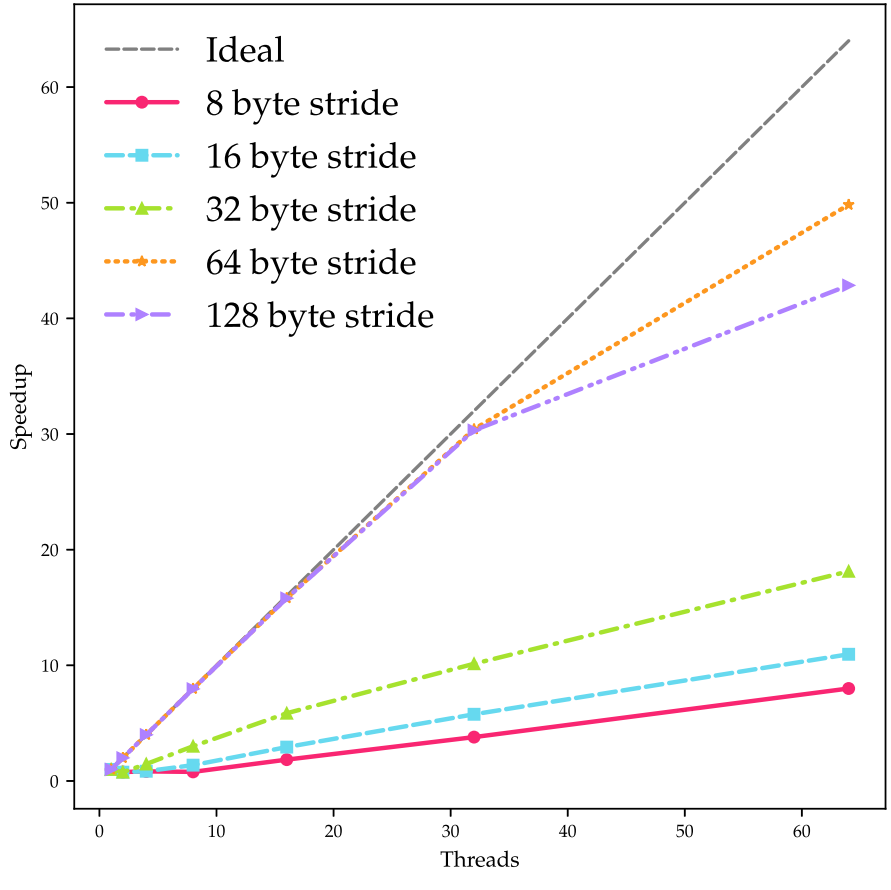
- Padding cache lines will ensure threads are not accessing the same cache lines and therefore eliminates false sharing.

FALSE SHARING

A proposed solution (with padding):

```cpp
#include <iostream>
#include <omp.h>
#define NTHREADS 8

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum[4 * NTHREADS] = {0};

#pragma omp parallel num_threads(NTHREADS)
    {
        const int tid = omp_get_thread_num();

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            sum[4 * tid] += 1.0 / (1.0 + x * x);
        }
    }

    // reduce per thread sum
    for (int i = 1; i < NTHREADS; ++i) {
        sum[0] += sum[4 * i];
    }
    std::cout << "pi = "
              << 4.0 * sum[0] * step << '\n';
    return 0;
}
```

Performance benchmark:



Cache line on a real system is 64 byte.

AVOID FALSE SHARING

...by preferring thread private variables when possible!

A better solution to compute π:

```cpp
#include <iostream>

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum = 0.0; // shared variable

#pragma omp parallel
    {
        double local_sum = 0.0; // register allocation

#pragma omp for nowait
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            local_sum += 1.0 / (1.0 + x * x);
        }

#pragma omp atomic
        sum += local_sum; // manual reduction
    }

    std::cout << "pi = "
              << 4.0 * sum * step << '\n';
    return 0;
}
```

The preferred OpenMP solution:

```cpp
#include <iostream>

int main(void)
{
    const int nsteps = 100000;
    const double step = 1.0 / nsteps;
    double sum = 0.0; // shared variable

#pragma omp parallel for reduction(+: sum)
        for (int i = 0; i < nsteps; ++i) {
            const double x = (i + 0.5) * step;
            // sum is thread private in reduction
            sum += 1.0 / (1.0 + x * x);
        }

    std::cout << "pi = "
              << 4.0 * sum * step << '\n';
    return 0;
}
```

- Be in favor of thread private variables whenever possible (easier to achieve if you write code from scratch).
- No synchronization, no race conditions, no false sharing, *code is parallel!*

RECAP

- Uniform and non-uniform memory access (UMA and NUMA)
- NUMA first-touch policy is important for efficient memory bank placement of your data
- To control this, we must control thread affinity
- Cache coherency protocols in shared memory take care that threads have consistent view of loaded cache lines
- False sharing can happen in multi-threaded execution when multiple threads request access to the same cache line. This will degrade your performance.

Further reading:

- Sections 2.6.2, 6.4, 6.5 and 6.13 *OpenMP API Specification v5.1*, OpenMP, 2020 (pdf in `git` repo as well)
- Section 2.4 in Eijkhout, *Introduction to High Performance Scientific Computing*, free PDF
- Chapter 25, 29, 30, 31 in Eijkhout, *"Parallel Programming for Science and Engineering"*, free PDF