NUMA-aware OpenMP Programming

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

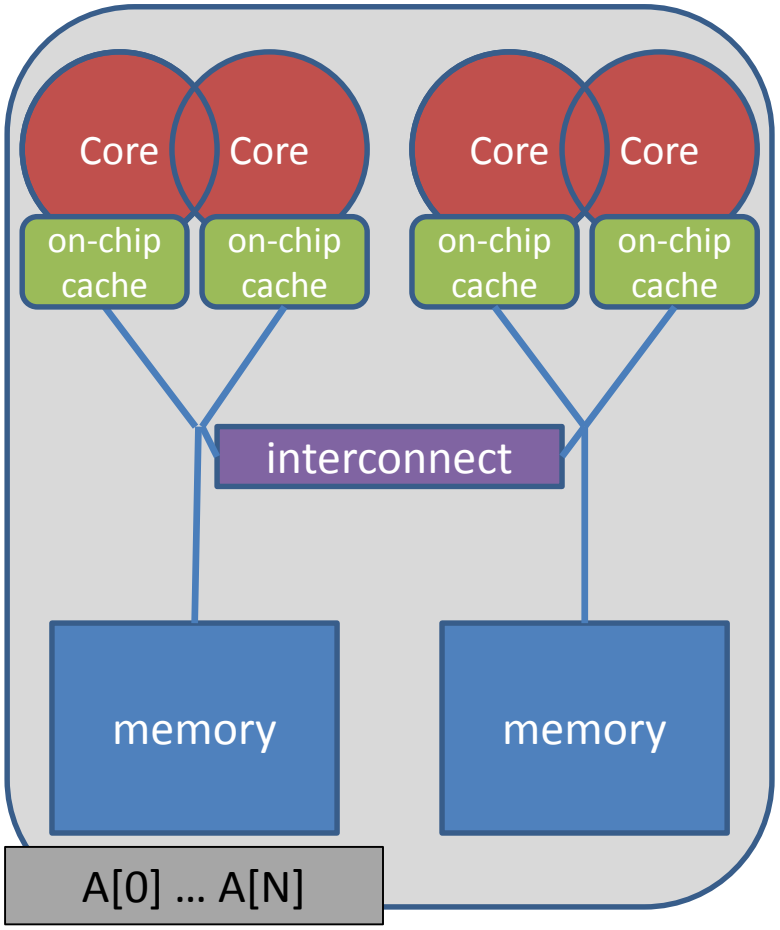
Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

NUMA Architectures

How To Distribute The Data ?

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



About Data Distribution

- **Important aspect on cc-NUMA systems**
  - → If not optimal, longer memory access times and hotspots
- **OpenMP does not provide support for cc-NUMA**
- **Placement comes from the Operating System**
  - → This is therefore Operating System dependent

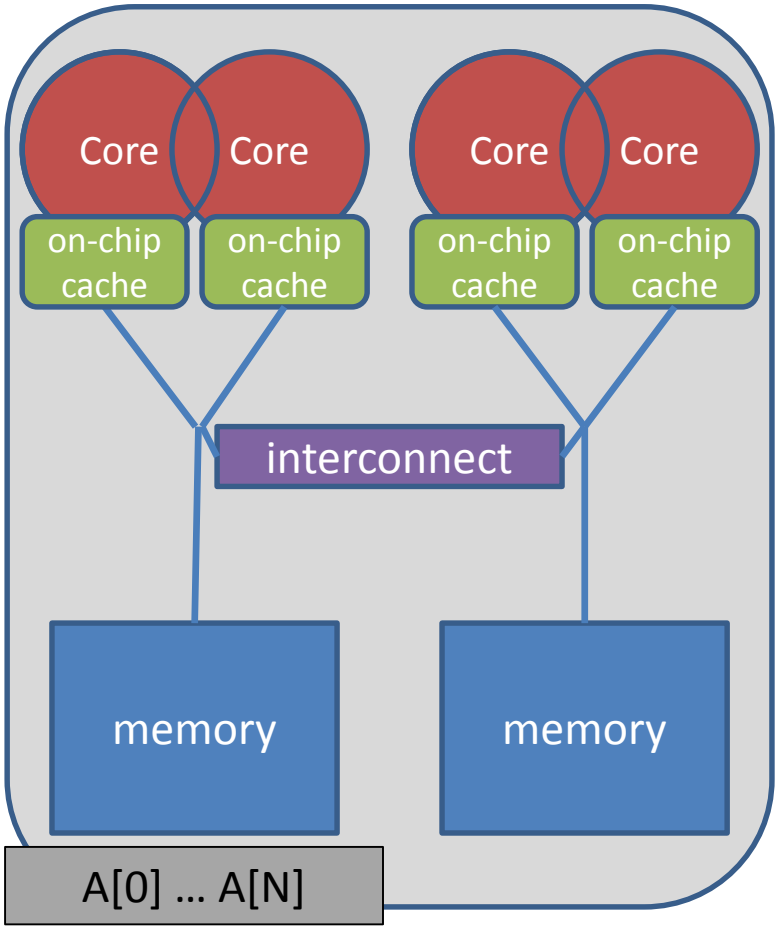
- **Windows, Linux and Solaris all use the “First Touch” placement policy by default**
  - → May be possible to override default (check the docs)

Non-Uniform Memory Arch.

- **Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



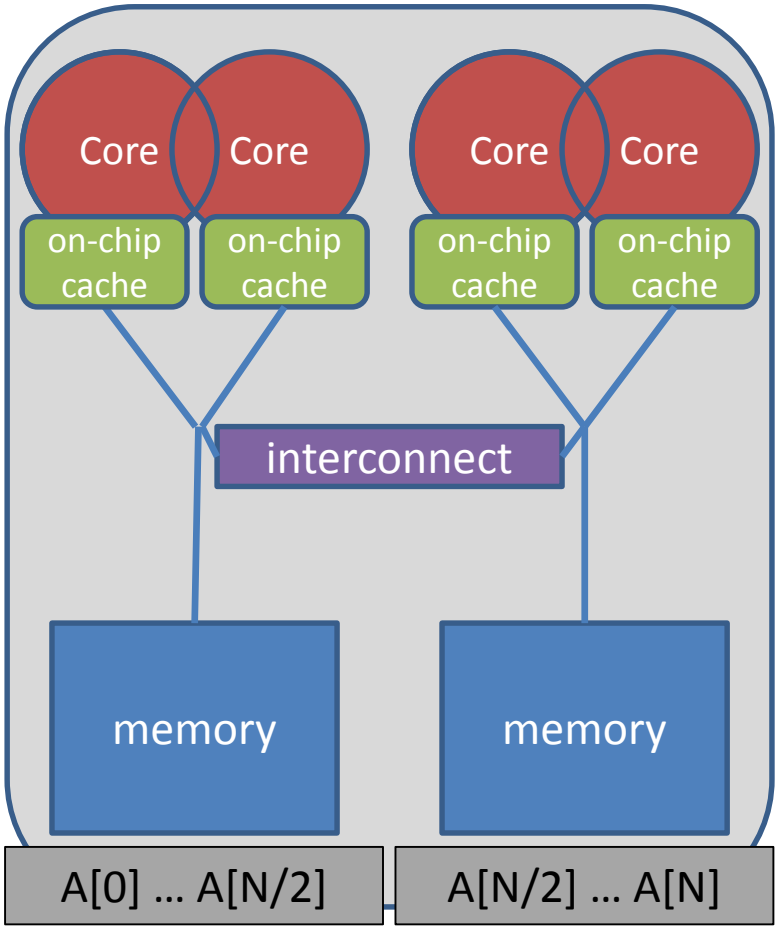
Non-Uniform Memory Arch.

- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



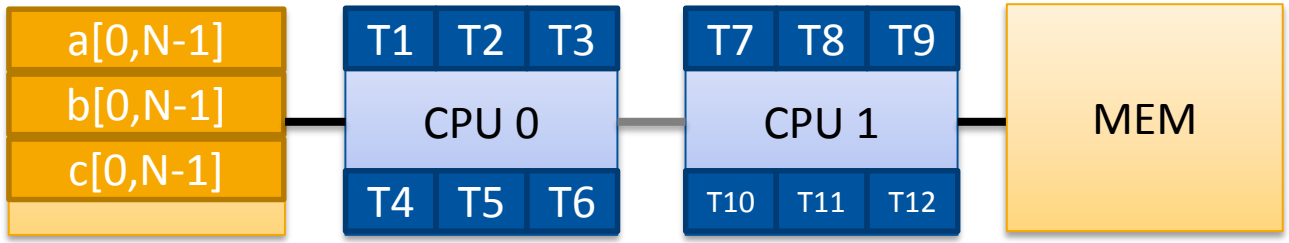
Non-Uniform Memory Arch.

- **Stream example with and without parallel initialization.**

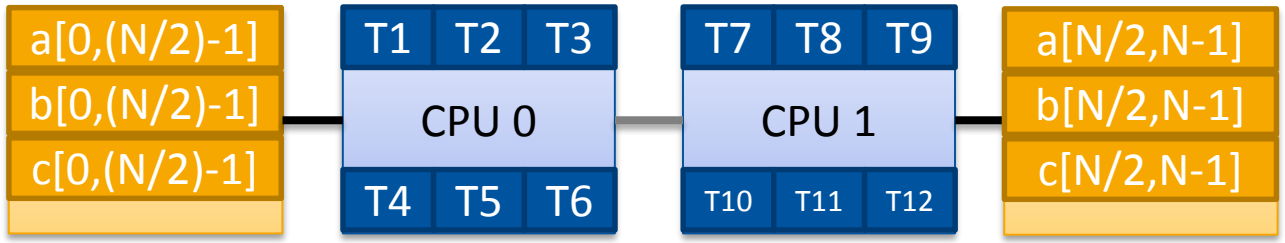
  → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

ser_init:



par_init:



Get Info on the System Topology

- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**
  - → Intel MPI's `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.
  - → hwlocs' `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.

Decide for Binding Strategy

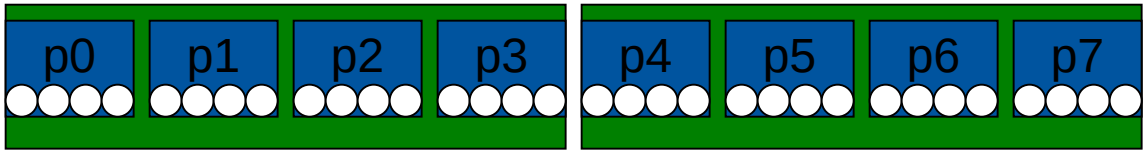
- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - → Putting threads far apart, i.e. on different sockets
    - →May improve the aggregated memory bandwidth available to your application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - → Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

OpenMP 4.0: Places + Binding Policies (1/2)

- **Define OpenMP Places**
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e. `OMP_PLACES=cores`
- **Define a set of OpenMP Thread Affinity Policies**
  - → SPREAD: spread OpenMP threads evenly among the places
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- **Goals**
  - → user has a way to specify where to execute OpenMP threads for
  - → locality between OpenMP threads / less false sharing / memory bandwidth

Places

- **Assume the following machine:**



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

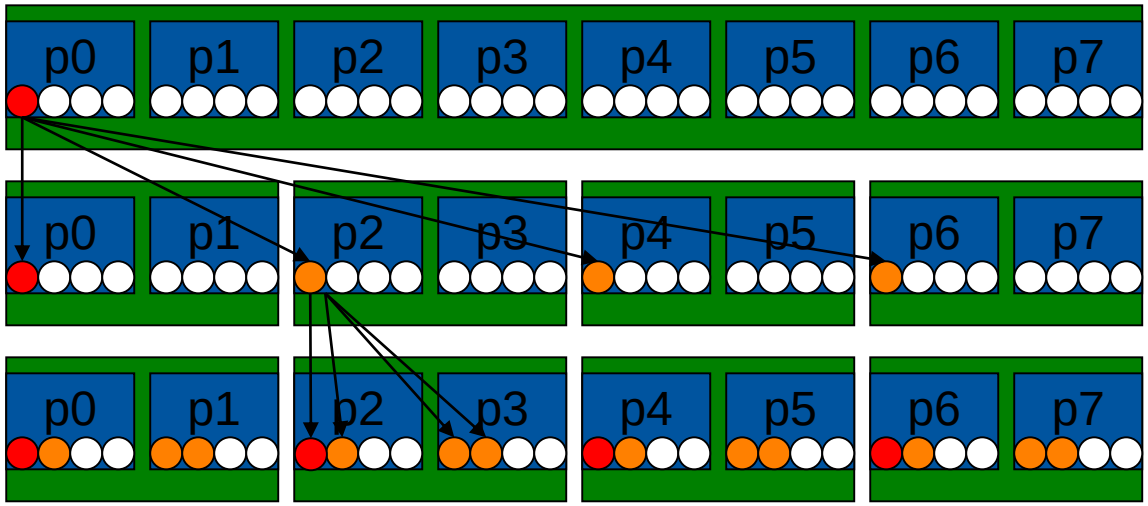
- **Abstract names for OMP_PLACES:**
  - → threads: Each place corresponds to a single hardware thread on the target machine.
  - → cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

OpenMP 4.0: Places + Binding Policies (2/2)

- **Example's Objective:**
  - → separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)**
  - → spread creates partition, compact binds threads within respective partition

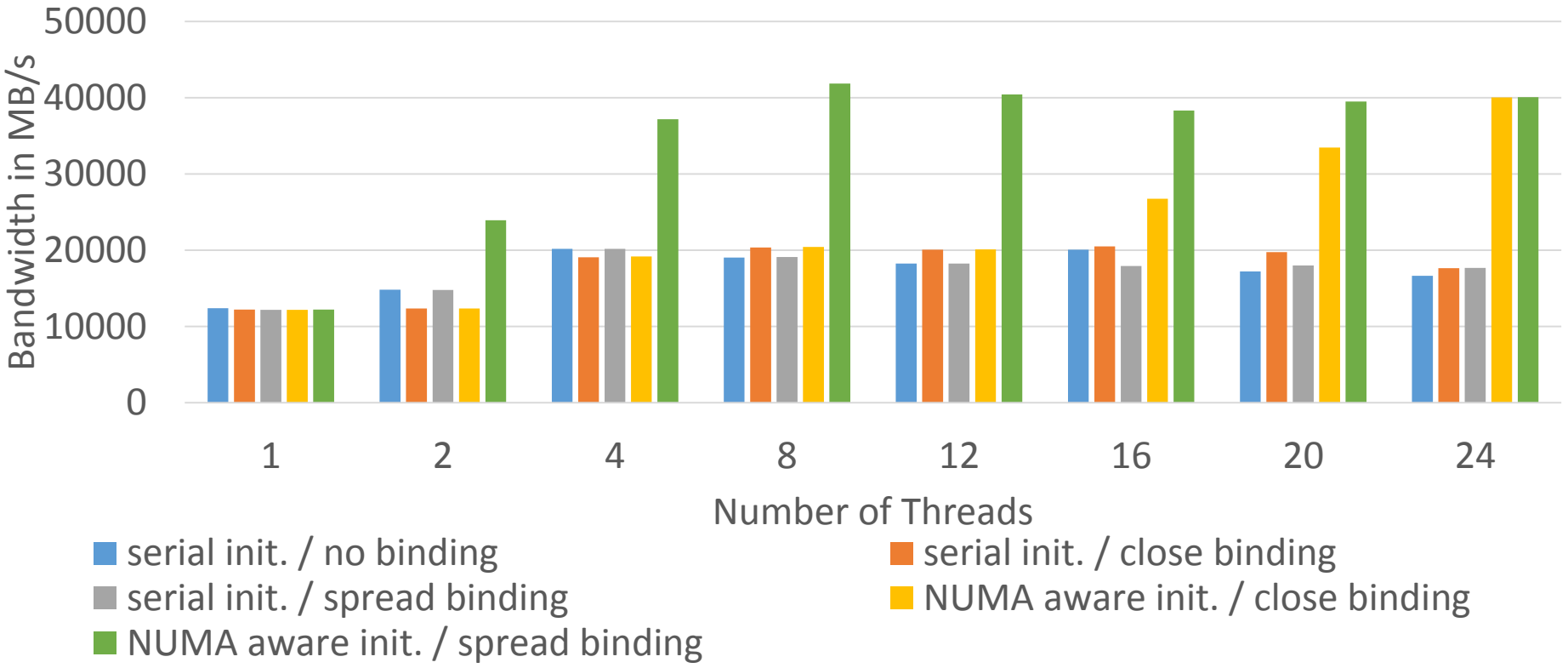
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4    = cores
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)
```

- **Example**
  - → initial
  - → spread 4
  - → close 4



Serial vs. Parallel Initialization

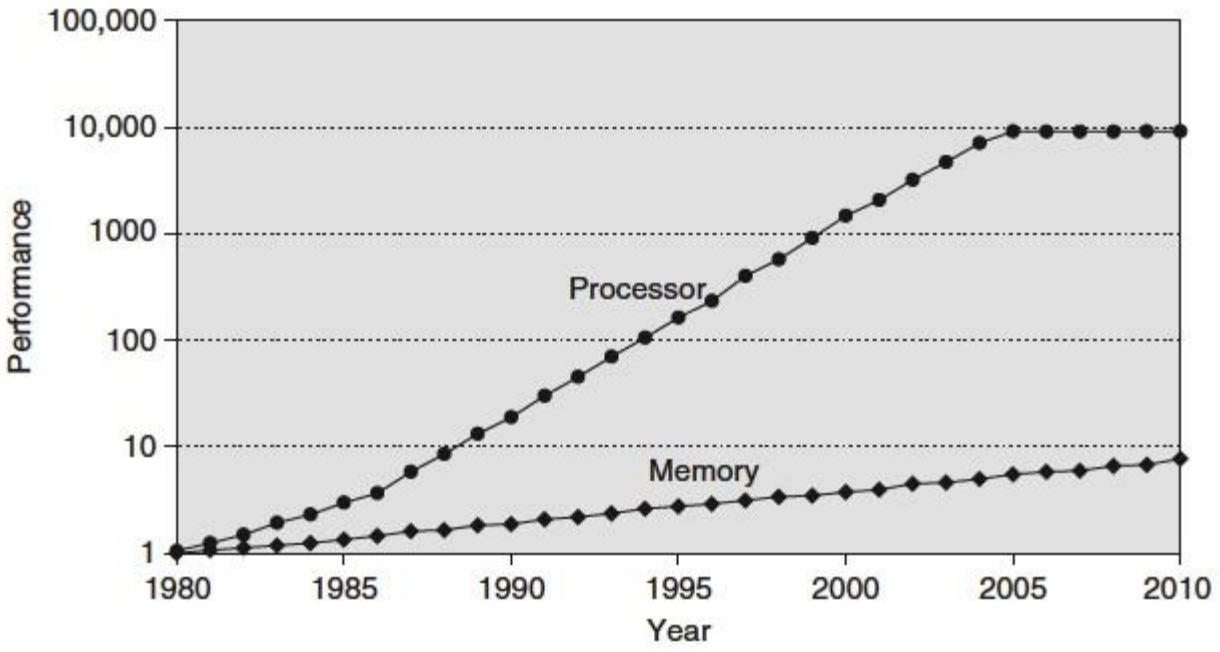
- **Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:**



False Sharing

Memory Bottleneck

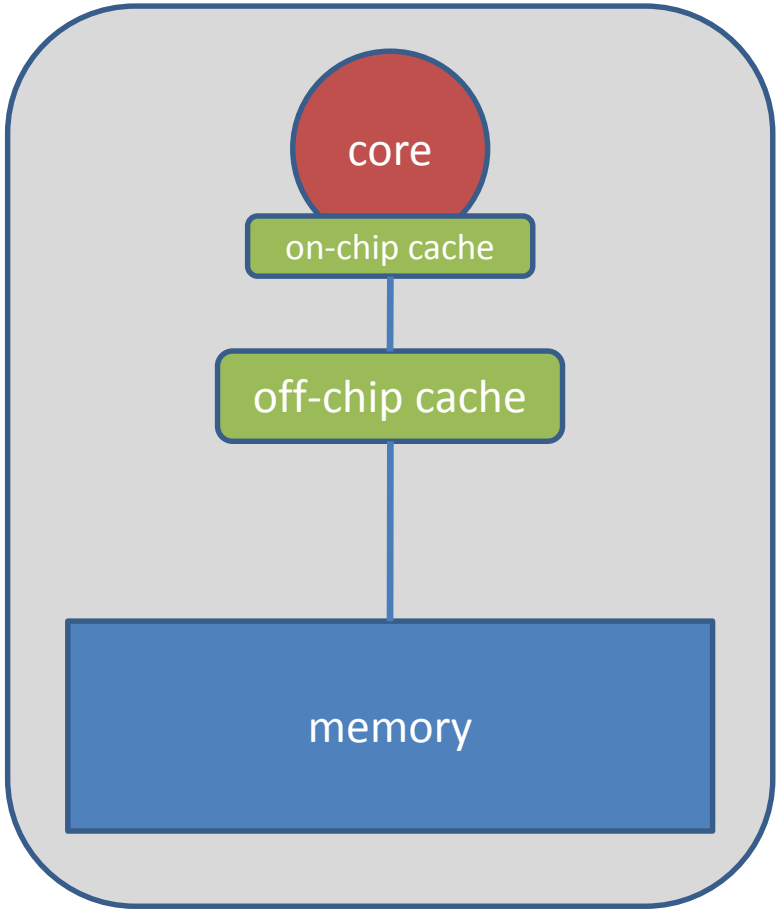
- **There is a growing gap between core and memory performance:**
  - → memory, since 1980: 1.07x per year improvement in latency
  - → single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a *per-core* basis



→ Source: John L. Hennessy, Stanford University, and David A. Patterson, University of California, September 25, 2012

Caches

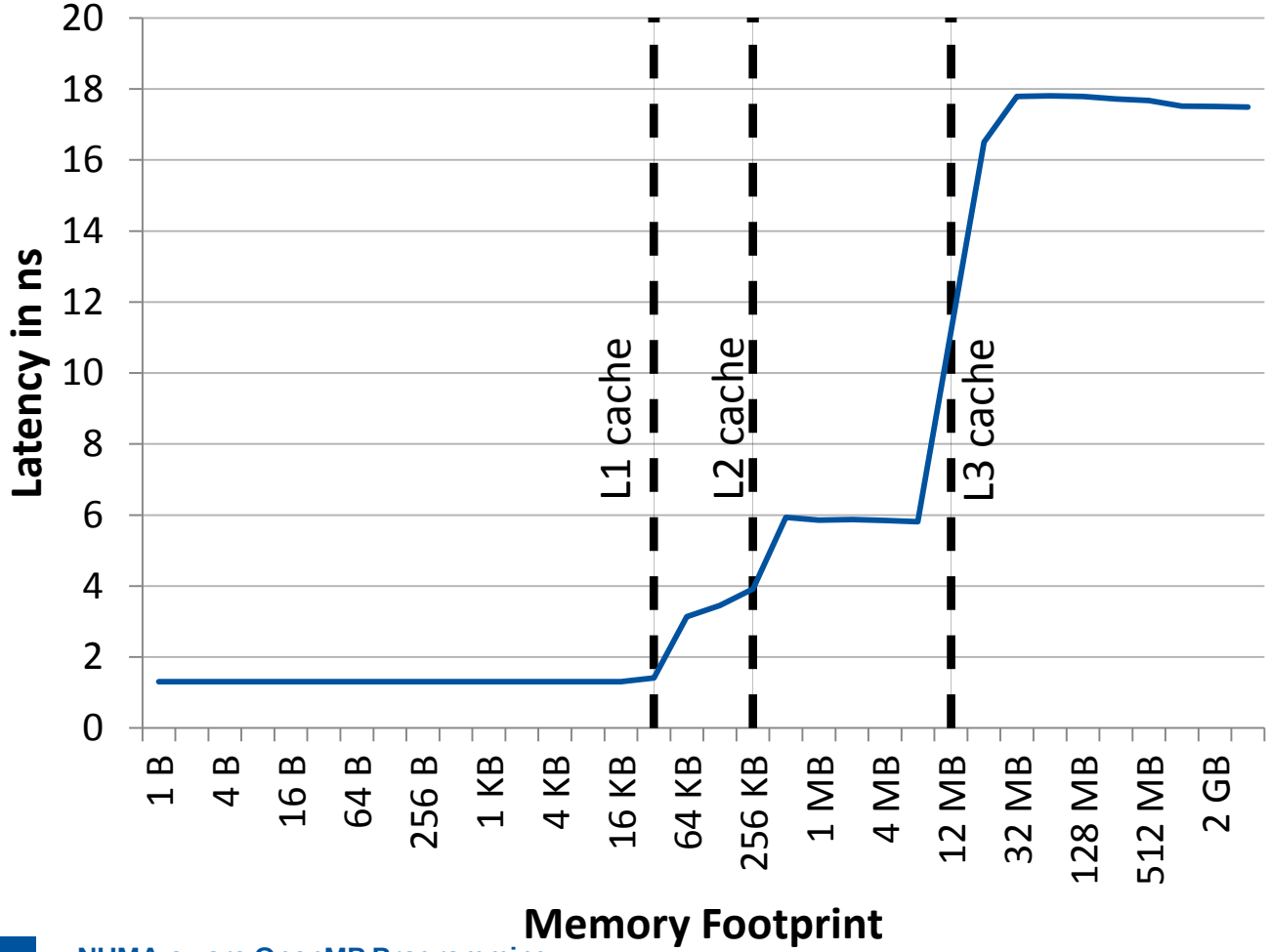
- **CPU is fast**
  - → Order of 3.0 GHz
- **Caches:**
  - → Fast, but expensive
  - → Thus small, order of MB
- **Memory is slow**
  - → Order of 0.3 GHz
  - → Large, order of GB



- **A good utilization of caches is crucial for good performance of HPC applications!**

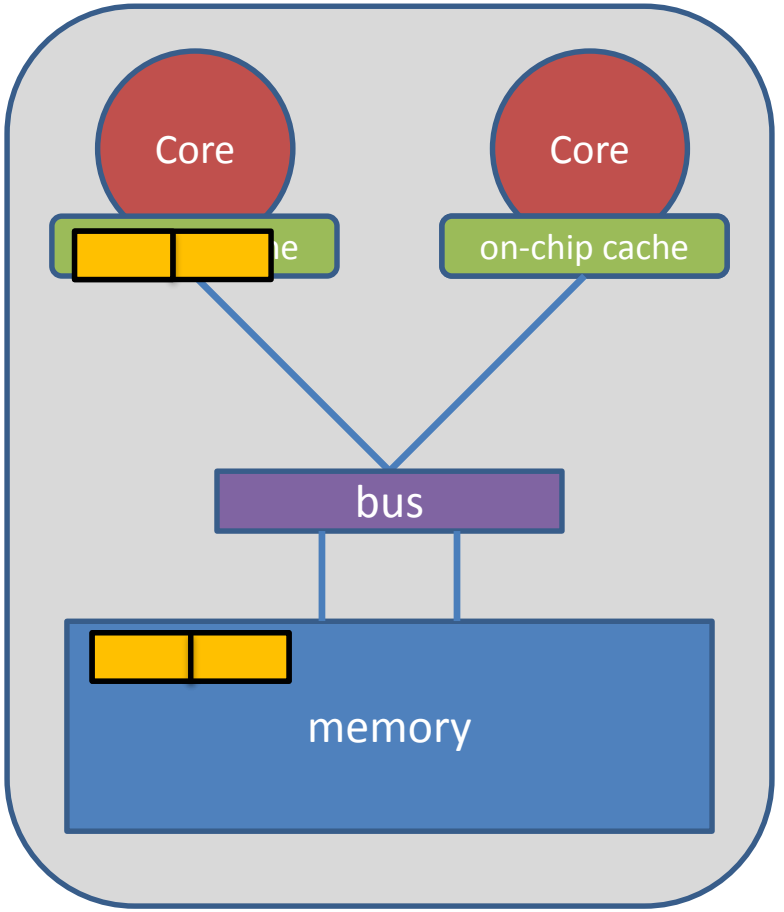
Visualization of the Memory Hierarchy

- **Latency on the Intel Westmere-EP 3.06 GHz processor**



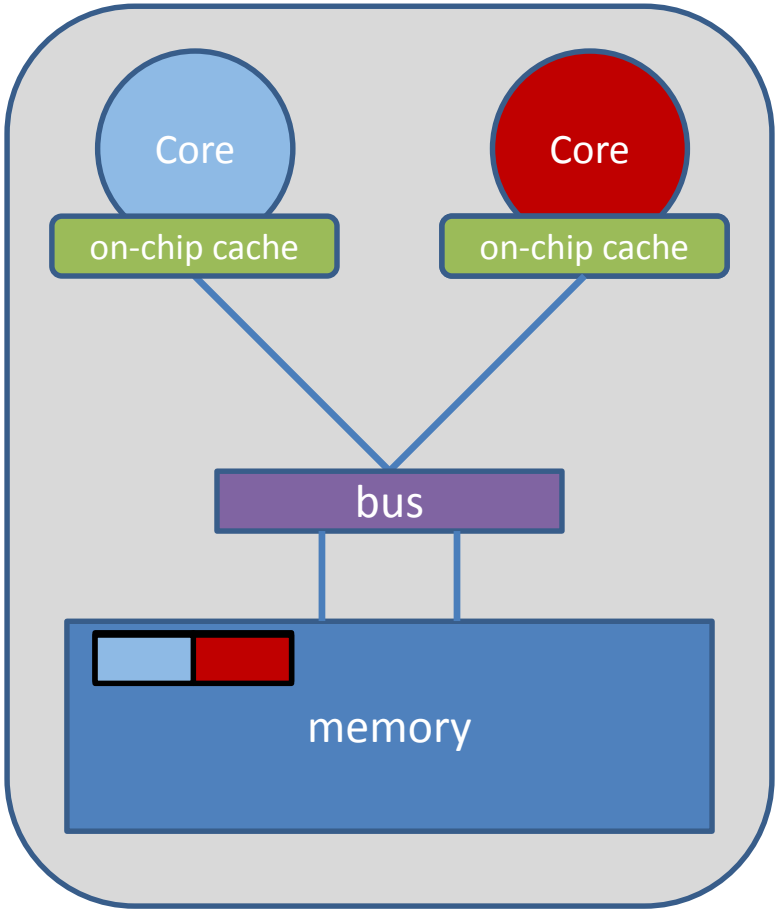
Data in Caches

- **When data is used, it is copied into caches.**
- **The hardware always copies chunks into the cache, so called *cache-lines*.**
- **This is useful, when:**
  - → the data is used frequently (temporal locality)
  - → consecutive data is used which is on the same cache-line (spatial locality)



False Sharing

- **False Sharing occurs when**
  - → different threads use elements of the same cache-line
  - → one of the threads writes to the cache-line
- **As a result the cache line is moved between the threads, also there is no real dependency**
- **Note: False Sharing is a performance problem, not a correctness issue**



Summing up vector elements again

It‘s your turn: Make It Scale!

```
#pragma omp parallel
{

#pragma omp for
   for (i = 0; i < 99; i++)
   {

         s   = s    + a[i];

   }

} // end parallel
```

```
do i = 0, 99
   s = s + a(i)
end do
```

```
do i = 0, 24
   s = s + a(i)
end do
```

```
do i = 25, 49
   s = s + a(i)
end do
```

```
do i = 50, 74
   s = s + a(i)
end do
```

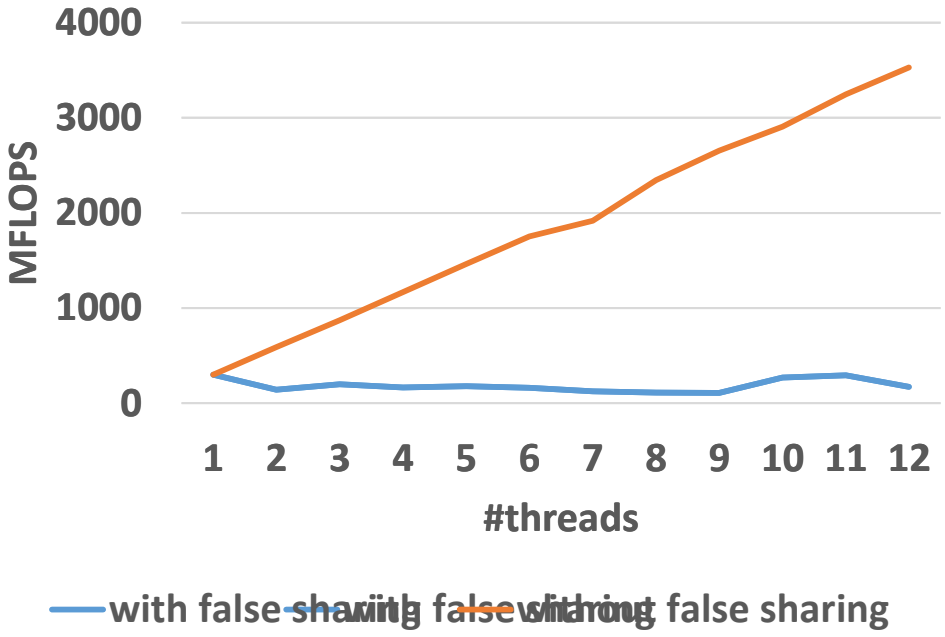
```
do i = 75, 99
   s = s + a(i)
end do
```

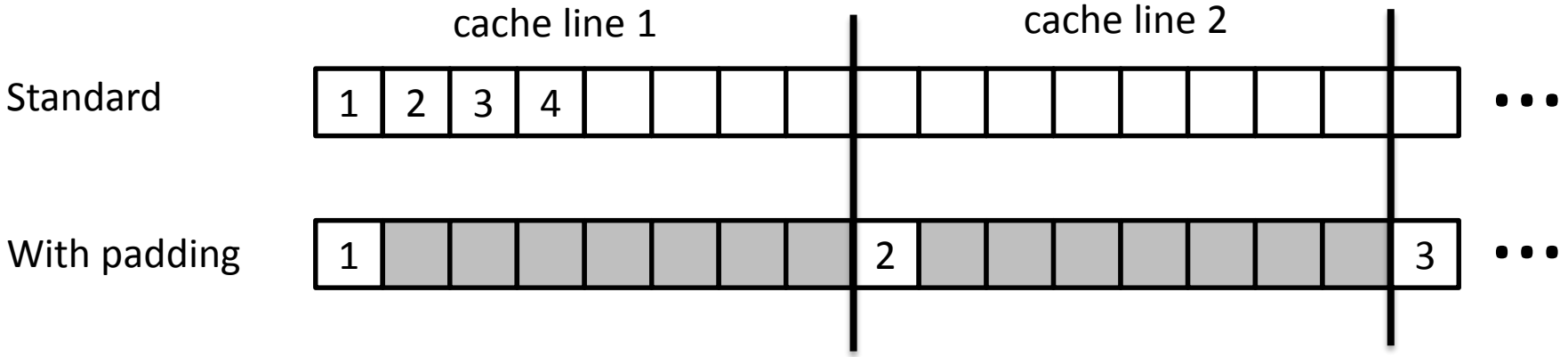
False Sharing

```
double s_priv[nthreads];
#pragma omp parallel num_threads(nthreads)
{
   int t=omp_get_thread_num();
   #pragma omp for
   for (i = 0; i < 99; i++)
   {
         s_priv[t]  += a[i];
   }
} // end parallel
for (i = 0; i < nthreads; i++)
{
      s += s_priv[i];
}
```

False Sharing

- no performance benefit for more threads
- Reason: false sharing of s_priv
- Solution: padding so that only one variable per cache line is used





Questions?