HPC2N, UmU

Performance aspects of OpenMP

Joachim Hein

LUNARC & Centre of Mathematical Sciences

Lund University

Pedro Ojeda May

HPC2N

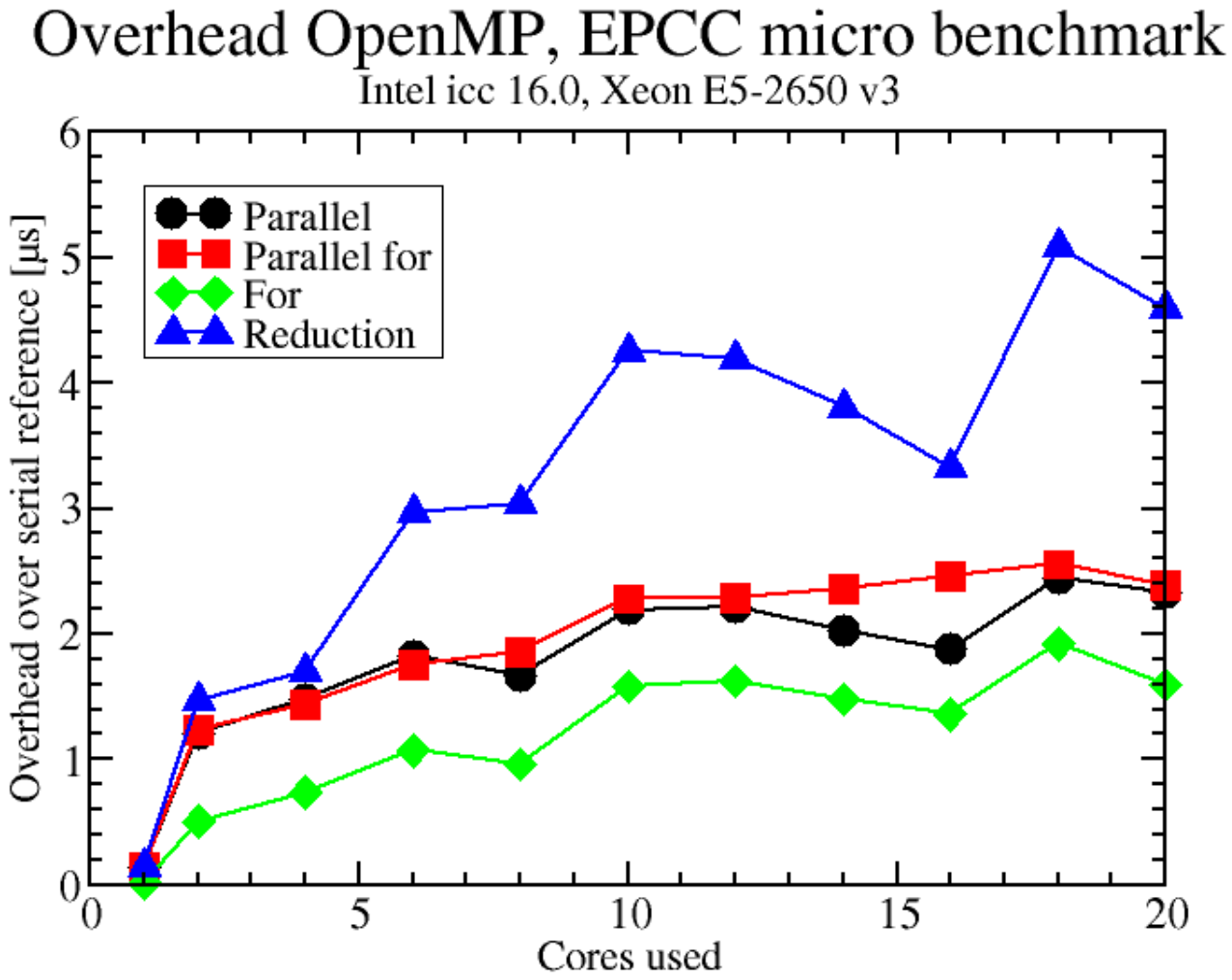
Umeå University

Outline

- How expensive are OpenMP constructs
  - Results from EPCC OpenMP microbenchmarks
- Cache memory and false sharing
- First touch

OVERHEADS OF OPENMP

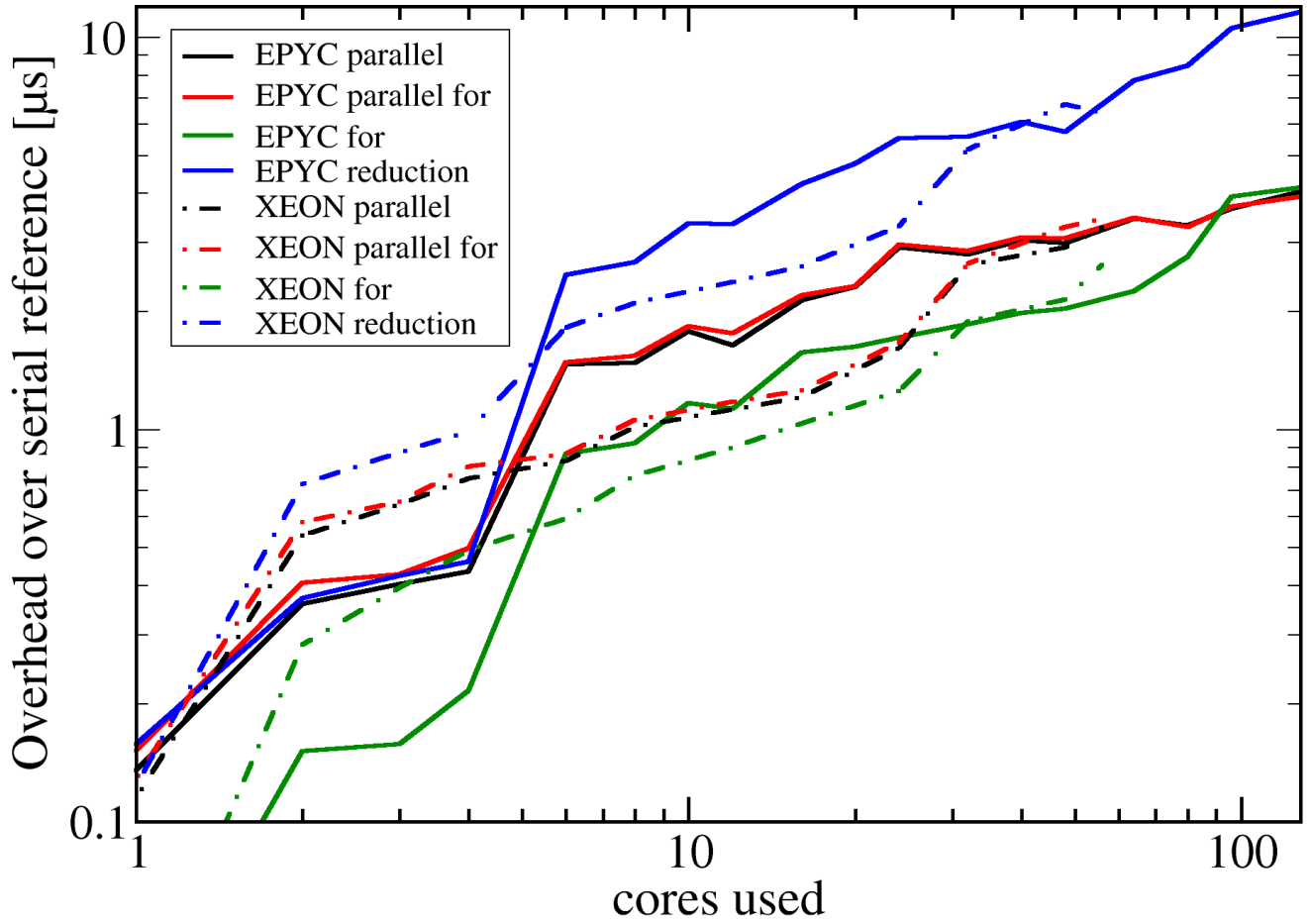
Performance of `parallel` and loop construct



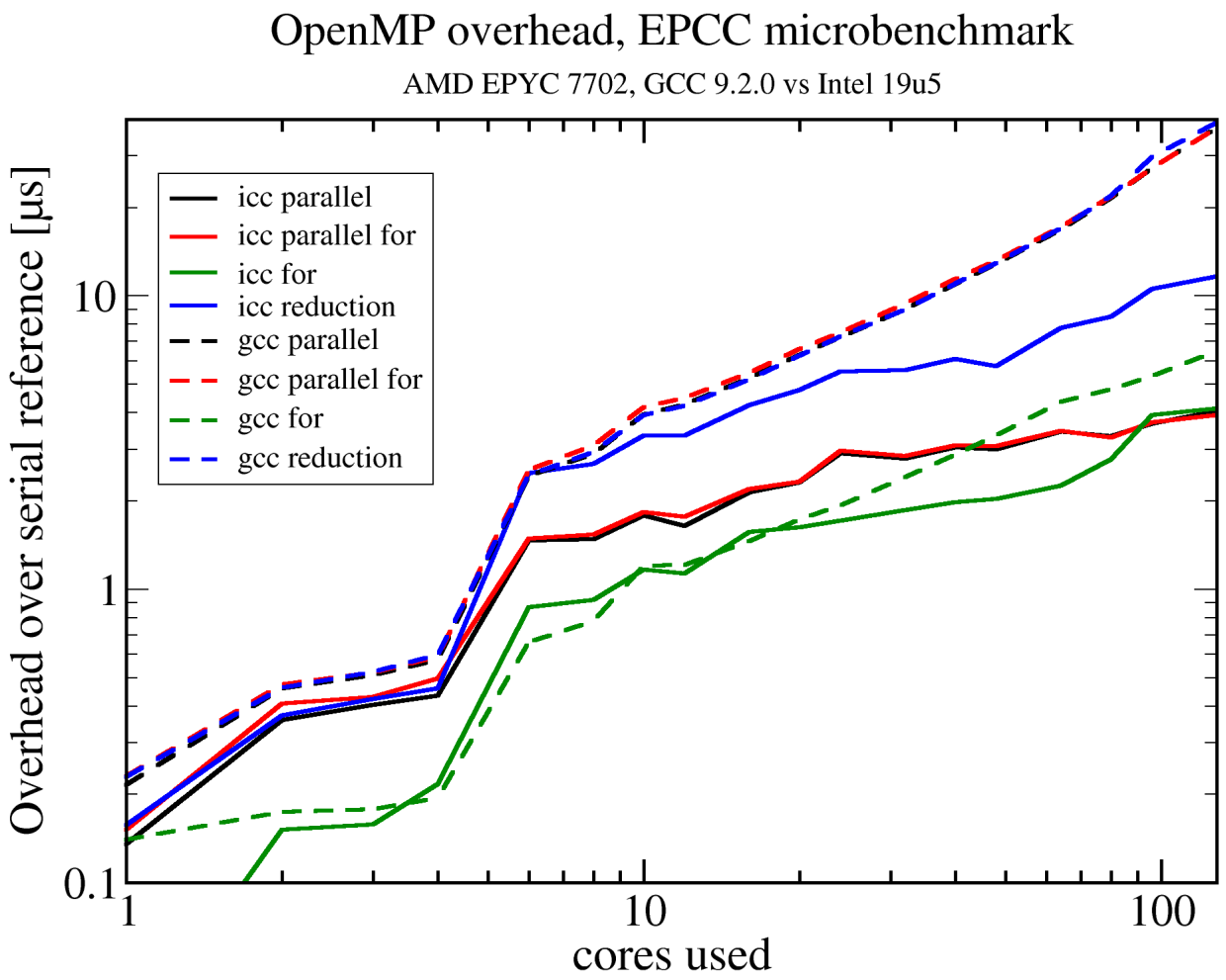
Performance: Large core count servers

OpenMP overhead, EPCC microbenchmark

AMD EPYC 7702 vs Intel Xeon 8280, Intel compiler 19u5



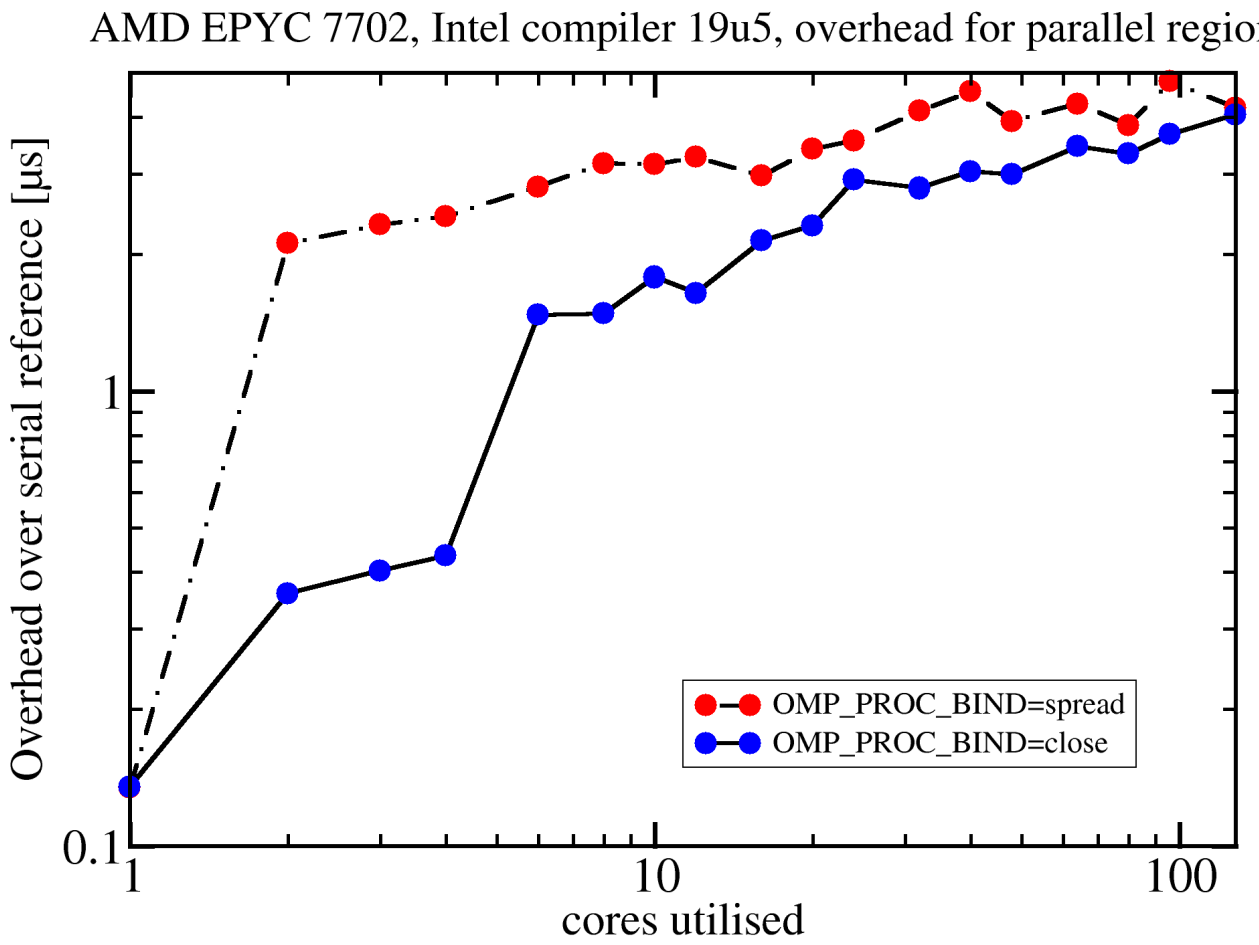
Performance: GCC vs Intel compiler



Discussion of `parallel` result

- Benchmark your own system to confirm findings
- Overhead of constructs increases steadily with core count
- Overhead for starting `parallel`:
  - around **1µs** small core count
  - several **µs** for large core count (up to 37µs for GCC)
  - Need significant work inside to pay for the time
  - Avoiding closing `parallel` regions – keep it running
- Loop construct cheaper than `parallel` (a lot for GCC)
  - Place multiple loop constructs in single `parallel`
- Currently: Intel compiler better than GCC
- Cost increase with `reduction` (Intel, larger task count)
- A `parallel for` is marginally dearer than `parallel`

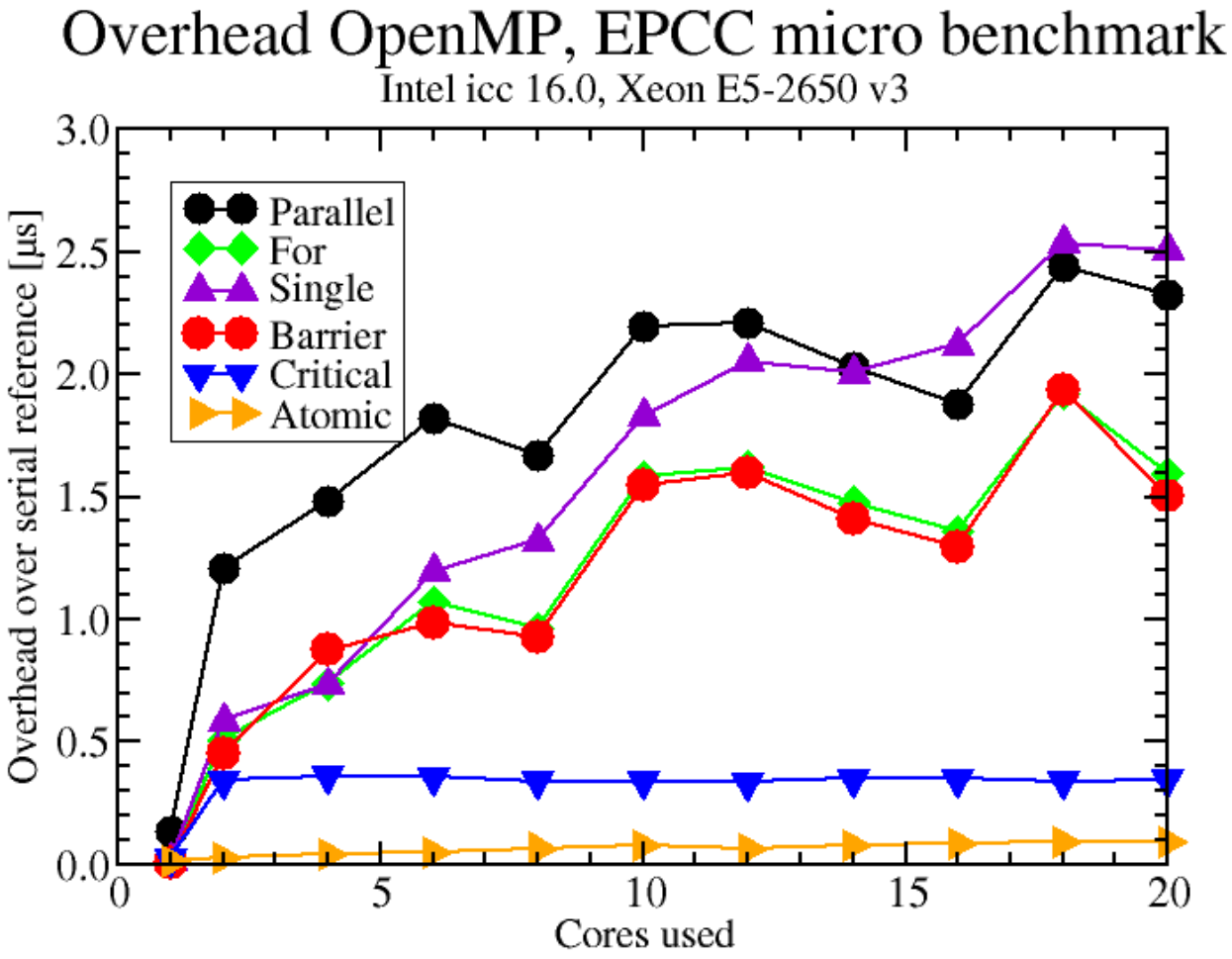
Effect of placement on performance spread vs close binding



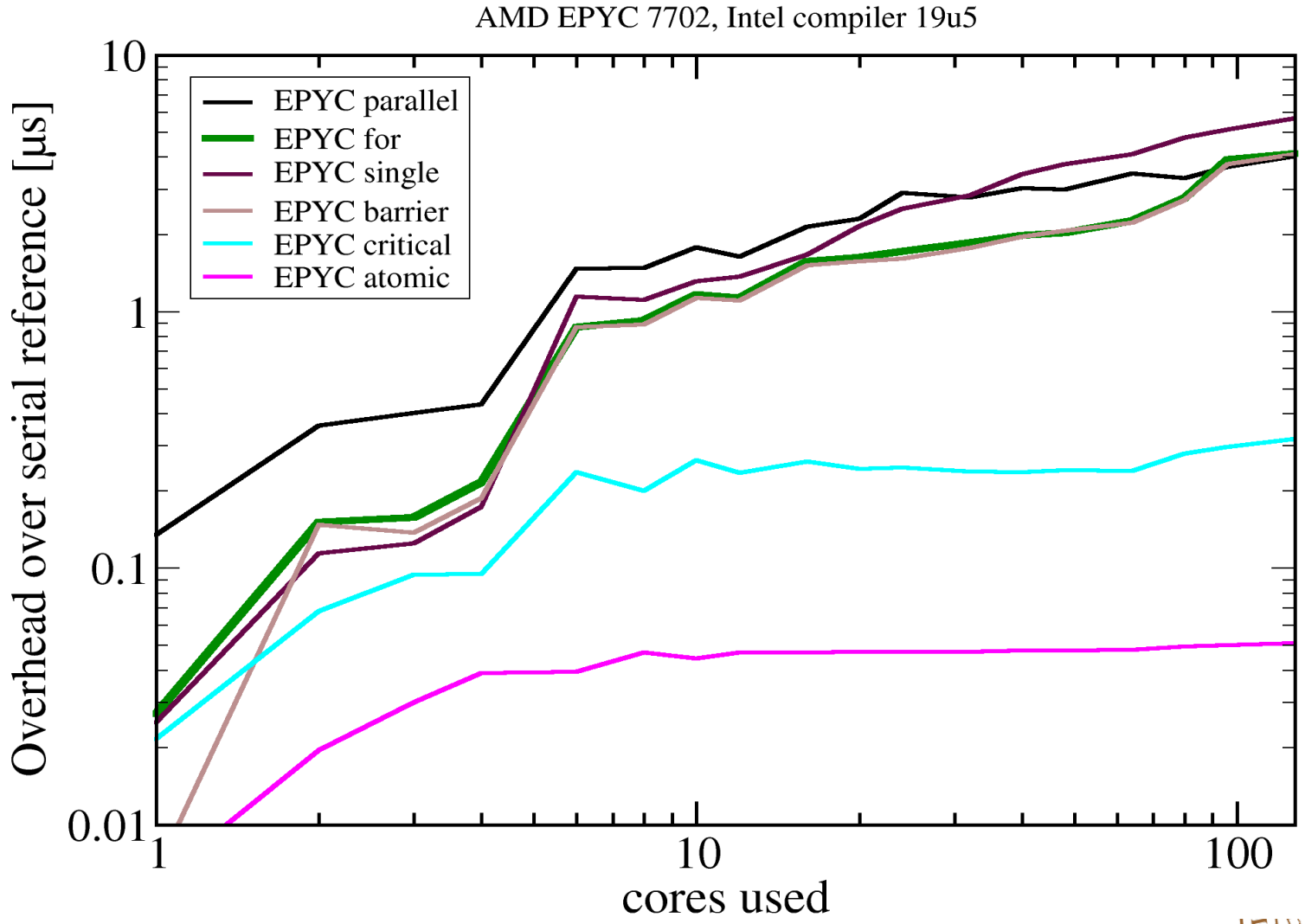
Discussion

- close binding superior
  - Huge effect for small thread count
    - Up to 4 threads on AMD EPYC2
      - inner chip communication
  - Smaller effect for large thread count

Effect of synchronisation



Effect of syncronisation



Discussion of synchronisation constructs

- Synchronisation quite expensive: **1 µs – 2 µs**
  - Cost increases with thread count

- Obvious benefit to `atomic` over `critical`

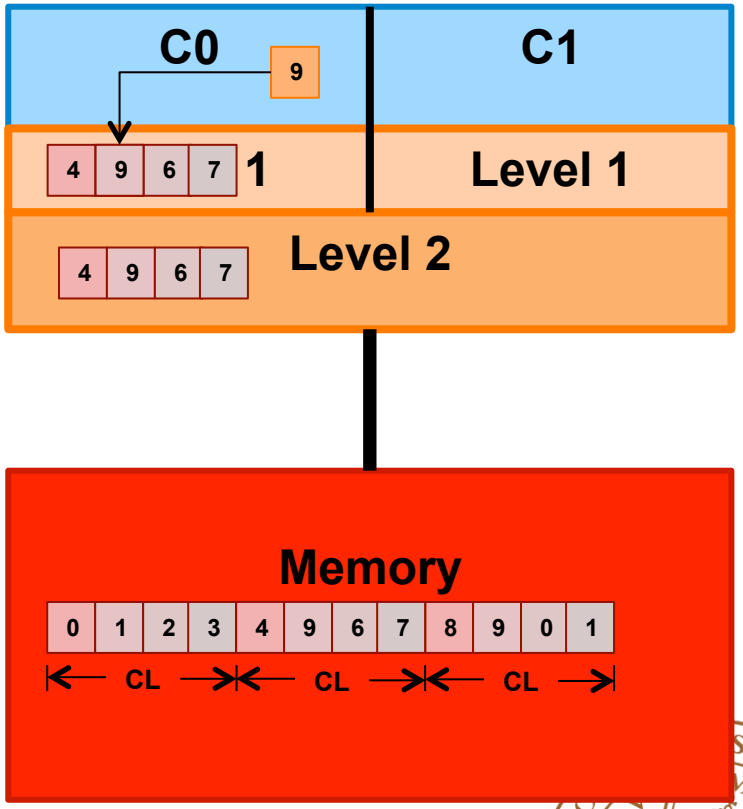
- Cost of `for` marginally higher than `barrier`
- Cost of `single` increases with thread count
  - For GCC `single` is faster then open/close `parallel`

- Overhead for `parallel` largest
  - Use `synchonisation` construct to avoid closing parallel region
  - Watch load balance for `critical` and `atomic`

CACHE MEMORY AND FALSE SHARING

A look into the memory system: Single multicore processors

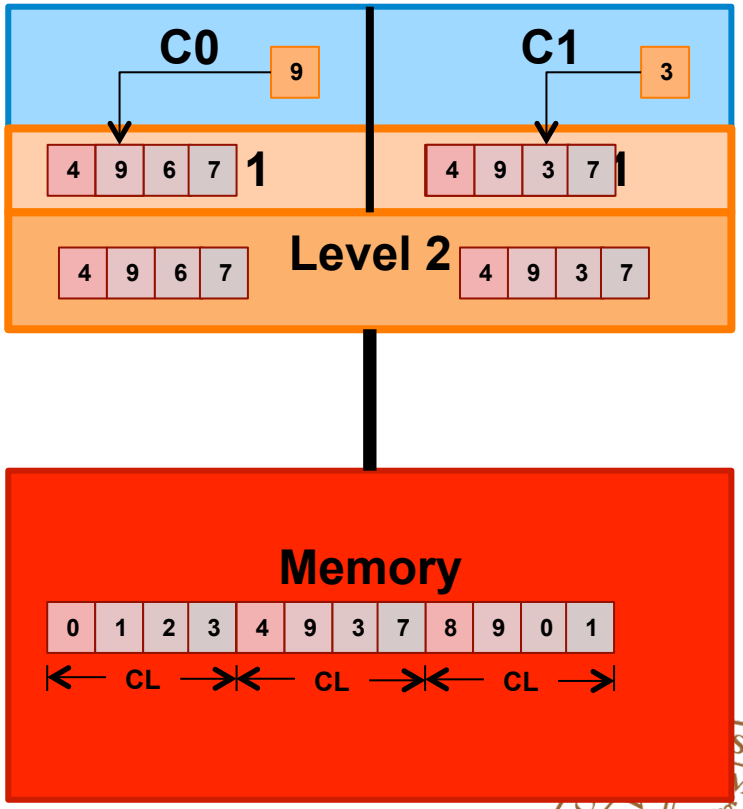
- Memory system hierarchy
  - Typically 2 to 3 cache levels
    - high levels private to core
    - low levels shared
  - Main memory
- Organised in cache lines
  - typically 4 or 8 doubles
- Example:
  - exchange 6th array value
  - load entire line into cache
  - replace individual value
  - write cache line back



Single multicore processors: Writing from different cores

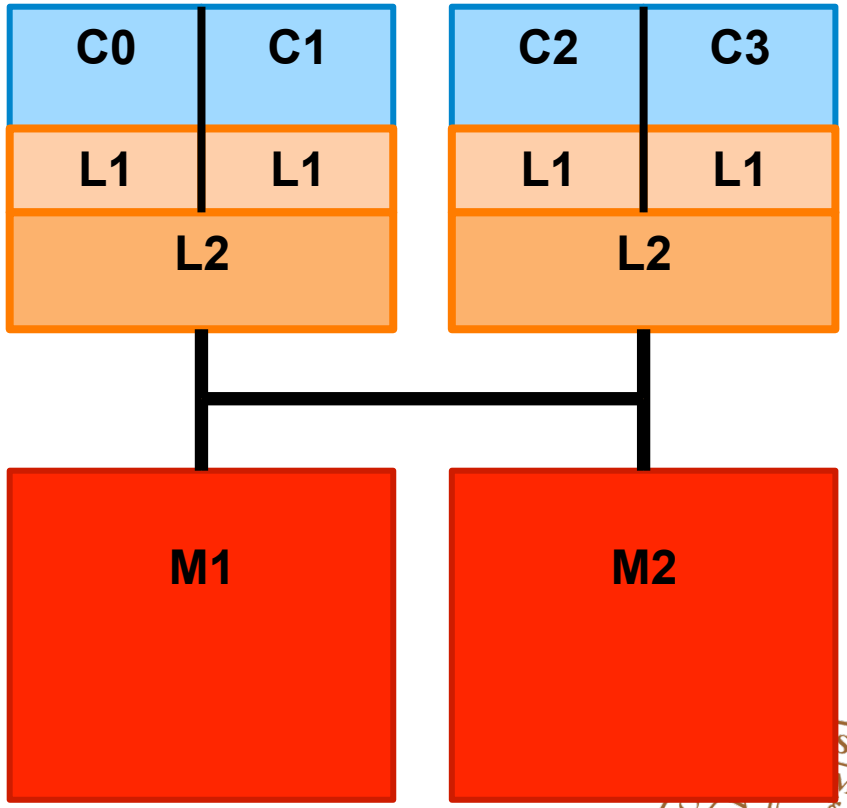
- C0 exchange 6th value
- C1 exchange 7th value

- Both need **same** cache line
- One core gets it and replaces
- Other core requests
- Move down to shared cache
- Load in L1 cache of other core
- Write cache line back to Memory
- The problem is commonly referred to as "***false sharing***"



System with multiple multicore processors False sharing even more of an issue!

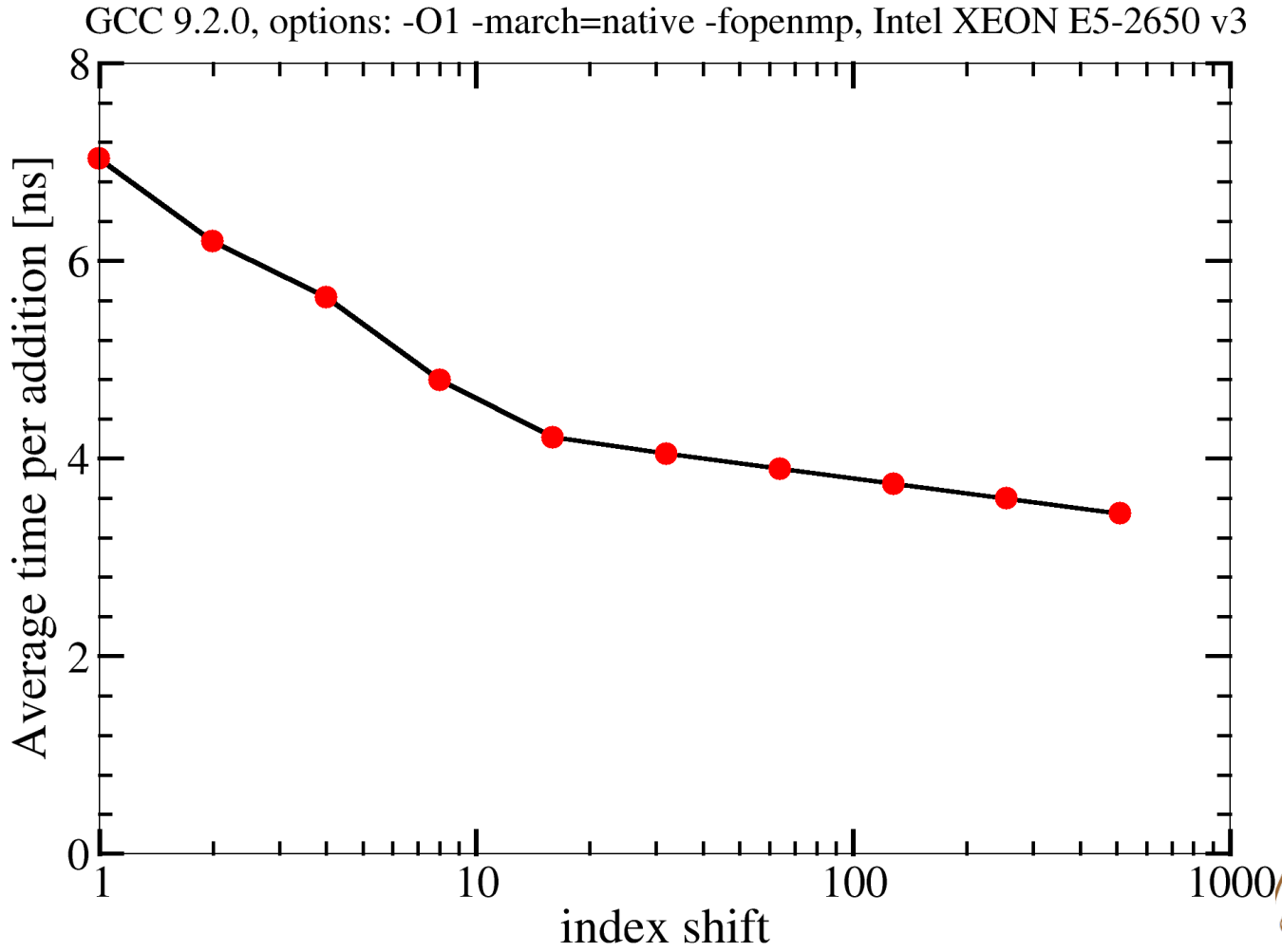
- Reading from remote memory carries penalty
- Typically flush to Mem when CL transfer between processors (e.g. C0➔C3)
- **Rem:** For reading no false sharing problems
  - CL can be held in more than one L1 cache for read
  - Exclusive copy required when writing!



Demonstrate: False sharing
Updating elements in an integer array

```
!$omp parallel default(none) shared(shift, a, times), &
!$omp    private(index, rep, ftime, stime)
     ! Loop shift from 1, 2, 4, …
     index = 1 + omp_get_thread_num() * shift
     a(index) = a(index) + 1    ! warm-up

     !$omp barrier
     stime = omp_get_wtime()
     do rep = 1, nrepeat
        a(index) = a(index) +1
     end do
     ftime = omp_get_wtime() - stime
     times(1+omp_get_thread_num()) = ftime/nrepeat
!$omp end parallel
```


Performance depending on size of shift
16 threads per node



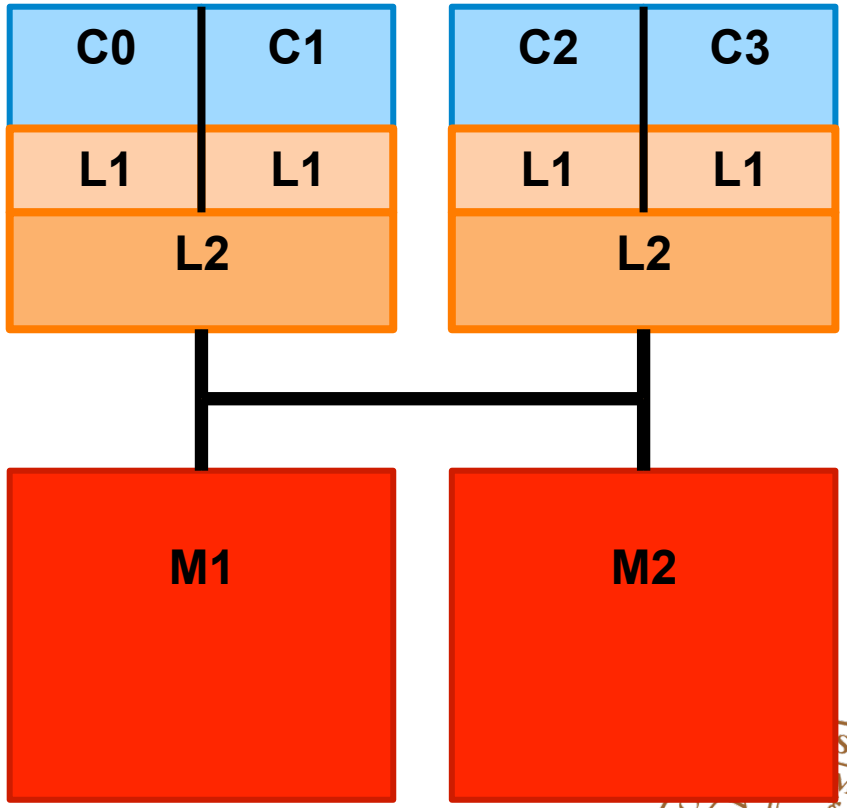
Discussion

- 16 integer per cache line
- Low level of optimisation required
  - Compiler find “short cut” at higher level
  - Most likely multiplication
- Huge performance penalty if multiple threads assessing same cache line
  - Minor improvement beyond shift=16

FIRST TOUCH

Memory system with multiple processors and multiple memory subsystems

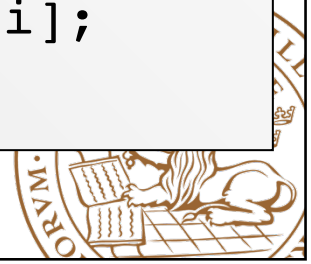
- Data allocated on memory local to one processor
- Typically happens when memory is “first touched”
  - E.g.: initialisation
  - **Not** malloc/allocate
- Avoid large data structures on:
  - Memory local to one of the processors
  - Memory local to wrong processor



Initialise data serially

- Serial thread on one core initialises
- Core places data on its local memory if possible
- Problems caused
  - Use part of memory systems (e.g. memory bus)
  - Some cores have to access remote memory

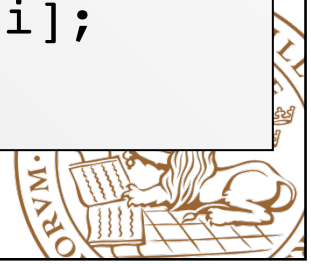
```
for (int i=0; i<VLENG; i++)
 { a[i] = 2.0 * (double)i;
   b[i] = 3.0 * (double)i;
   c[i] = (double)i;
 }

#pragma omp parallel for \
  shared(a, b, c)
 for(int i=0; i<VLENG; i++)
  { c[i] = a[i] + b[i];
  }
```



Initialise data in parallel

- Put parallelisation on initialisation
- Each thread places its share of data on its local memory
- Potential benefit in work loop:
  - Use all memory systems
  - Use thread local memory (requires same static schedule and binding or affinity for best performance)

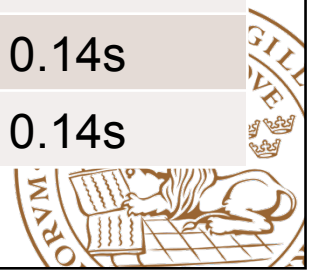
```
#pragma omp parallel for \
  shared (a, b)
for (int i=0; i<VLENG; i++)
 { a[i] = 2.0 * (double)i;
   b[i] = 3.0 * (double)i;
   c[i] = (double)i;
 }

#pragma omp parallel for \
  shared(a, b, c)
 for(int i=0; i<VLENG; i++)
  { c[i] = a[i] + b[i];
  }
```



Performance study

- Initialise 3 vectors of 500 million doubles
- Add two of them and store result in parallel in 3rd vector
- Compare time for add & store loop when using serial or parallel initialisation

Processor	Proc/ node	Threads	Compiler	Time with serial intial.	Time with parallel intial.
AMD 6220	2	16	icc 13.1.1	0.62s	0.25s
AMD 6238	4	48	icc 13.1.1	1.32s	0.10s
Intel E5-2660	2	16	gcc 4.7.2	0.59s	0.22s
Intel E5-2660	2	16	icc 12.1.4	0.47s	0.18s
Intel E5-2650 v3	2	20	gcc 4.9.3	0.29s	0.14s
Intel E5-2650 v3	2	20	icc 16.0.1	0.29s	0.14s



Performance study – current CPUs

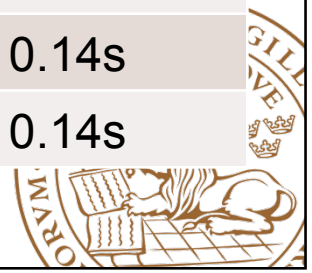
- Initialise 3 vectors of 2000 million doubles
- Add two of them and store result in parallel in 3rd vector
- Compare time for add & store loop when using serial or parallel initialisation

Processor	Proc/ node	Cores	Compiler	Time with serial intial.	Time with parallel intial.
AMD EPYC 7702	2	128	gcc 9.2.0	2.51s	0.23s
AMD EPYC 7702	2	128	icc 19u5	2.51s	0.23s
AMD EPYC 7702	2	64 of 128	icc 19u5	1.57s	0.46s
AMD EPYC 7542	2	64	gcc 9.2.0	2.75s	0.27s
Intel Xeon 8280	2	56	icc 19u5	1.83s	0.29s
Intel Xeon 8280	2	28 of 56	icc 19u5	1.17s	0.59s

Using `calloc` vs `malloc` (C-issue)

- Both functions are used to allocate memory
  - `calloc` zeros out the memory, `malloc` does not
- Compare time for parallel add & store loop when using parallel initialisation and either `malloc` or `calloc`
- Problemsize: 500 million

Processor	Proc/ node	Tasks	Compiler	Time for `malloc`	Time for `calloc`
AMD 6220	2	16	icc 13.1.1	0.25s	0.56s
AMD 6238	4	48	icc 13.1.1	0.10s	0.60s
Intel E5-2660	2	16	gcc 4.7.2	0.23s	0.23s
Intel E5-2660	2	16	icc 12.1.4	0.18s	0.22s
Intel E5-2650 v3	2	20	gcc 4.9.3	0.14s	0.14s
Intel E5-2650 v3	2	20	icc 16.0.1	0.17s	0.14s



Using `calloc` vs `malloc` (C-issue)

- Both functions are used to allocate memory
  - `calloc` zeros out the memory, `malloc` does not
- Compare time for parallel add & store loop when using parallel initialisation and either `malloc` or `calloc`
- Problemsize: 2000 million

Processor	Proc/ node	Cores	Compiler	Time for `malloc`	Time for `calloc`
AMD EPYC 7702	2	128	gcc 9.2.0	0.23s	0.23s
AMD EPYC 7702	2	128	icc 19u5	0.23s	0.75s
Intel Xeon 8280	2	56	gcc 7.2.0	0.30s	0.30s
Intel Xeon 8280	2	56	icc 19u5	0.29s	0.67s

- Performance implications system dependent

Summary

- OpenMP constructs overheads on the µs scale
  - This is O(1000 cycles)
  - Need decent amount of work to amortise
- False sharing can severely degrade performance
  - Avoid access to cache line recently modified on another thread
- First touch
  - Absolutely crucial on cc-numa architectures