Shared Memory Architecture

Troels Haugbølle
haugboel@nbi.ku.dk

February 28, 2022

UNIVERSITY OF COPENHAGEN

Overview

Today:

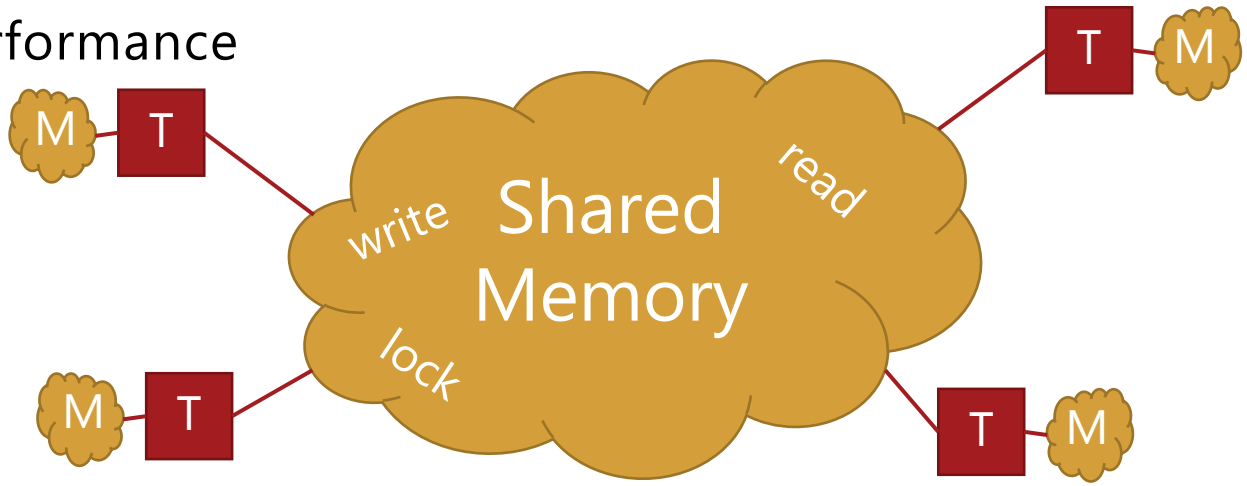
- Architecture: Shared memory systems
- Cache: coherence and false sharing
- Software vs Hardware: Programming models and optimal use of SHMP

Learning Objectives for this module:

- UMA, ccNUMA, and MP shared memory, limitations of shared memory
- Threads: different thread packages, working with threads, and coordination.
- Hands-on experience with OpenMP

Why parallel programming in shared memory architecture

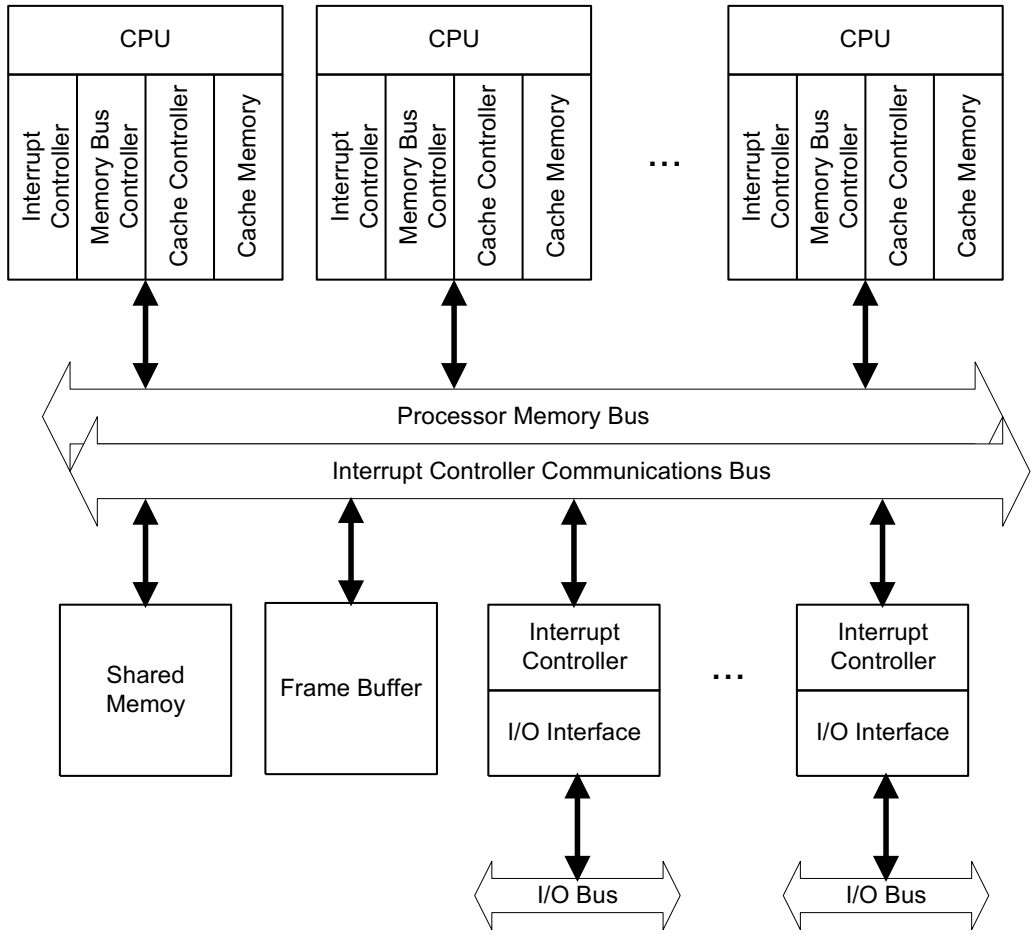
- Ease of use
- Incremental programming model
- Pervasive in clusters and workstations
- Today, even relevant for laptops or phones
- Good starting point for learning parallel programming
- Inconveniences:
  - Implicit update of memory can (will!) lead to race conditions
  - Hard to get good performance
  - Limited scalability



Architecture

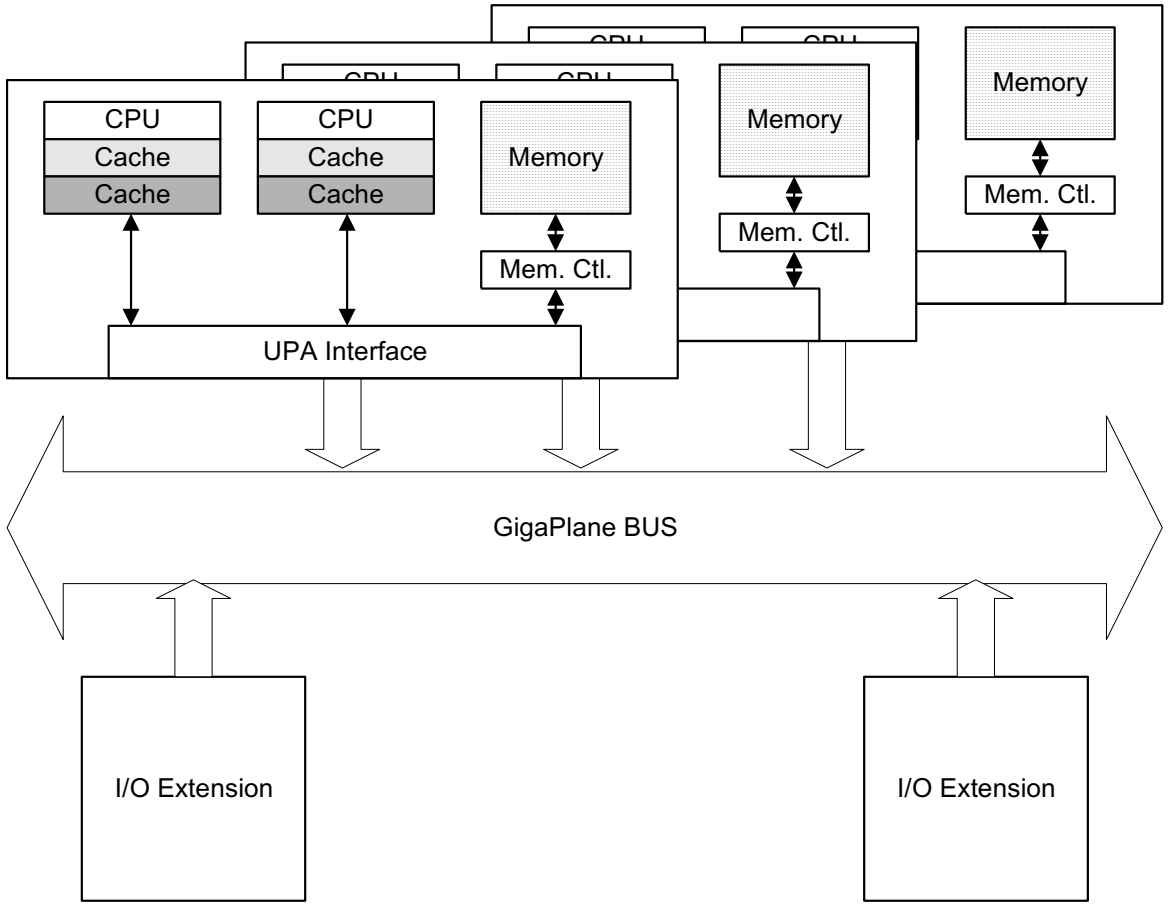
UMA: Unified Memory Architecture

- Shared bus architecture
- Historically dominant SHM architecture
- Not very scalable
- Simple memory structure



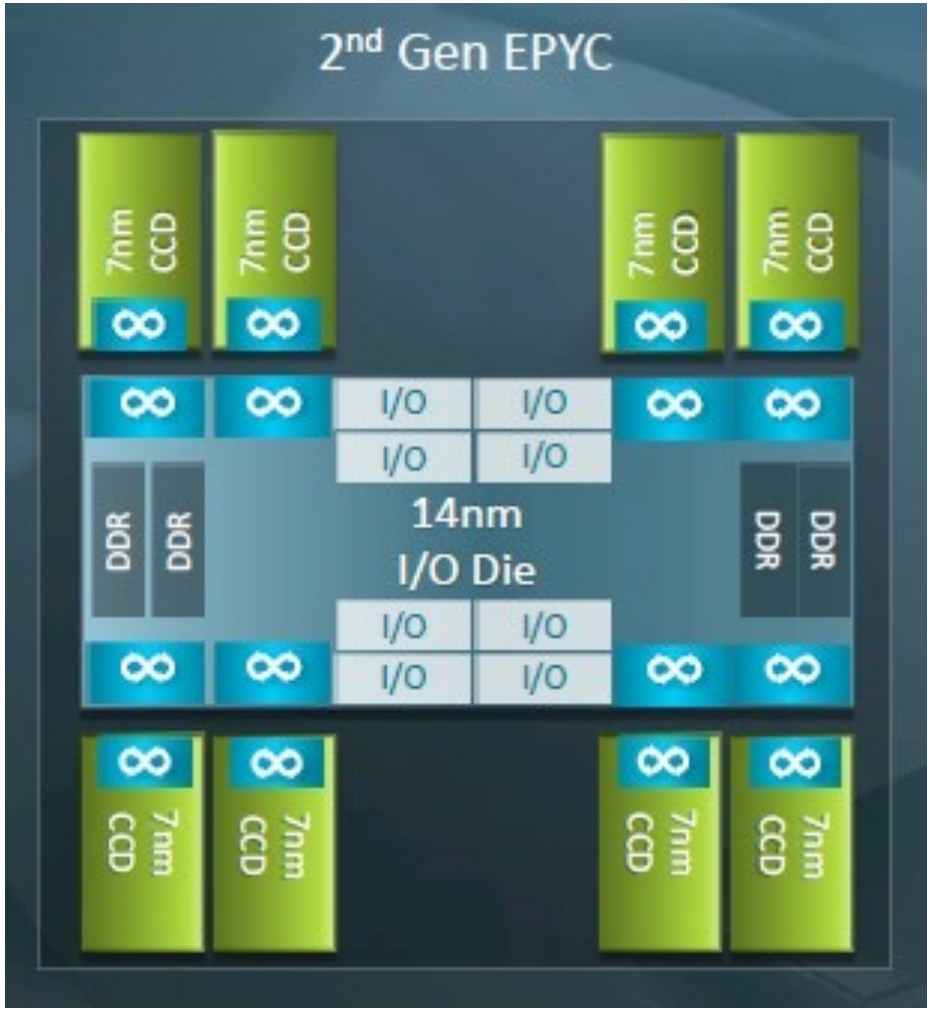
NUMA: Non-Uniform Memory Architecture

- Crossbar or network switched
- Rather complex
- Quite expensive (Fujitsu, HP, IBM, NEC, SGI, …)
- Can scale to hundreds of processors
- Memory consistency protocol major part of headache

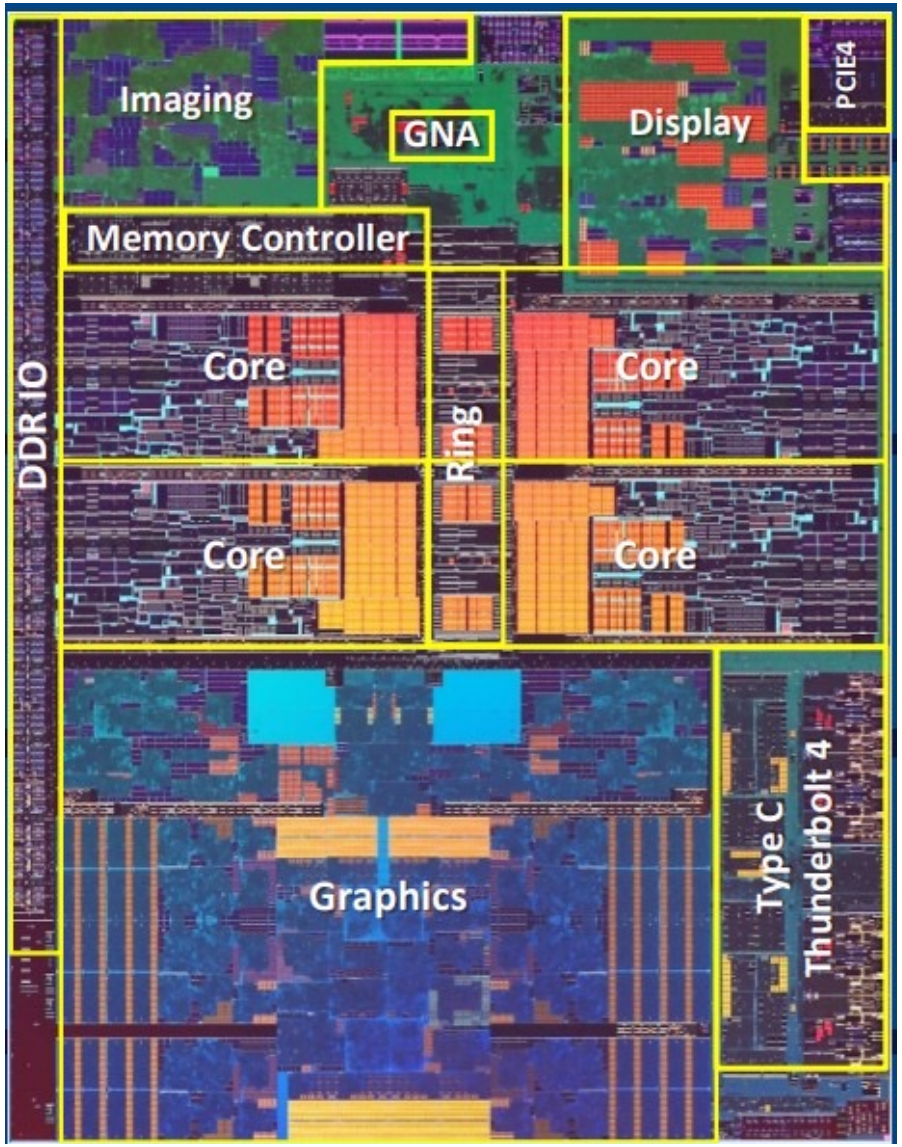


Modern variant: multi processor on a chip

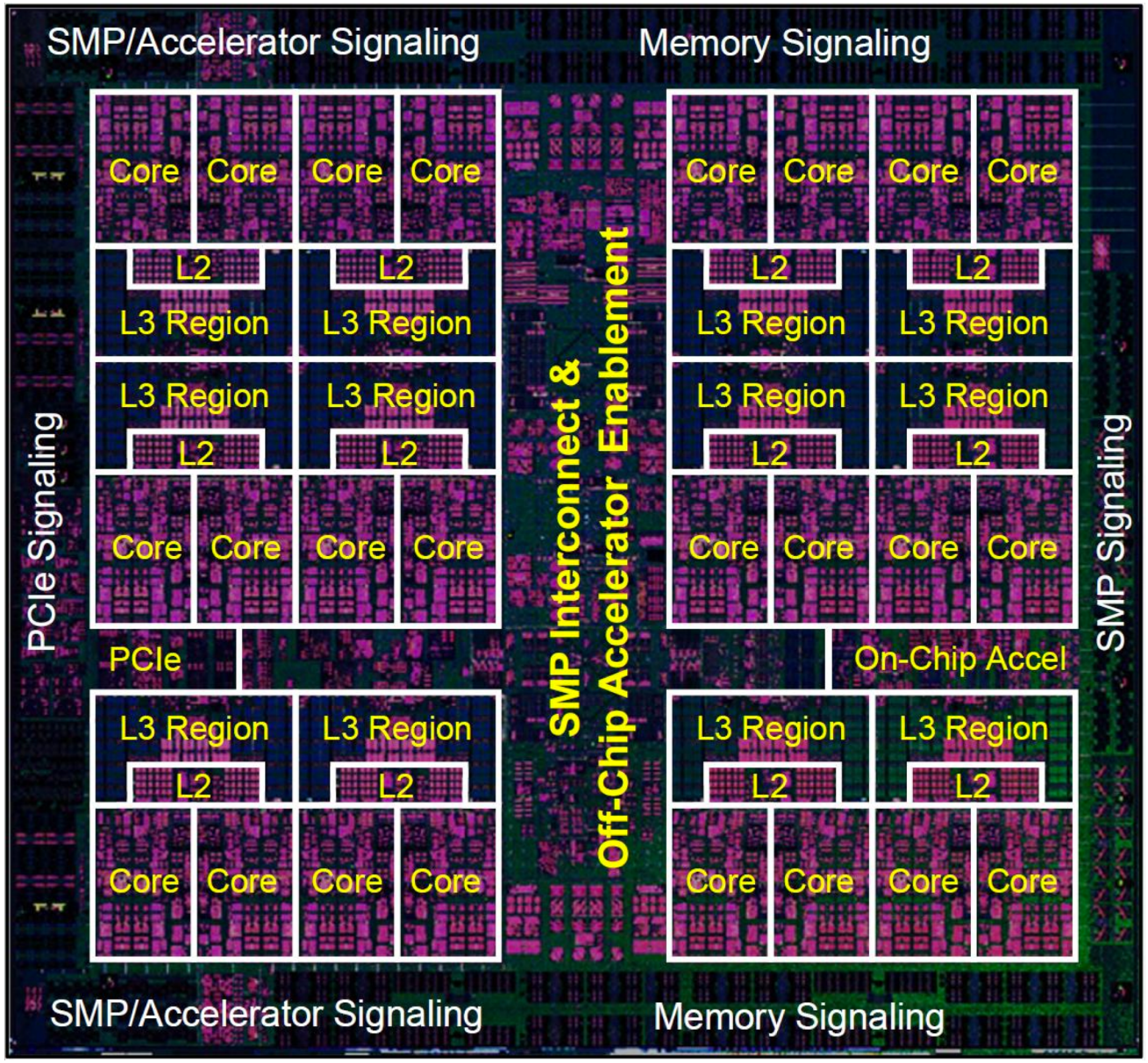
- Traditionally UMA but evolving towards NUMA
- Baked into silicon ➔ cheap
- Can scale to +100 cores
- Very low latency between cores
- Allow for more fine-grained (loop-level) parallelism
- Cluster nodes of dual-socket multi-core nodes ➔ (two+) levels of connectivity between cores



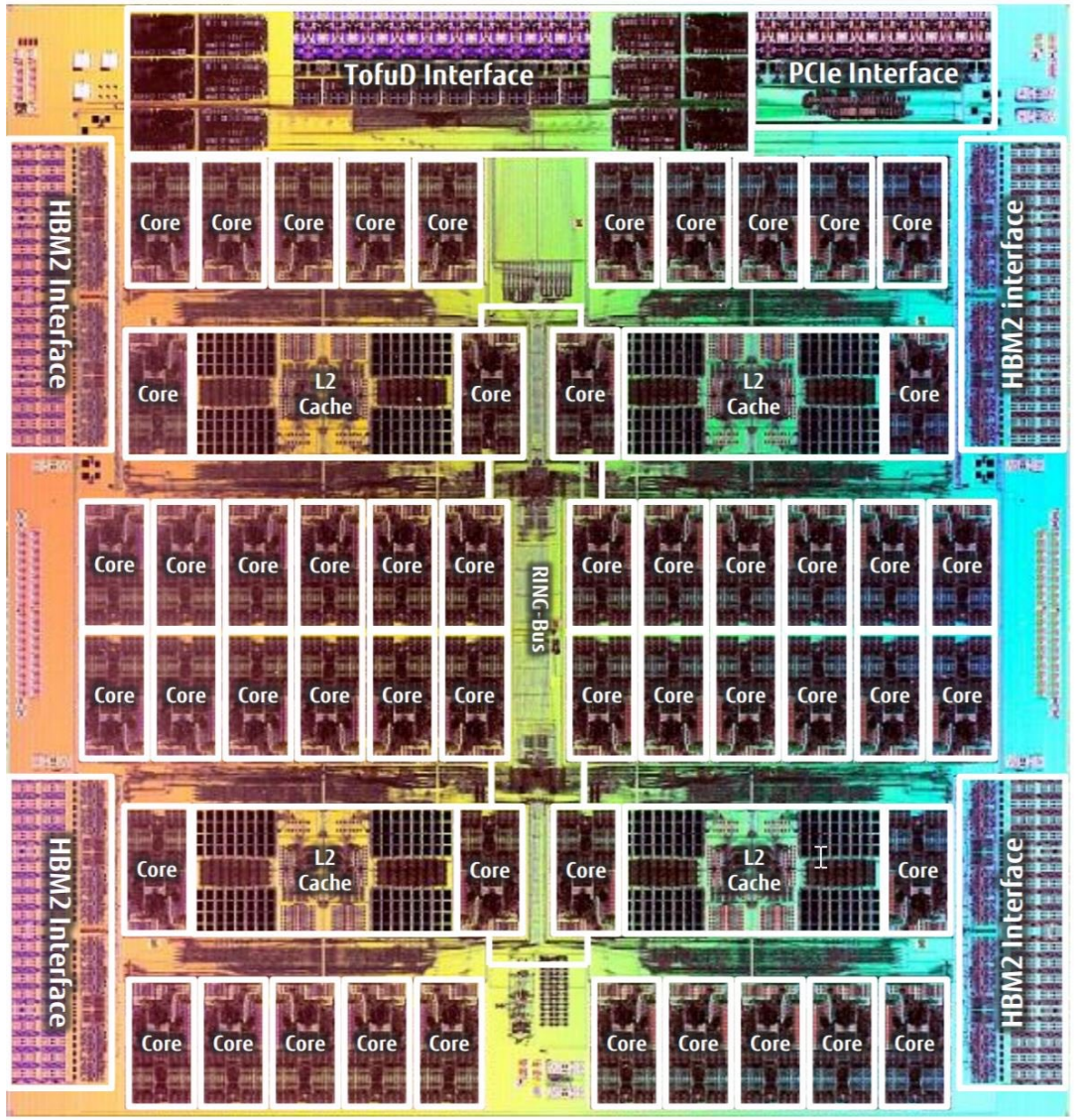
Intel Tiger-Lake



IBM Power 9


SMP/Accelerator Signaling
Memory Signaling
Core
L2
L3 Region
PCIe Signaling
SMP Interconnect &
Off-Chip Accelerator Enablement
SMP Signaling
PCIe
On-Chip Accel
SMP/Accelerator Signaling
Memory Signaling

Fujitsu ARM A64FX (Fugaku CPU)

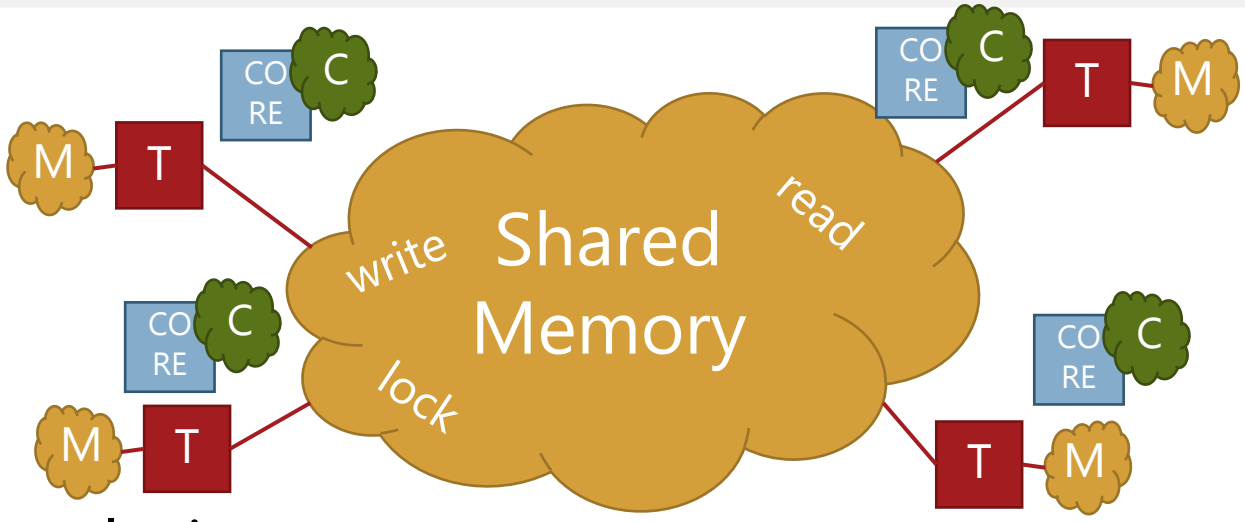


Virtual MP – Hyperthreading

- Idea: computation limited by memory stalls
- Cheap: add extra registers per virtual core and some control logic
- Lower transistor usage for each virtual core
- Hyperthreading / SMT (Intel x86, AMD):
  - Hardware threads shifts are activated on cache miss
  - Has inherent overhead and adapts to fewer threads
- Hardware Simultaneous MultiThreading (IBM, intel KNL)
  - Threads shift on every cycle
  - Simpler and has less overhead
  - Requires many threads per physical core – typically 4 to 8
- Advantage: can improve performance if memory access is non-uniform
- Disadvantage: more threads share resources (cache, registers)

Cache

Cache and multiprocessing



- Simple picture of shared memory:
  - Each thread access memory shared
  - Order of execution depend on timing
- Cache introduces an extra layer of complexity
  - Cache helps with
    - Temporal locality: data that is reused with small separation in instruction stream
    - Spatial locality: if data is used, then data nearby in memory may be reused
  - CPUs reads and writes from cache, not from main memory
- How can view of memory be kept coherent for all CPU cores?

  ➔ Cache coherence!
- Strict definition: any read should return latest write

Cache Coherence

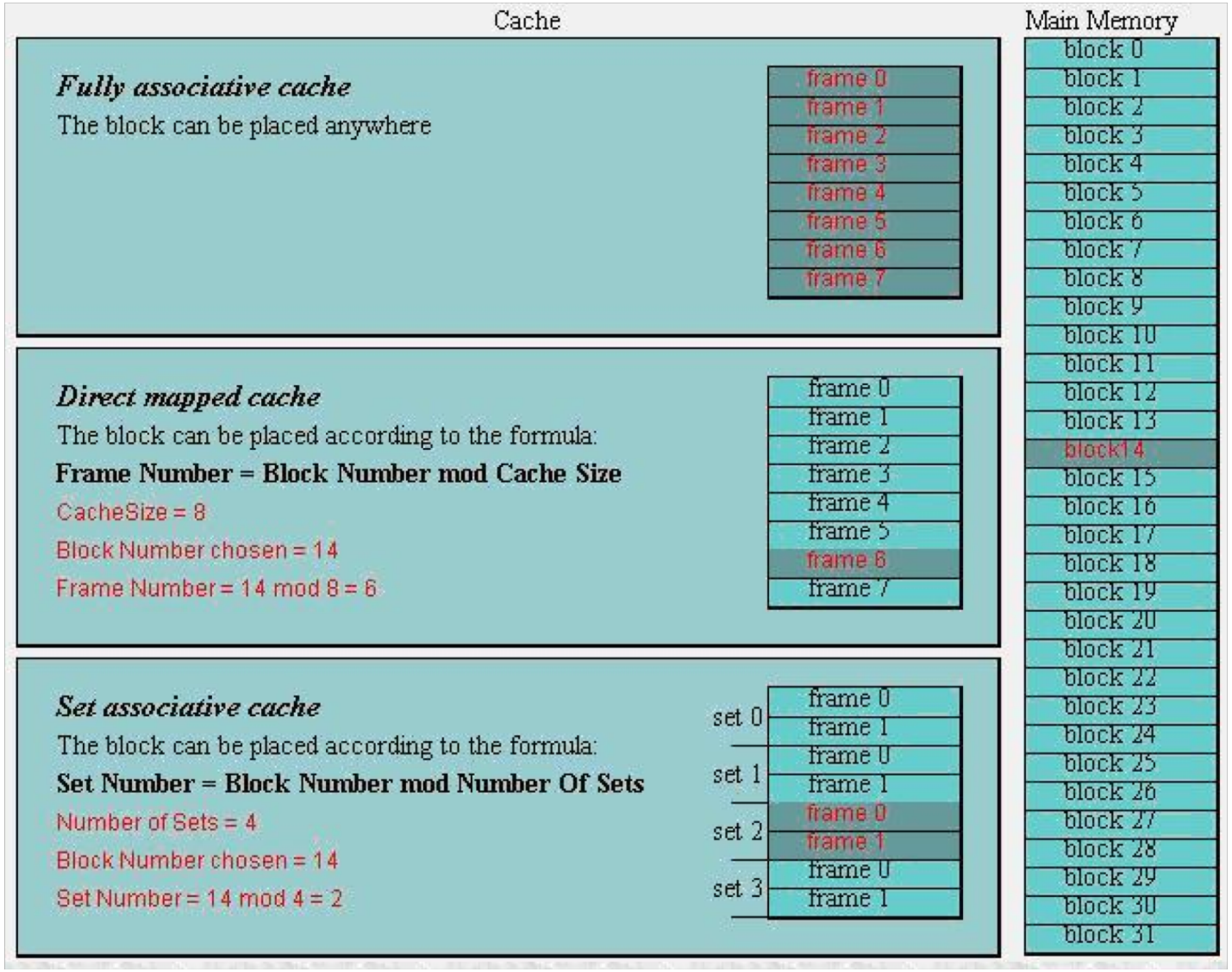
- Strict definition: any read should return latest write

- In practice:
  - A write to memory must eventually be seen by a read from memory
  - All writes are seen in the order they are committed

- Cache protocols implement algorithms to
  - Broadcast knowledge about data present in cache
  - Keep track of data present in multiple caches
  - Invalidate data in cache that differs from data in main memory

- Consequence:
  - If (a lot of) memory is updated by different processors (data present in multiple caches), the system bus can be overwhelmed by cache coherency traffic

Cache memory



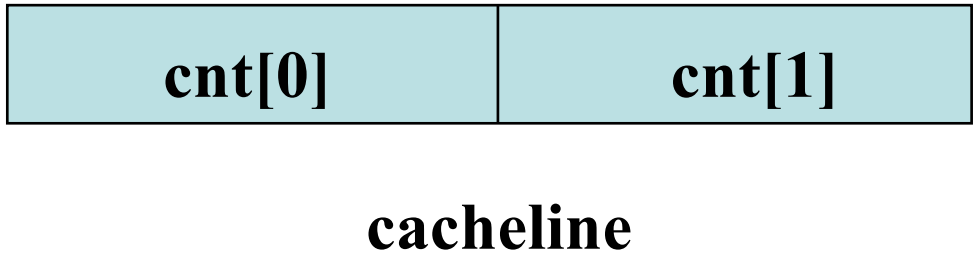
1 block is a cache line
Size: 64 bytes

A cache line corresponds to
8 double precision floats

False Sharing through cache lines

- Unfortunate memory layout may cause performance problems due to *false sharing*

```
Int cnt[2];
Thread A: cnt[0]++;
Thread B: cnt[1]++;
```



False Sharing through cache lines

```
#define threads  (8)
volatile int count[threads];

#ifdef FALSE
worker(void * arg)
{
  int i,index=(int)arg;
  for(i=0;i<1000000000; i++)
    count[index]++;
}
#else
worker(void * arg)
{
  int i,index=(int)arg;
  int temp=0;

  for(i=0;i<1000000000; i++)
    temp++;

    count[index]+=temp;
}
#endif
```

```
main()
{
  pthread_t t[threads];
  void *val;
  int i;

  for(i=0;i<threads;i++)
    pthread_create(&(t[i]),NULL,worker,(void *)i);

  for(i=0;i<threads;i++)
    pthread_join(t[i], &val);
}
```

False Sharing

real	1m1.848s
user	7m29.350s
sys	0m0.000s

No False Sharing

real	0m3.672s
user	0m29.200s
sys	0m0.000s

Software vs Hardware

Parallelism in Software mapped to Hardware

Different levels of coordination

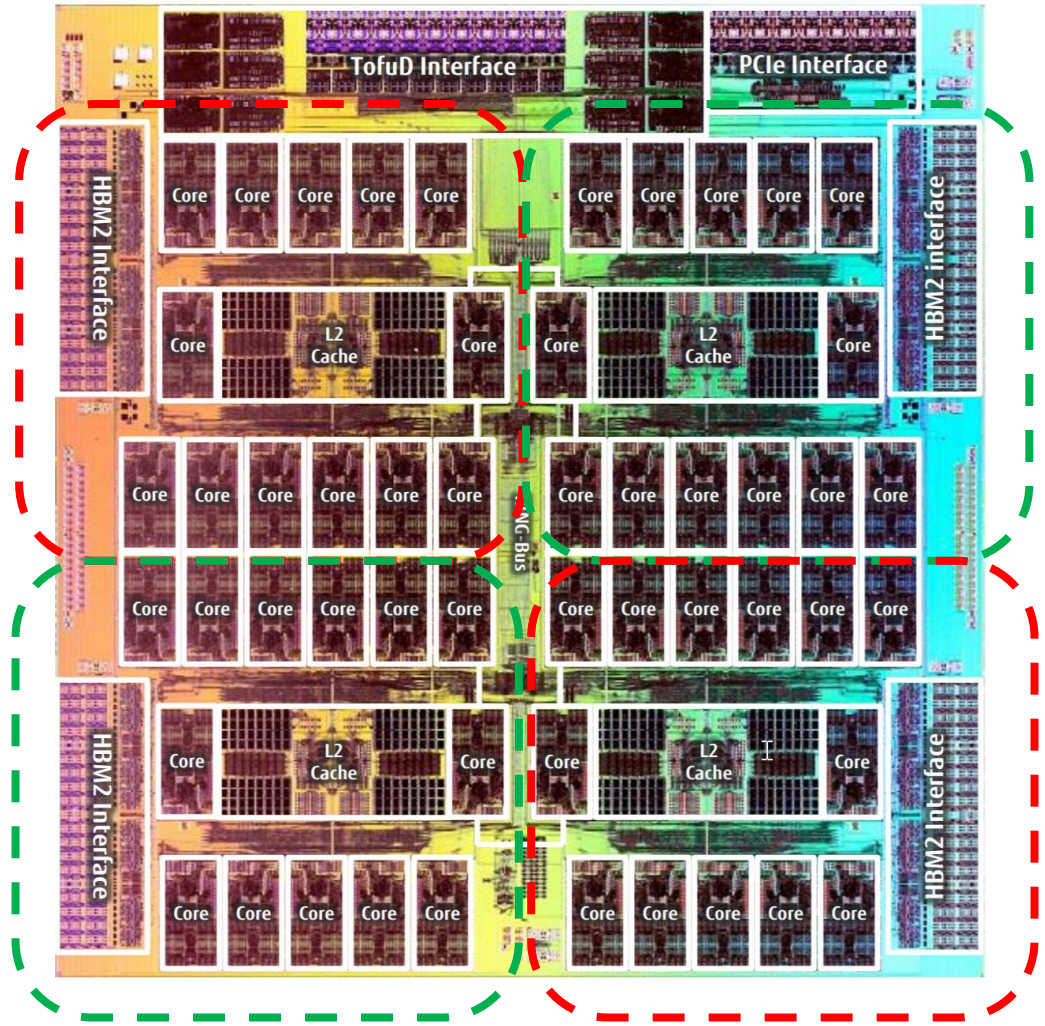
- fine-grained: SIMD and vectorization, instruction level
  - Data elements split out in packages in the *inner loop*. Typically <10 elements wide. *Register level* sharing.

- Medium-grained: Shared memory, thread coordination
  - Data is shared but partitioned in chunks. Typically ≈1000 elements per chunk. Inner and outer loops. *Implicit sharing* through main memory.

- Coarse-grained: message passing, process coordination
  - Data is partitioned at the program level. *Explicit sharing* through exchange of messages.

Parallelism in Software mapped to Hardware

- Shared memory processing on modern processors
  - HPC nodes typically ccNUMA dual-processor. Access times very different on socket and off socket
  - Inside socket access time may be non-uniform too
  - Cache domains logical boundary for deployment of shared memory
- <u>Rule of thumb:</u>
  - limit shared memory processing to largest uniform-memory domain
  - test benefit of hyper-threading



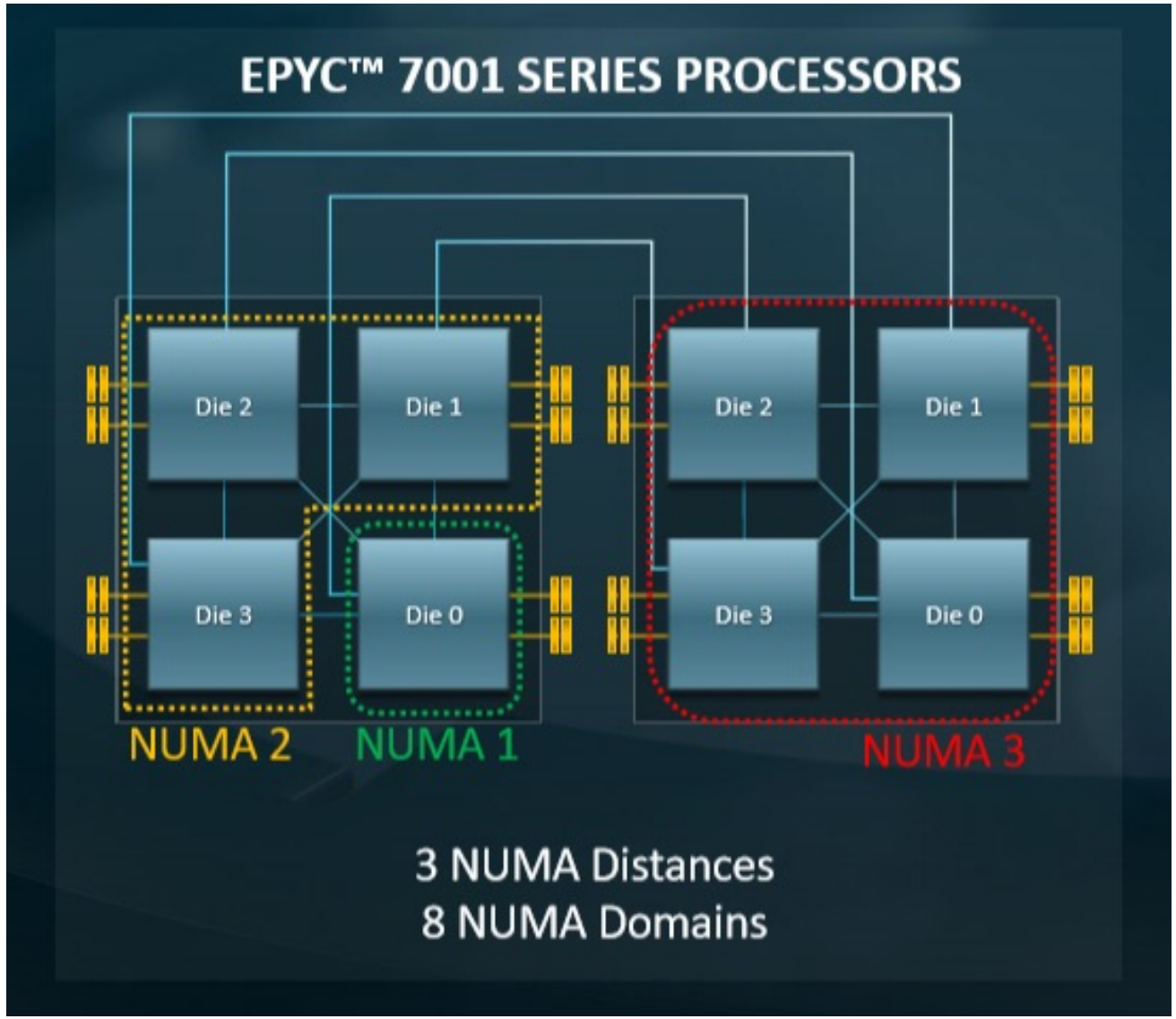
ARM A64FX: four on-chip UMA domain: 12 + 1 cores

Parallelism in Software mapped to Hardware

MODI cluster - Epyc Naples:

- Each CPU socket has 4 dies with its own memory controller
- 8 NUMA domains, 8 cores / 16 threads per domain
- Best options:
  - 8 cores per process
  - maybe 32 cores works ?
  - 64 cores suboptimal for most workloads
- In Linux you can see system topology with: `numactl -H`

```
node   0   1   2   3   4   5   6   7
  0:  10  16  16  16  32  32  32  32
  1:  16  10  16  16  32  32  32  32
  2:  16  16  10  16  32  32  32  32
  3:  16  16  16  10  32  32  32  32
  4:  32  32  32  32  10  16  16  16
  5:  32  32  32  32  16  10  16  16
  6:  32  32  32  32  16  16  10  16
  7:  32  32  32  32  16  16  16  10
```



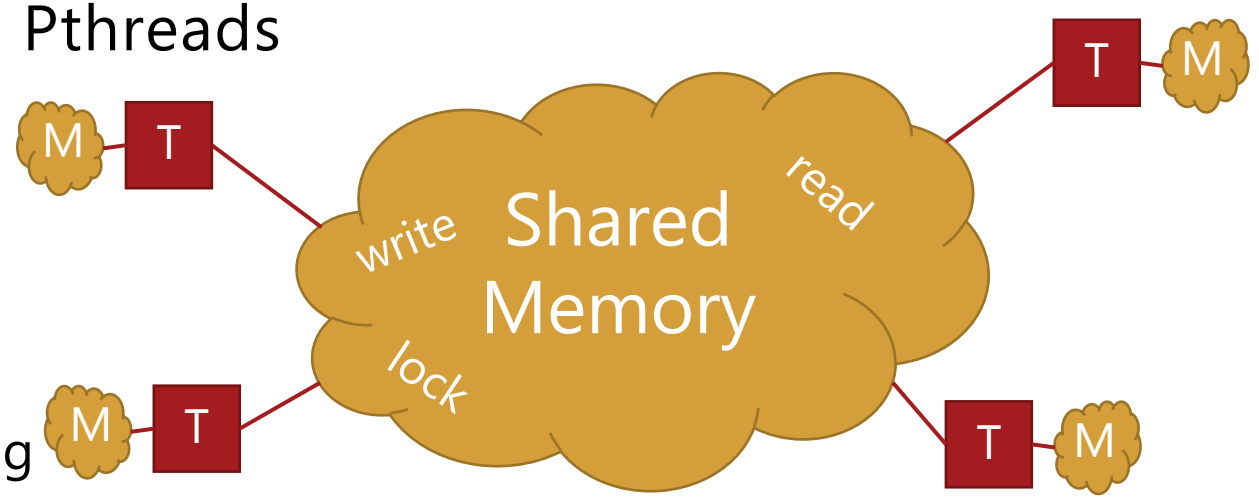
Shared memory Programming Models

- Thread based models (MIMD):
  - A program has a single shared memory space
  - Threads operate concurrently and independently on the data
  - Threads can be created and destroyed dynamically
  - Examples: Pthreads and *OpenMP*
- Data Parallel models ("software SIMD"):
  - Parallel work is done on a partitioned data set
  - Each task does the same work on different sections of the data
  - Example: OpenACC, Python Parallel Map
- Compiler Auto Parallelisation
  - Add compiler flag; hope for the best: only works for very simple for-loops
- Libraries: linear algebra, search, FFT, …

OpenMP for Shared Memory

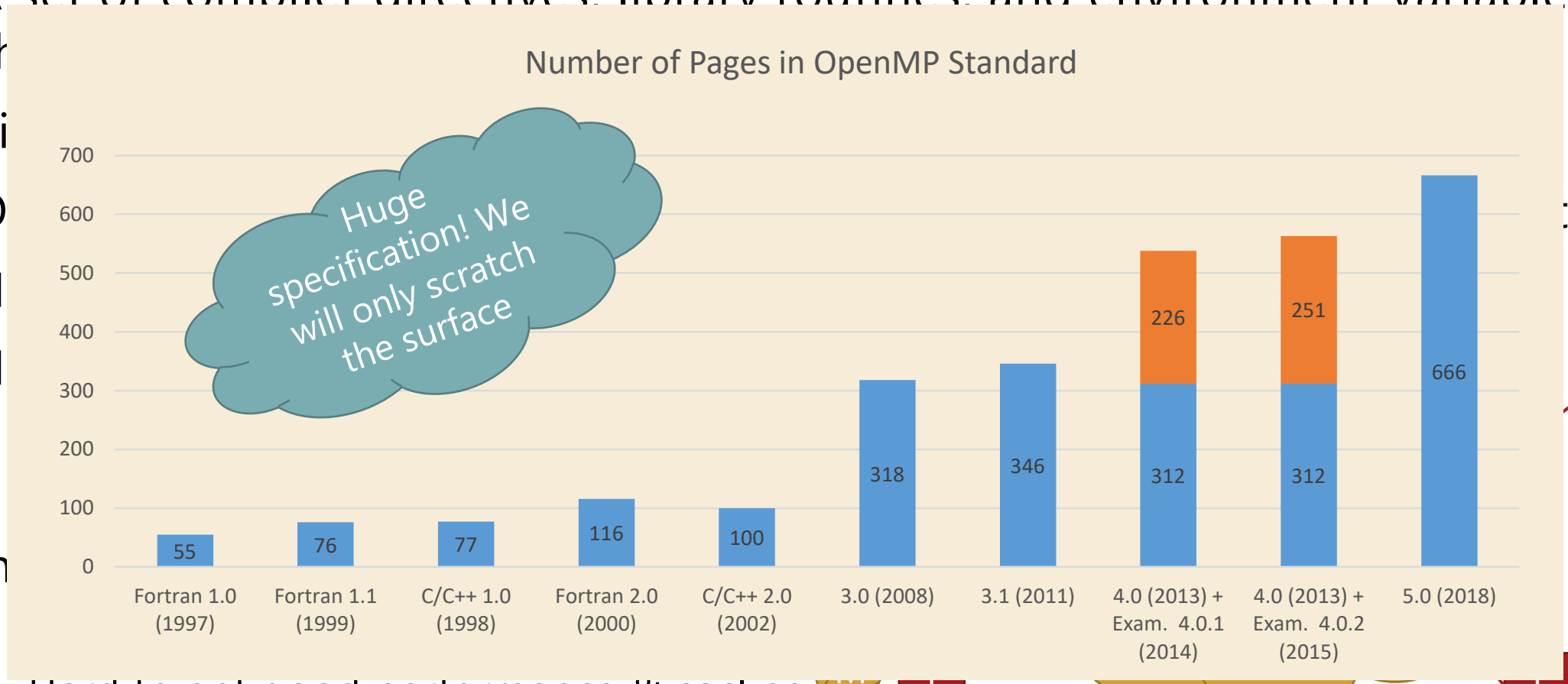
What is OpenMP?

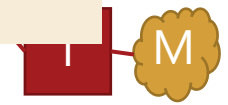
- A set of compiler directives, library routines, and environment variables for shared memory programming in C, C++, and Fortran
- Simplifies writing multi-threaded programs
- Open standard defined in 1997 to merge a plethora of proprietary options
- Has grown to cover SMPs, vectorization and accelerators
- Handles the boilerplate code of e.g. Pthreads
  - Fork and joining of threads
  - Thread Pool, distribution of work
- Inconveniences:
  - Can (will!) lead to race conditions
  - Hard to get good performance & scaling
  - OpenMP parallelize the loops you tell it to indiscriminately



What is OpenMP?

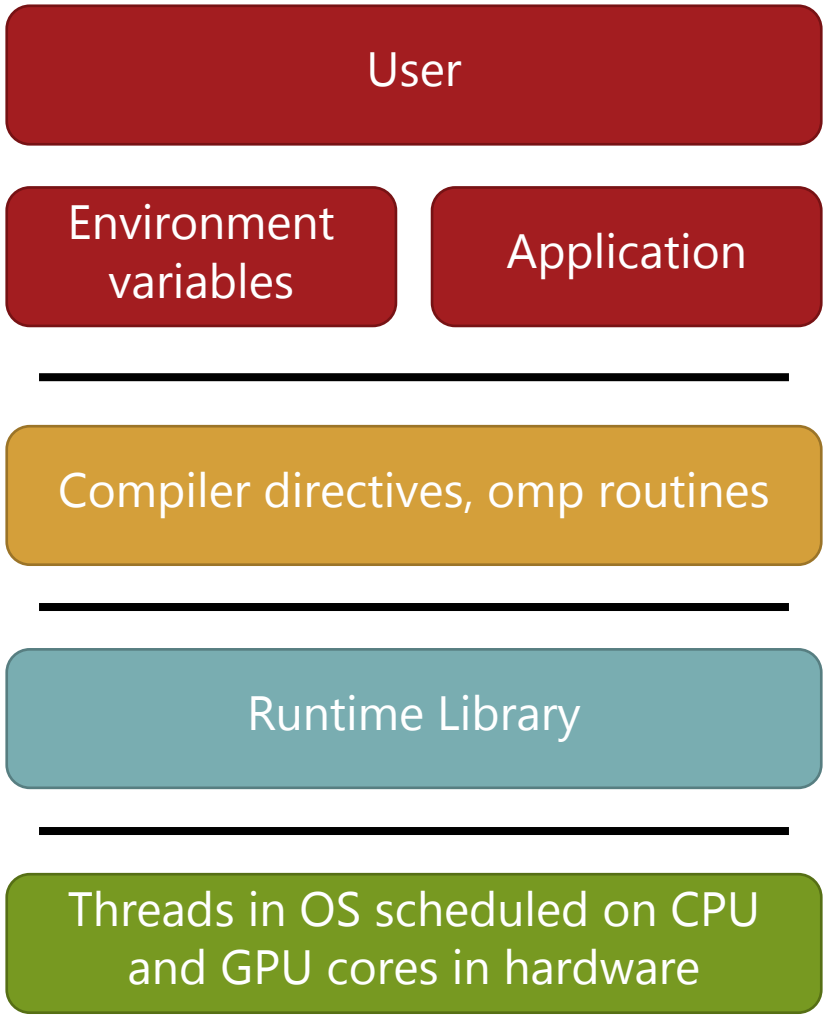
- A set of compiler directives, library routines, and environment variables for sh
- Si
- O tions
- H
- H
  -
  -
- In
  -
  - Hard to get good performance & sacling
  - OpenMP parallelize the loops you tell it to indiscriminately



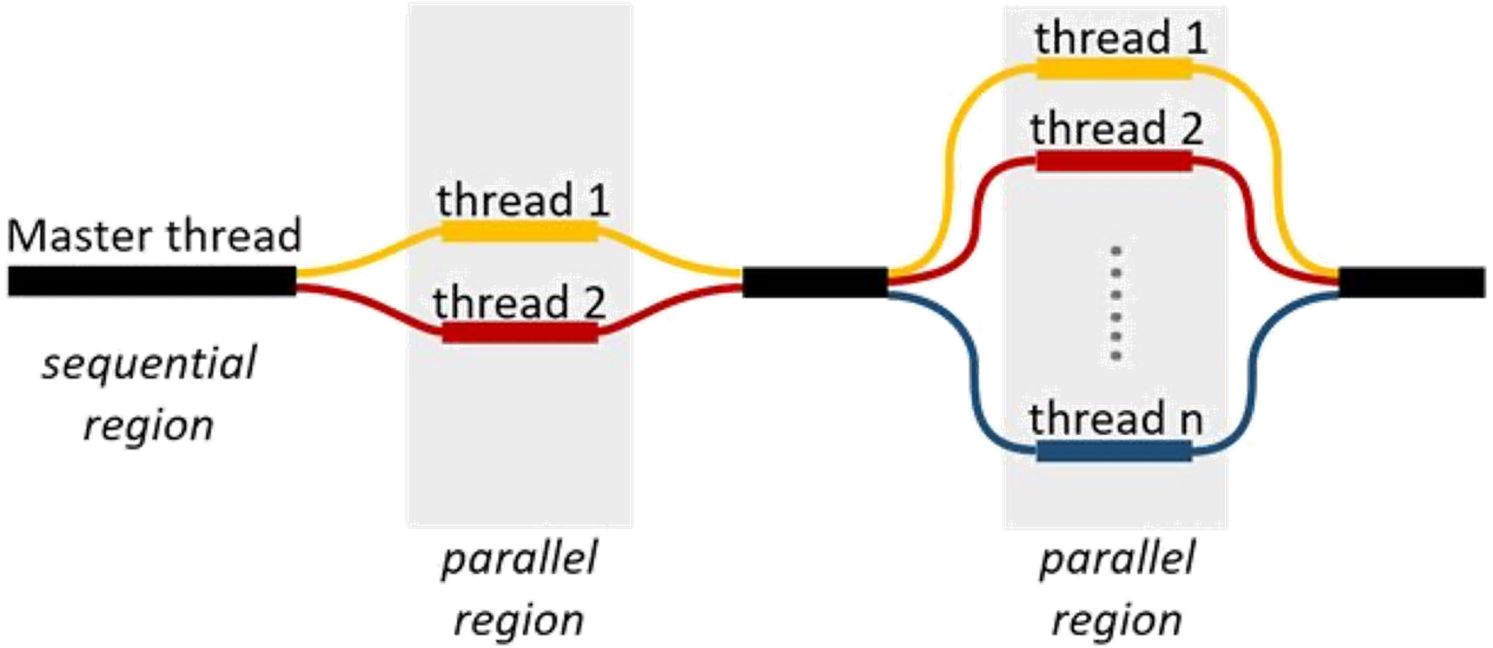


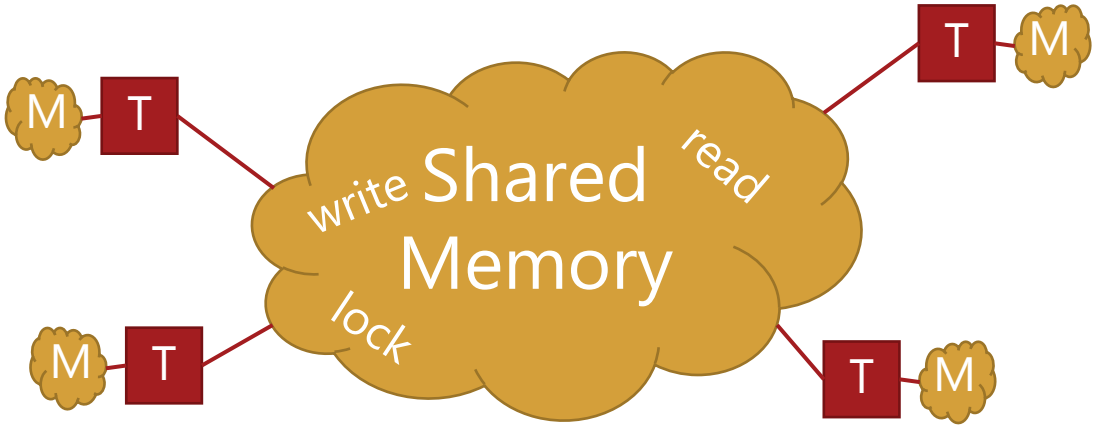
OpenMP Stack Overview

- User view: Environment variables set
  - How many threads to use: OMP_NUM_THREADS=4
  - Private memory per thread: OMP_STACKSIZE=2m
  - Affinity between threads and cores: OMP_PLACES=cores
  - How to bind to cores: OMP_PROC_BIND=close
  - Display information: OMP_DISPLAY_AFFINITY=true
- Developer view:
  - Directives: comments in code to instruct parallelism
    - #pragma omp parallel, critical, atomic, barrier, ...
  - Library routines for settings and info:
    - omp_get_num_threads()
    - omp_get_thread_num()
    - ...
- Low-level view:
  - OpenMP Runtime library: Schedule threads to do parallel work



Memory and fork-join execution model





- Program start: only master thread runs
- Parallel region: generate team of threads ("fork")
- Threads synchronize and leave parallel region ("join")
- Only master executes sequential region
- Directives
  - Task & data distribution
  - Synchronization
  - Data dependencies

Summary

- Shared memory programming is increasingly important due to the increasing number of CPU cores on a single chip
- Performance and scaling requires understanding the memory hierarchy:
  - shared memory gives easy access to all memory
  - if all threads needs simultaneous access to same memory location, system is overwhelmed
  - Beware! Granularity of memory access is on cache lines: false sharing impacts performance as much as true sharing
- Minimum size of data chunks is ≈1000 elements
- In practice, best scalability is usually obtained inside NUMA domains or CPU sockets to minimize cache coherency traffic; know thy hardware!
- Typically we program shared memory computers using the concept of "threads".