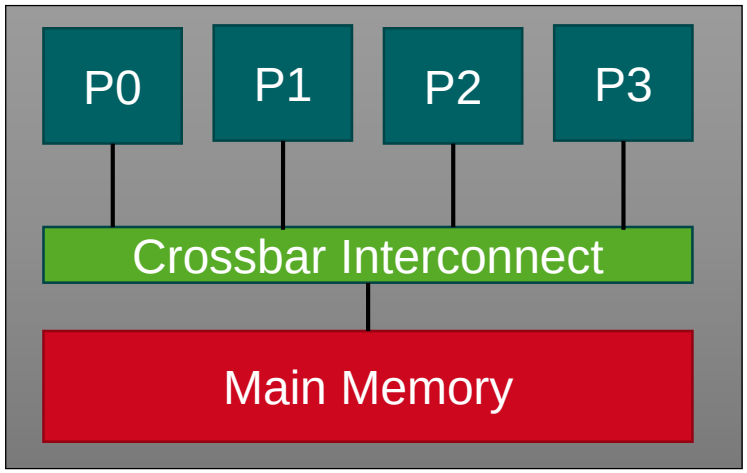
NUMA Architectures

Dirk Schmidl
schmidl@itc.rwth-aachen.de

Thanks to the following people for providing parts of the slides:
- Christian Terboven (RWTH Aachen)
- Sandra Wienke (RWTH Aachen)
- Michael Klemm (Intel)

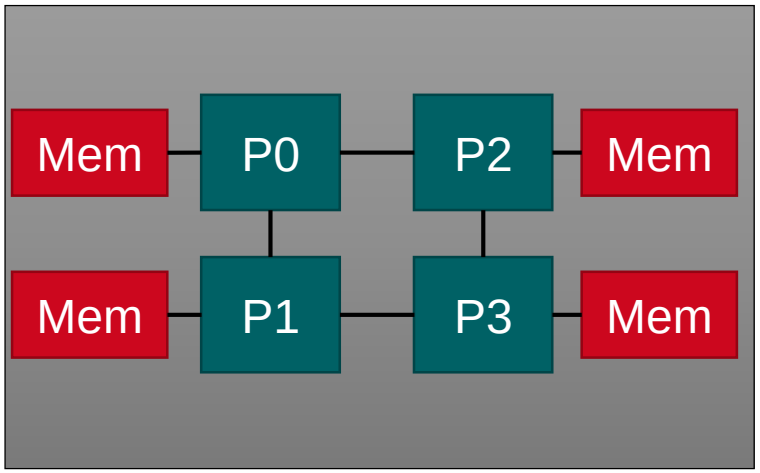
NUMA Architectures

- Uniform Memory Access (UMA):



- Pro:
  - Easier to program
- Con:
  - Main Memory bandwidth is a bottleneck
  - Limits system size

- Non Uniform Memory Access (NUMA):

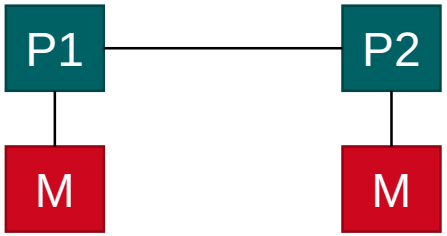


- Pro:
  - Higher overall memory bandwidth
  - Every processor adds bandwidth to the system
- Con:
  - Needs to be considered be programmers

NUMA Architectures

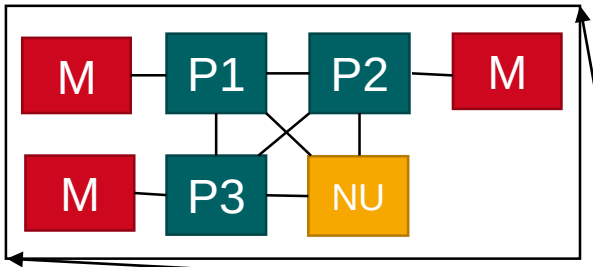
Standard Server:

- Blade-Server
- 2 Processors
- NUMA Architecture



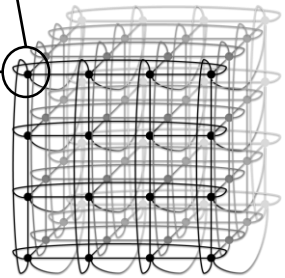
Numascale System (Running at the University of Oslo):

- 4 Racks
- 144 Processors
- 1728 Cores
- 4,6 TB Main Memory



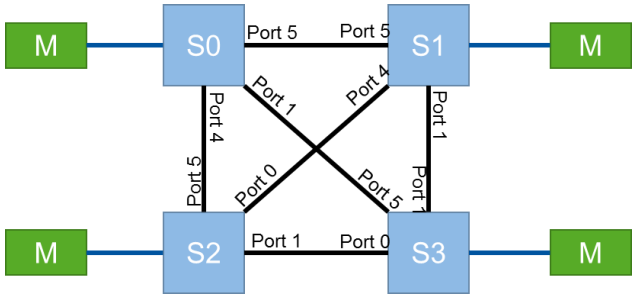
- 3 Sockets
- Numascale Chip

- 3D Torus Topology



NUMA Architectures

4 Socket Intel Xeon System:

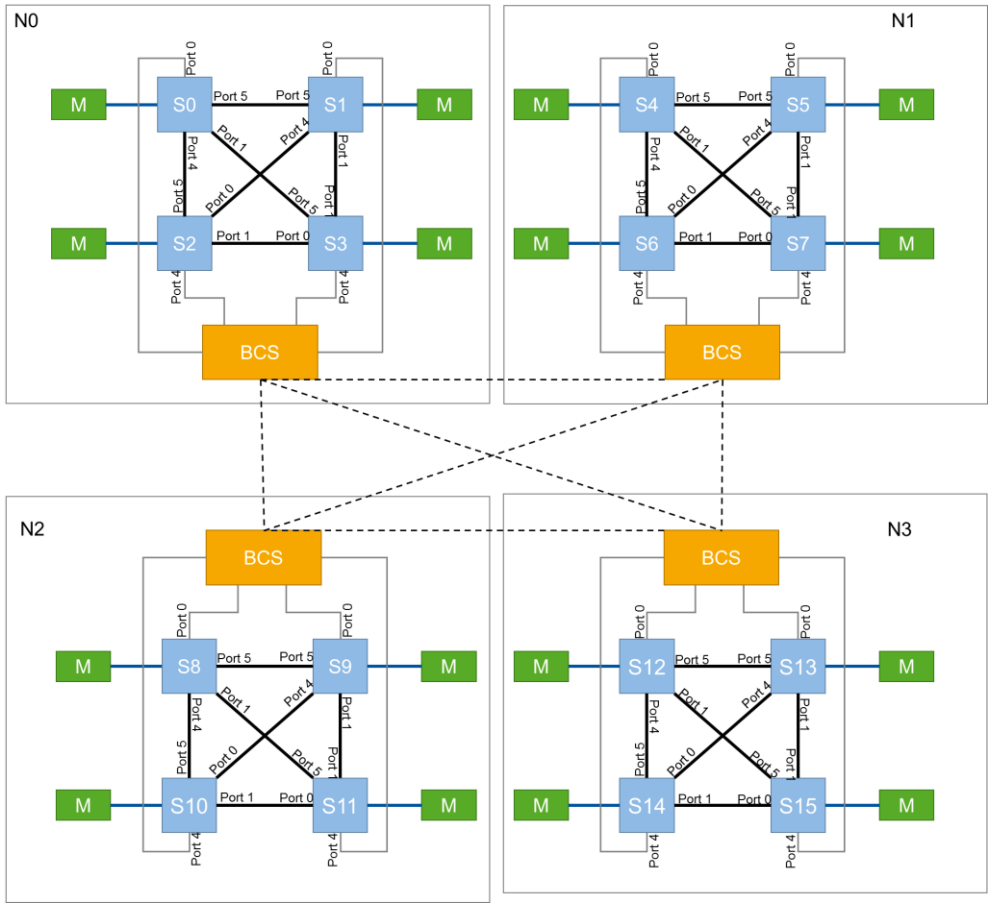


Hierarchical NUMA System:

- 2 Levels of Cache-coherent interconnects
- different protocols on different levels

	bandwidth	latency
lokal	~ 15,1 GB/s	~ 115 ns
QPI	~ 12,8 GB/s	~ 144 ns
BCS	~ 3,4 GB/s	~ 300 ns

16 Socket Bull Coherence Switch (BCS) System:

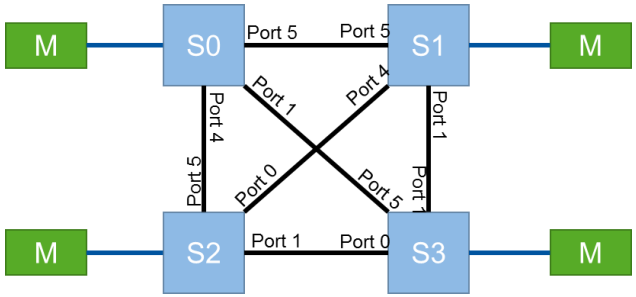


Investigating NUMA topologies

numactl - command line tool to investigate and handle NUMA under Linux

- $ numactl --hardware - prints information on NUMA nodes in the system
- $ numactl --show - prints information on available recourses for the process

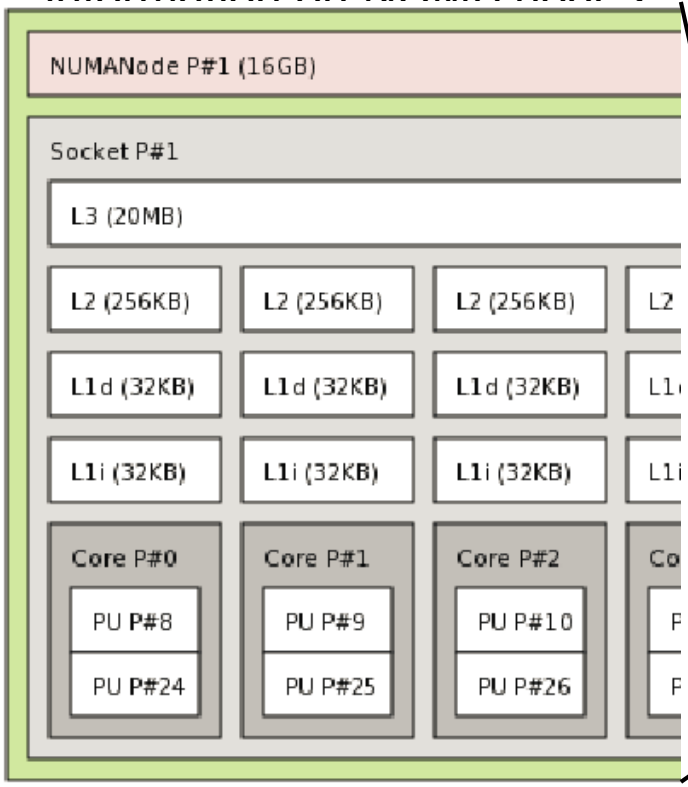
```
ds534486@linuxbsc001 [/home/ds534486] $ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
node 0 size: 65501 MB
node 0 free: 50533 MB
node 1 cpus: 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61
node 1 size: 65536 MB
node 1 free: 58763 MB
node 2 cpus: 2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62
node 2 size: 65536 MB
node 2 free: 52232 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63
node 3 size: 65536 MB
node 3 free: 46185 MB
node distances:
node   0   1   2   3
  0:  10  15  15  15
  1:  15  10  15  15
  2:  15  15  10  15
  3:  15  15  15  10
```

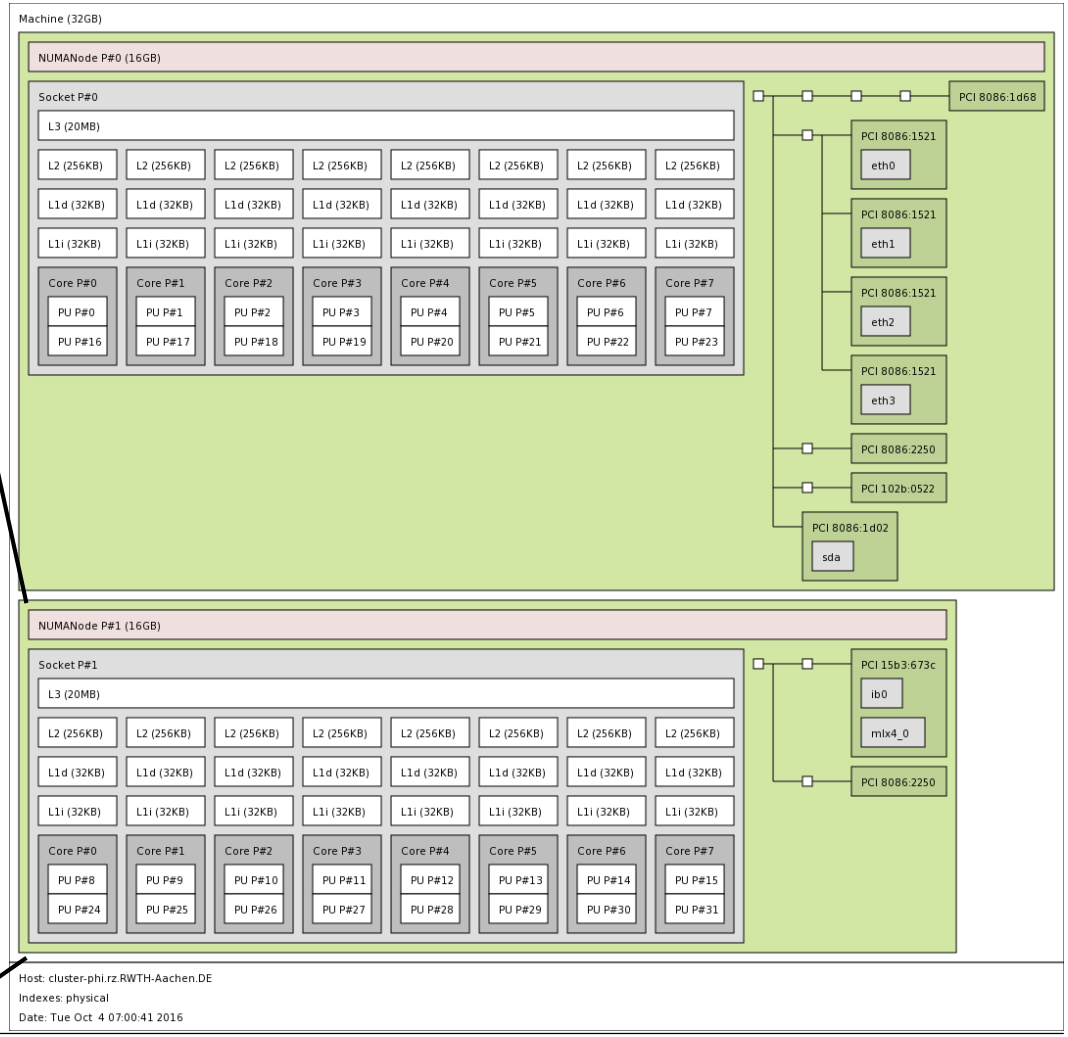


Investigating NUMA topologies

lstopo - tool to show the system topology

- information on NUMA nodes

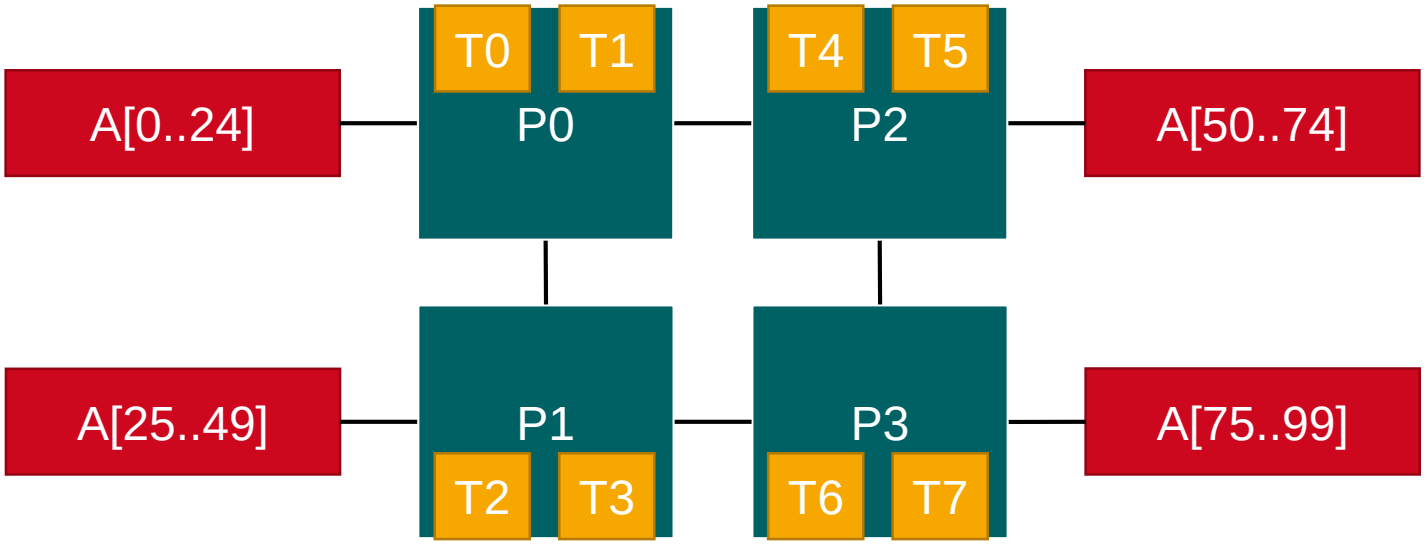




Optimizing NUMA accesses

Goal: Minimize the number of remote memory accesses as much as possible!

1. How are threads distributed on the system?
2. How is the data distributed on the system?
3. How is work distributed across threads?



Thread Placement in OpenMP

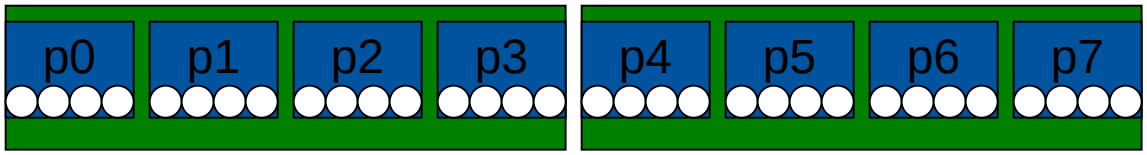
Thread Placement in OpenMP

- 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

- Available strategies:
  - **close**: put threads close together on the system
  - **spread**: place threads far apart from each other
  - **master**: run on the same place as the master thread

Thread Placement in OpenMP

- Assume the following machine:



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

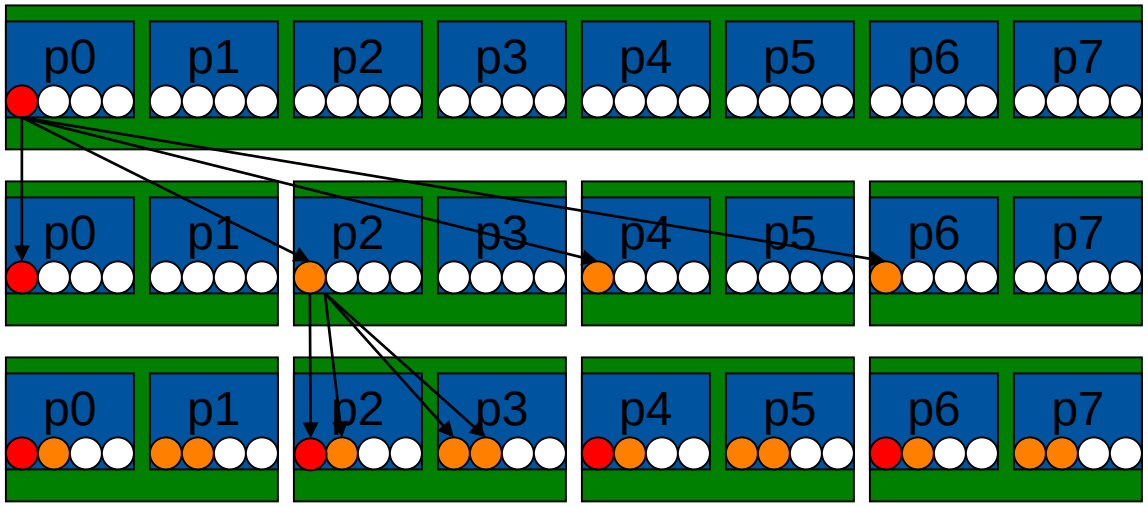
- Abstract names for OMP_PLACES:
  - threads: Each place corresponds to a single hardware thread on the target machine.
  - cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - sockets: Each place corresponds to a single socket (consisting of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

Thread Placement in OpenMP

- 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}, ... = {0:4}:8:4    = cores
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)
```

- Example
  - initial
  - spread 4
  - close 4

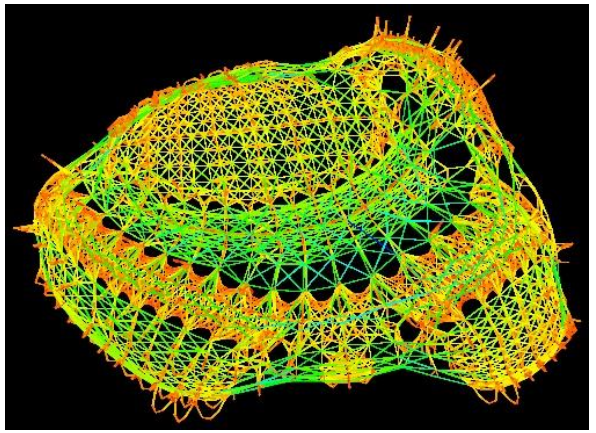


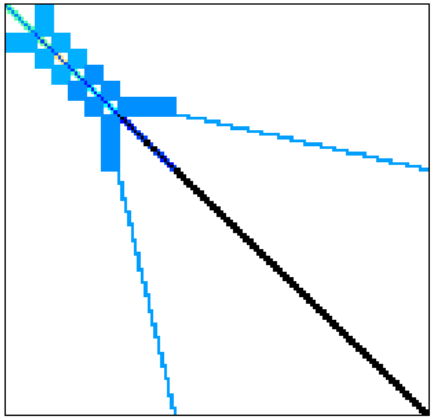
Thread Placement in OpenMP

- `int omp_get_place_num(void);`
  - returns the place number of the place where the encountering thread is bound to
- `void omp_place_get_num_procs(int place_num);`
  - returns the number of processors in place `place_num`
- `void omp_get_place_proc_ids(int place_num, int *ids);`
  - returns the ids of processors in place `place_num`
- `int omp_get_partition_num_places(void);`
  - returns the number of places of the partition of the encountering thread
- `void omp_get_partition_place_nums(int *place_nums);`
  - returns the number of places in the partition of the encountering thread

Case Study: CG

- Sparse Linear Algebra
  - Sparse Linear Equation Systems occur in many scientific disciplines.
  - Sparse matrix-vector multiplications (SpMxV) are the dominant part in many iterative solvers (like the CG) for such systems.
  - number of non-zeros << n*n





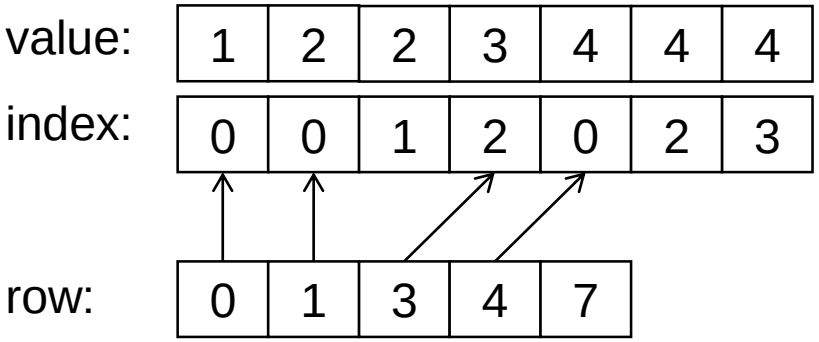
Case Study: CG

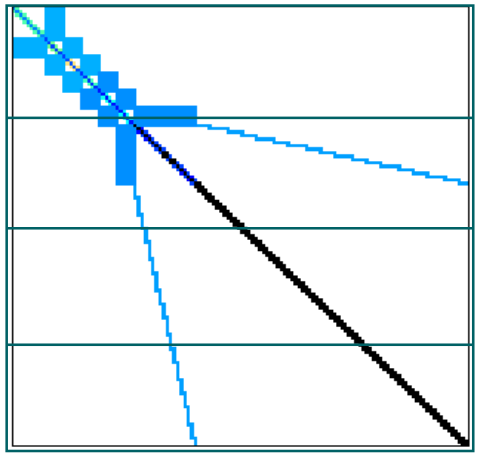
- $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 4 & 4 \end{pmatrix}$

```
for (i = 0; i < A.num_rows; i++){
  sum = 0.0;
  for (nz=A.row[i]; nz<A.row[i+1]; ++nz){
    sum+= A.value[nz]*x[A.index[nz]];
  }
  y[i] = sum;
}
```

$\vec{y} = A * \vec{x}$

- Format: compressed row storage
  - store all values and columns in arrays (length nnz)
  - store beginning of a new row in a third array (length n+1)

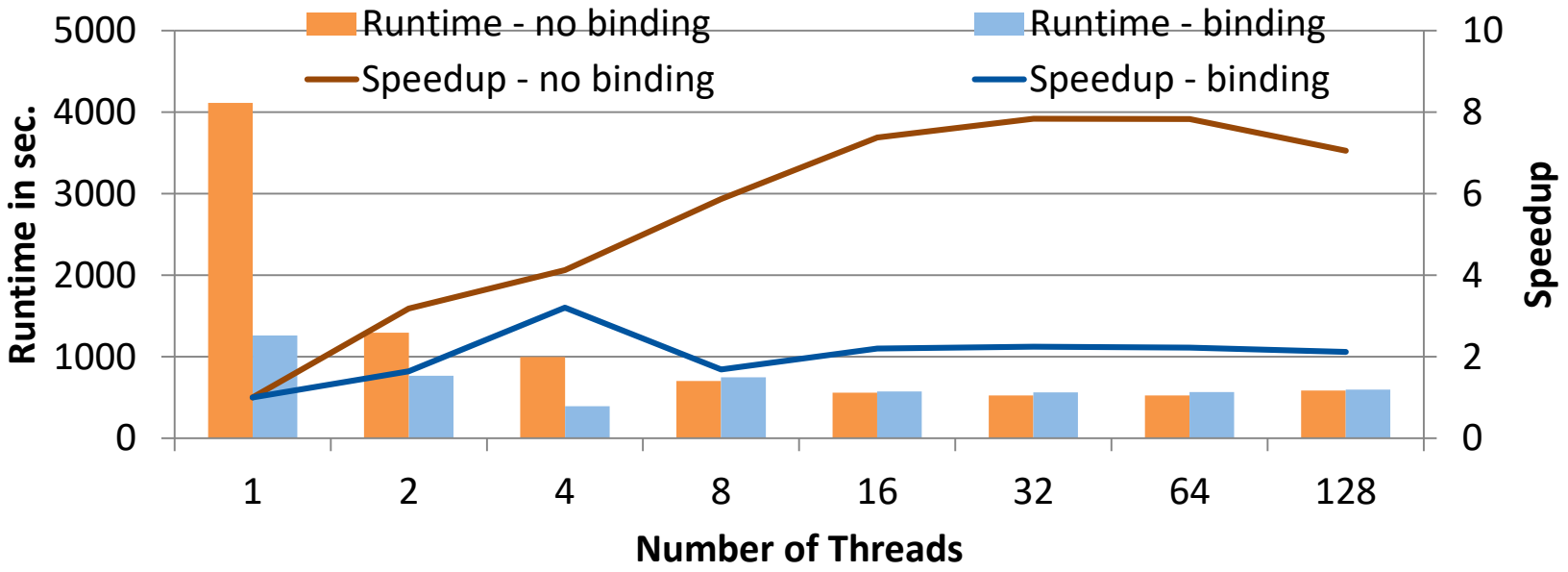




Case Study: CG

Implementation:

- parallelize all hotspots with a parallel for construct
- use a reduction for the dot-product
- activate thread binding



Data Placement

Data Placement

- 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

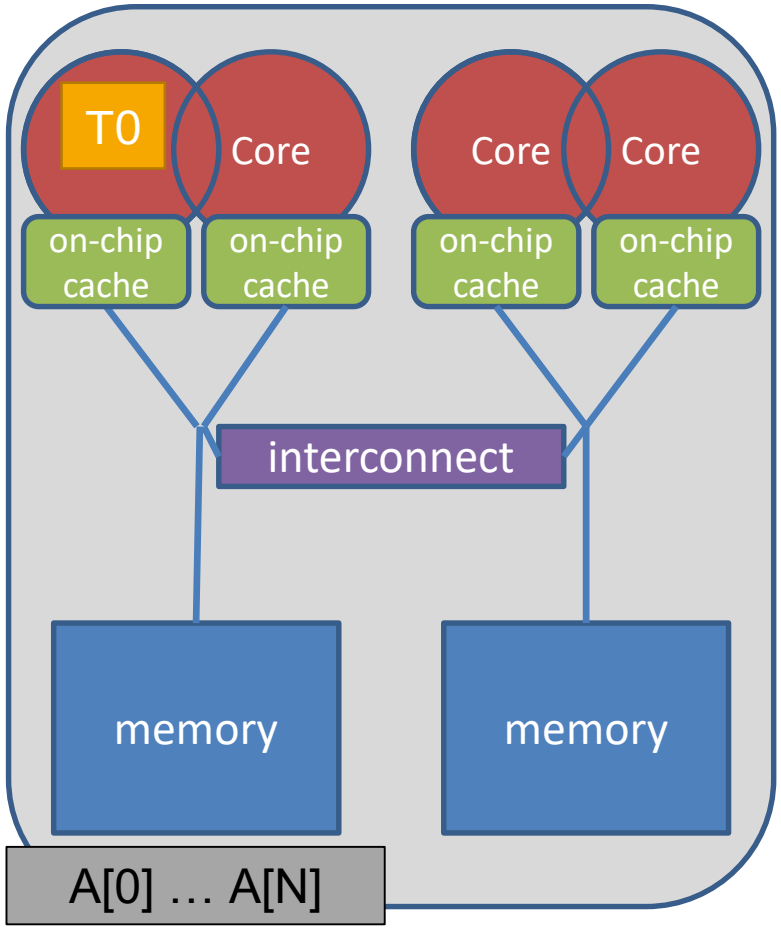
- Windows, Linux and Solaris all use the “First Touch” placement policy by default

First-touch in action

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;
}
```



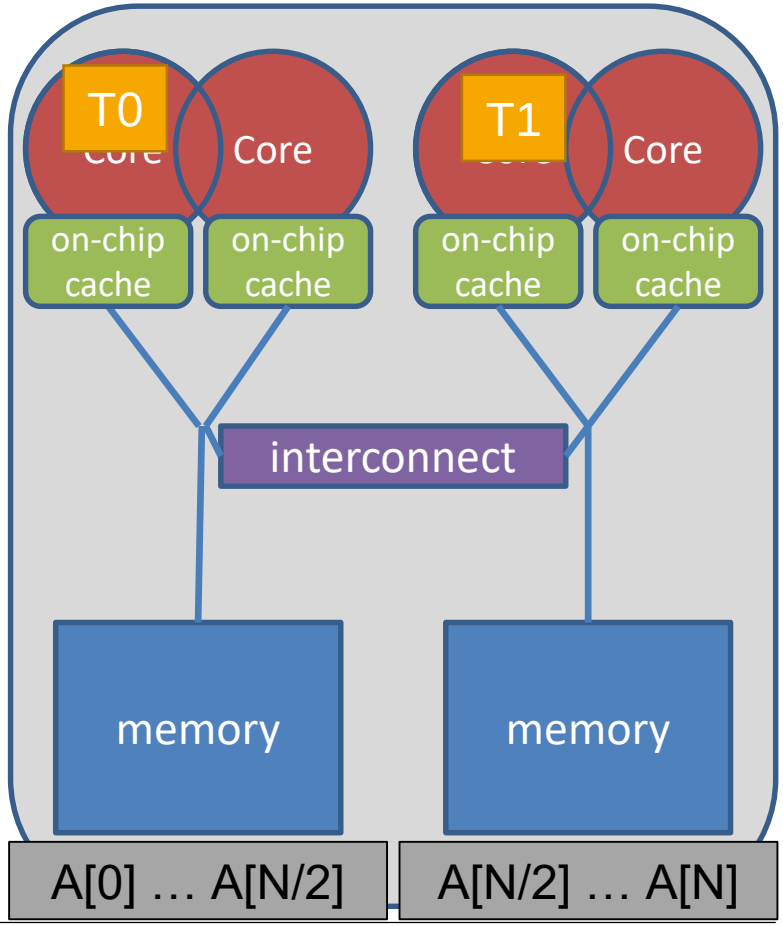
First-touch in action

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));


omp_set_num_threads(2);

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

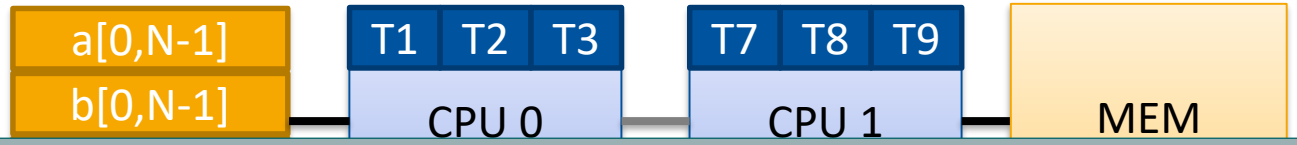


First-touch in action

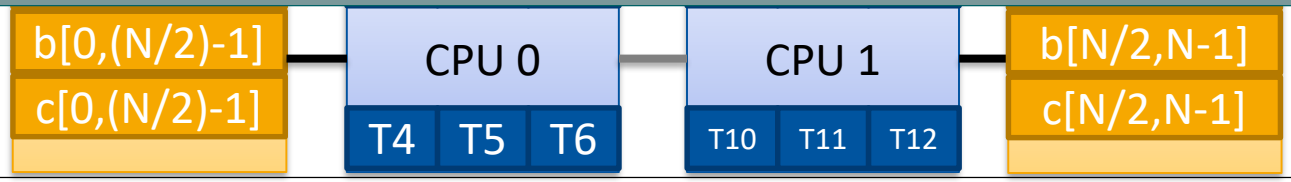
- Stream example ($\vec{a} = \vec{b} + s * \vec{c}$) with and without parallel initialization.
  - 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	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:



Note: Later Linux Kernels support „automatic NUMA balancing“. With this enabled, results look better for the ser_init case.



Memory- and Thread-placement in Linux

numactl - command line tool to investigate and handle NUMA under Linux

- `$ numactl --cpunodebind 0,1,2 ./a.out`
  - only use cores of NUMA node 0-2 to execute a.out
- `$ numactl --physcpubind 0-17 ./a.out`
  - only use cores 0-17 to execute a.out
- `$ numactl --membind 0,3 ./a.out`
  - only use memory of NUMA node 0 and 3 to execute a.out
- `$ numactl --interleave 0-3 ./a.out`
  - distribute memory pages on NUMA nodes 0-3 in a round-robin fashion
  - overwrites first-touch policy

Memory- and Thread-placement in Linux

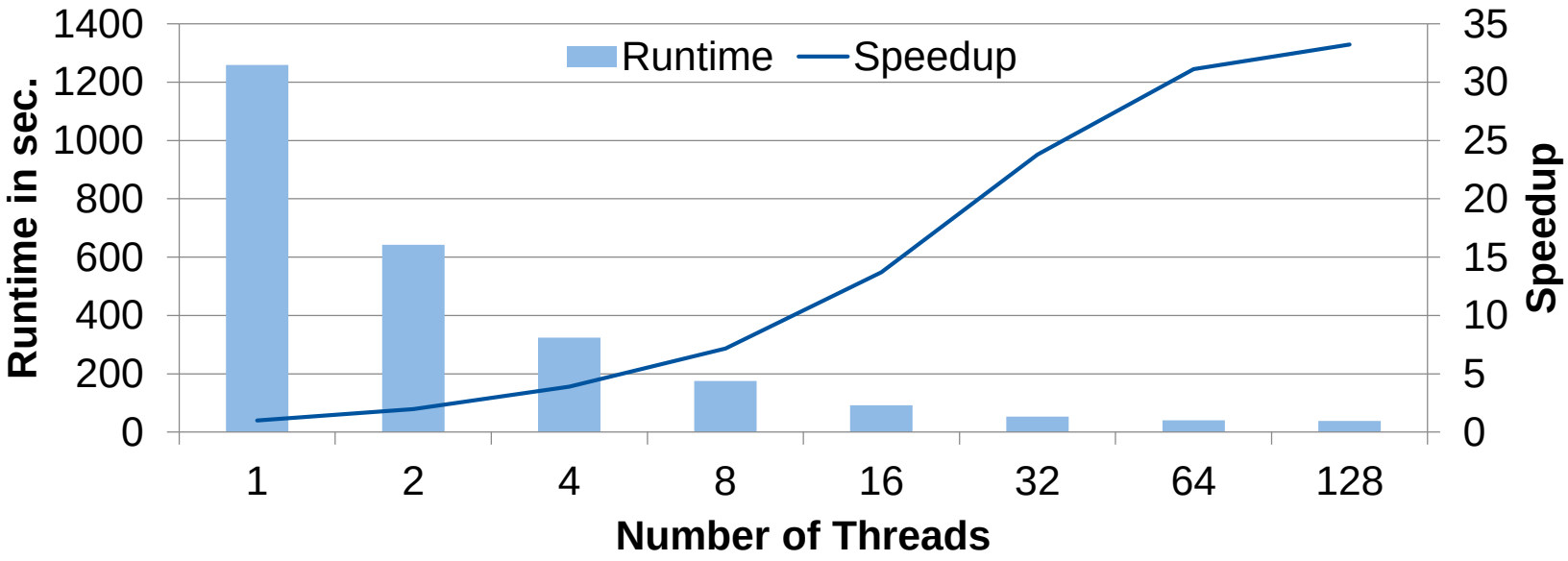
libnuma - library for NUMA control
(include numa.h and link -lnuma)

- `void *numa_alloc_local(size_t size);`
  - allocate memory on the local NUMA node
- `void *numa_alloc_onnode(size_t size, int node);`
  - allocate memory on NUMA node node
- `void *numa_alloc_interleaved(size_t size);`
  - allocate memory distributed round-robin on all NUMA nodes
- `int numa_move_pages(int pid, unsigned long count, void **pages, const int *nodes, int *status, int flags);`
  - migrate memory pages at runtime to different NUMA nodes

Case Study: CG

Tuning:

- Use first-touch initialization for data placement
- Parallelize all initialization loops
- Always use a static schedule



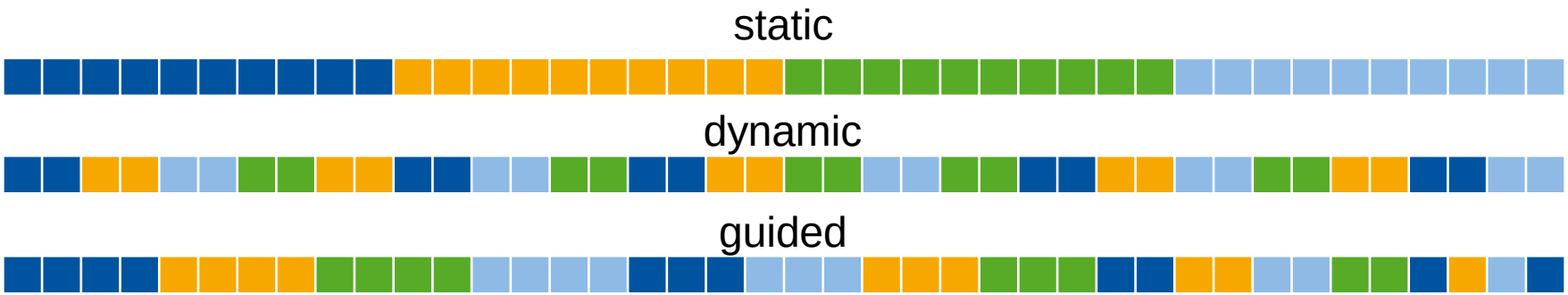
- Scalability improved a lot by this tuning on the large machine.

Work Distribution

Work Distribution

- For loop worksharing constructs the assignment of iterations to threads depends on the schedule used.

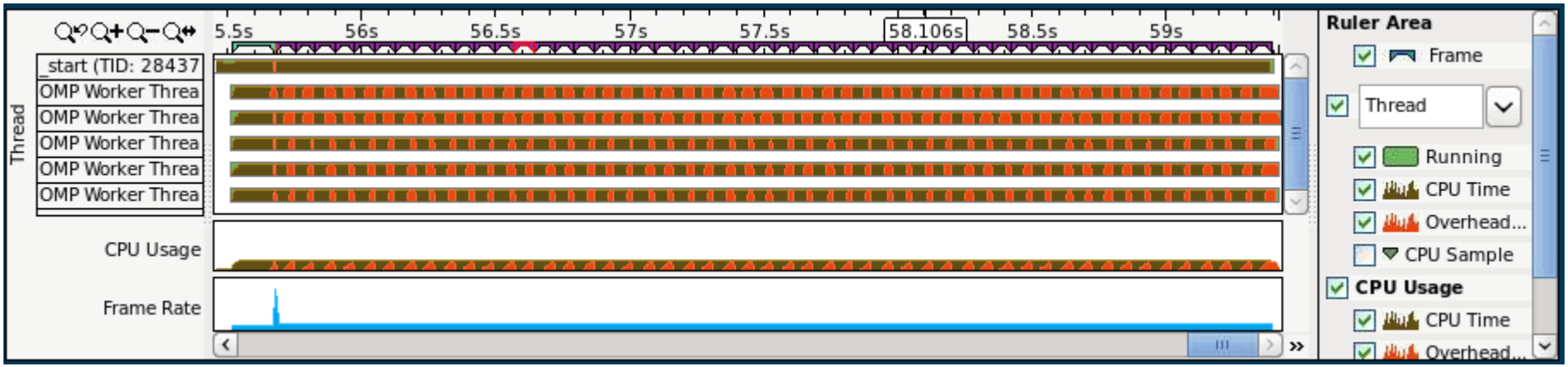
```
#pragma omp parallel for schedule(...)
for (i=0 ; i < 40 ; i++){
        A[i]=42;
}
```



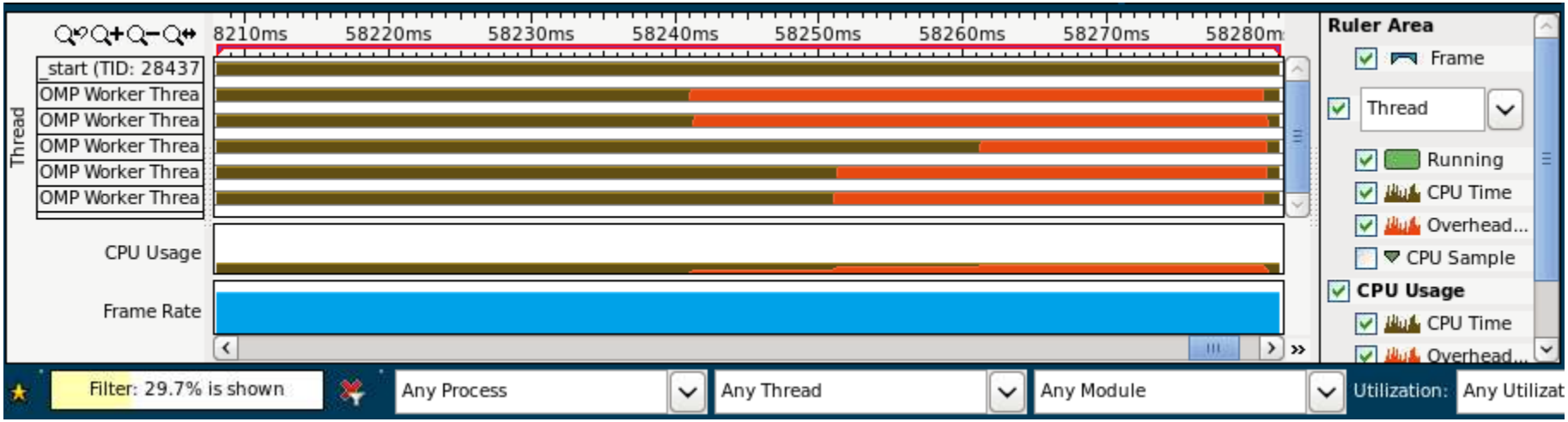
- For tasking no fixed mapping is provided.

Case Study:CG

- Different iterations of the CG Solver

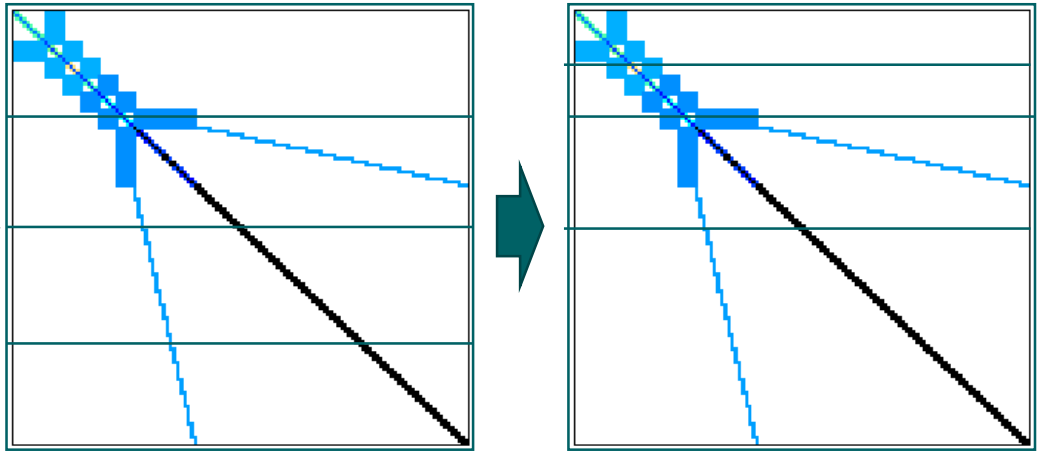


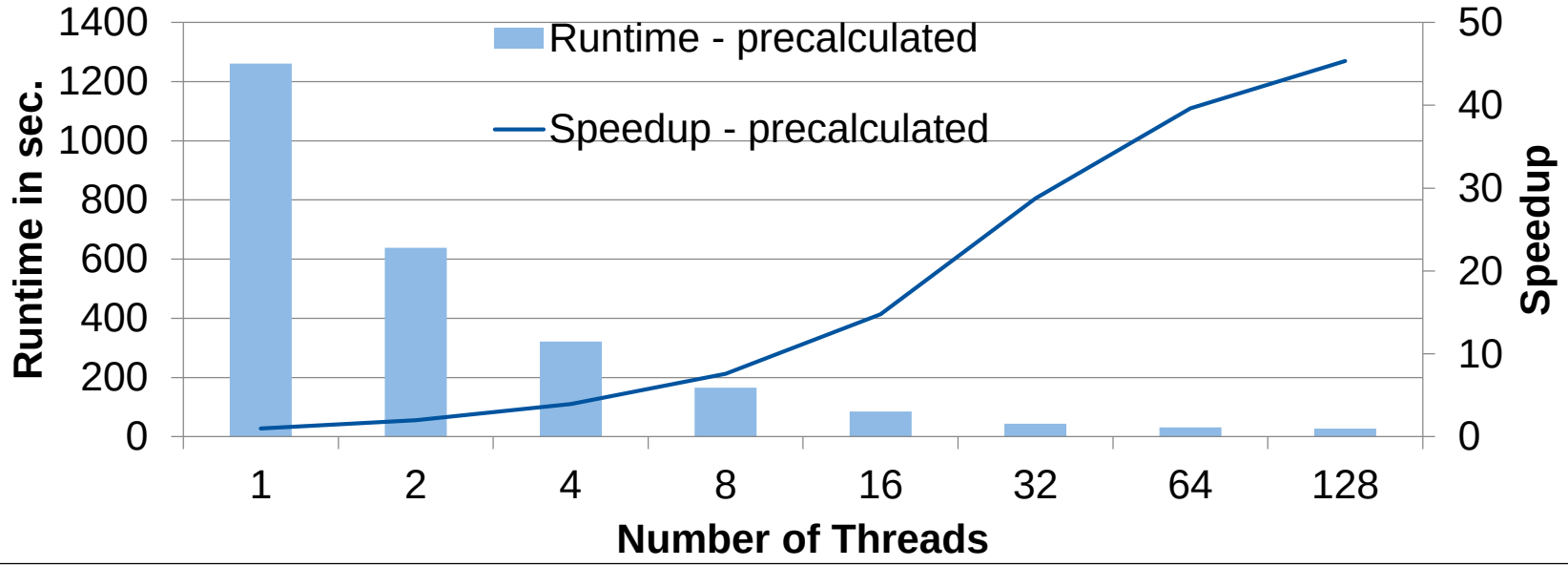
- Zoomed in on one iteration



Case Study: CG

- Tuning:
  - pre-calculate a schedule for the matrix-vector multiplication, so that the non-zeros are distributed evenly instead of the rows

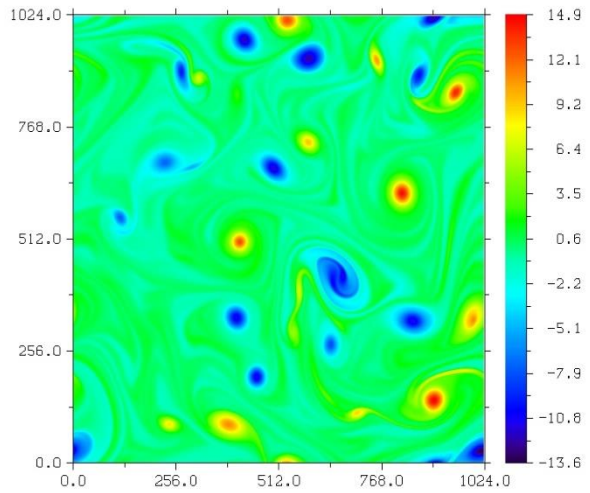


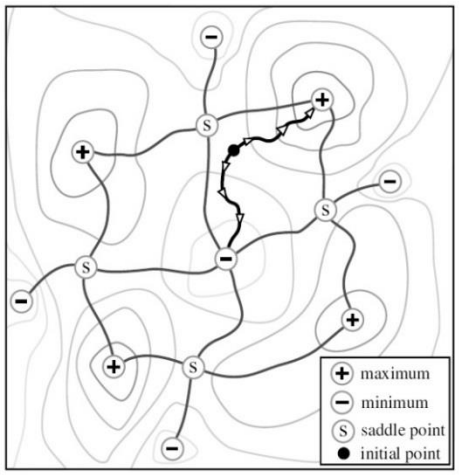


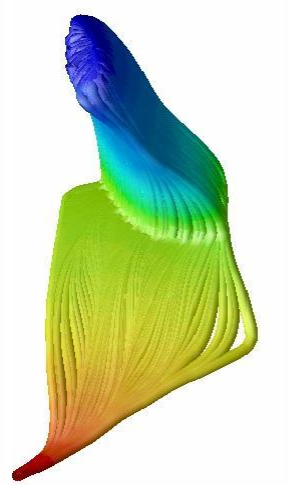
Application Case Study

TrajSearch

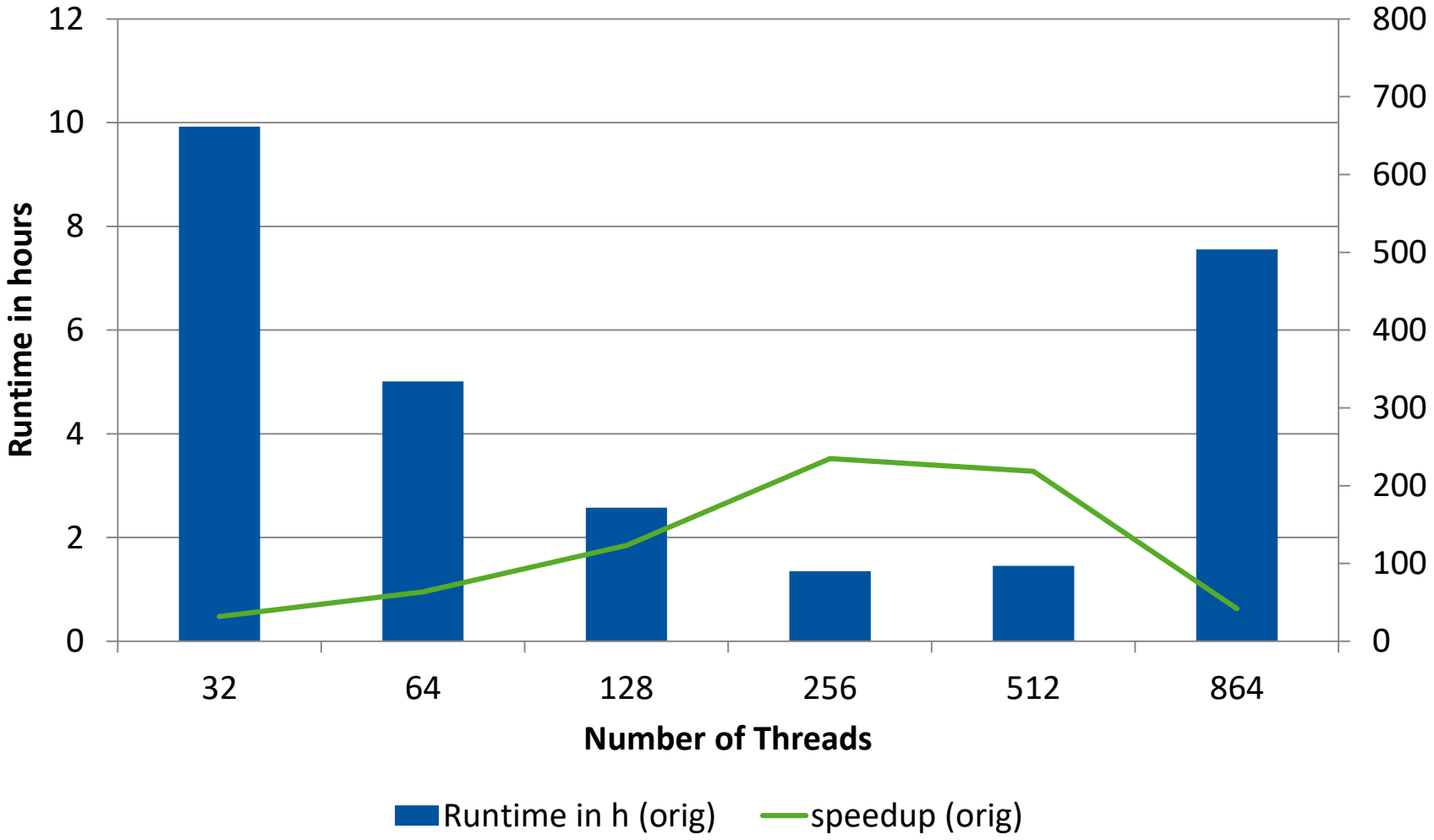
- Post-processing code for dissipation element analysis
- Follows Trajectories starting at each gridcell in direction of ascending and descending gradient of an passive 3D scalar field
- Trajectories lead to a local maximum or minimum respectively
- The composition of all gridcells of which trajectories end in the same pair of extremal points defines a dissipation element
- Developed at the Institute for Combustion Technology at RWTH Aachen





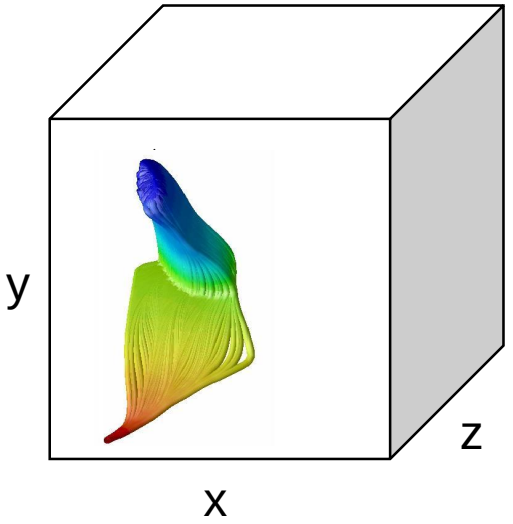


Performance Results on Numascale system



TrajSearch

- “Computer Science View on the Code”
  - Input is a large 3D Array
  - Independent search process through the array with read only access
  - Search processes differ in length
  - Memory access is unknown, since it depends on the direction
  - Writing reached minima and maxima to a list
  - Writing points crossed during the search in a second large array



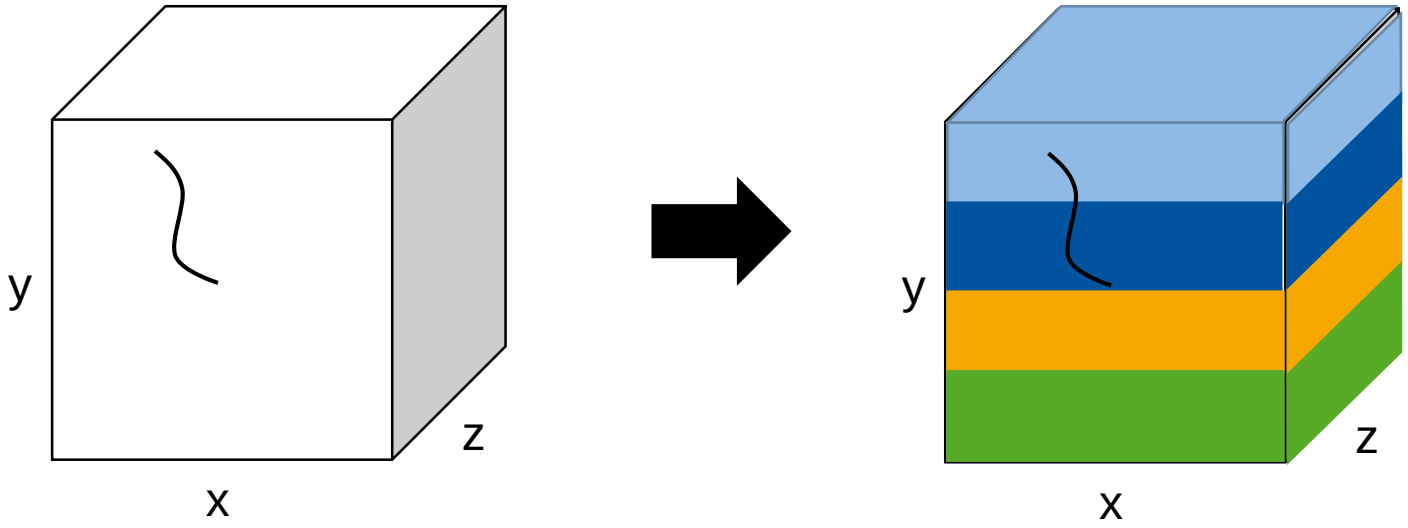
Optimization Steps (1/3)

Reduce Synchronization:

- Local Buffers per Thread for the result data
- Using multi-threading optimized memory allocation, like kmp_malloc
- Replaced the Fortran random number generator with a simple RNG generating independent streams per thread

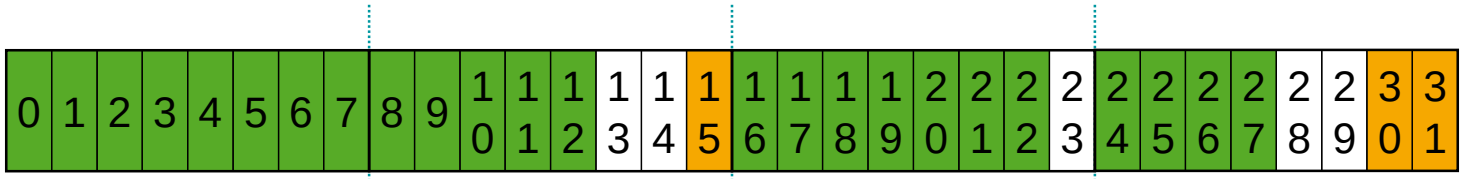
Optimization Steps (2/3)

- Data placement
  - Starting point of trajectories are well known
  - Trajectories starting in neighbor grid cells will often need near data
  - Compact thread pinning is needed to avoid thread migration
  - Remote accesses cannot be avoided completely
  - NUMA Caches might help to reuse data



Optimization Steps (3/3)

- Load imbalance
  - each trajectory has a different length (-> computational load imbalance)
  - the data placement is fix (-> dynamic scheduling is not sufficient)
- Numa-aware scheduling
  - start with a static load balance
  - instead of idling “help” other threads when work is done
  - to reduce interference work of foreign nodes will get iterations from the highest index backwards



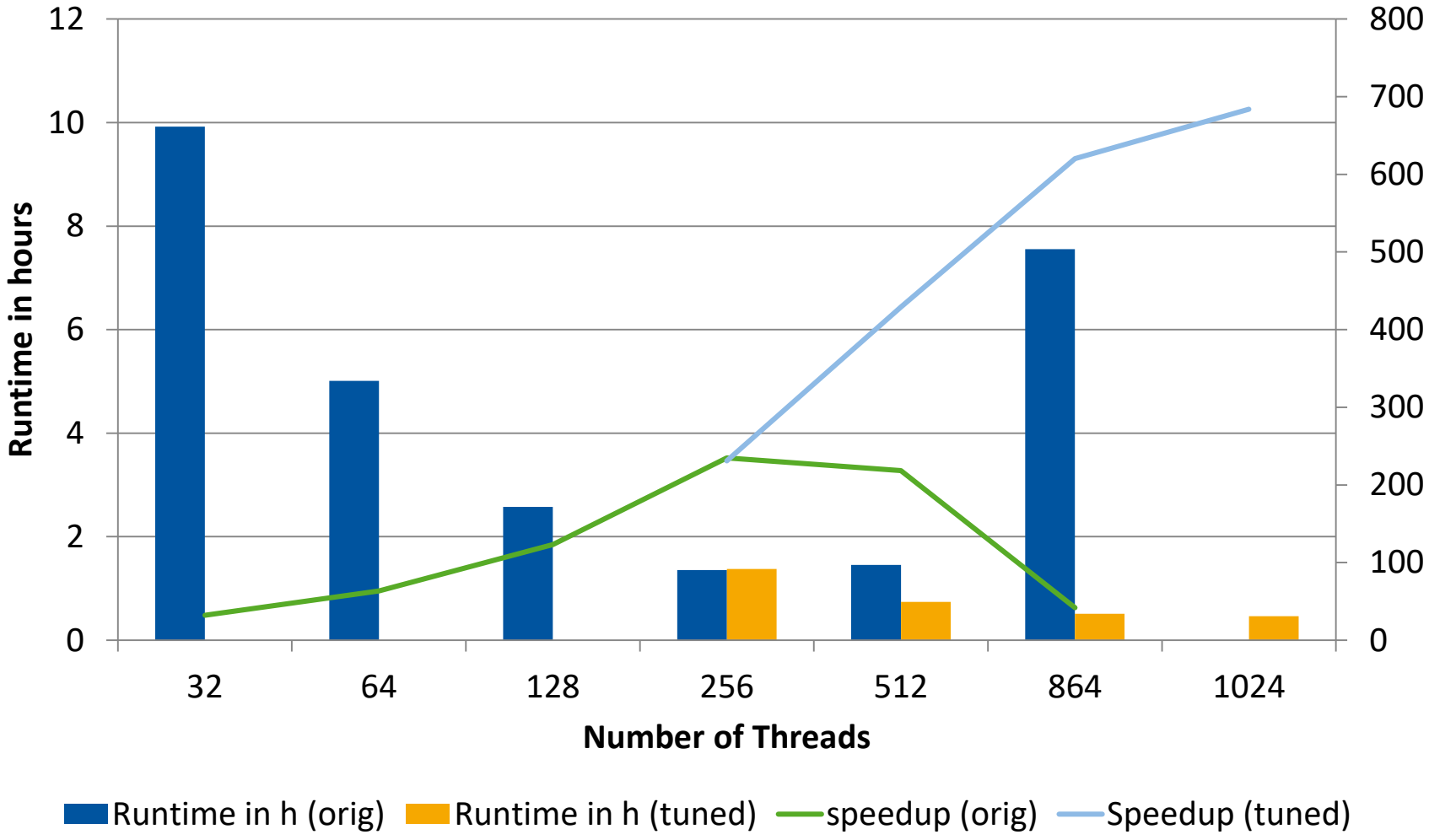
node 0	
core1	core2
0, 4, 6	1, 5, 31
core3	core4
2, 30	3, 7, 15

node 1	
core1	core2
8	9, 12
core3	core4
10	11

node 2	
core1	core2
17	18,20
core3	core4
16,22	19, 21

node 3	
core1	core2
26	24
core3	core4
27	25

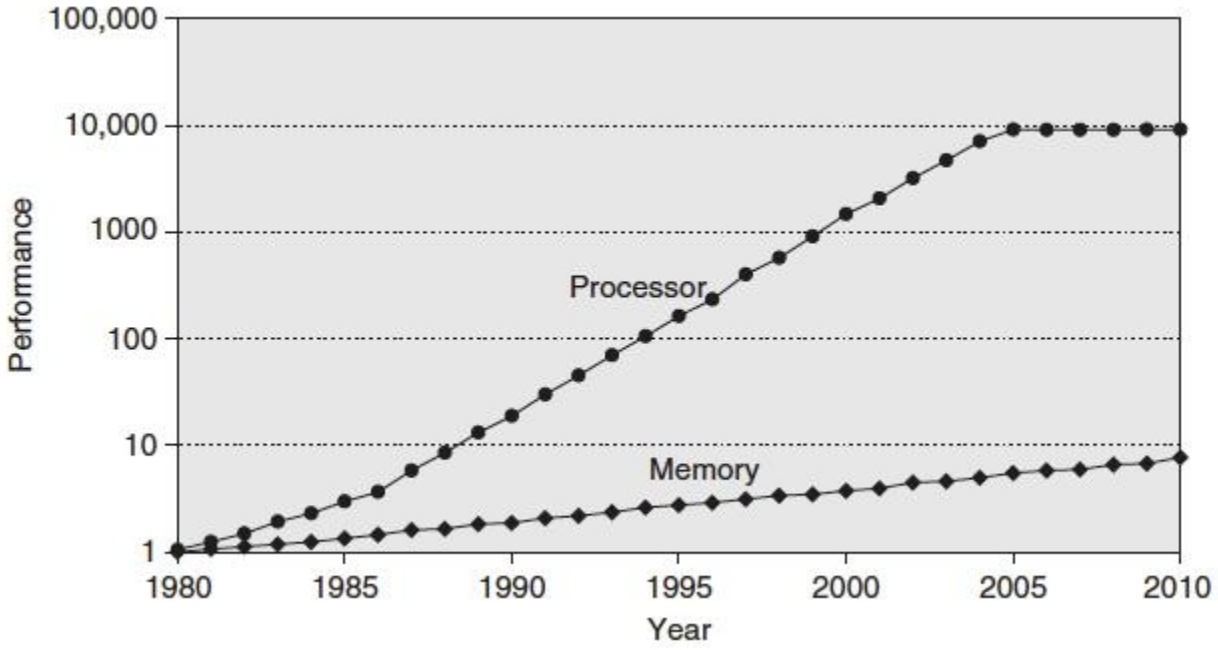
TrajSearch on Numascale system



False-sharing

Memory Bottleneck

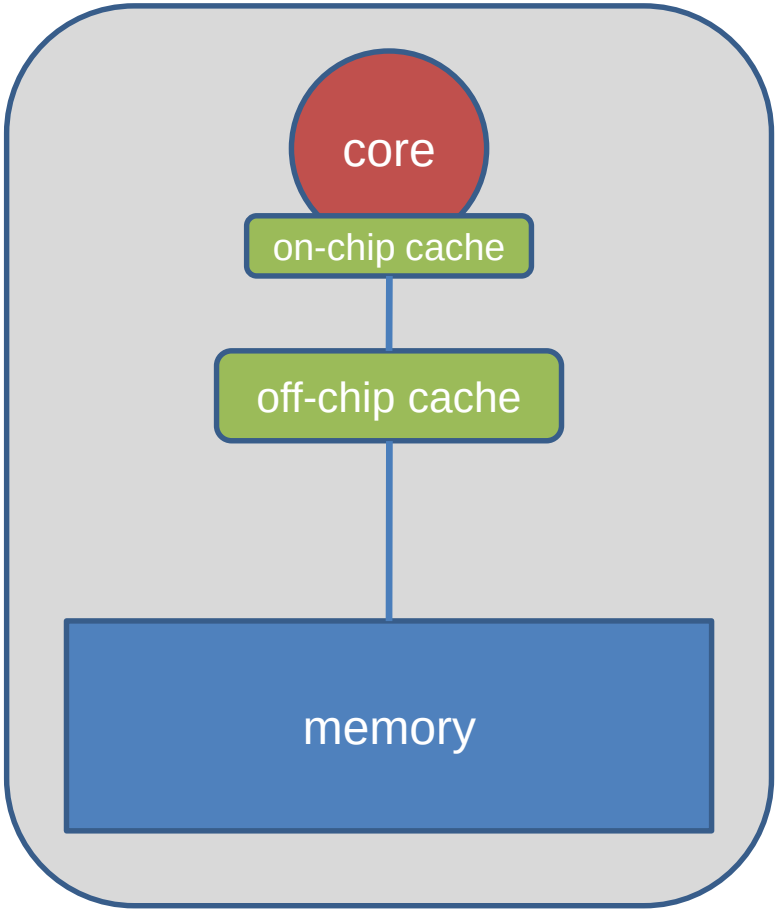
- There is a growing gap between core and memory performance:
  - memory, since 1980: 1.07x per year improvement in latency
  - single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a *per-core* basis



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

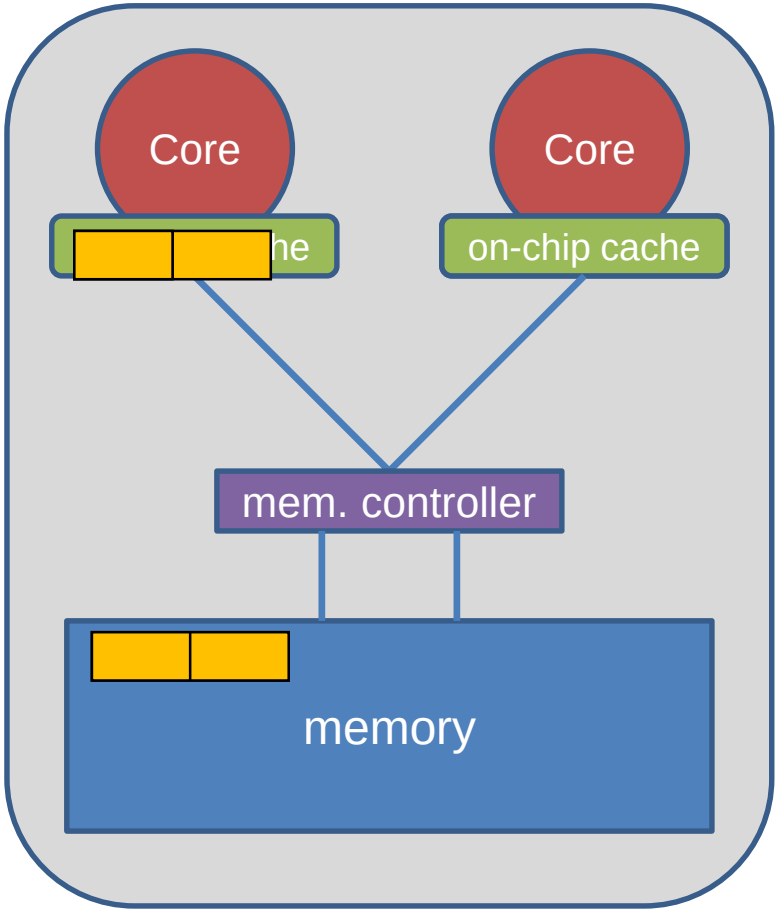
Caches

- 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!



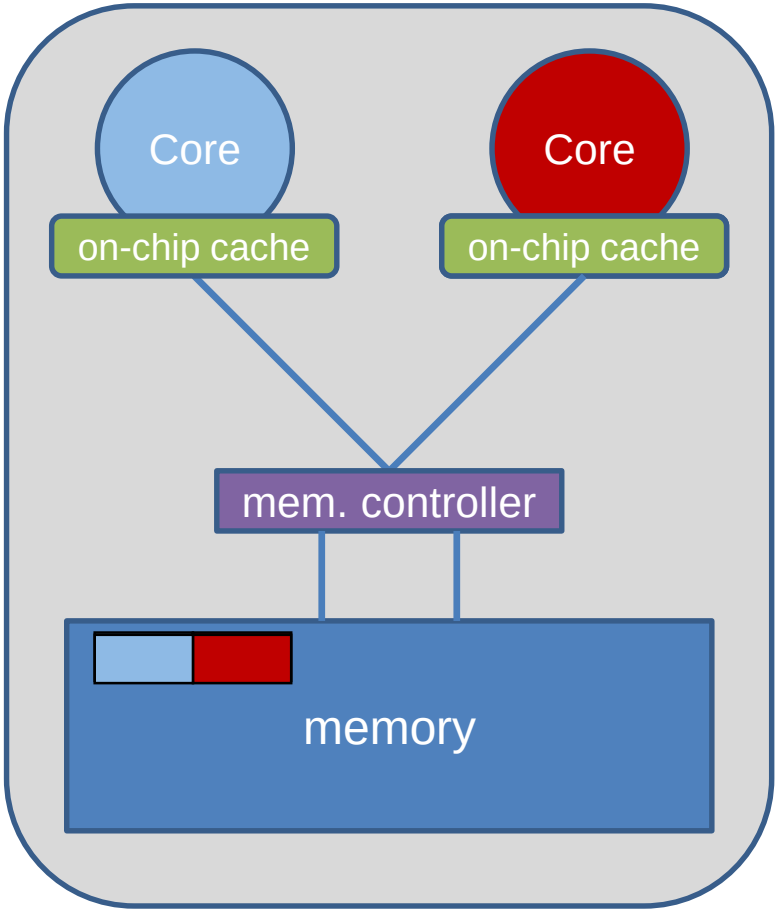
Caches

- 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

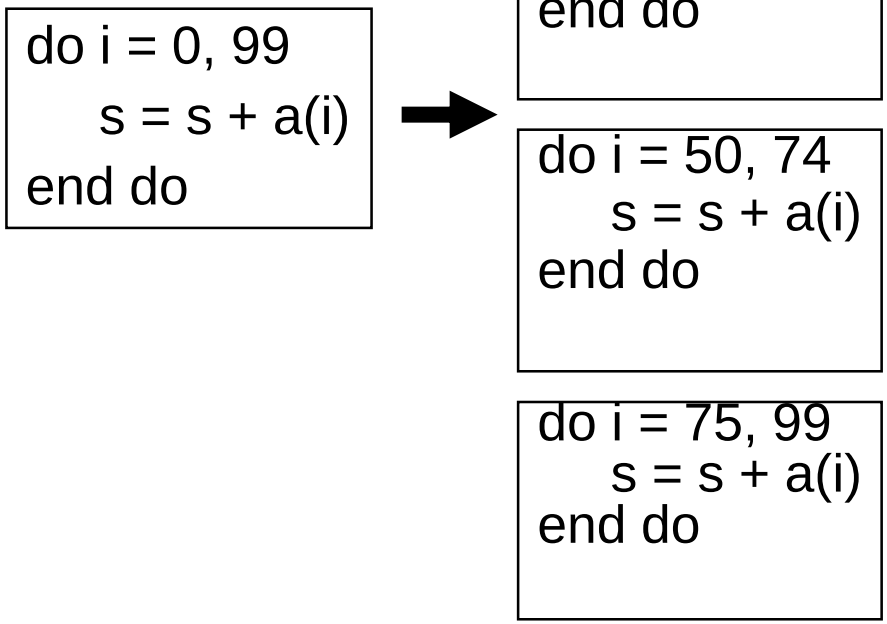
- False Sharing occurs when
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line
- As a result the cache line is moved between the threads, also there is no real dependency



- Note: False Sharing is a performance problem, not a correctness issue

Summing up vector elements

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < 99; i++)
  {
        s  = s   + a[i];
  }
} // end parallel
```

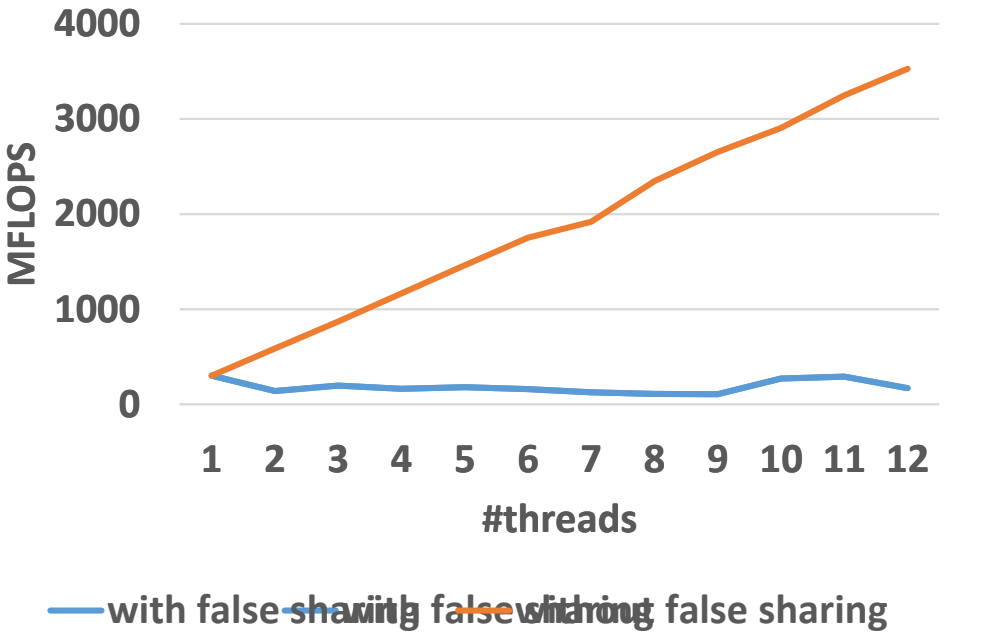


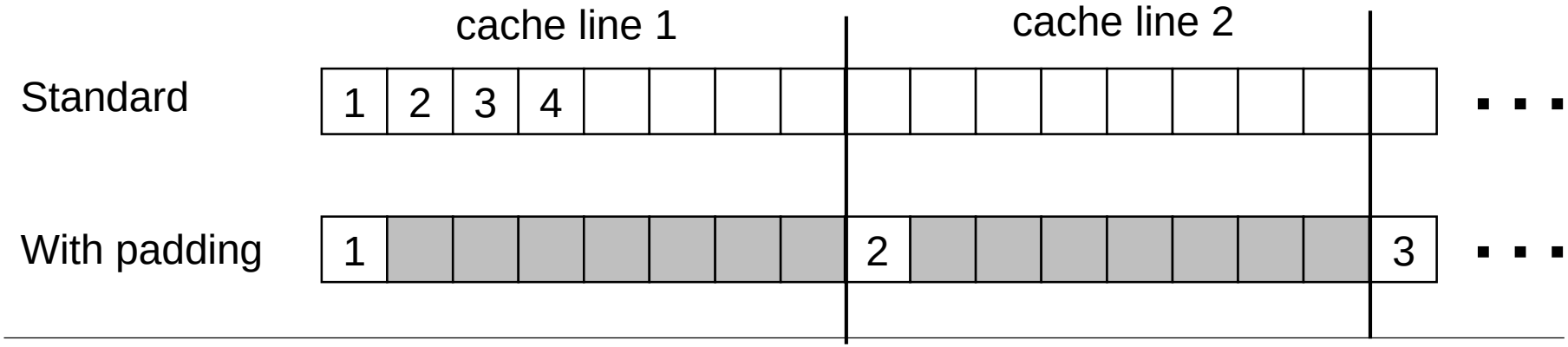
Summing up vector elements

```
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];
}
```

Summing up vector elements

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





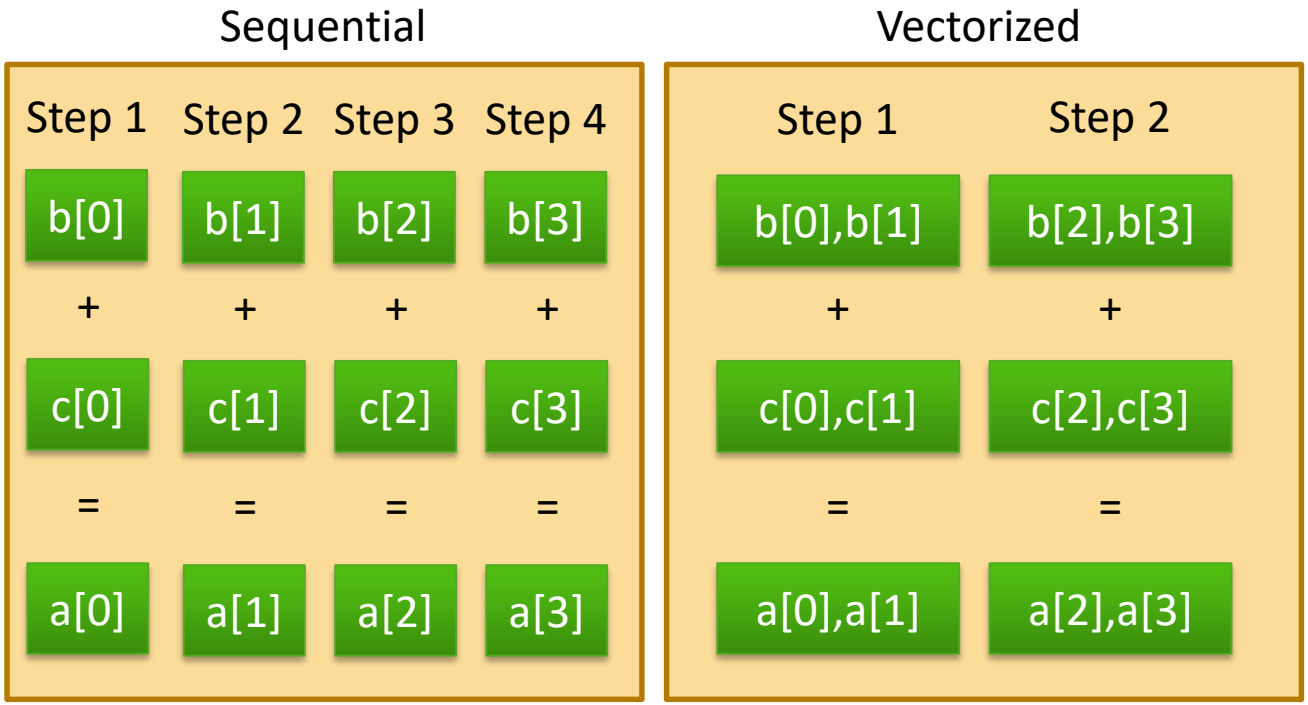
Vectorization with OpenMP

Vectorization

- SIMD = Single Instruction Multiple Data
  - Special hardware instructions to operate on multiple data points at once
  - Instructions work on vector registers
  - Vector length is hardware dependent

```
double a[4],b[4],c[4];
…

for(i=0 ; i < 4 ; i++)
{
      a[i]=b[i]+c[i];
}
```



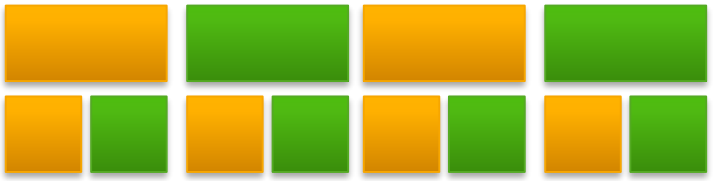
Vectorization

- Vector lengths on Intel architectures
  - 128 bit: SSE = Streaming SIMD Extensions

2 x double

4 x float

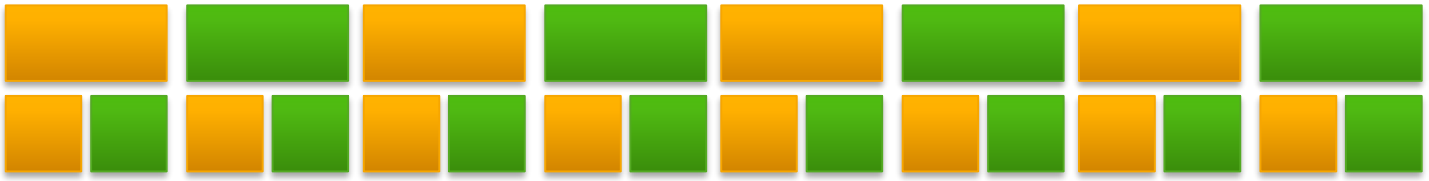
  - 256 bit: AVX = Advanced Vector Extensions



4 x double

8 x float

  - 512 bit: AVX-512



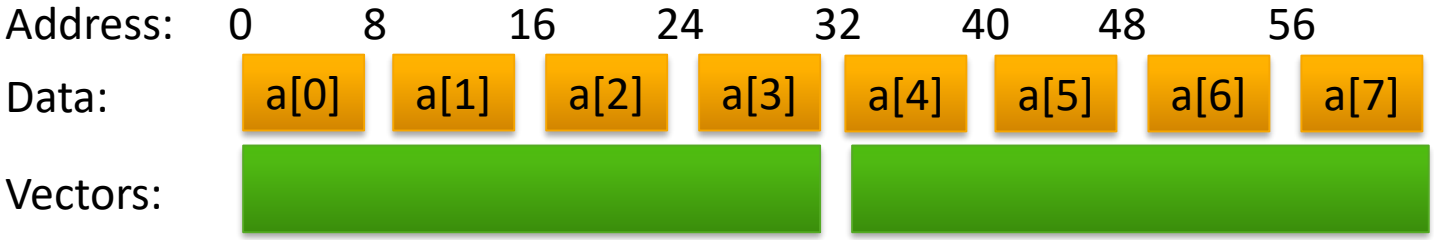
8 x double

16 x float

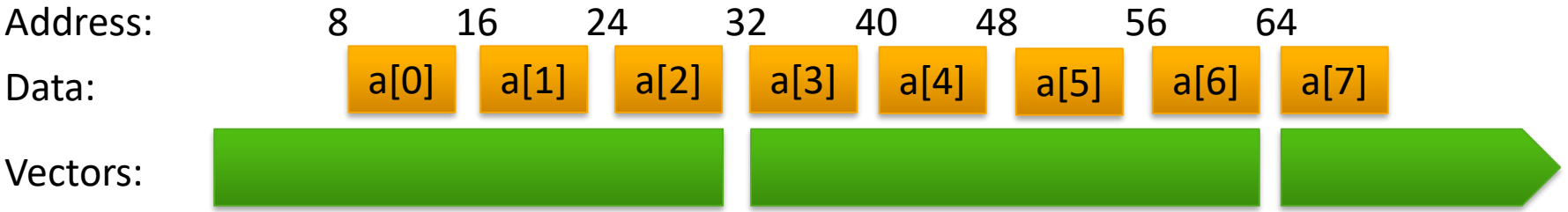
Data Alignment

- Vectorization works best on aligned data structures.

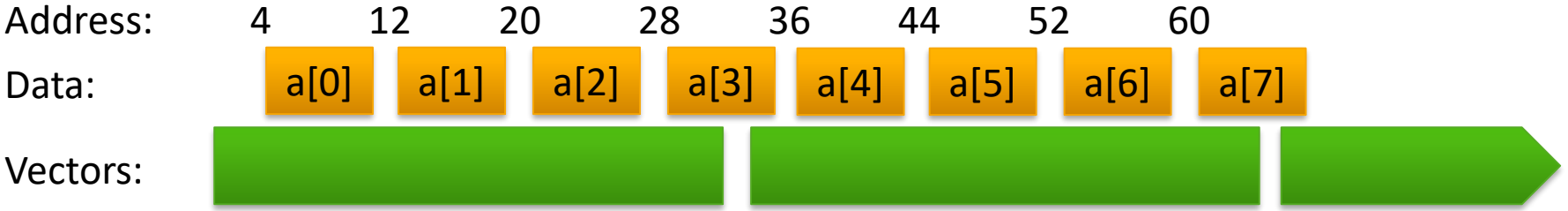
Good alignment



Bad alignment



Very bad alignment



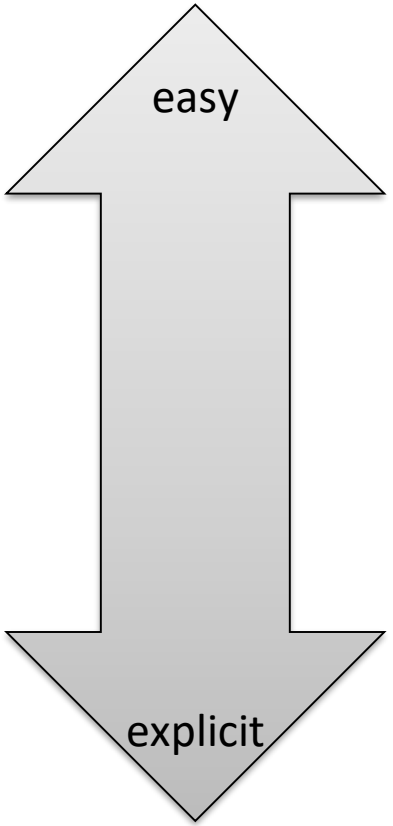
Ways to Vectorize

Compiler
auto-vectorization

Explicit Vector Programming
(e.g. with OpenMP)

Inline Assembly
(e.g.)

Assembler Code
(e.g. addps, mulpd, …)



The OpenMP SIMD constructs

The SIMD construct

- The SIMD construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.

C/C++:
```
#pragma omp simd [clause(s)]
  for-loops
```

Fortran:
```
!$omp simd [clause(s)]
  do-loops
[!$omp end simd]
```

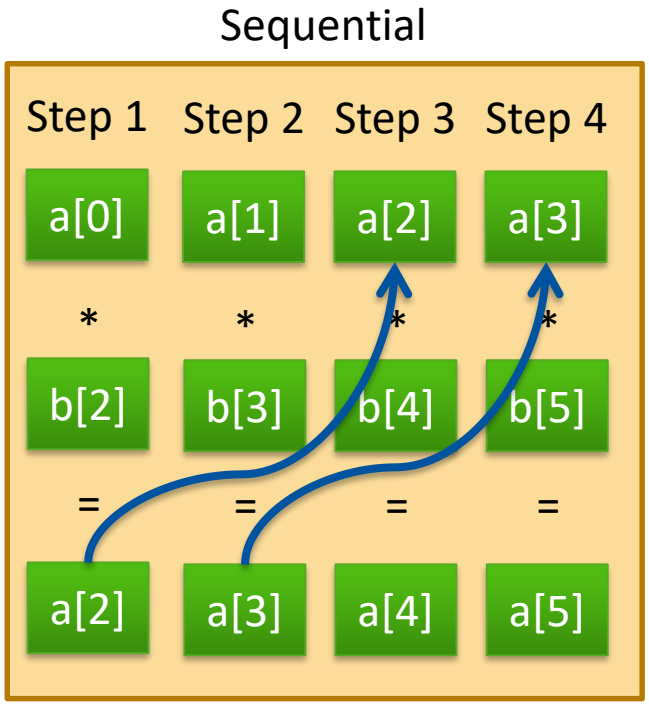
- where clauses are:
  - linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - aligned(*list[:alignment]*), specifies that data is aligned
  - private(*list*), as usual
  - lastprivate(*list*) , as usual
  - reduction(*reduction-identifier:list*) , as usual
  - collapse(*n*), collapse loops first, and than apply SIMD instructions

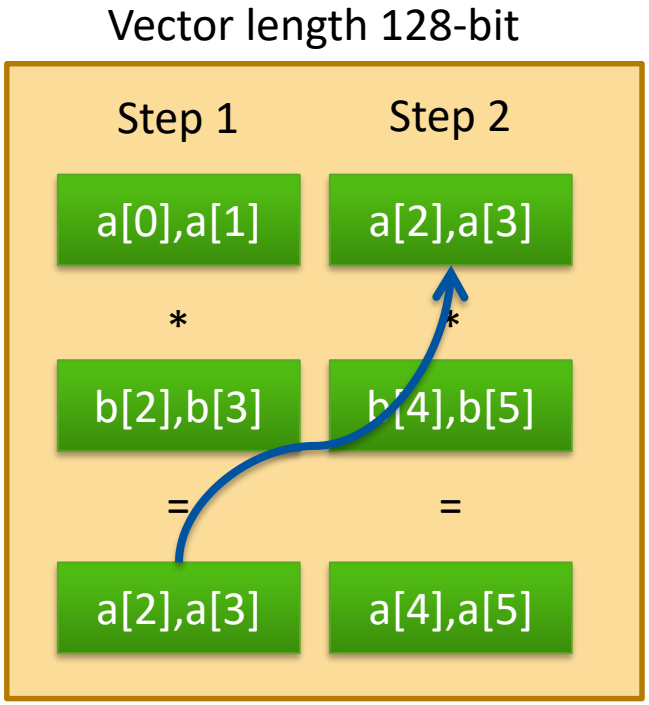
The SIMD construct

- The *safelen* clause allows to specify a distance of loop iterations where no dependencies occur.

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```



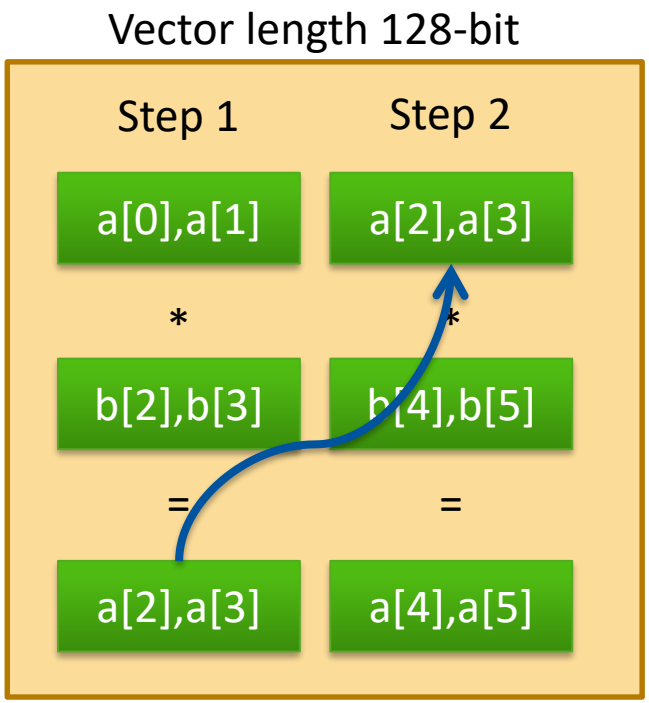


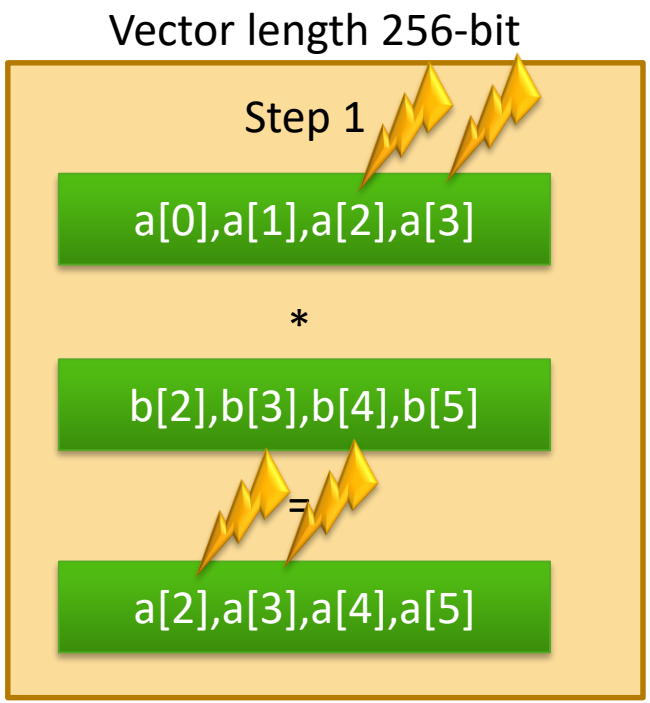
The SIMD construct

- The *safelen* clause allows to specify a distance of loop iterations where no dependencies occur.

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```





- Any vector length smaller than or equal to the length specified by safelen can be chosen for vectorization.
- In contrast to parallel for/do loops the iterations are executed in a specified order.

The loop SIMD construct

- The loop SIMD construct specifies a loop that can be executed in parallel by all threads and in SIMD fashion on each thread.

C/C++:

```
#pragma omp for simd [clause(s)]
  for-loops
```

Fortran:

```
!$omp do simd [clause(s)]
  do-loops
[!$omp end do simd [nowait]]
```

- Loop iterations are first distributed across threads, then each chunk is handled as a SIMD loop.
- Clauses:
  - All clauses from the *loop*- or SIMD-construct are allowed
  - Clauses which are allowed for both constructs are applied twice, once for the threads and once for the SIMDization.

The declare SIMD construct

- Function calls in SIMD-loops can lead to bottlenecks, because functions need to be executed serially.

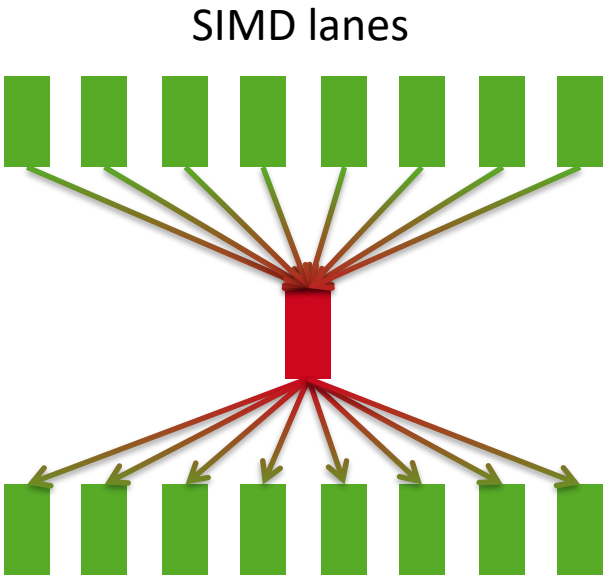
```
for(i=0 ; i < N ; i++)
{

      a[i]=b[i]+c[i];

      d[i]=sin(a[i]);

      e[i]=5*d[i];

}
```



Solutions:

- avoid or inline functions
- create functions which work on vectors instead of scalars

The declare SIMD construct

- Enables the creation of multiple versions of a function or subroutine where one or more versions can process multiple arguments using SIMD instructions.

C/C++:

```
#pragma omp declare simd [clause(s)]
[#pragma omp declare simd [clause(s)]]
  function definition / declaration
```

Fortran:

```
!$omp declare simd (proc_name)[clause(s)]
```

- where clauses are:
  - simdlen(*length*), the number of arguments to process simultanously
  - linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - aligned(*argument-list[:alignment]*), specifies that data is aligned
  - uniform(*argument-list*), arguments have an invariant value
  - inbranch / notinbranch, function is always/never called from within a conditional statement

Thank you for your attention!

Questions?