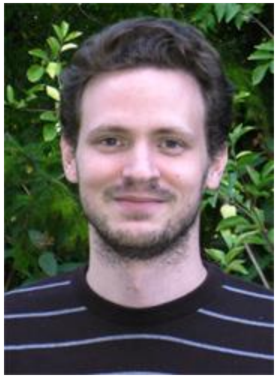
OpenMP on NUMA Architectures



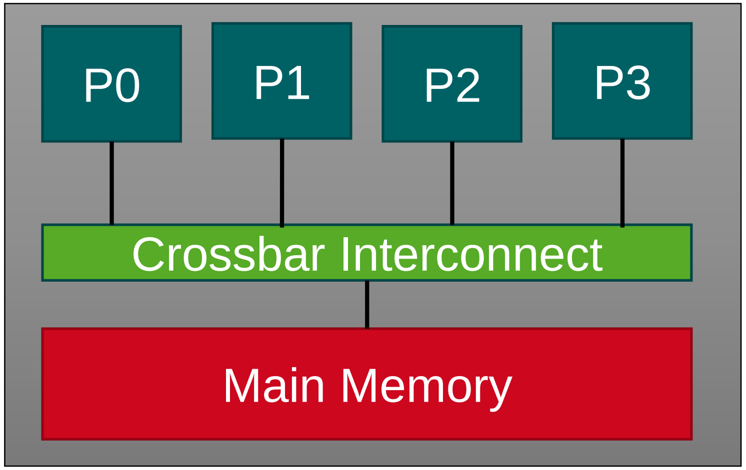
Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

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

IT Center | RWTH AACHEN UNIVERSITY

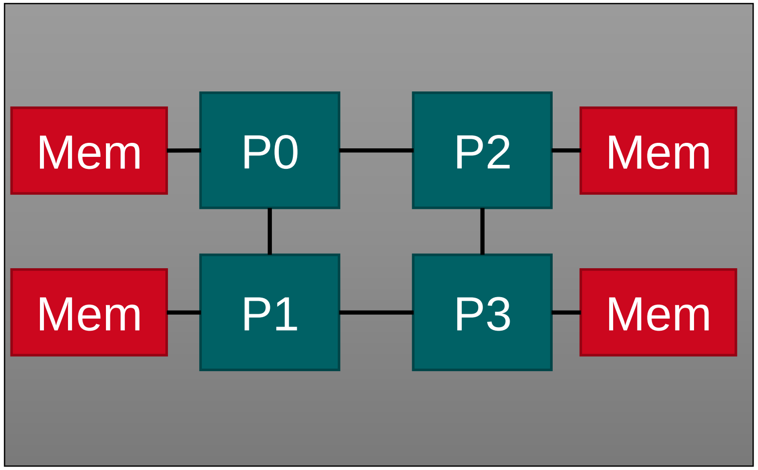
NUMA Architectures

- 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

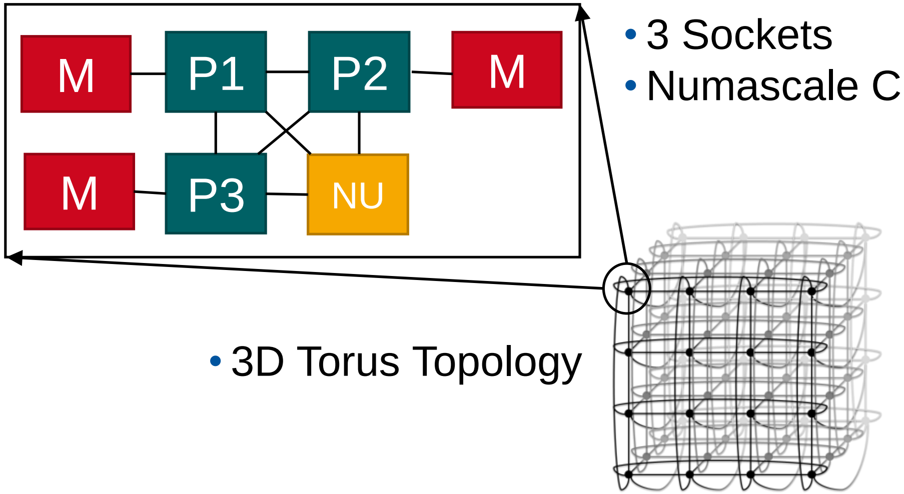
NUMA Architectures

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

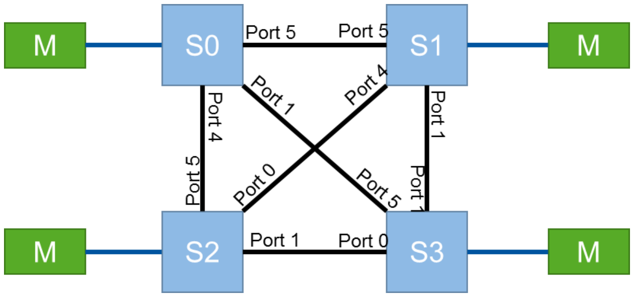


- 3 Sockets
- Numascale Chip

- 3D Torus Topology

NUMA Architectures

4 Socket Intel Xeon System:

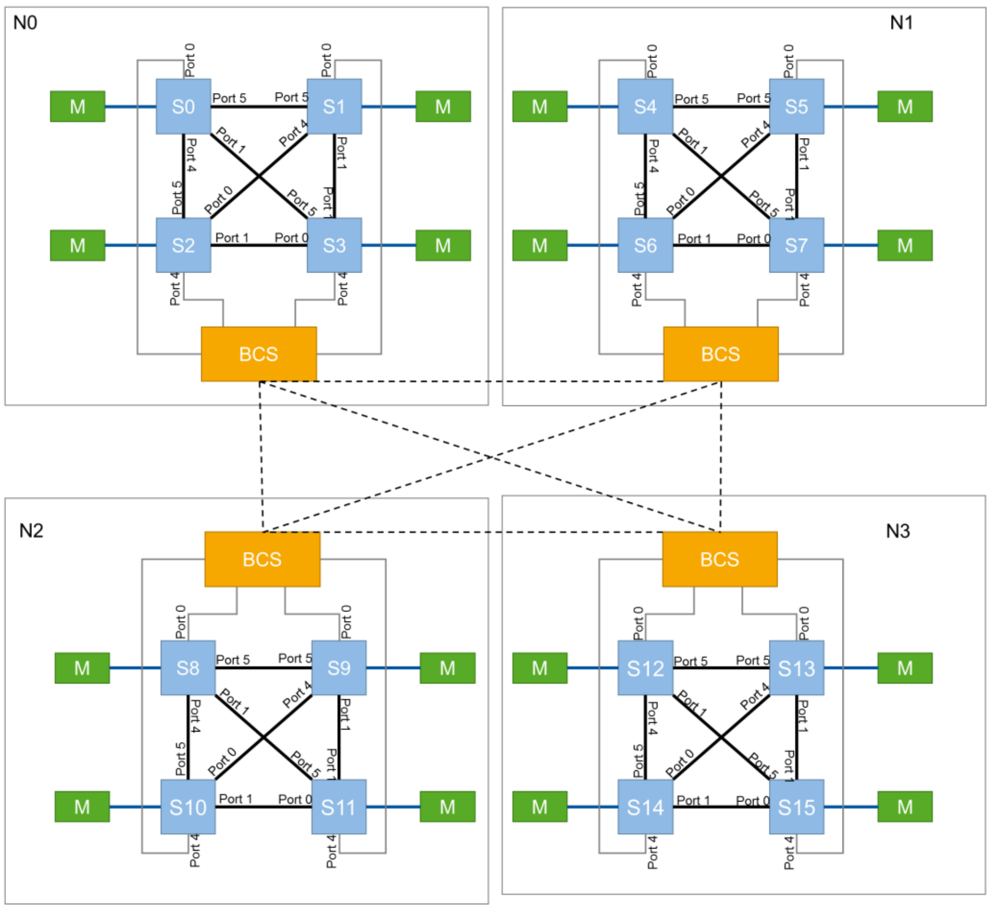


Hierarchical NUMA System:

- 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:

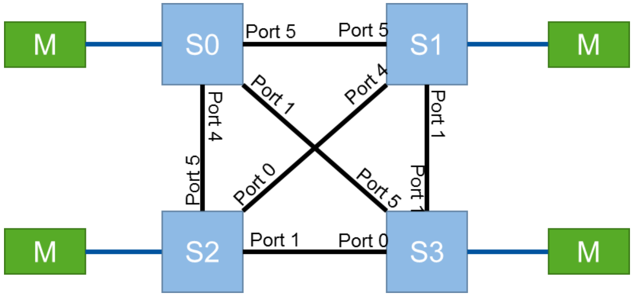


Investigating NUMA topologies

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

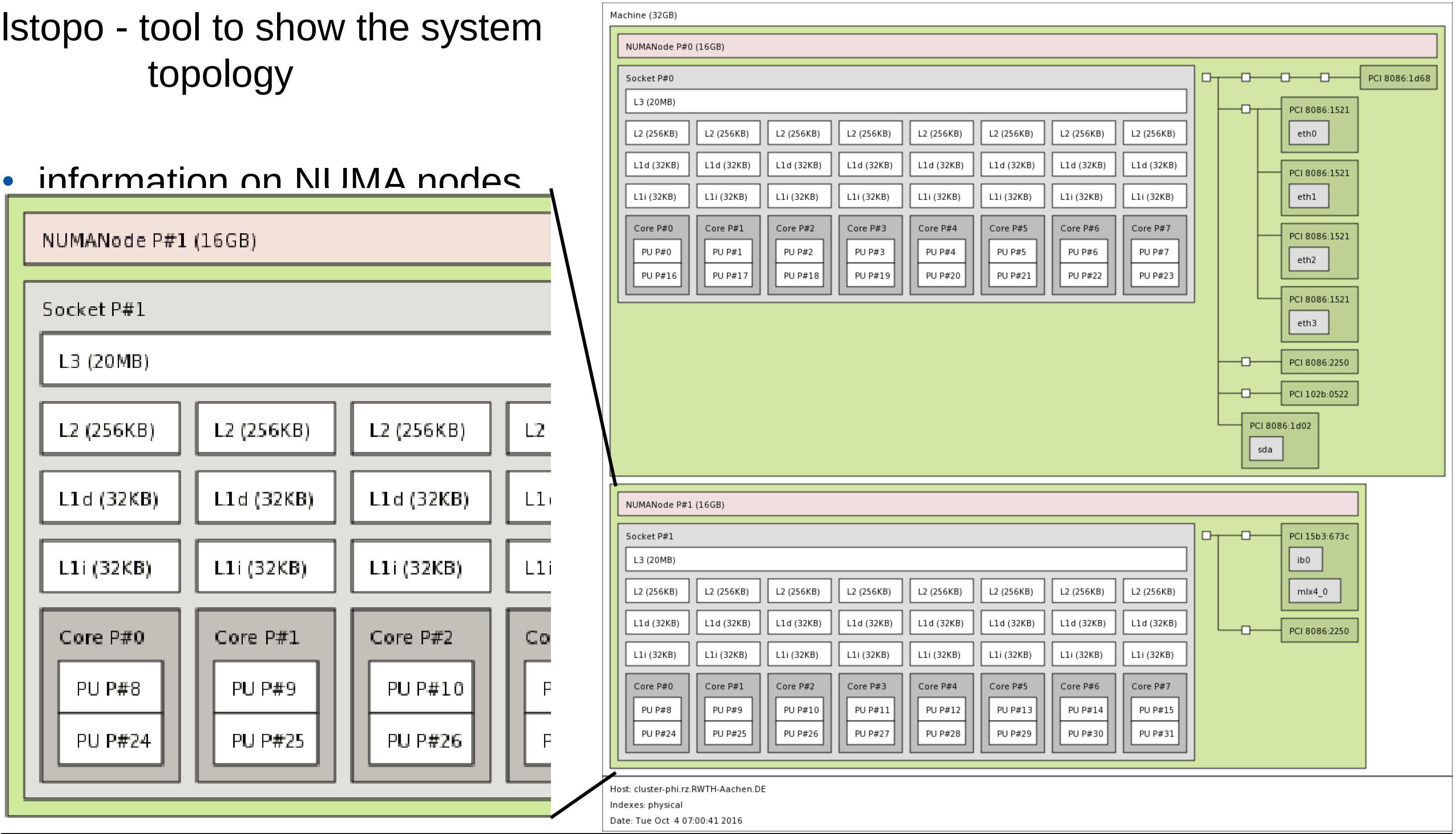
```
ds534486@linuxbsc001 [/home/ds534486] $ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
node 0 size: 65501 MB
node 0 free: 50533 MB
node 1 cpus: 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61
node 1 size: 65536 MB
node 1 free: 58763 MB
node 2 cpus: 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

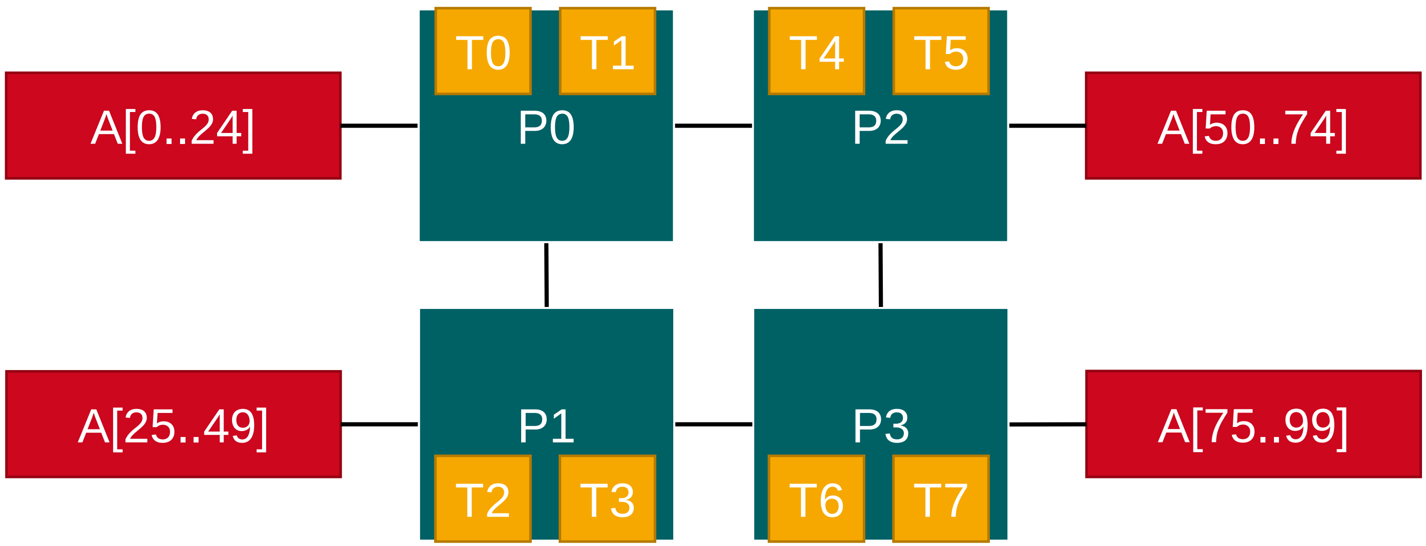
- information on NUMA nodes



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

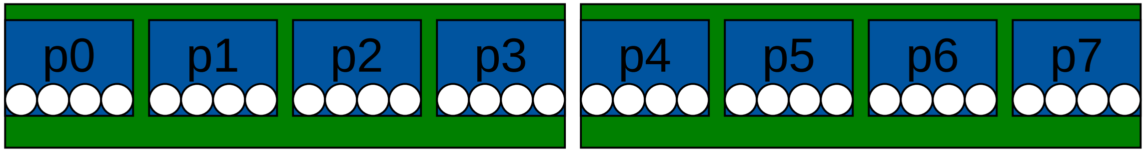
Thread Placement in OpenMP

- 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

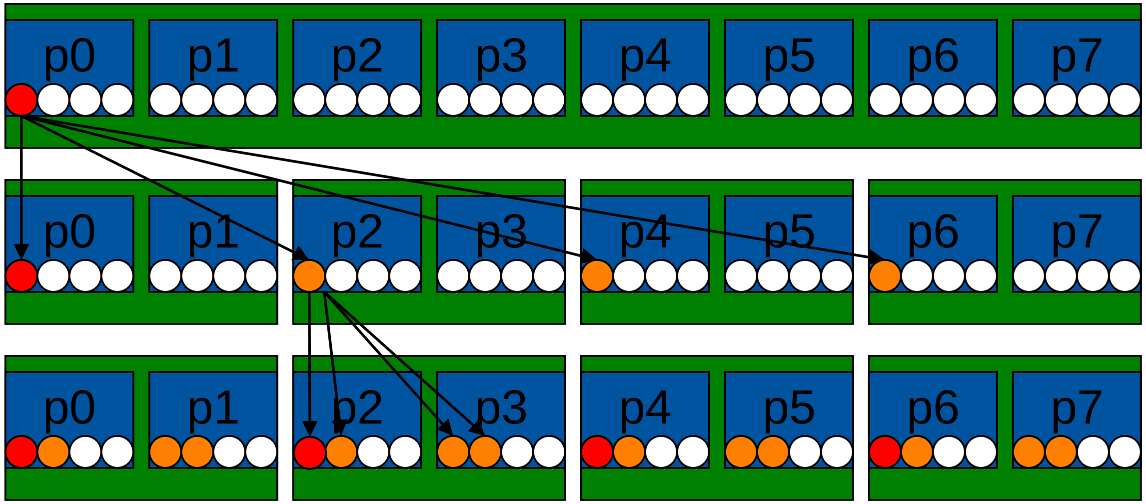
- Abstract names for OMP_PLACES:
  – threads: Each place corresponds to a single hardware thread on the target machine.
  – cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  – sockets: Each place corresponds to a single socket (consisting of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

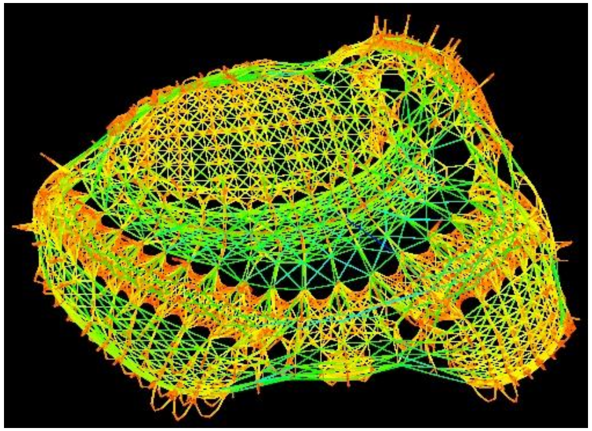


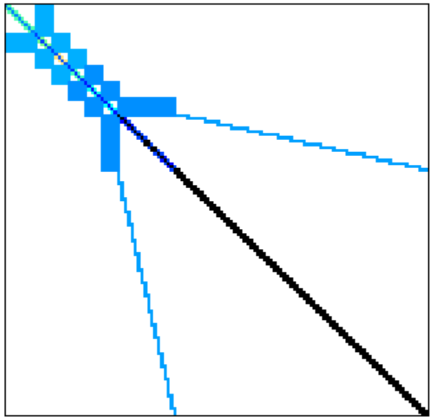
Thread Placement in OpenMP

- `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

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

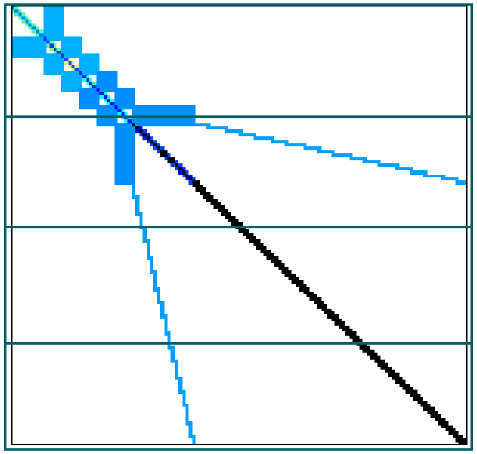
```
for (i = 0; i < A.num_rows; i++){
  sum = 0.0;
  for (nz=A.row[i]; nz<A.row[i+1]; ++nz){
    sum+= A.value[nz]*x[A.index[nz]];
  }
  y[i] = sum;
}
```

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

- 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)

value:	1	2	2	3	4	4	4
index:	0	0	1	2	0	2	3

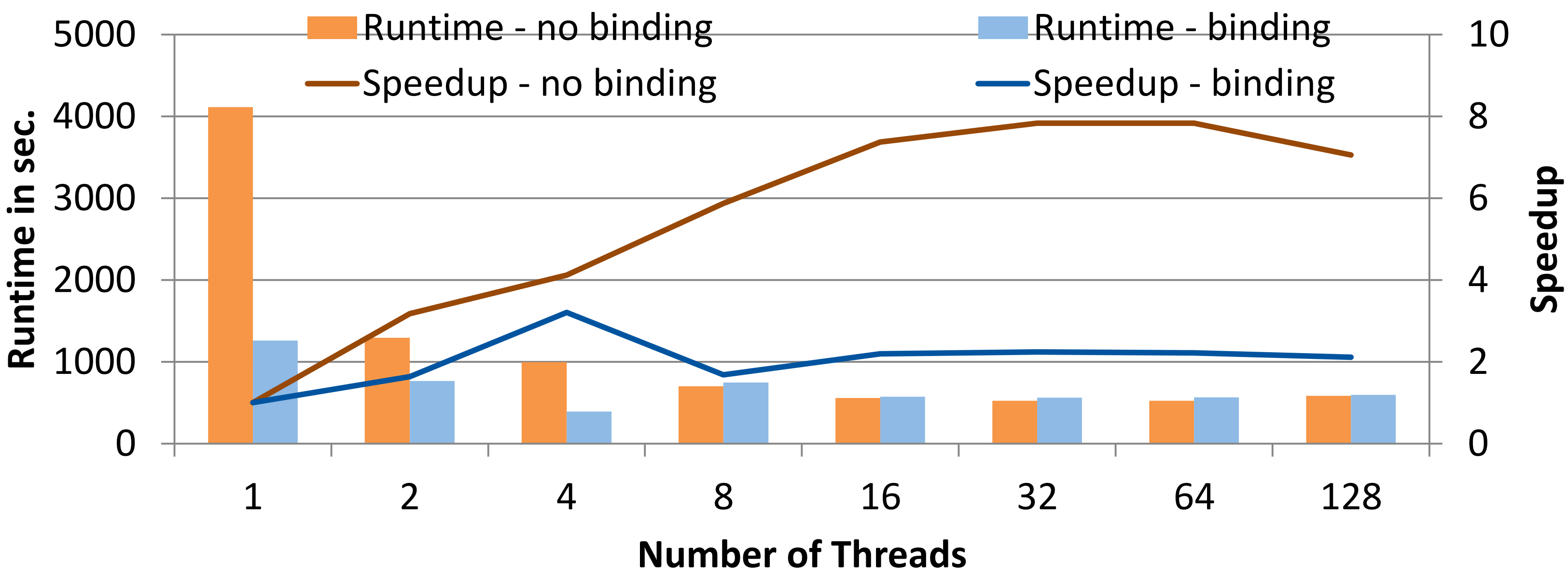
row:	0	1	3	4	7



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

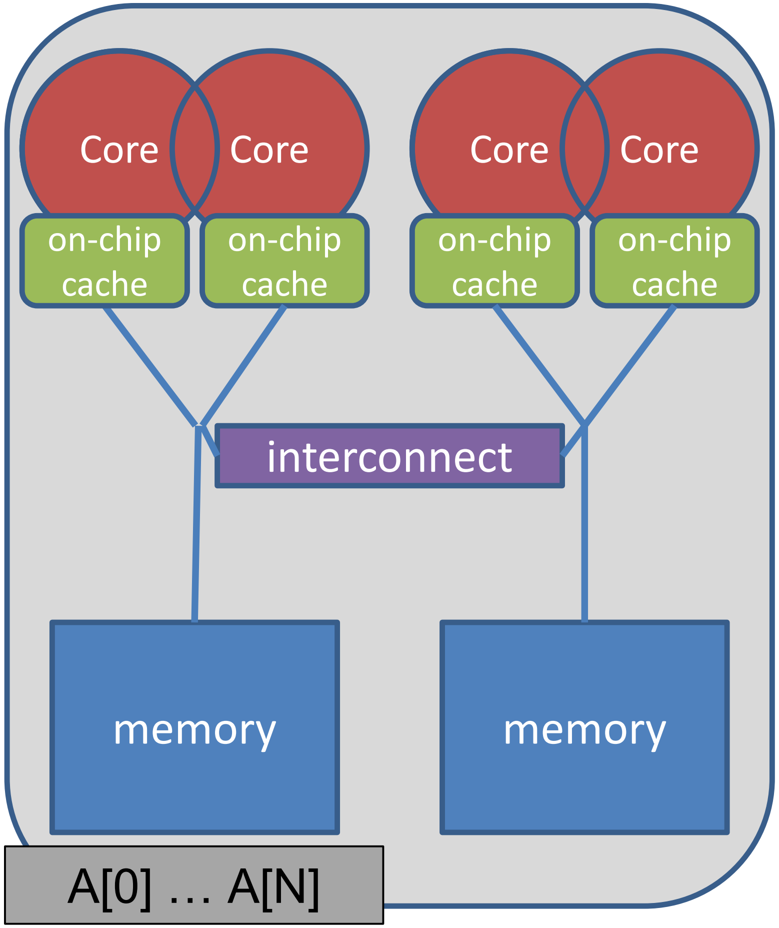
- Windows, Linux and Solaris all use the “First Touch” placement policy by default

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;
}
```



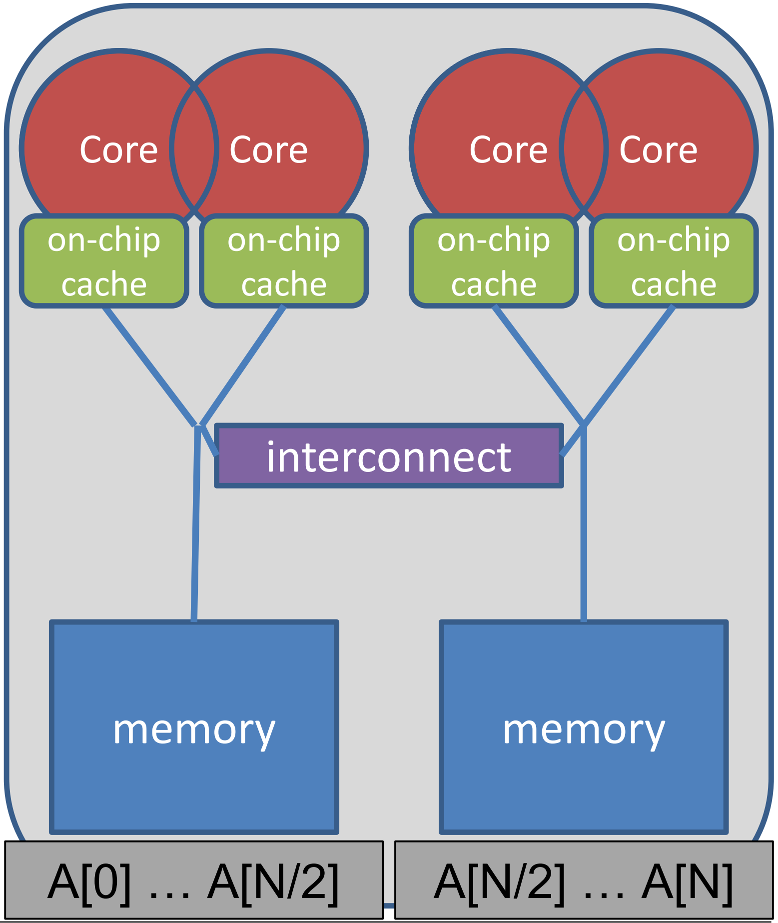
First-touch in action

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;
}
```

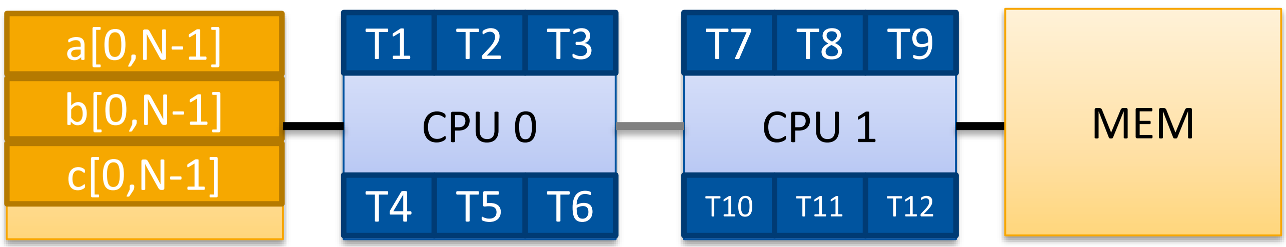


First-touch in action

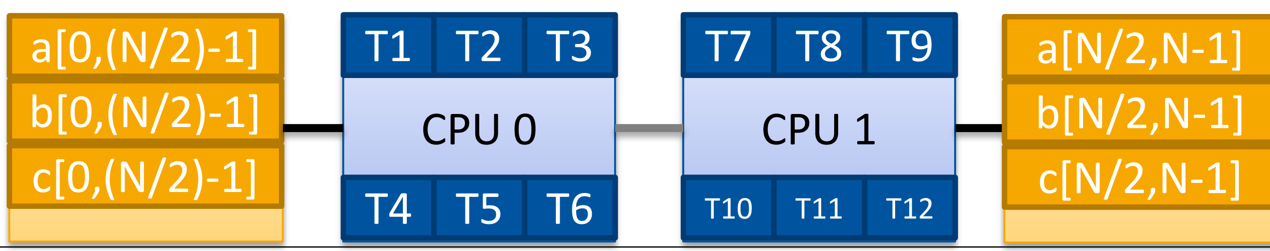
- Stream example ($\vec{a} = \vec{b} + s * \vec{c}$) with and without parallel initialization.
  – 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	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:



par_init:



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

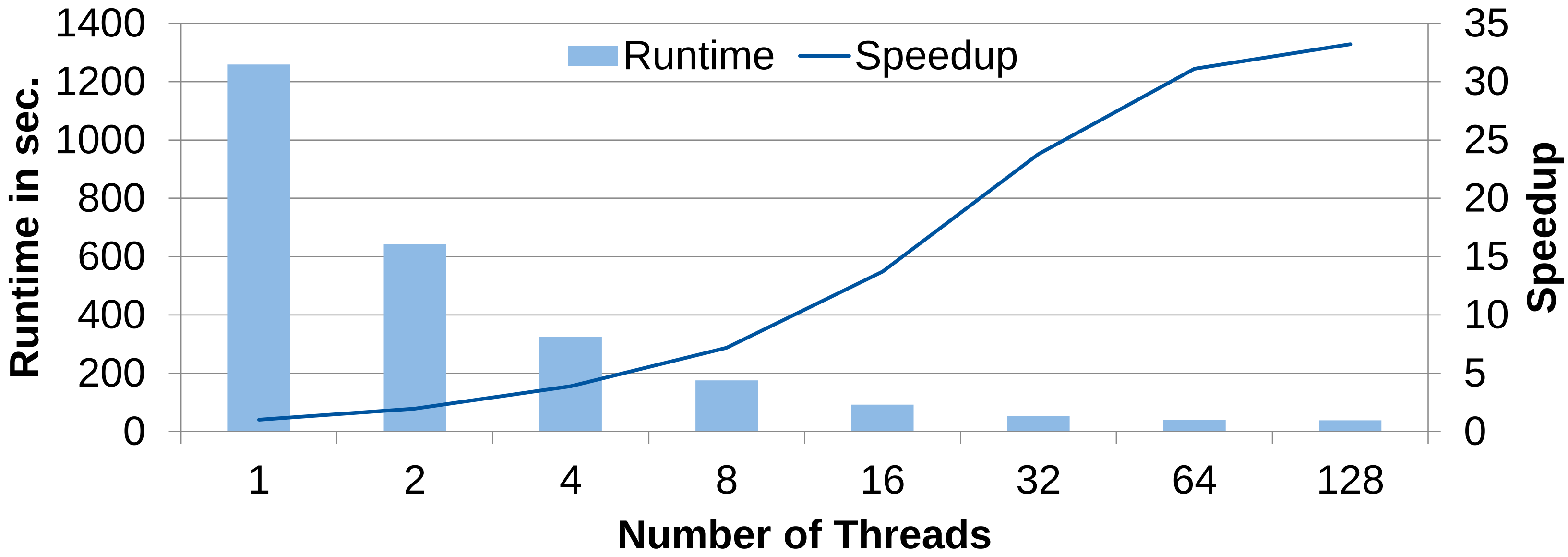
libnuma - library for NUMA control
(include numa.h and link -lnuma)

- `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



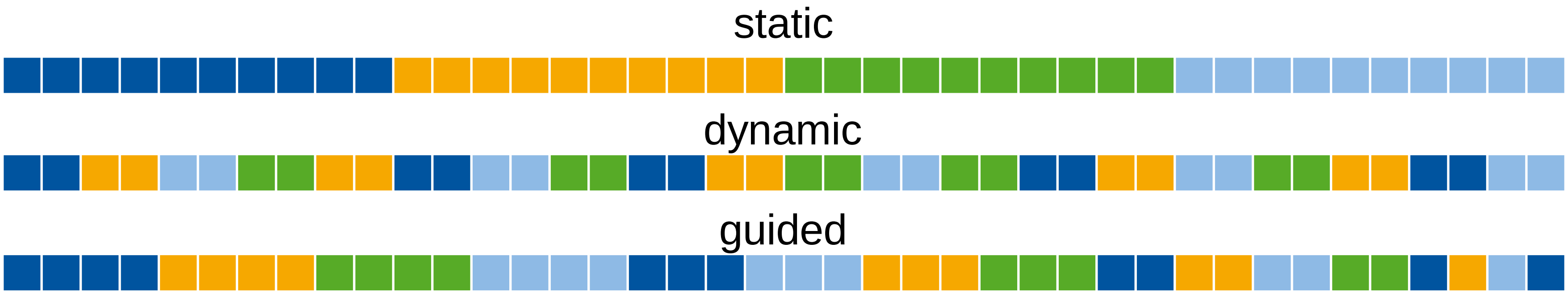
- Scalability improved a lot by this tuning on the large machine.

Work Distribution

Work Distribution

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

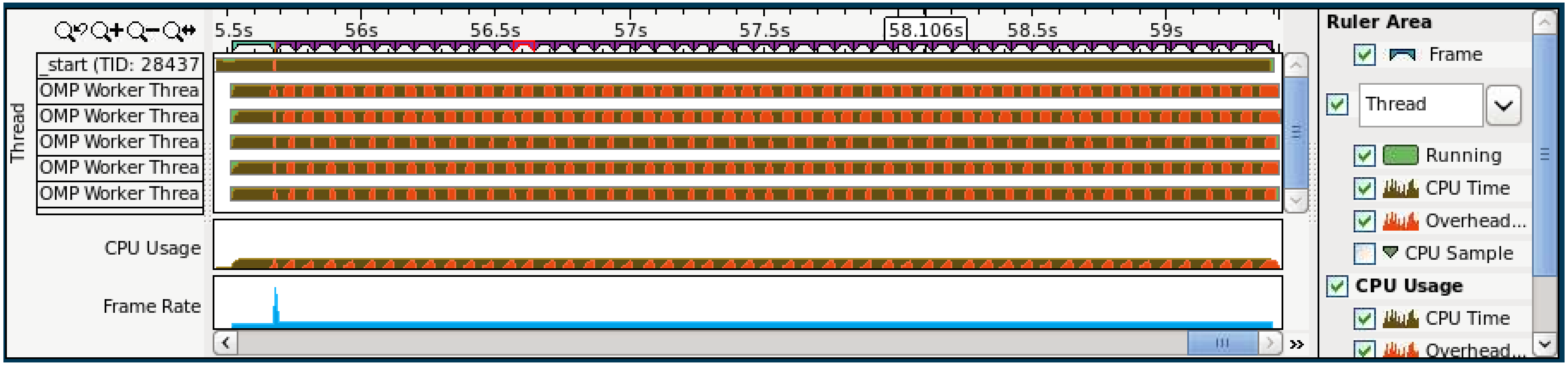
```
#pragma omp parallel for schedule(...)
for (i=0 ; i < 40 ; i++){
            A[i]=42;
}
```



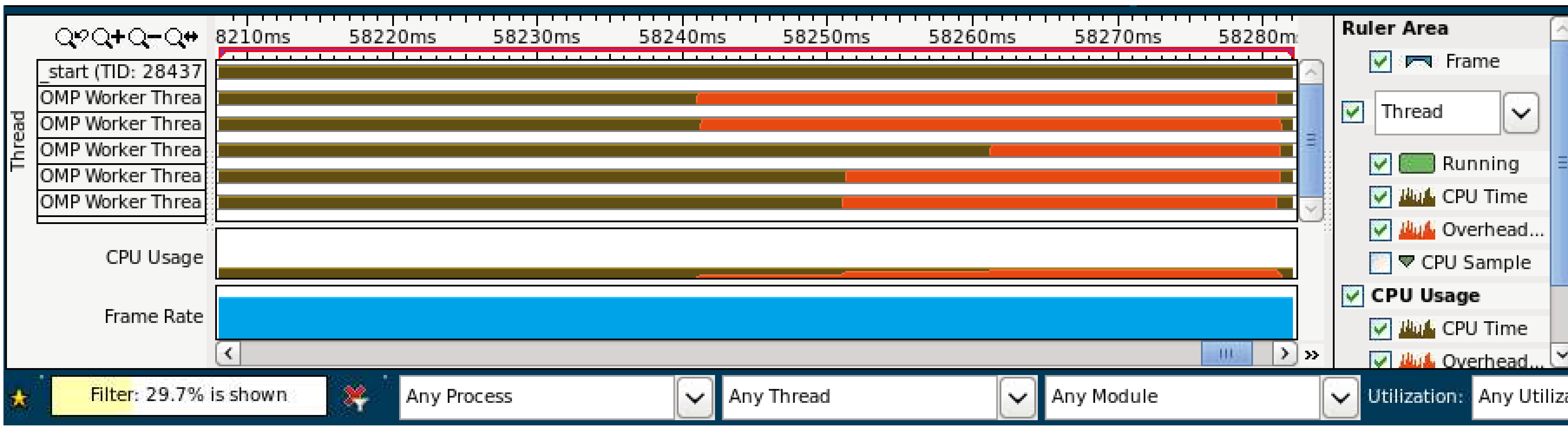
- For tasking no fixed mapping is provided.

Case Study:CG

- Different iterations of the CG Solver

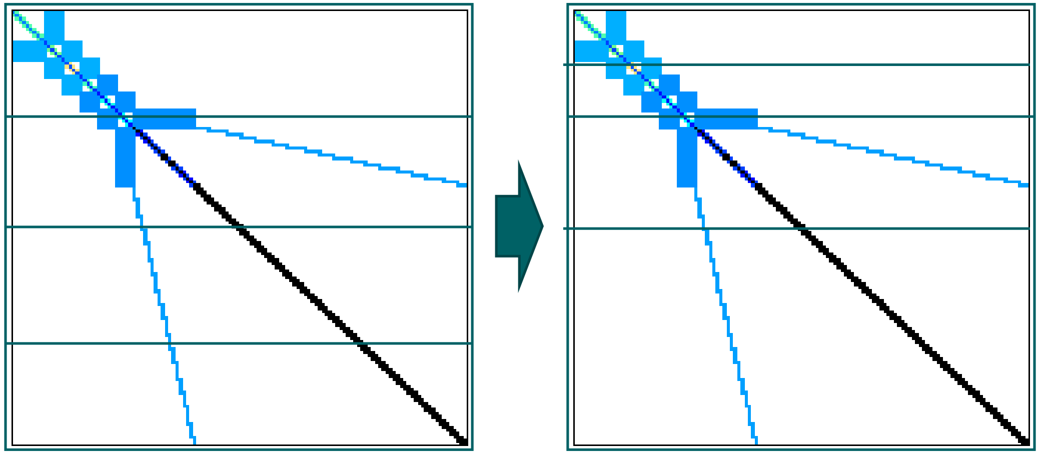


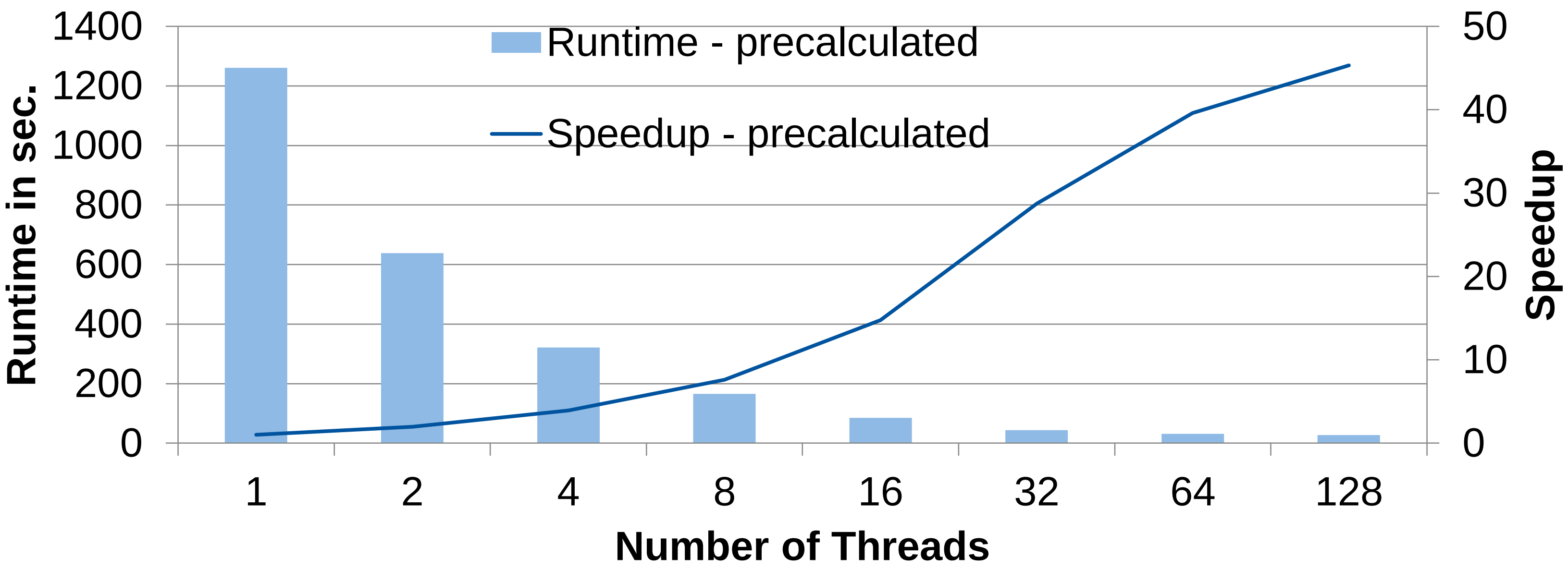
- Zoomed in on one iteration



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

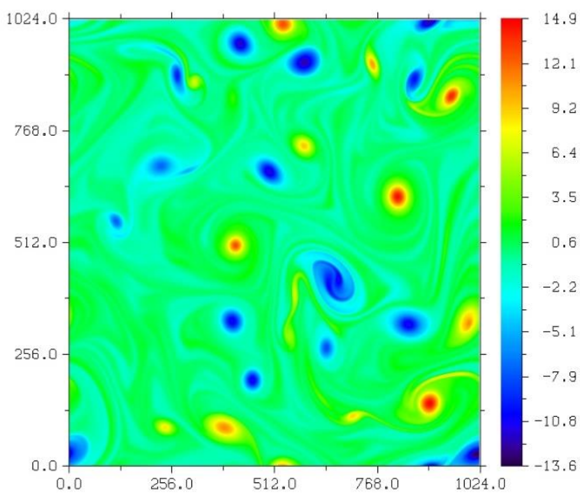


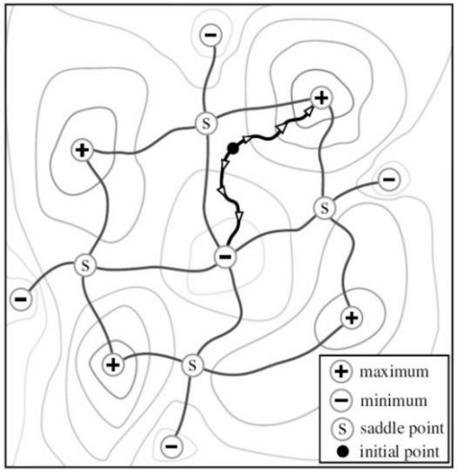


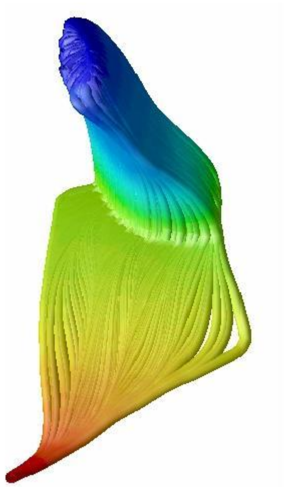
Application Case Study

TrajSearch

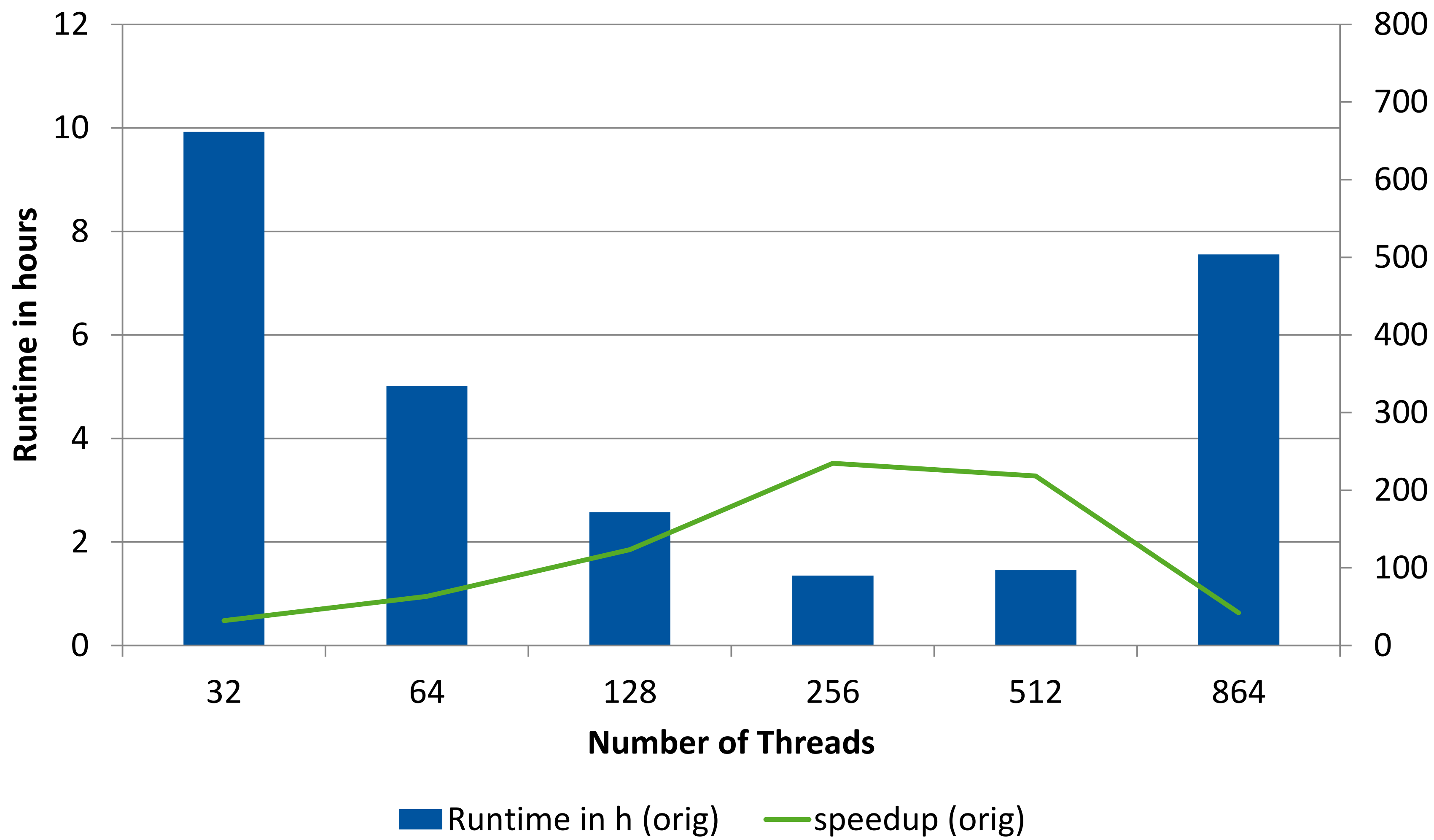
- Post-processing code for dissipation element analysis
- Follows Trajectories starting at each gridcell in direction of ascending and descending gradient of an passive 3D scalar field
- Trajectories lead to a local maximum or minimum respectively
- The composition of all gridcells of which trajectories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





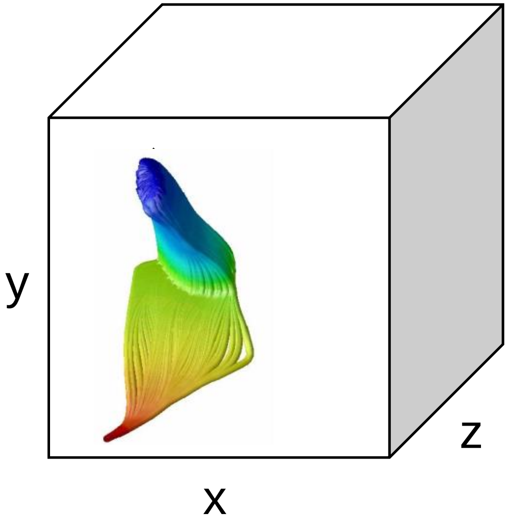


Performance Results on Numascale system



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



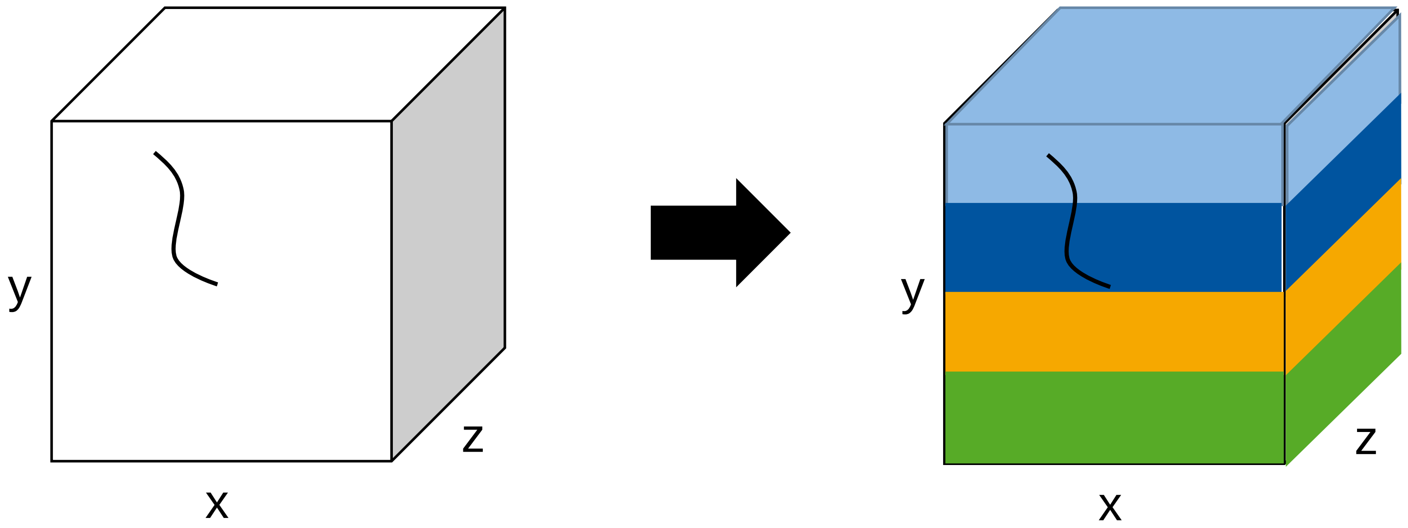
Optimization Steps (1/3)

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

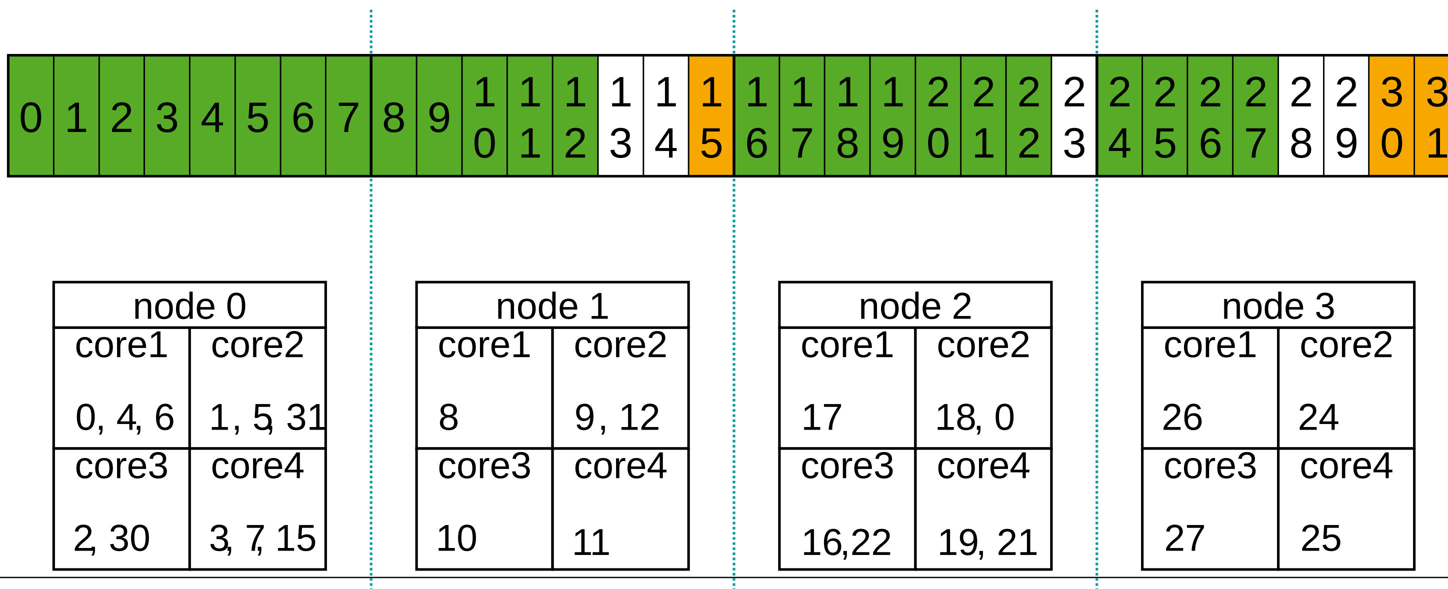
Optimization Steps (2/3)

- 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



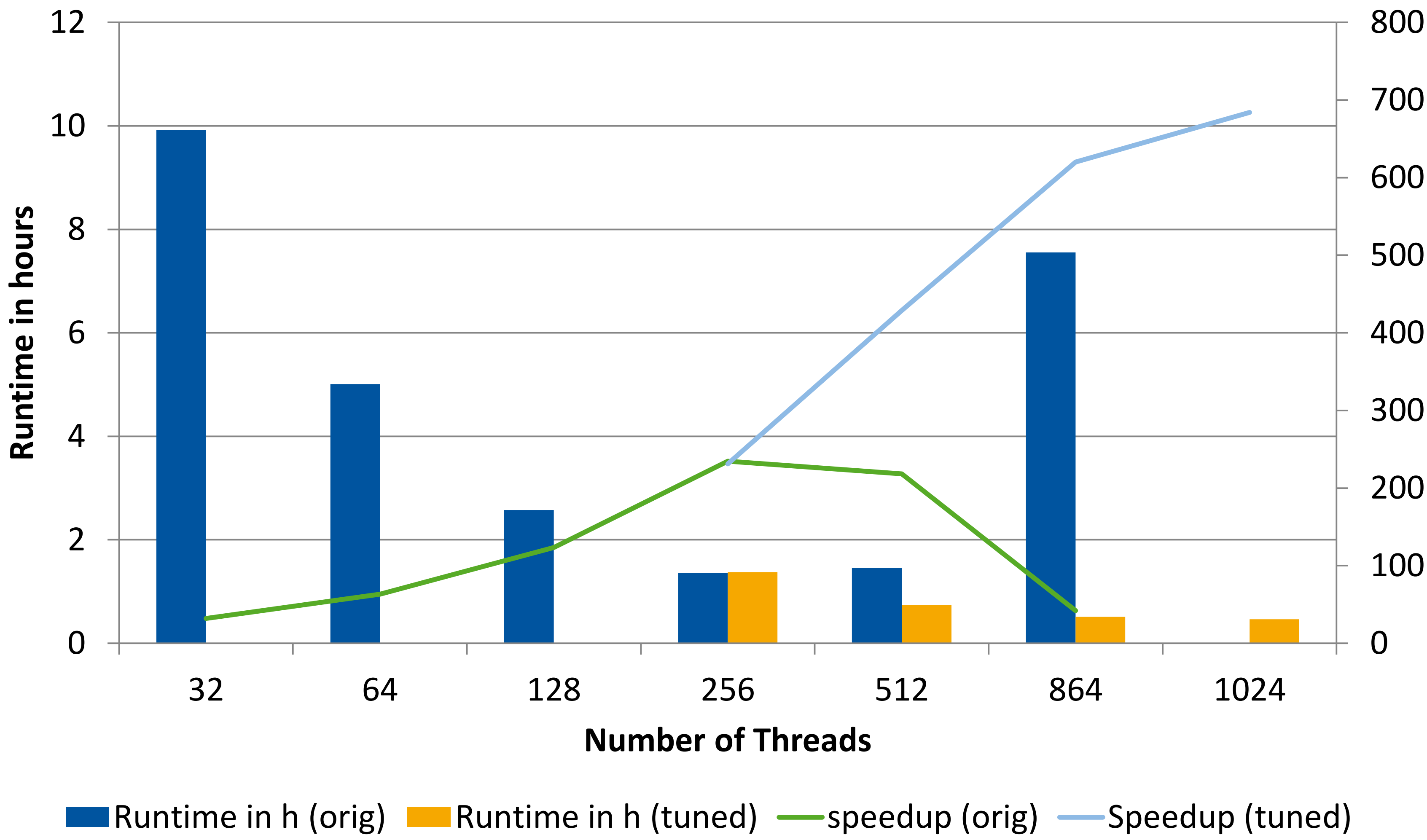
Optimization Steps (3/3)

- 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



IT Center
RWTH AACHEN UNIVERSITY

TrajSearch on Numascale system



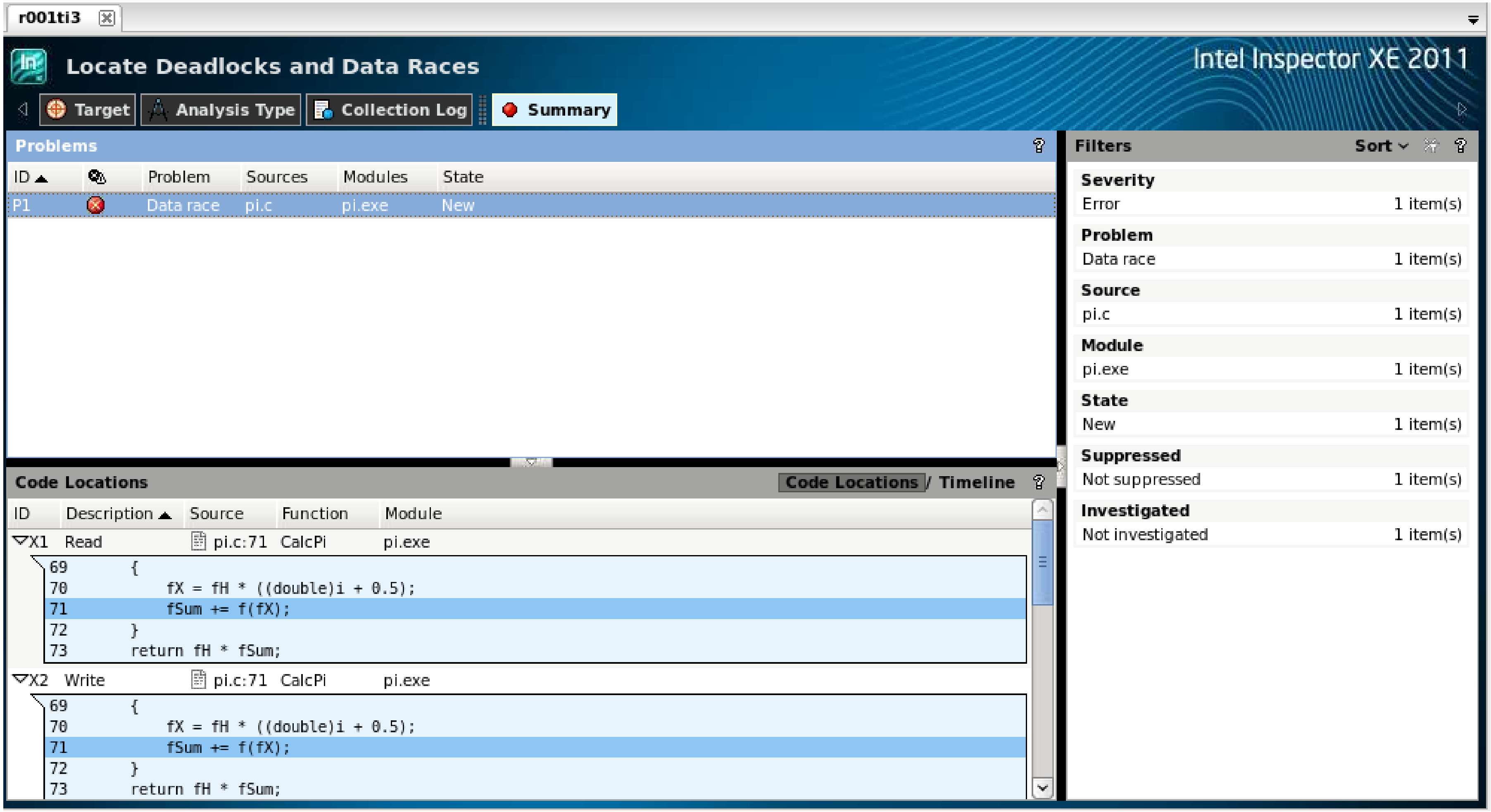
Tools for OpenMP

Data race

- Data Race: the typical OpenMP programming error, when:
  - two or more threads access the same memory location, and
  - at least one of these accesses is a write, and
  - the accesses are not protected by locks or critical regions, and
  - the accesses are not synchronized, e.g. by a barrier.
- Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run
- In many cases *private* clauses, *barriers* or *critical regions* are missing
- Data races are hard to find using a traditional debugger
  - Use the *Intel Inspector XE* or similar tool

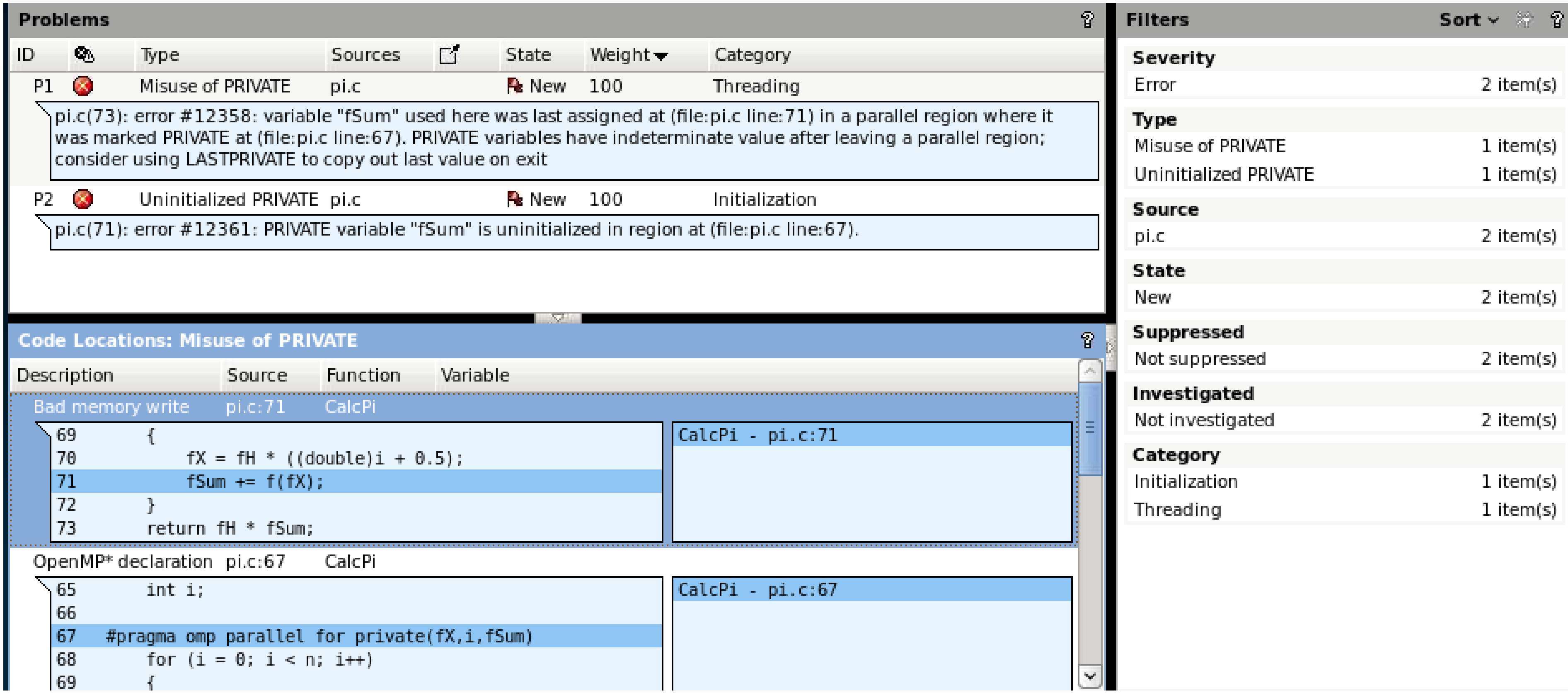
Inspector XE

- Runtime detection of data races



Inspector XE - Static Security Analysis

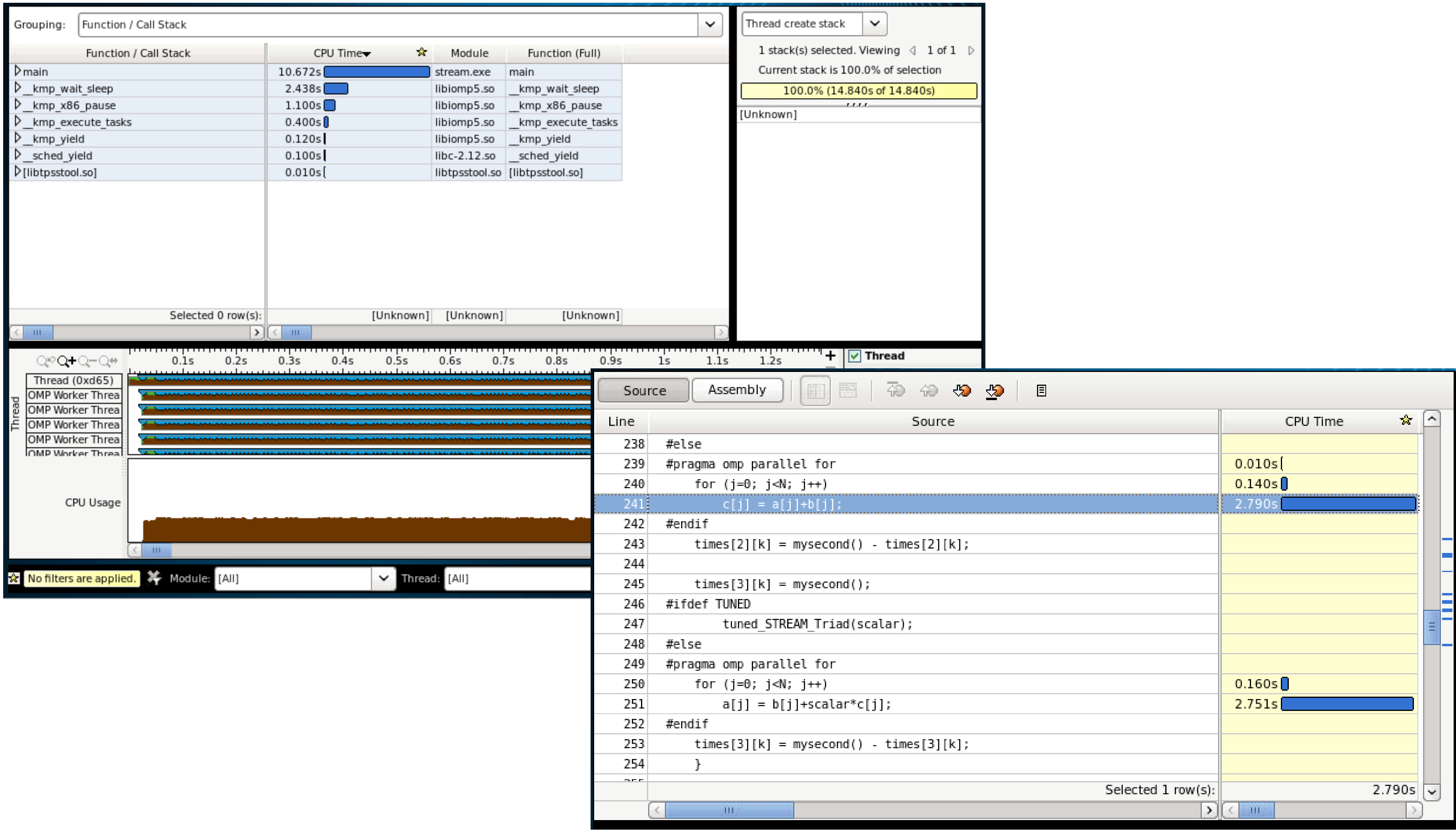
- Compile time checks with SSA (compile with "“-diag-enable sc-full")



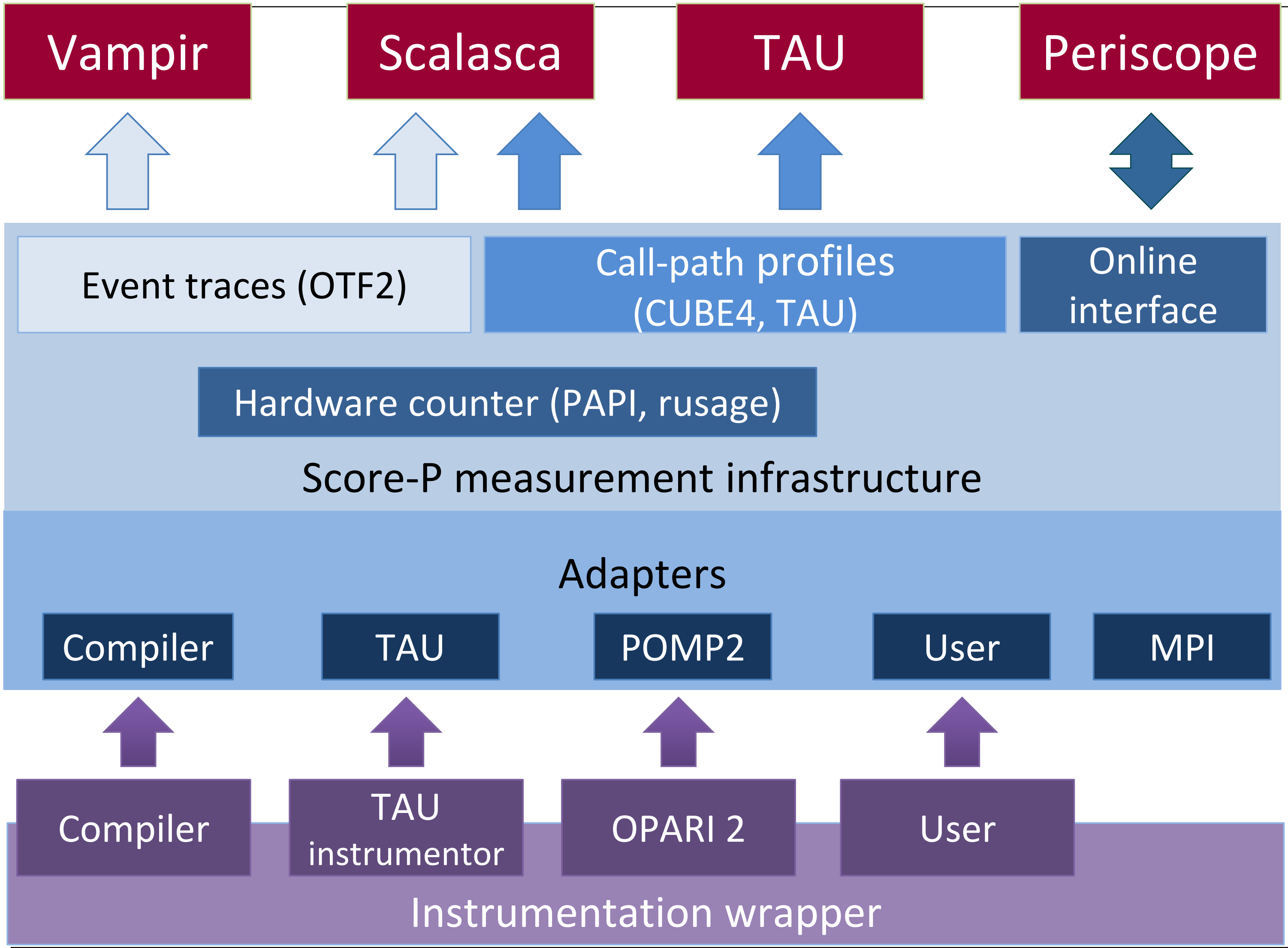
Intel VTune Amplifier XE

- Performance Analyses for
  - Serial Applications
  - Shared Memory Parallel Applications
- Sampling Based measurements
- Features:
  - Hot Spot Analysis
  - Concurrency Analysis
  - Wait
  - Hardware Performance Counter Support

Performance tools - VTune



Performance tools - Score-P



Advanced OpenMP - NUMA Architectures
Dirk Schmidl, Christian Terboven | IT Center RWTH Aachen University
OpenMPCon / IWOMP , October 5, 2016, Nara, Japan

Performance Tools Score-P / Cube

1. **Metric tree**
2. **Call tree**
3. **Topology tree**

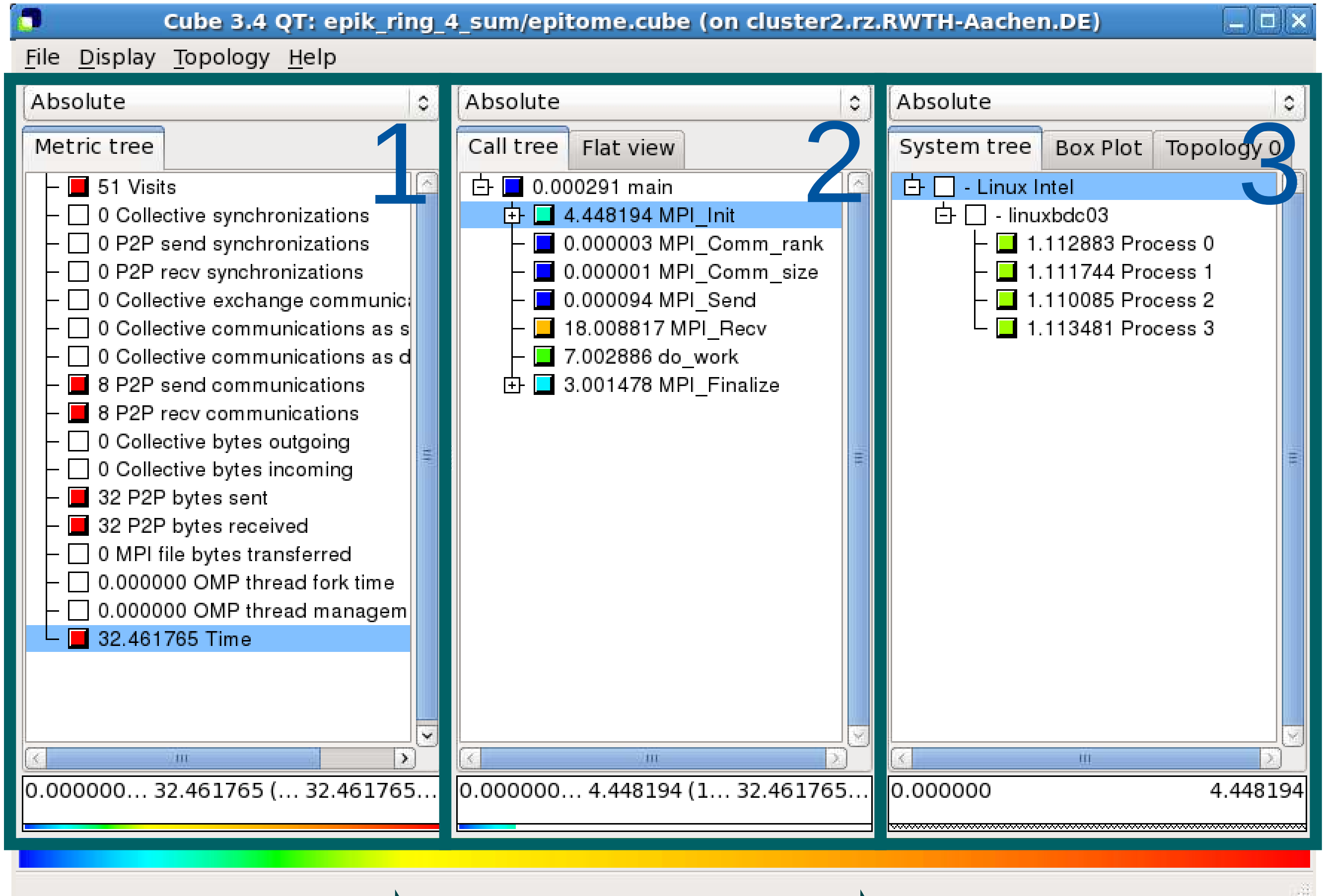
- **All views are coupled from left to right:**

1. **choose a metric**

- **-> this metric is shown for all functions**

2. **choose a function**

- **-> the right view shows the distribution over processes**



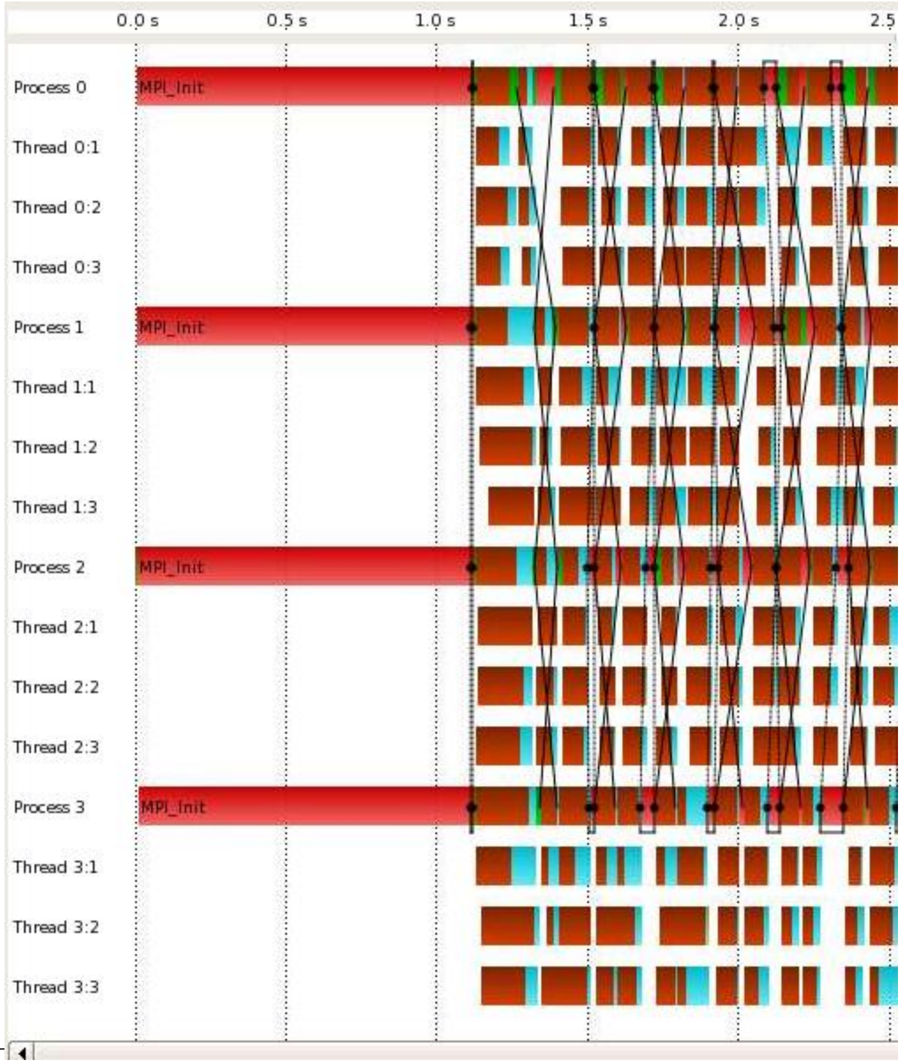
Total execution time is 32 sec.

Out of these 4.4 sec. are spent in MPI_Init().

Out of these 1.1 sec is spent by every process.

Performance Tools Score-P / Vampir

- The Timeline gives a detailed view of all events.
- Regions and Messages of all Processes and Threads are shown.
- Zoom horizontal or vertical for more detailed information.
- Click on a message or region for specific details.



Thank you for your attention!

Questions?