



# OpenMP Overview

Dirk Schmidl

[schmidl@itc.rwth-aachen.de](mailto:schmidl@itc.rwth-aachen.de)

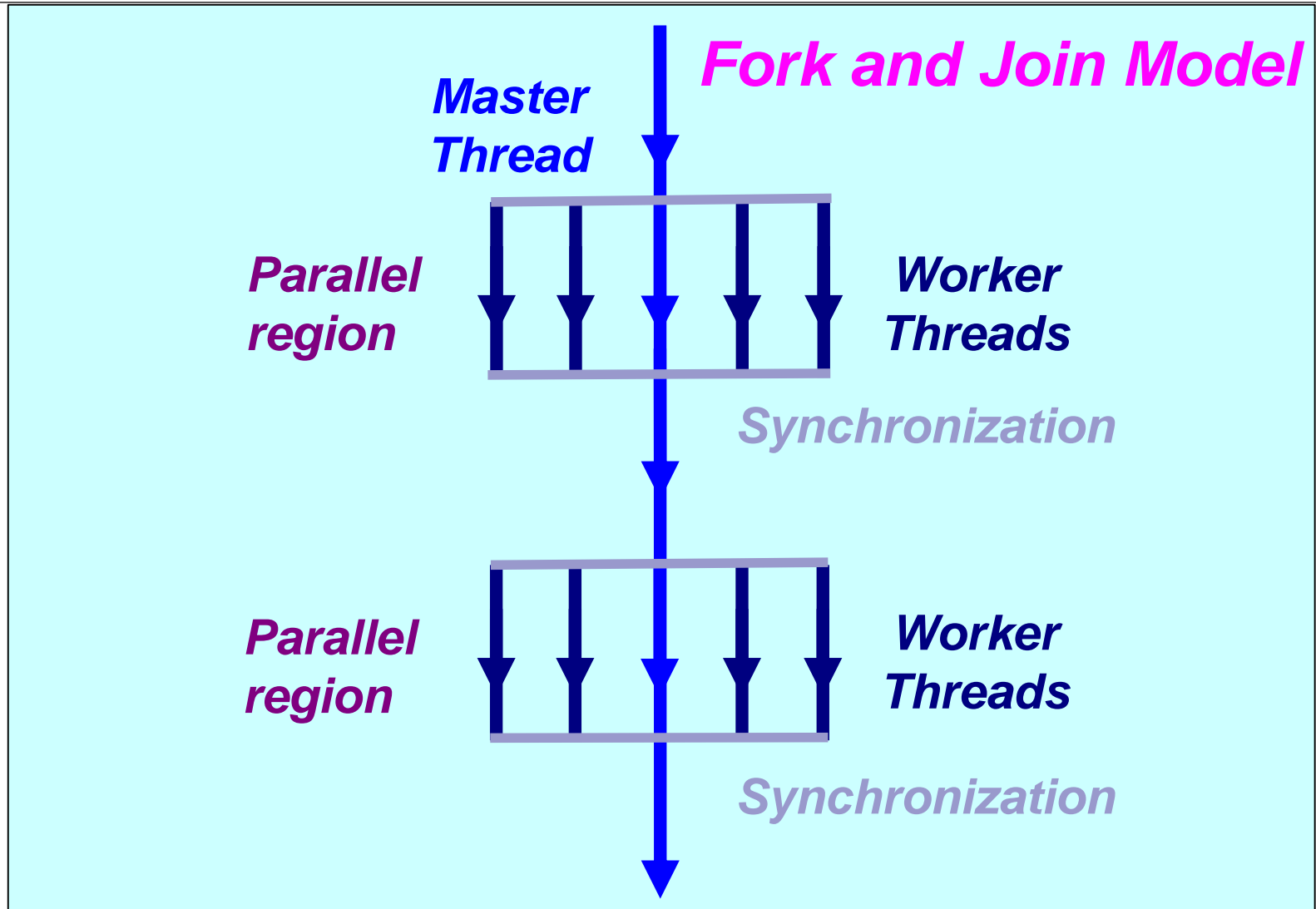
Thanks to the following people for providing parts of the slides:

- Christian Terboven (RWTH Aachen)
- Sandra Wienke (RWTH Aachen)
- Michael Klemm (Intel)

# Core concept

---

# The OpenMP Execution Model



## Defining Parallelism in OpenMP

---

*A parallel region is a block of code executed by all threads in the team*

```
#pragma omp parallel [clause[,] clause] ...  
{  
    "this code is executed in parallel"  
} // End of parallel section (note: implied barrier)
```

```
!$omp parallel [clause[,] clause] ...  
    "this code is executed in parallel"  
!$omp end parallel (note: implied barrier)
```

# The Worksharing Constructs

```
#pragma omp for
{
    ....
}
```

```
!$OMP DO
```

```
    ....
!$OMP END DO
```

```
#pragma omp sections
{
    ....
}
```

```
!$OMP SECTIONS
```

```
    ....
!$OMP END SECTIONS
```

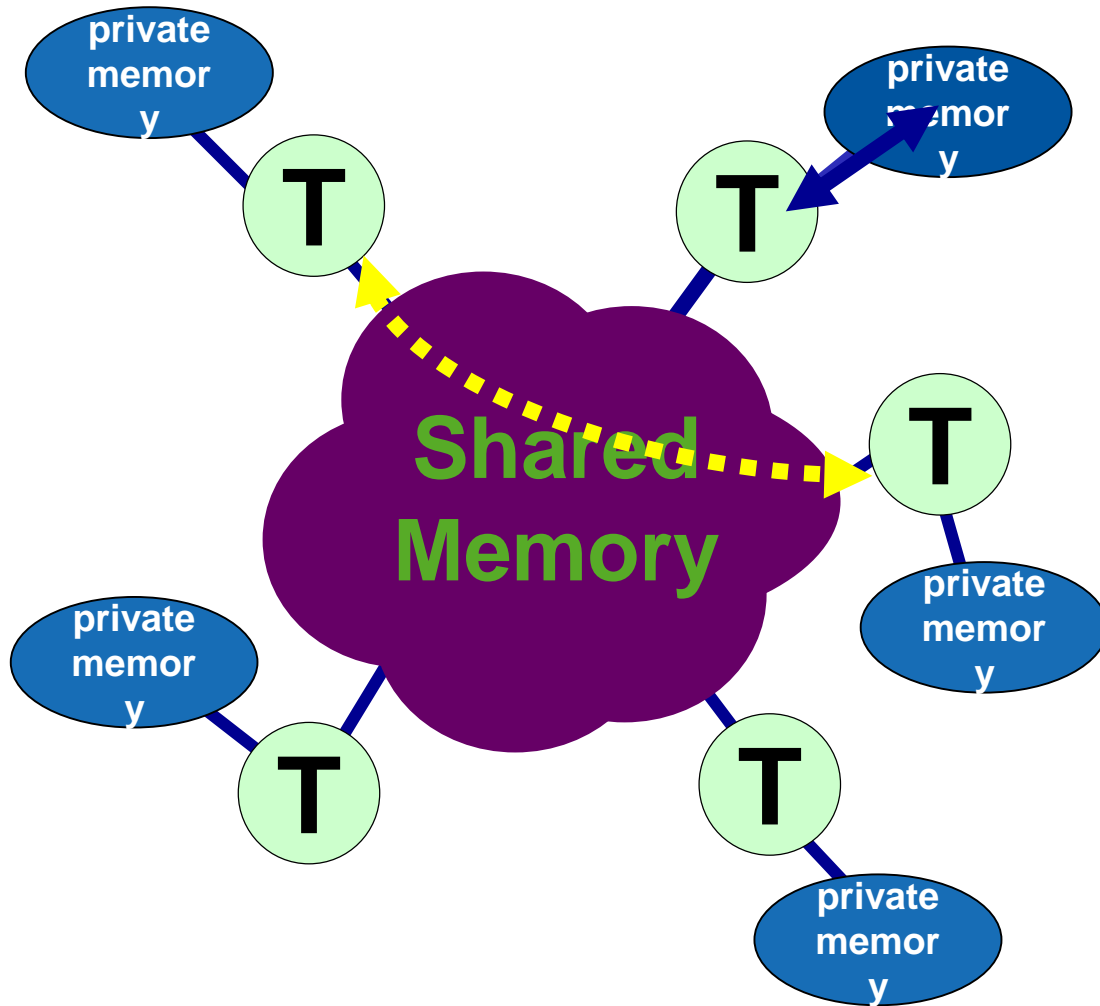
```
#pragma omp single
{
    ....
}
```

```
!$OMP SINGLE
```

```
    ....
!$OMP END SINGLE
```

- ✓ *The work is distributed over the threads*
- ✓ *Must be enclosed in a parallel region*
- ✓ *Must be encountered by all threads in the team, or none at all*
- ✓ *No implied barrier on entry*
- ✓ *Implied barrier on exit (unless the nowait clause is specified)*
- ✓ *A work-sharing construct does not launch any new threads*

# The OpenMP Memory Model



- ◆ *All threads have access to the same, globally shared memory*
- ◆ *Data in private memory is only accessible by the thread owning this memory*
- ◆ *No other thread sees the change(s) in private memory*
- ◆ *Data transfer is through shared memory and is 100% transparent to the application*

# Scoping Rules

---

- Managing the Data Environment is the challenge of OpenMP.
- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master's value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

# Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread\_key\_create (Posix-Threads), keyword `__thread` (GNU extension)

**Really: try to avoid the use of threadprivate and static variables!**

C/C++

```
static int i;  
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

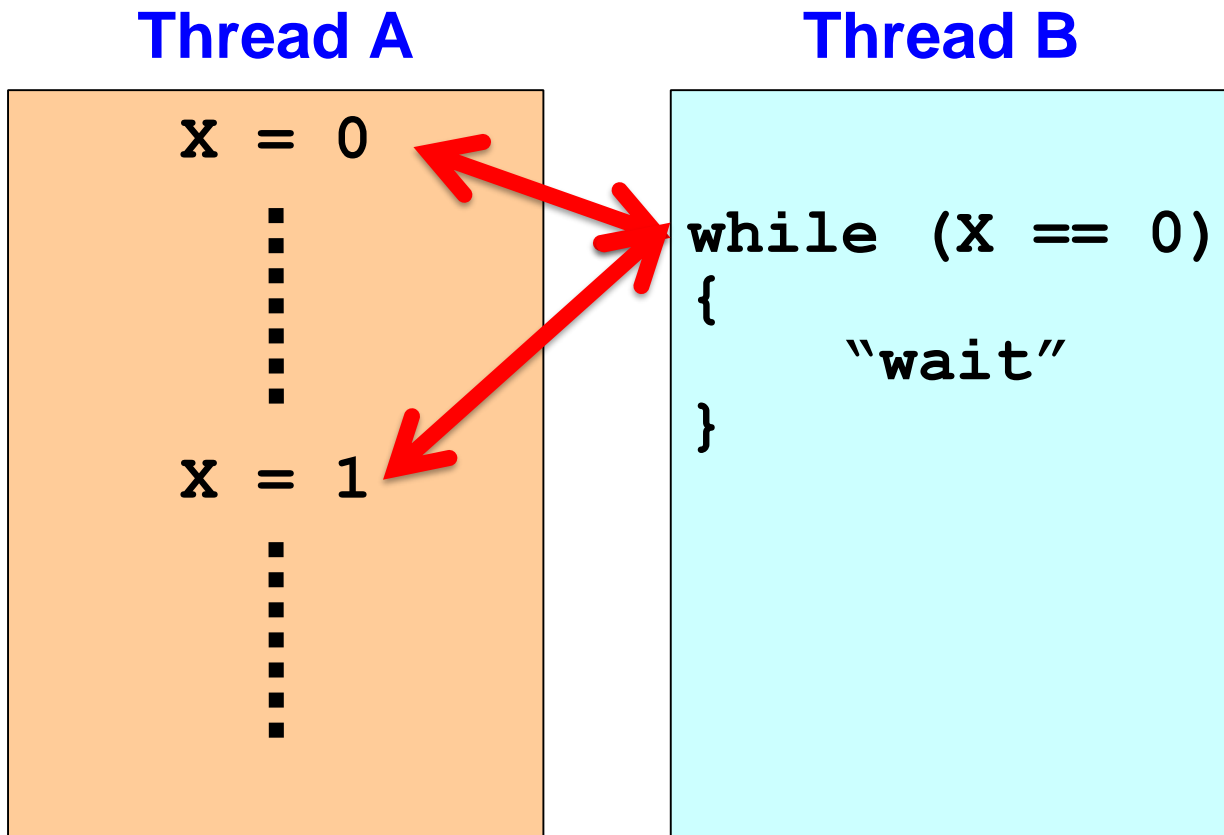


# Gotcha's

---

- Need to get this right
  - Part of the learning curve
- Private data is undefined on entry and exit
  - Can use firstprivate and lastprivate to address this
- Each thread has its own temporary view on the data
  - Applicable to shared data only
  - Means different threads may temporarily not see the same value for the same variable ...
  - Let me explain

# The Flush Directive



***If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)***

# The Flush construct

---

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

- Strongly recommended: do **not** use this directive
  - ... unless really necessary. Really 😊.
  - Could give very subtle interactions with compilers
  - If you insist on still doing so, be prepared to face the OpenMP language lawyers
- Implied on many constructs
  - A good thing
  - This is your safety net

# The OpenMP Barrier

---

- Several constructs have an implied barrier
  - This is another safety net (has implied flush by the way)
- In some cases, the implied barrier can be left out through the “nowait” clause
- This can help fine tuning the application
  - But you’d better know what you’re doing
- The explicit barrier comes in quite handy then

```
#pragma omp barrier
```

```
!$omp barrier
```

# The Nowait Clause

---

- To minimize synchronization, some directives support the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct
- In C, it is one of the clauses on the pragma
- In Fortran, it is appended at the closing part of the construct

```
#pragma omp for nowait
{
    :
}
```

```
!$omp do
    :
    :
!$omp end do nowait
```

# Mutual exclusion

---

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- OpenMP also provides locks und locking routines
  - omp\_lock\_t
  - omp\_init\_lock()
  - omp\_set\_lock()
  - omp\_unset\_lock()
  - omp\_test\_lock()
  - omp\_init\_lock()

# Performance Optimization for Locking

---

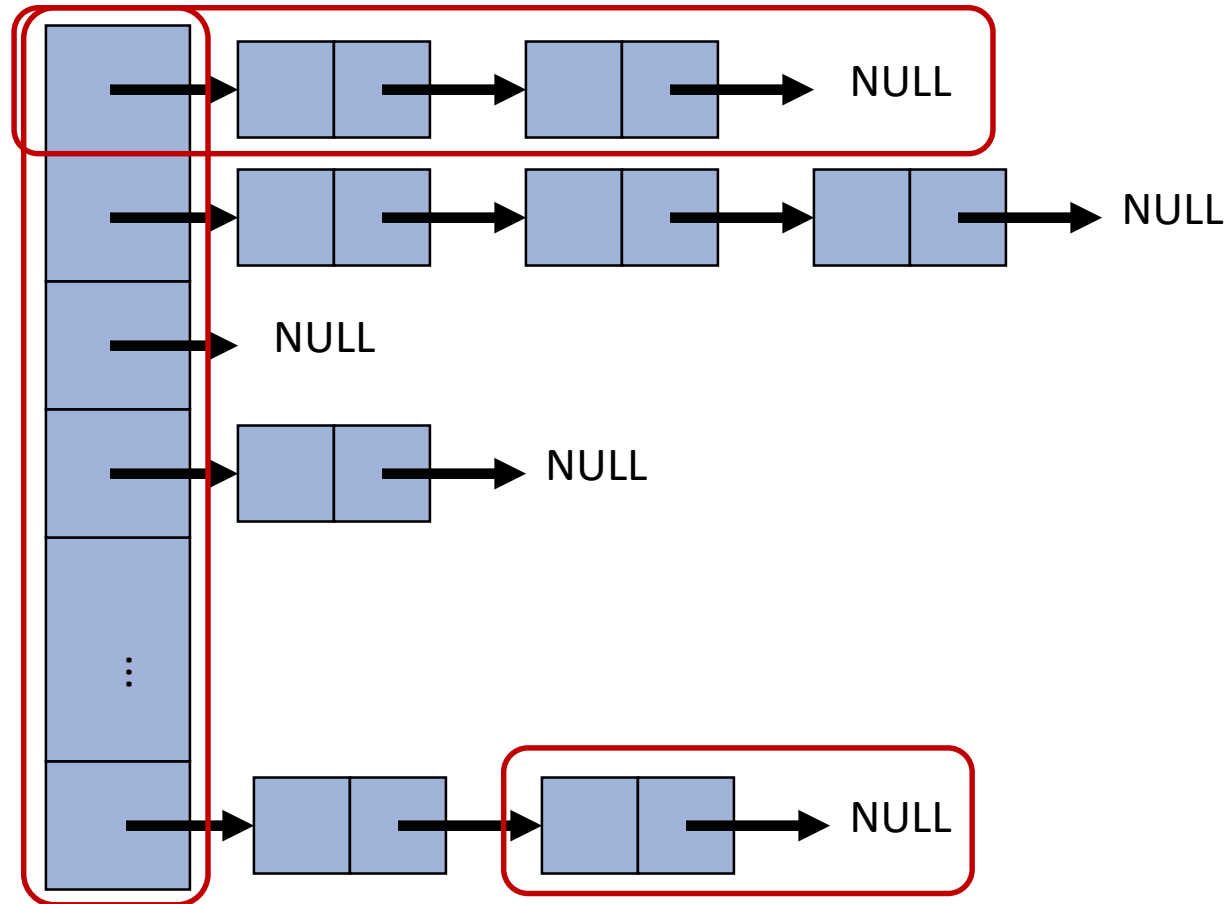
- Don't use locks 😊
- Fine-grained locking
  - Push locks towards the finest granularity of data access (if possible)
  - May avoid mutual exclusion of lengthy sequences of execution
- Lock-free data structures
  - Don't use locks, but use atomic instruction of the machine
  - Advice: do not attempt to implement such a data structure yourself
- Use transactional memory
  - Speculate on the mutual exclusion (increased parallelism if no conflicts)
  - Pay extra if a conflict happens

# Fine-grained Locking

- Example: hash table with linked lists for buckets

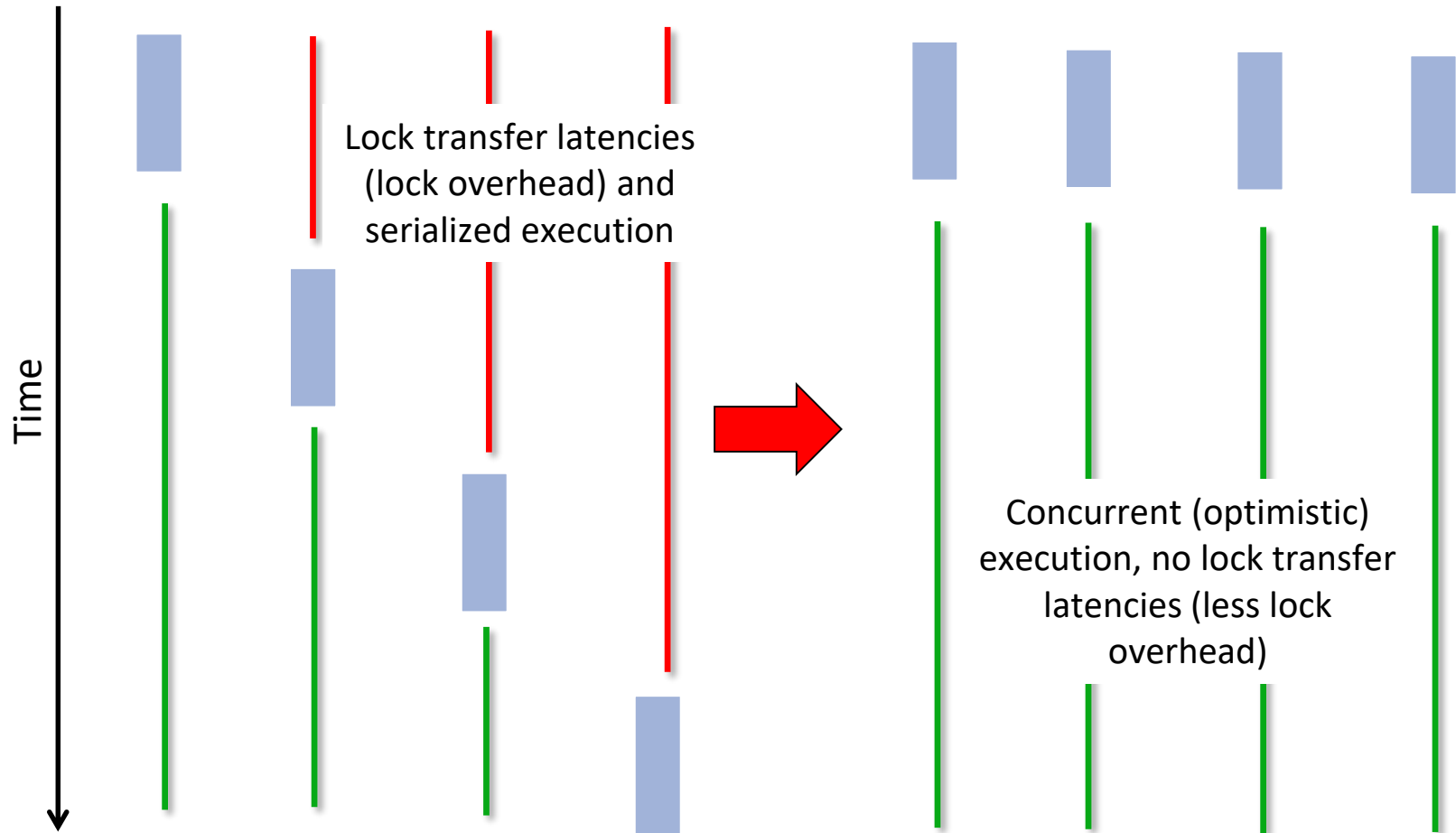
- Tradeoff:

- (Expected) degree of parallelism
  - Number of individual locks required
  - Implementation complexity
- Can be combined with TM
    - See next slide





# Transactional memory



# Lock hints in OpenMP

- Lock hints can help the Runtime to choose the best implementation of a lock.
- Hints are:
  - `omp_lock_hint_none`
  - `omp_lock_hint_uncontended`
  - `omp_lock_hint_contended`
  - `omp_lock_hint_nonspeculative`
  - `omp_lock_hint_speculative`
- Hints can be combined with `+` or `|`.

## C/C++

```
omp_lock_t lck;  
omp_init_lock_with_hint(&lck);  
#pragma omp parallel  
{  
    omp_set_lock(&lck);  
    /* mutual exclusion here...*/  
    ...  
    omp_unset_lock(&lck);  
}  
omp_destroy_lock(&lck);
```

## C/C++

```
#pragma omp critical hint(...)  
{  
    ...  
}
```

# 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

The screenshot displays the Intel Inspector XE 2011 interface. The main window is titled "Locate Deadlocks and Data Races". The "Problems" pane on the left shows a single problem, P1, identified as a "Data race" in the source file "pi.c" within the module "pi.exe". The "Code Locations" pane below it shows the source code for the "CalcPi" function in "pi.c". The code is as follows:

```
69 {  
70     fX = fH * ((double)i + 0.5);  
71     fSum += f(fX);  
72 }  
73 return fH * fSum;
```

The "Filters" pane on the right shows the following counts:

Severity	Count
Error	1 item(s)

Problem	Count
Data race	1 item(s)

Source	Count
pi.c	1 item(s)

Module	Count
pi.exe	1 item(s)

State	Count
New	1 item(s)

Suppressed	Count
Not suppressed	1 item(s)

Investigated	Count
Not investigated	1 item(s)

# 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

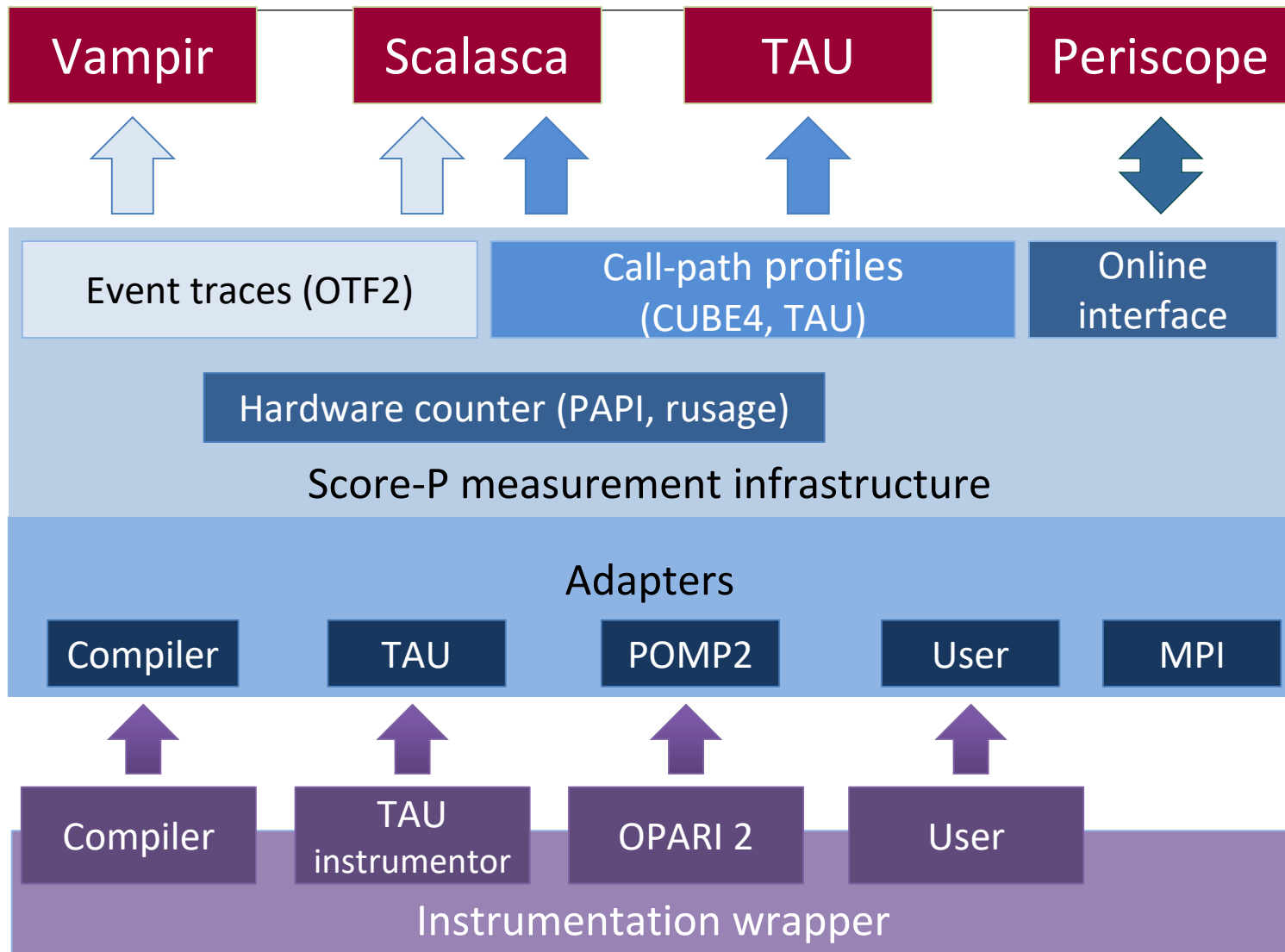
The screenshot displays the Intel VTune Performance Analyzer interface, divided into several panels:

- Top Left Panel (Function / Call Stack):** Shows a list of functions and their CPU times. The 'main' function in 'stream.exe' has the highest CPU time at 10.672s.
- Top Right Panel (Thread create stack):** Shows the thread create stack for the selected thread. It indicates that 1 stack(s) is selected, viewing 1 of 1. The current stack is 100.0% of the selection, with a total CPU time of 100.0% (14.840s of 14.840s).
- Bottom Left Panel (Thread and CPU Usage):** Displays a timeline of thread execution and CPU usage. The threads shown are Thread (0xd65), OMP Worker Threa, and OMP Worker Threa. The CPU usage is shown as a bar chart.
- Bottom Right Panel (Source Code):** Shows the source code of the selected thread. The code is in C and includes OpenMP directives. The CPU time for each line is displayed on the right side of the code editor.

Function / Call Stack	CPU Time	Module	Function (Full)
main	10.672s	stream.exe	main
_kmp_wait_sleep	2.438s	libiomp5.so	_kmp_wait_sleep
_kmp_x86_pause	1.100s	libiomp5.so	_kmp_x86_pause
_kmp_execute_tasks	0.400s	libiomp5.so	_kmp_execute_tasks
_kmp_yield	0.120s	libiomp5.so	_kmp_yield
_sched_yield	0.100s	libc-2.12.so	_sched_yield
[libtpstool.so]	0.010s	libtpstool.so	[libtpstool.so]

Line	Source	CPU Time
238	#else	
239	#pragma omp parallel for	0.010s
240	for (j=0; j<N; j++)	0.140s
241	c[j] = a[j]+b[j];	2.790s
242	#endif	
243	times[2][k] = mysecond() - times[2][k];	
244		
245	times[3][k] = mysecond();	
246	#ifdef TUNED	
247	tuned_STREAM_Triad(scalar);	
248	#else	
249	#pragma omp parallel for	0.160s
250	for (j=0; j<N; j++)	
251	a[j] = b[j]+scalar*c[j];	2.751s
252	#endif	
253	times[3][k] = mysecond() - times[3][k];	
254	}	

# Performance tools - Score-P



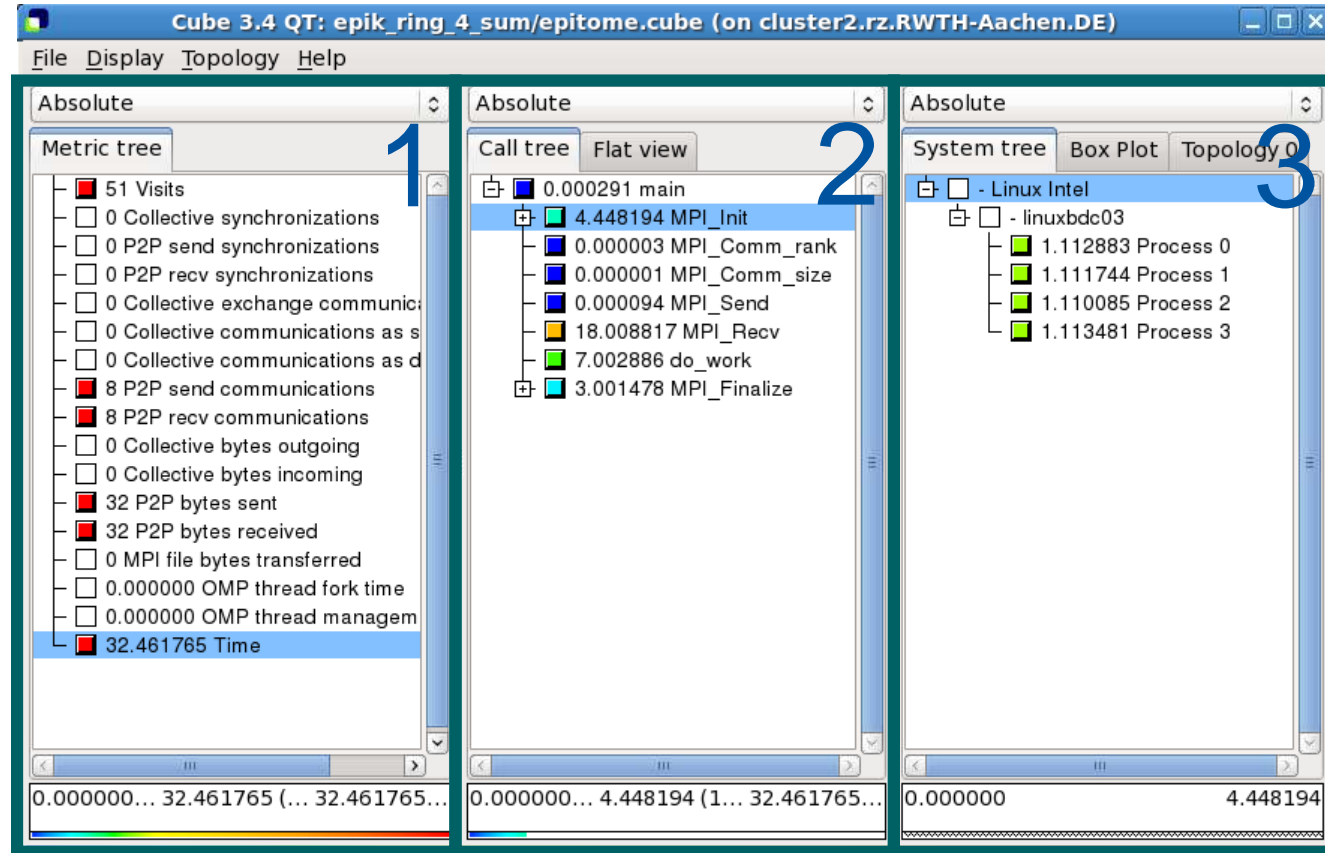


# Performance Tools Score-P / Cube

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



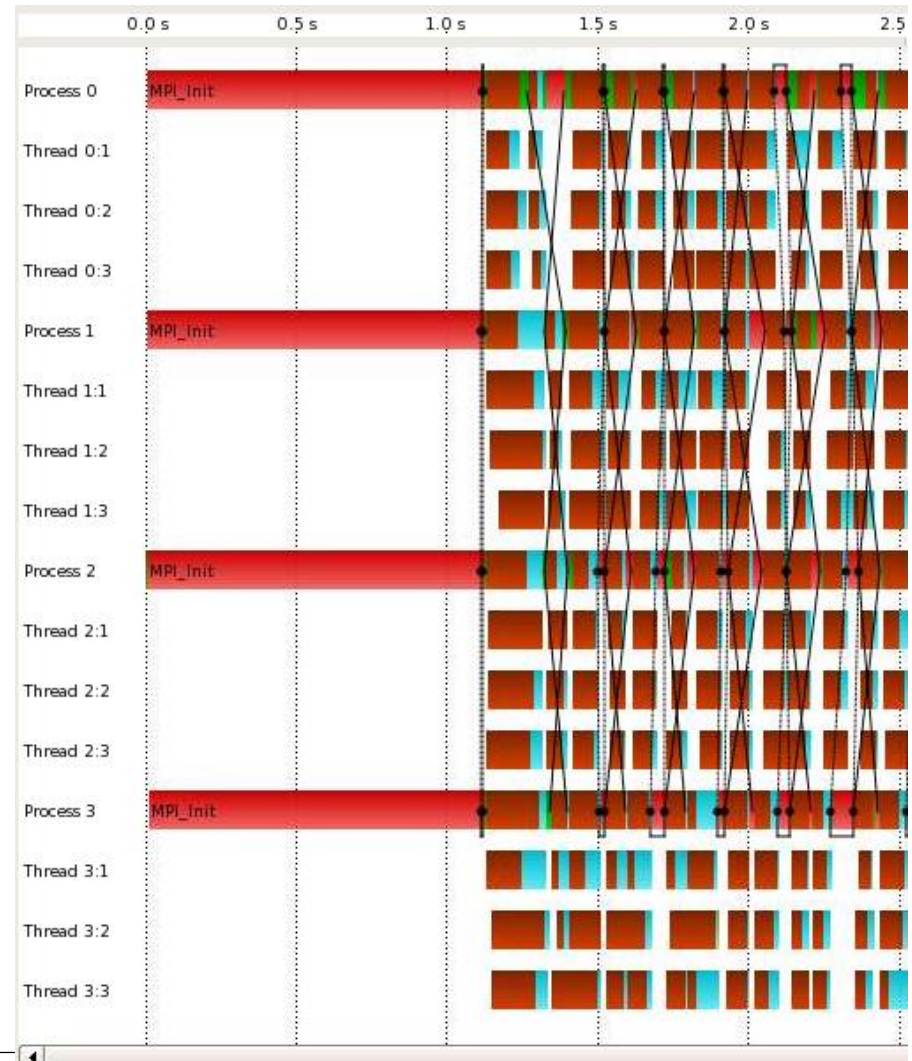
Total execution time is 32 sec.

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?**