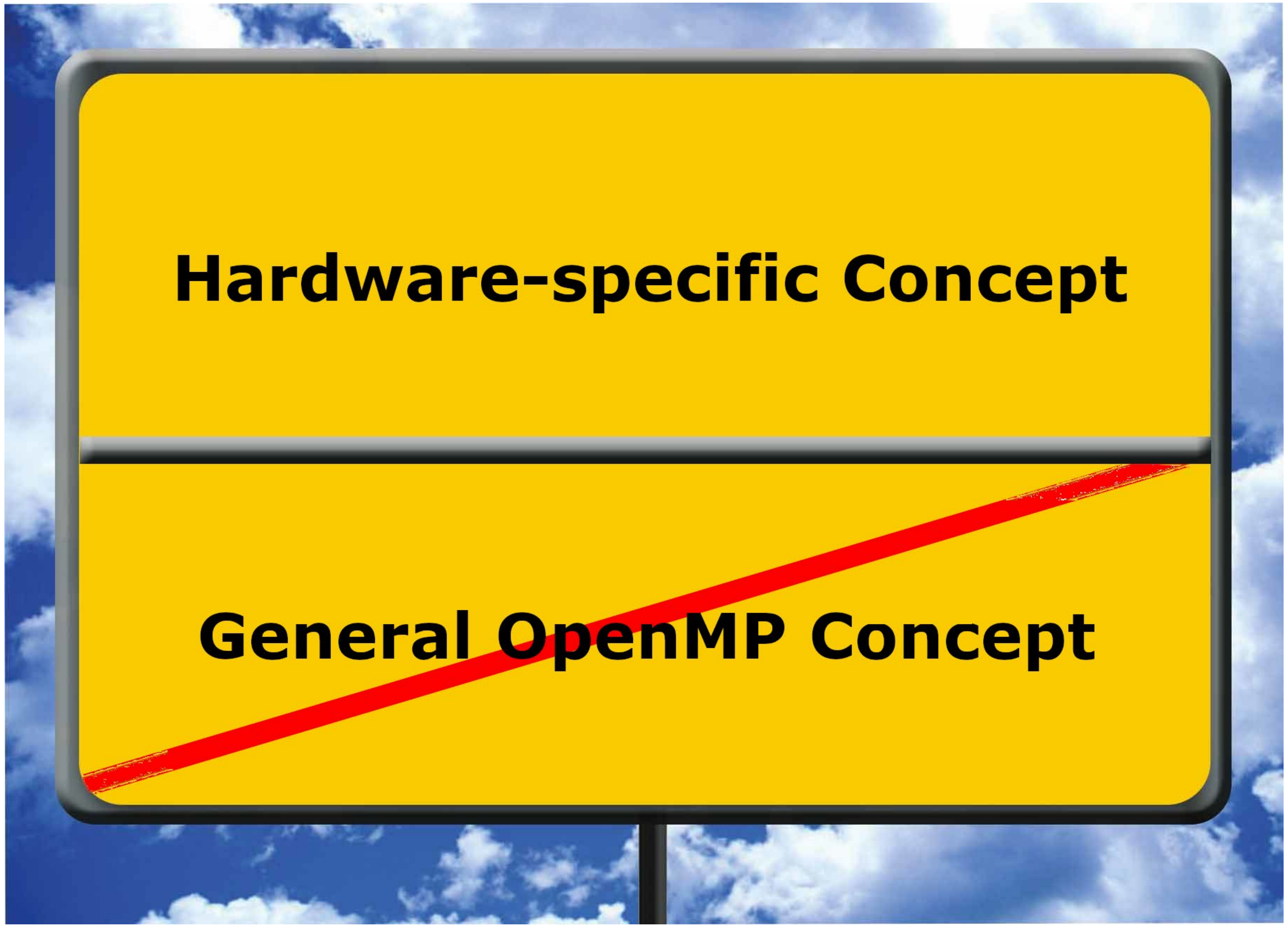
Introduction to OpenMP

Dr. Christian Terboven

Introduction to OpenMP

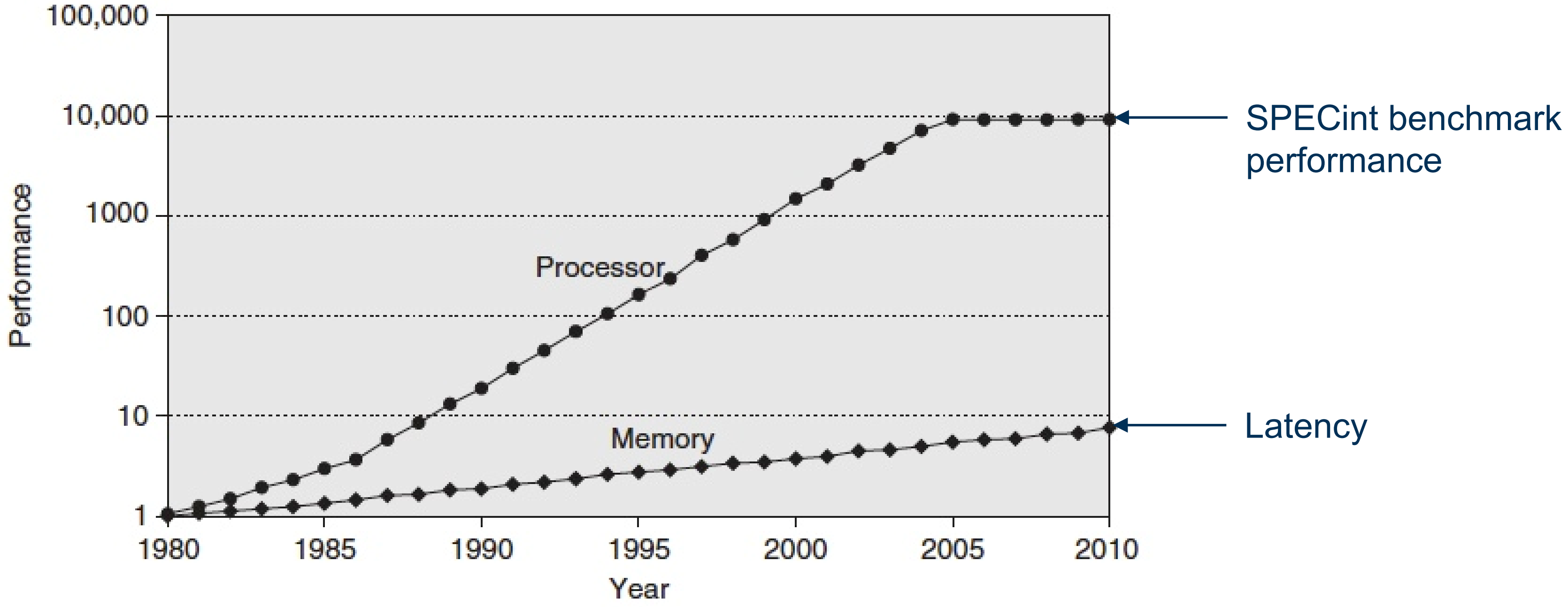
Dr. Christian Terboven

False Sharing


Hardware-specific Concept
General OpenMP Concept

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

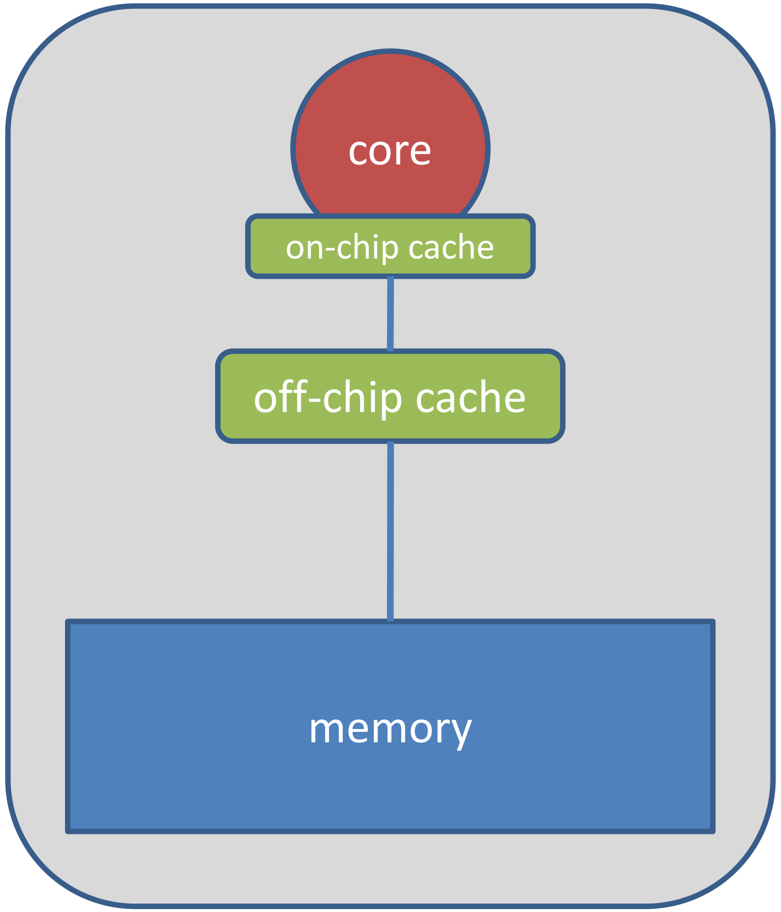


- Source: John L. Hennessy, Stanford University, and David A. Patterson, University of California, September 25, 2012

Caches

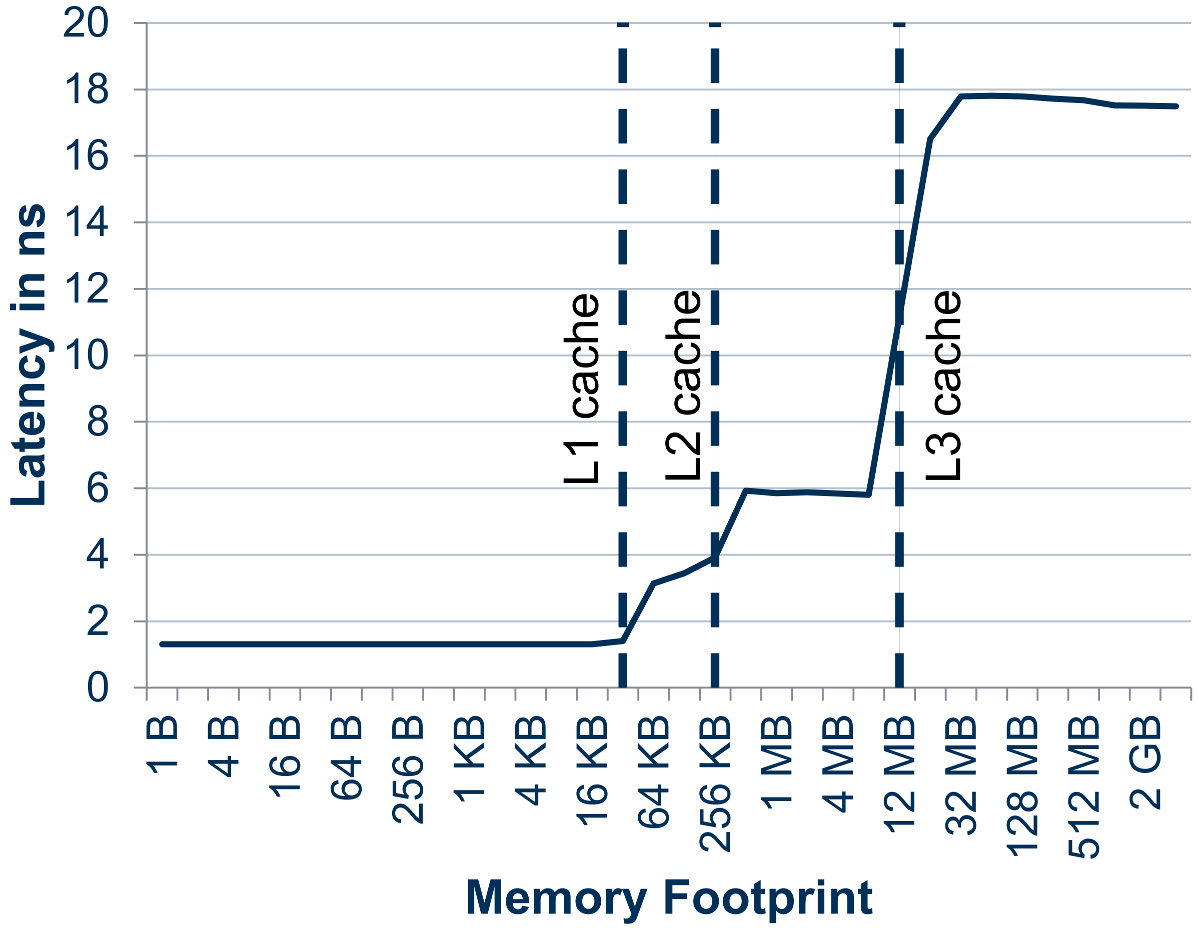
HPC.NRW

- 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!



Visualization of the Memory Hierarchy

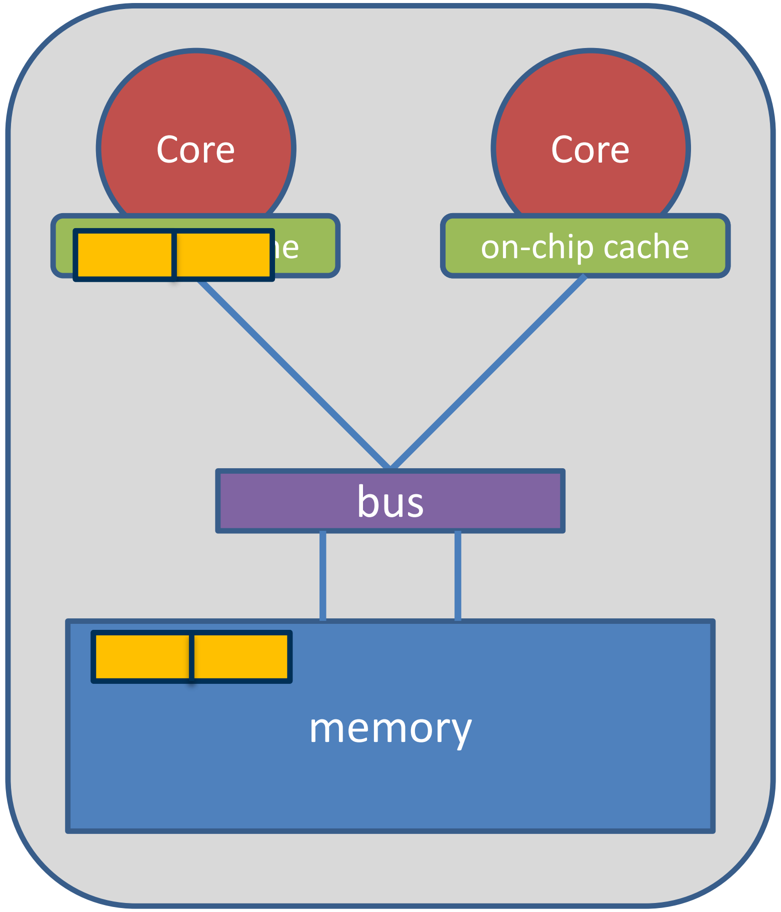
– Latency on the Intel Westmere-EP 3.06 GHz processor



Data in Caches

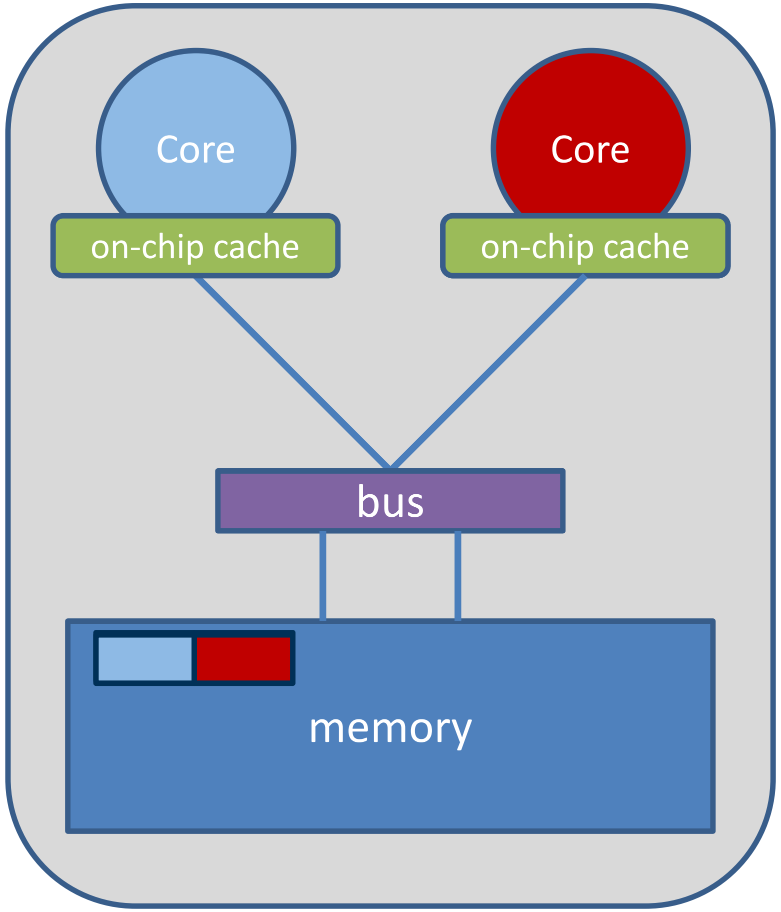
HPC.NRW

- 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, although there is no real data dependency
- Note: False Sharing is a performance problem, not a correctness issue



Summing up vector elements again

```c
#pragma omp parallel
{


    #pragma omp for
    for (i = 0; i < 99; i++)
    {


        s  = s + a[i];


    }

} // end parallel
```

```
do i = 0, 99
    s = s + a(i)
end do
```

→

```
do i = 0, 24
    s = s + a(i)
end do
```

```
do i = 25, 49
    s = s + a(i)
end do
```

```
do i = 50, 74
    s = s + a(i)
end do
```

```
do i = 75, 99
    s = s + a(i)
end do
```

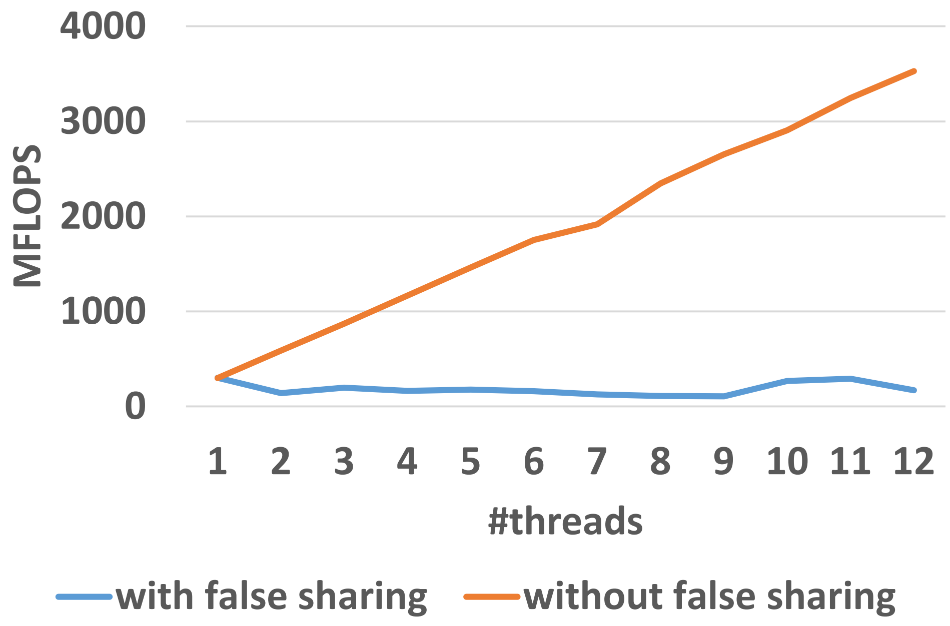
False Sharing

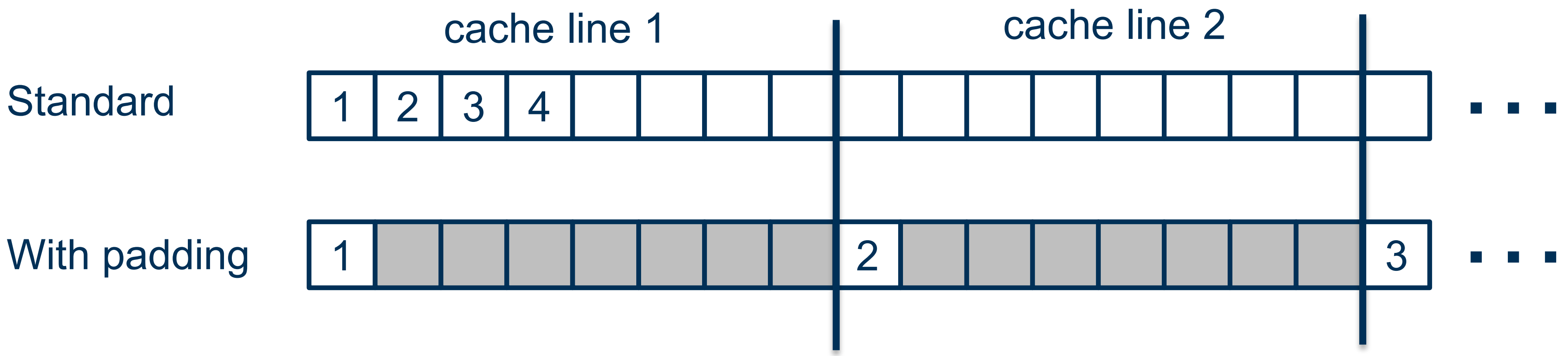
HPC.NRW

```
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];
}
```

False Sharing

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

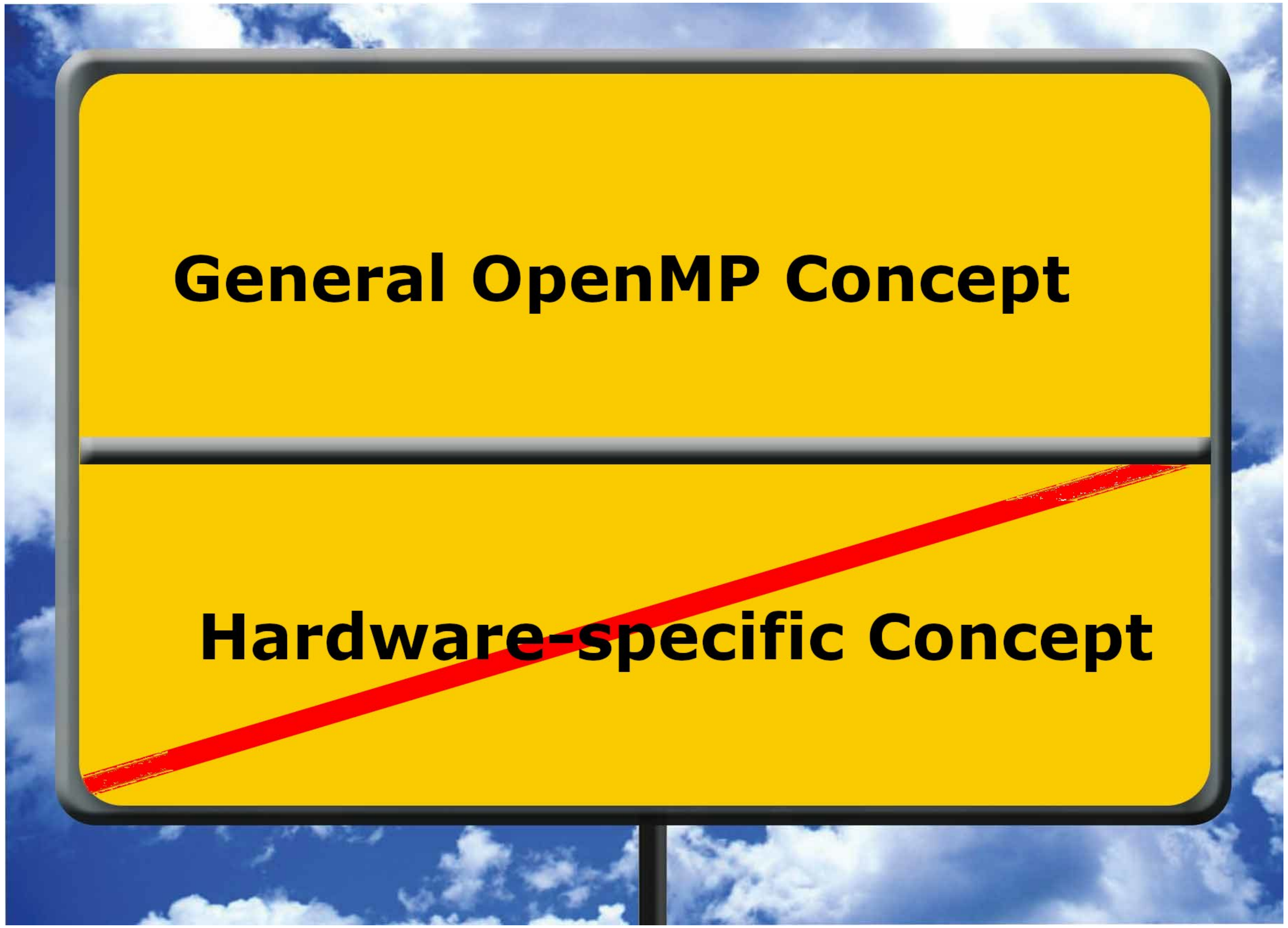




False Sharing avoided

HPC.NRW

```
double s_priv[nthreads * 8];
#pragma omp parallel num_threads(nthreads)
{
    int t=omp_get_thread_num();
    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t * 8]  += a[i];
    }
} // end parallel
for (i = 0; i < nthreads; i++)
{
    s += s_priv[i * 8];
}
```


General OpenMP Concept
~~Hardware-specific Concept~~

Introduction to OpenMP

Dr. Christian Terboven

Example: PI

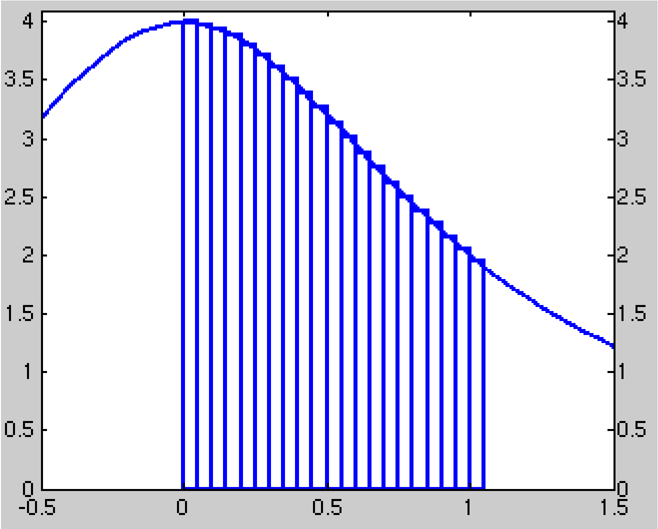
Example: Pi

```c
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Example: Pi

HPC.NRW

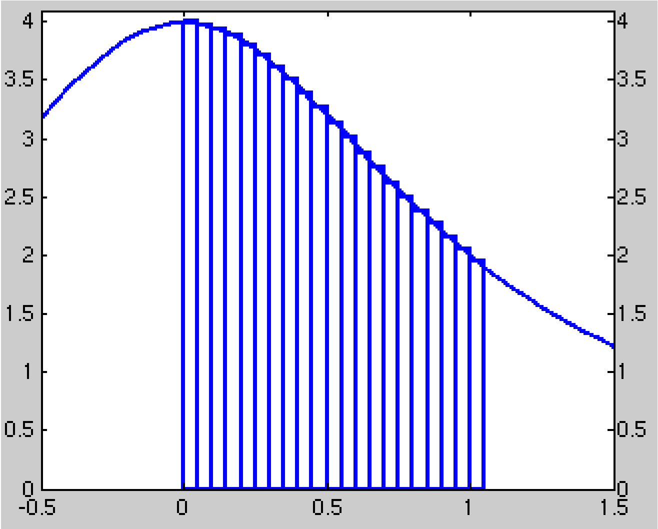
```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

What is wrong with this code?

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Example: Pi

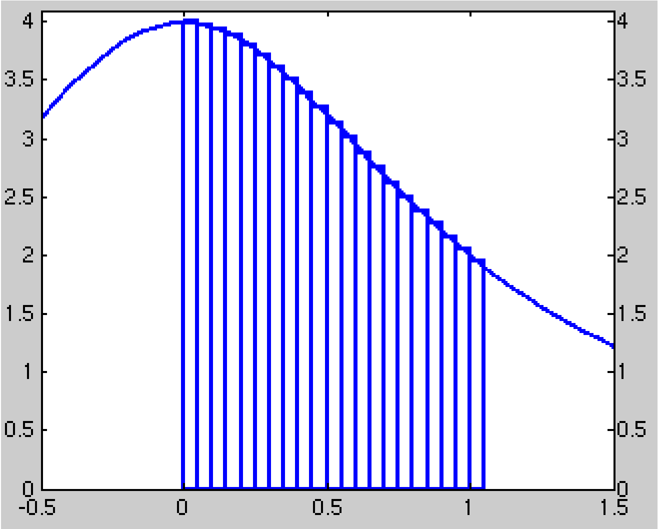
HPC.NRW

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Example: Pi

HPC.NRW

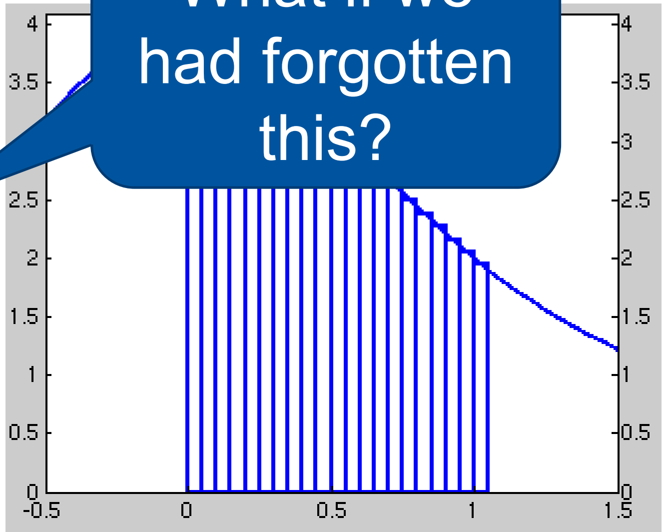
```c
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

What if we had forgotten this?



Race Condition

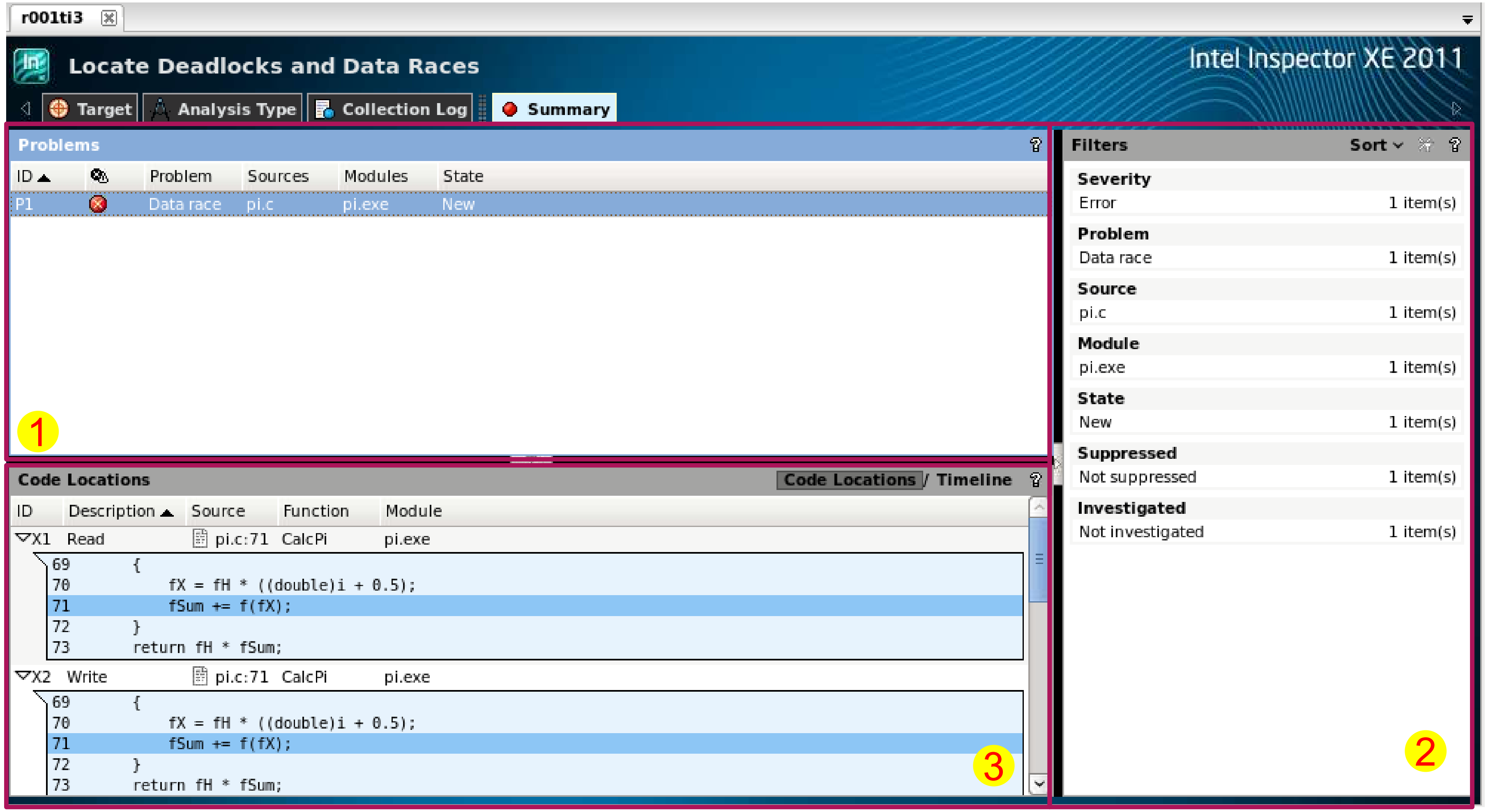
HPC.NRW

- 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 tools like *Intel Inspector XE, ThreadSanitizer, Archer*

Inspector XE – Results

HPC.NRW

1 detected problems
2 filters
3 code location



The missing reduction is detected.

Example: Pi

HPC.NRW

```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

#pragma omp parallel for private(fX,i,fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

What if we just made the fSum variable private?

fSum == 0 (no update to global variable)

Example: Pi Scalability

– Results for $n = 2 \cdot 10^9$:

# Threads	Runtime [sec.]	Speedup
1	1.141	1.00
2	0.575	1.96
4	0.298	3.93
8	0.161	7.08

System: CLAIX2018 Node (Intel Xeon 8160)
Compiler: Intel Compiler 19.0, Flags: –fopenmp –O3

– Scalability is good (for up to 8 threads):
  – About 100% of the runtime has been parallelized.
  – As there is just one parallel region, there is virtually no overhead introduced by the parallelization.
  – Problem is parallelizable in a trivial fashion ...

Questions?