Prof. Dr. D. Kranzlmüller, Dr. K. Fürlinger

Parallel Computing

WS 2017/18

Session 10: OpenMP Tasks, Tracing, Scalability

Tobias Fuchs, M.Sc.

tobias.fuchs@nm.ifi.lmu.de


TOUCH & CACHE LINE:
FOOL US

Some irritating code

→ http://hpc.wiki/lab/session-10/omp-eztrace

Bringing it all together

Revisiting Profiling, Tracing, Scalability and Hardware Models

Profiling

- **Aggregates performance events** and timings for the execution as a whole.
- **No chronology** of the events (no timestamps).
- Some profilers record relative order of events.

Examples: IPM, ompP

Example of profiling output of IPM

call	orank	ncalls	buf_size	t_tot	t_min	t_max	%comm
MPI_Recv	2	17	131072	5.96e+00	6.43e-02	5.92e-01	75.5
MPI_Recv	7	18	4	1.82e+00	8.45e-06	4.17e-01	23.0
MPI_Barrier	*	2	*	1.04e-01	7.14e-05	1.03e-01	1.3
MPI_Sendrecv	8	18	504	4.56e-03	6.31e-05	3.19e-04	0.1
MPI_Send	1	36	4	1.84e-03	1.75e-05	1.62e-04	0.0
MPI_Sendrecv	16	18	504	1.55e-03	3.18e-05	2.80e-04	0.0

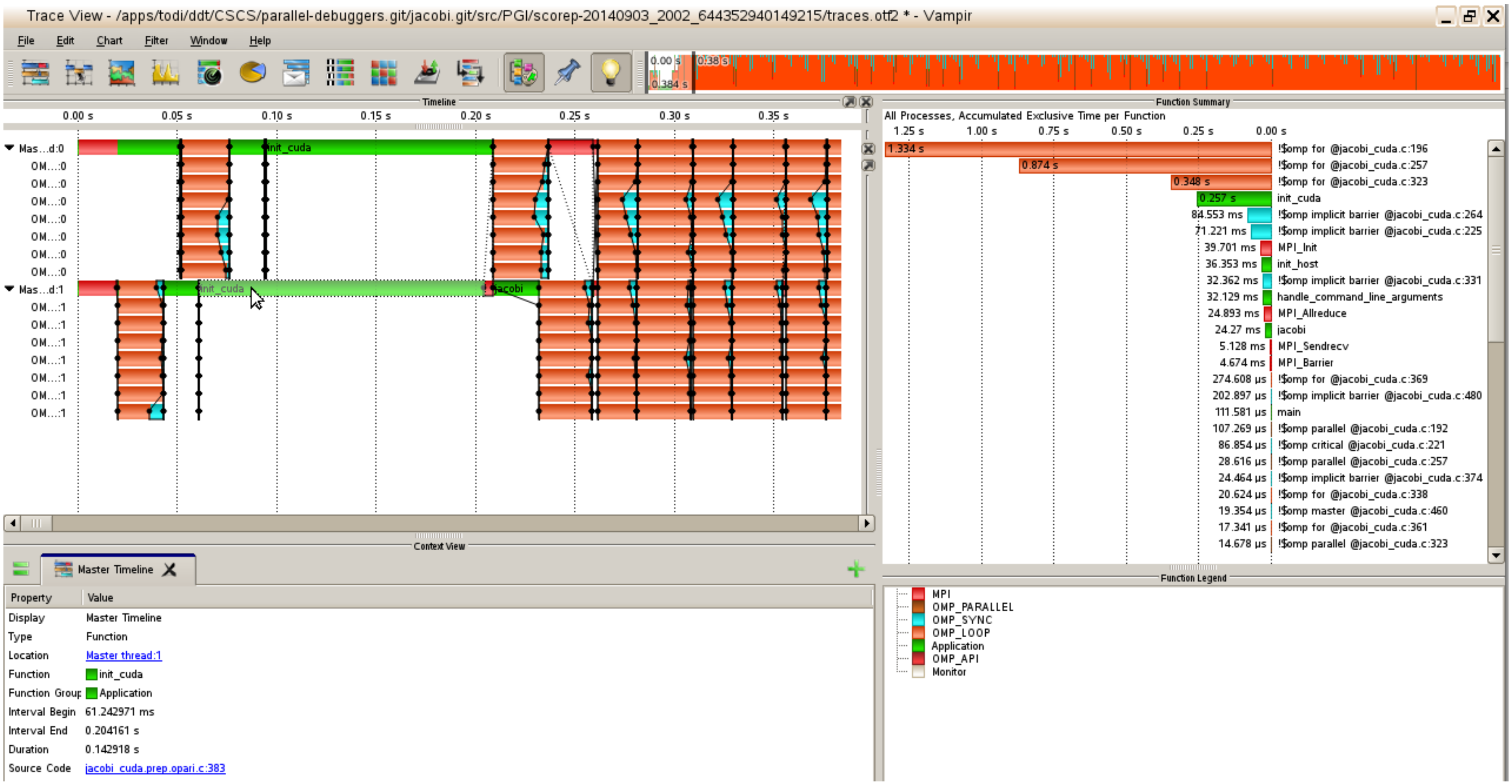
Here: 23% of communication time spent in 4 byte MPI_Recv.

Tracing

- Records the **chronology**, often with timestamps.
- **Extensive in time**: code is instrumented so all events are enclosed between timestamps.
- **Extensive in data**: amount of data in trace increases with runtime. Typically, trace data is periodically written to disk or network.

Example: Score-P (tracing) + Vampir (trace data viewer)

Profiling vs. Tracing



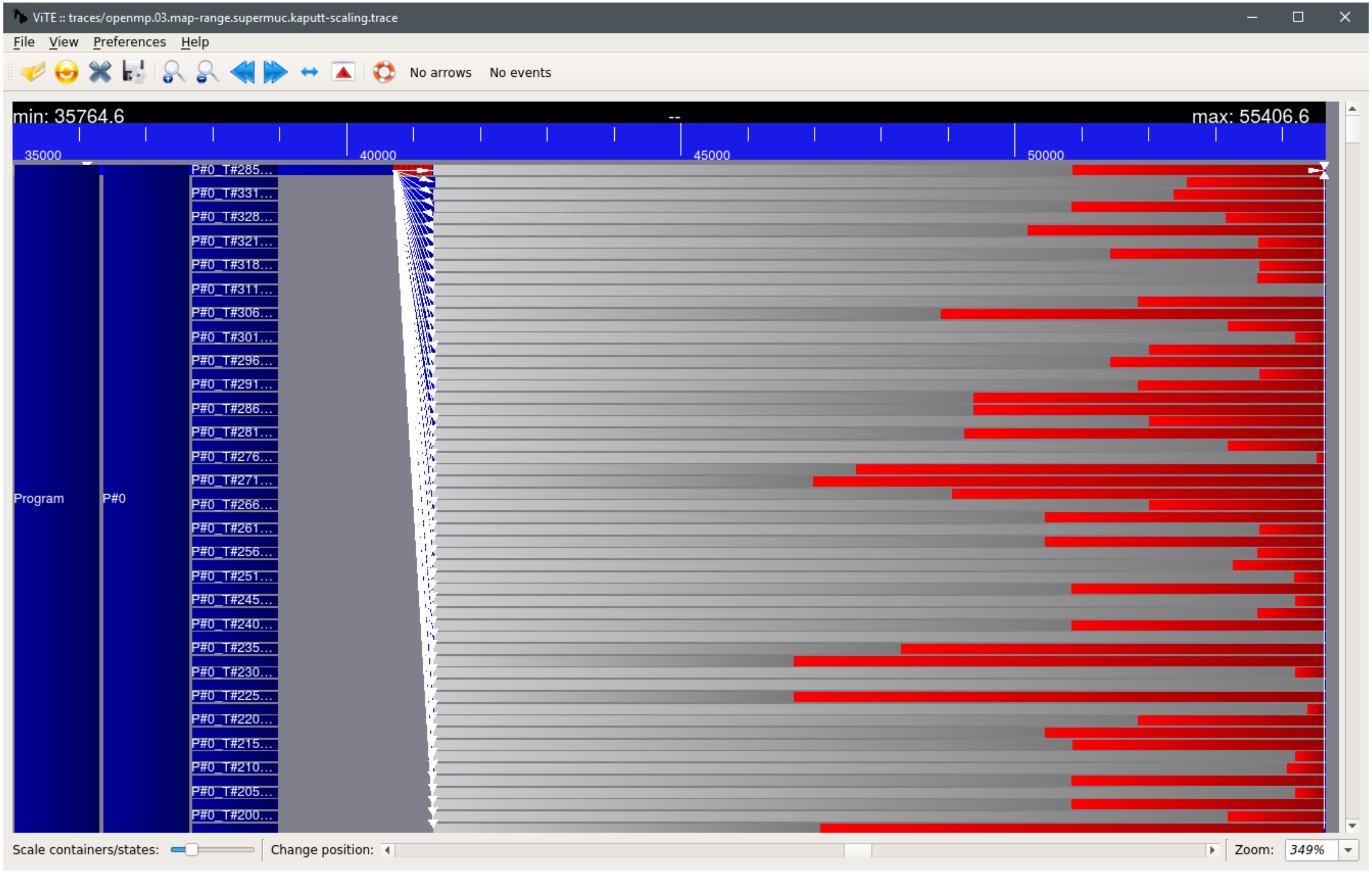
Free Open Source toolkit: eztrace

- Lightweight, portable and reasonably feature-rich
  (also an irritating sensation in the derrière to configure and build,
  a tad annoying to use and, of course, ugly as ten Scotsmen, probably
  just to remind you that it's free)
- Supports OpenMP, MPI, pthreads, CUDA, …
- Plug-and-play build for SuperMuc provided for this course at
  http://hpc.wiki/lab/session-10/omp-eztrace/eztrace-supermuc-homeinst.tgz
  for your convenience (just unpack the tarball to ~/opt/eztrace)
- How-to and sources from this session:
  http://hpc.wiki/lab/session-10/

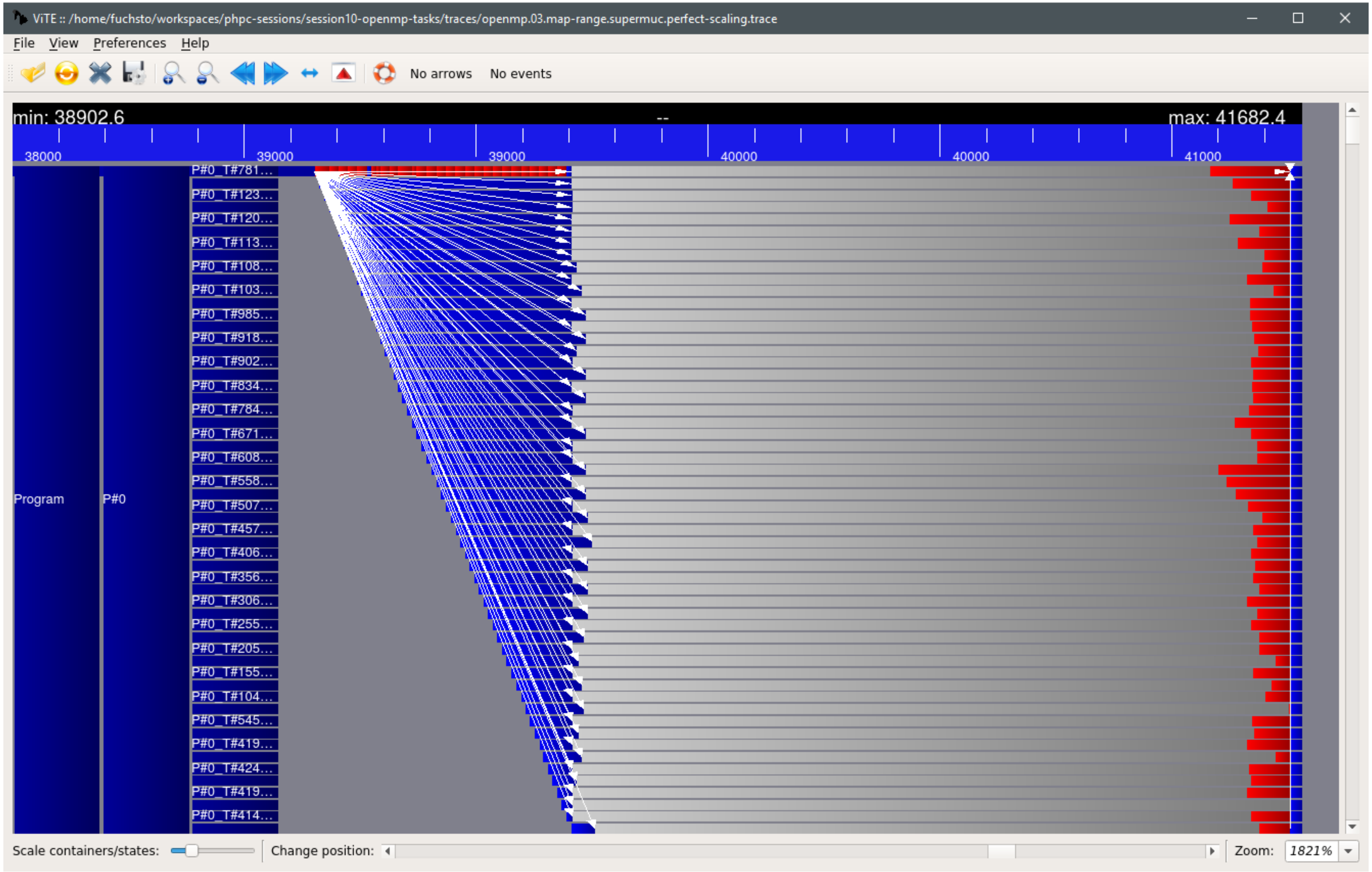
Example: openmp.03.map-ranges

... now let‘s demystify the code examples we saw earlier
(http://hpc.wiki/lab/session-10/omp-eztrace)

Example: openmp.03.map-ranges



Example: openmp.03.map-ranges



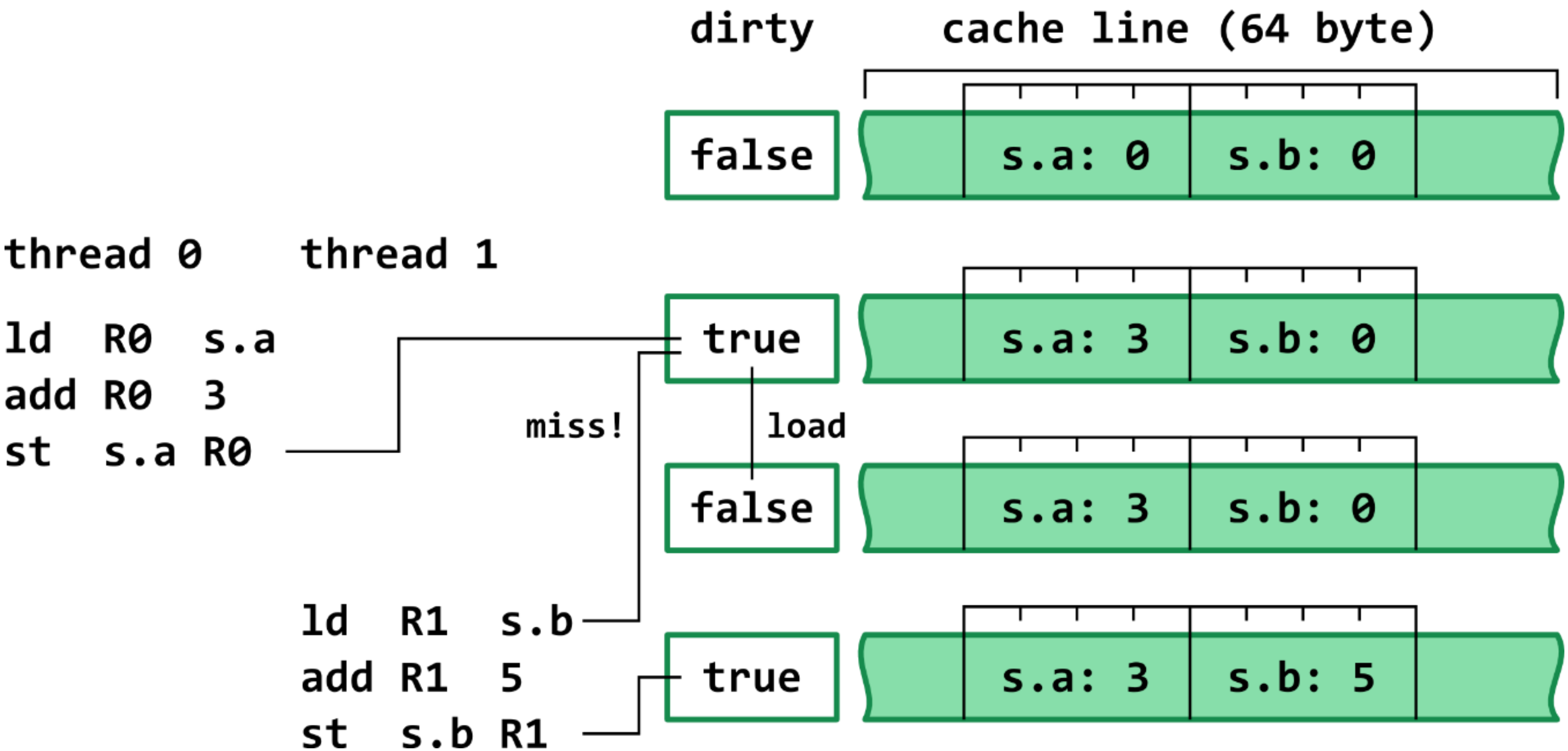
False Sharing

- Two or more threads write to seemingly unrelated variables:

```
shared: struct {        |   thread 0:      |   thread 1:
    int a;              |   s.a += 3;      |   s.b += 5;
    int b;              |                  |
} s;                    |                  |
```

- Cache-miss in both threads:
  s.a and s.b are in same cache line, += is read and write.
- One of the most important usage patterns to spot.

False Sharing



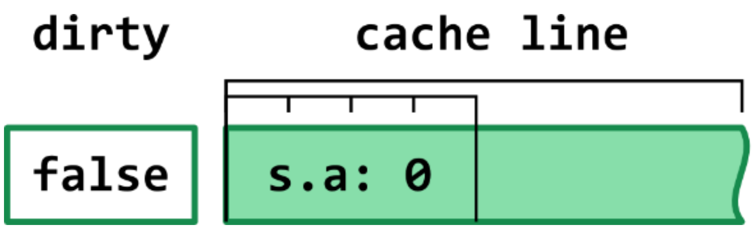
False Sharing

- Solution: alignment

```
shared: struct {
    aligned(c) int a;
    aligned(c) int b;
} s;
```

- Aligned variables (here: s.a, s.b) will be allocated and aligned at least on a c-byte boundary.
- If c = width of a cache line (typically 64 B): no false sharing.

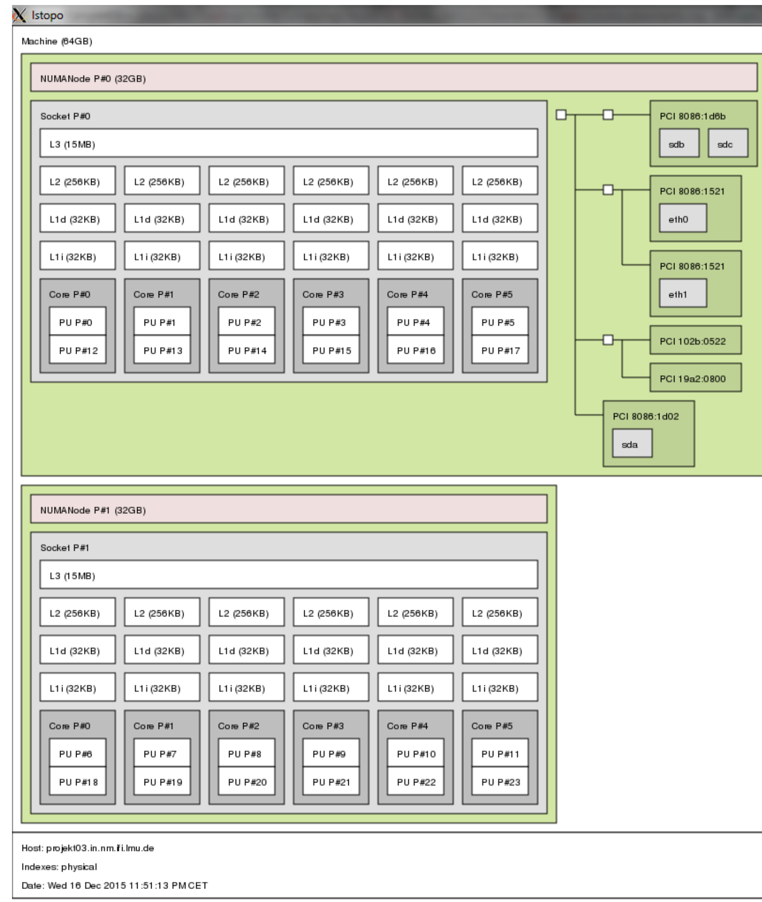
False Sharing



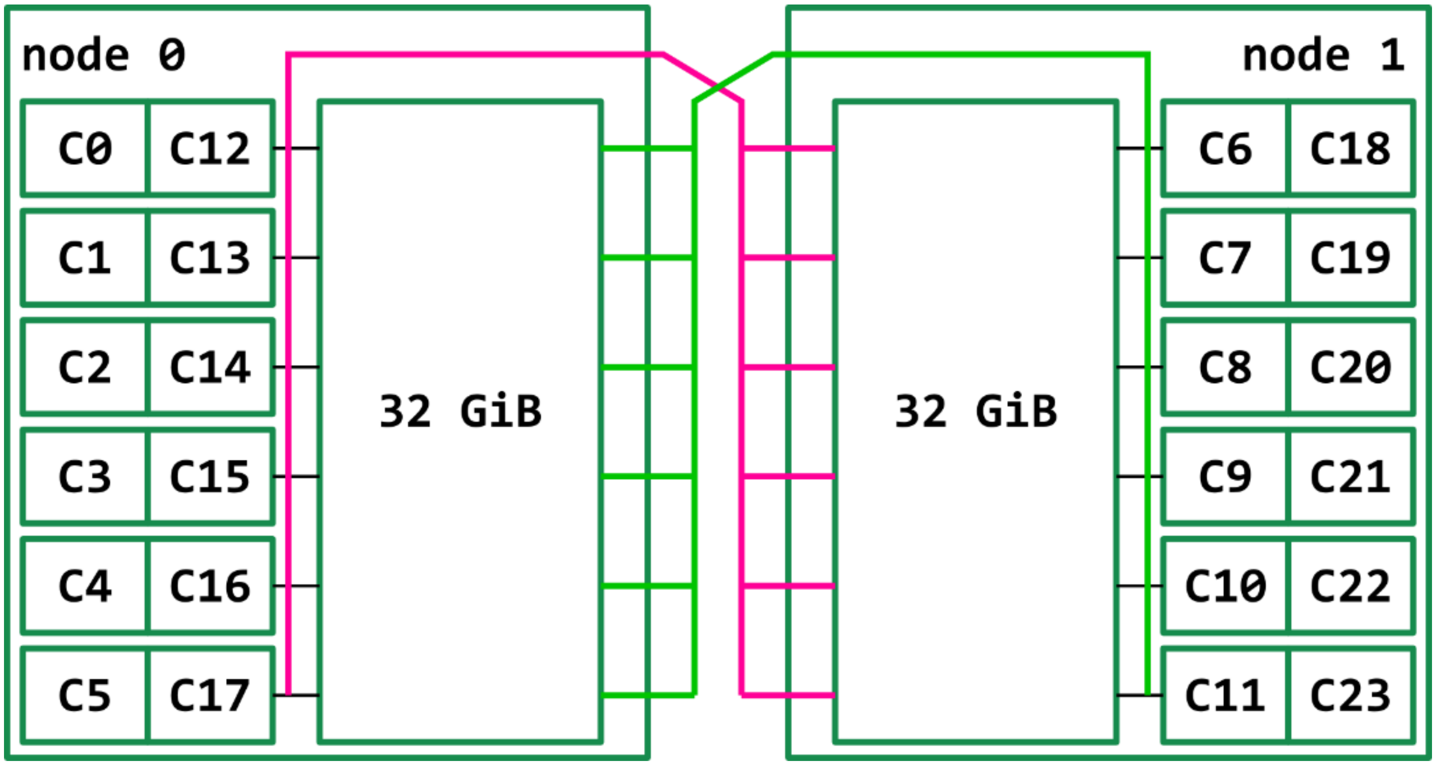
NUMA and numactl

NUMA Nodes

Let’s analyze this output from lstopo on a NUMA system …



Task- and Data Placement



numactl

What‘s the advantage of using

numactl --localalloc --physcpubind=0-11

compared to not specifying allocation policy / affinity?

numactl

What‘s the advantage of using

numactl --localalloc --physcpubind=0-11

compared to not specifying allocation policy / affinity?

→ Data placement:
Nodes may only allocate in local memory (only 32 GiB available on each node)

→ Task placement:
Hyperthreading cores disabled, every thread on dedicated physical core

numactl on SuperMUC Ph.2


```
SUPERMUC login22 $ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 28 29 30 31 32 33 34
--<SNIP>--
node distances:
node   0   1   2   3
  0:  10  21  31  31
  1:  21  10  31  31
  2:  31  31  10  21
  3:  31  31  21  10
```

This is obtained from **SLIT (System Locality Information Table)**

Three locality domains, here indicated by distances 10, 21, 31

Course Assignment

- Redesign partial sum algorithm variants
  (code: http://hpc.wiki/lab/session-09/)
  using OpenMP tasking
- Measure weak- and strong scaling as before
  (benchmark boilerplate available in code section)
- Visualize task dependencies using eztrace (or any other tracing tools, Vampir is available on SuperMUC)
- Use traces to explain scaling

Scalability

Example:
OpenMP Matrix Multiplication

Matrix Product with OpenMP


```c
void mmult_naive_par(double A[M][N], double B[N][K], double C[M][K]) {
    int    i, j, k;
    double sum;
    #pragma omp parallel for private(j,k,sum)
    for (i = 0; i < M; i++) {
        for (j = 0; j < K; j++) {
            sum = 0.0;
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

These are actual measurements submitted for a highly optimized implementation of mmult.

Spot the effects of

- Amdahl’s Law
- Gunther’s Law
- Gustafson’s Law

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Amdahl's Law

Performance increase (speedup) is **limited by sequential sections and degree of parallelism**.

Note that super-linear speedup is rare but possible!

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gunther’s Law

Performance may **worsen** if degree of parallelism is increased because contention rate increases.

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gustafson's Law

Higher degree of parallelism can yield performance benefit **when problem size is increased**. For example:

N = 1000 → GLFOPS drop from 8 to 16 threads.

N = 2000 → GLFOPS saturated from 8 to 16 threads.

N = 4000 → GFLOPS increased from 8 to 16 threads.

Matrix Product with OpenMP

N/T	1	2	4	8	16	32
100	0.89	0.78	0.82	0.50	0.12	0.11
1000	14.12	19.90	32.91	32.32	26.22	21.66
2000	14.30	28.74	42.63	64.76	65.04	35.31
4000	14.68	29.09	57.93	82.64	156.84	138.94

Spot the effects of

- Gunther's Law + Gustafson's Law

Speedup saturation as predicted by Amdahl's and Gustafson's models **shifted** to higher degrees of parallelism **with increasing problem size**.

N = 1000 → Saturates with 4 threads

N = 2000 → Saturates with 8 threads

N = 4000 → Saturates with 16 threads

Vector Triad


```
void vector_blyad(double A[N],
                  double B[N],
                  double C[N],
                  double D[N]) {
    for (j=0; j < REPEAT; j++) {
      for (i=0; i < N; i++) {
        A[i] = B[i] + C[i] - D[i];
      }
    }
}
```

We fail to achieve peak performance because we have the wrong problem. Bad luck.

Vector Triad

```c
void vector_triad(double A[N],
                  double B[N],
                  double C[N],
                  double D[N]) {
    for (j=0; j < REPEAT; j++) {
      for (i=0; i < N; i++) {
        A[i] = B[i] + C[i] * D[i];
      }
    }
}
```

Vector Triad is a common benchmark, uses floating point addition and -multiplication as otherwise FLOPS peak performance could not be reached ("fused multiply-add", I prefer the term "multiply-accumulate")

Read this

Intel: *"Optimizing Applications for NUMA"*
https://software.intel.com/en-us/articles/optimizing-applications-for-numa

"What Every Programmer Should Know About Memory". Drepper, Ulrich. November 2007.

"Local and Remote Memory: Memory in a Linux/NUMA System". Lameter, Christoph. June 2006.

Tobias Fuchs

tobias.fuchs@nm.ifi.lmu.de

www.mnm-team.org/~fuchst