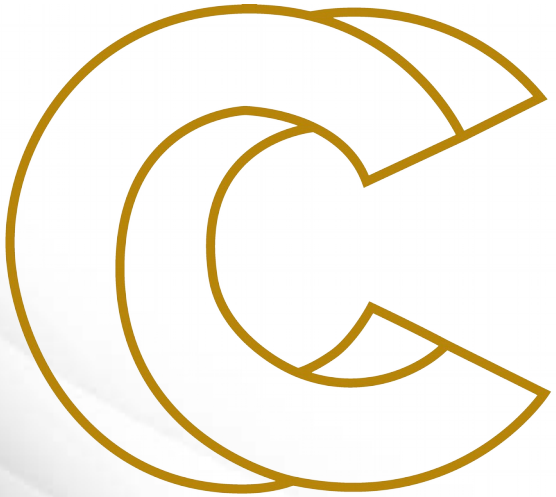
OpenMP and MPI applications

Profiling, performance evaluation and optimization



EURO

Leon Kos

University of Ljubljana, FME, LECAD lab

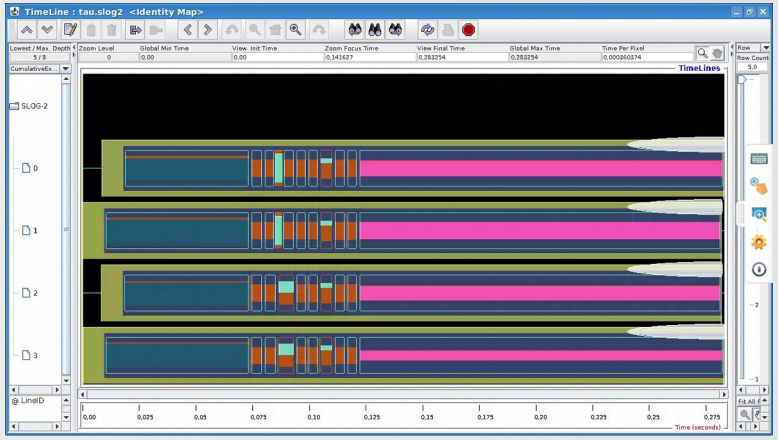
Methods for performance evaluation

- **Profiling**: gives information about time spent in each part or routine of the program

Metric: TIME
Value: Exclusive
Units: seconds

0,115	main [{/home/bogdanl/PTC_OpenMP_prof/examples/openmpi/simple-work/simple.c} {15,0}]
0,058	MPI_Finalize()
0,03	MPI_Init()
0,02	foo [{/home/bogdanl/PTC_OpenMP_prof/examples/openmpi/simple-work/simple.c} {8,0}] [THROTTLED]
0,01	.TAU application
2,0E-6	MPI_Get_processor_name()
0	MPI_Comm_size()
0	MPI_Comm_rank()

- **Tracing**: gives information about when events take place on a timeline



TAU performance system

- **TAU** (Tuning and Analysis Utilities):

https://www.cs.uoregon.edu/research/tau/home.php

- **scalable and flexible** performance analysis toolkit
- comprehensive **performance profiling and tracing**
- supports **all parallel programming/execution paradigms**:

  MPI, MPI-IO, OpenMP CPU, OpenMP GPU, OpenACC, CUDA, OpenCL etc.

TAU usage

- **Instrumentation**:
  - adding probes to perform measurements
  - source code instrumentation
  - wrapping external libraries (I/O, CUDA, OpenACC, OpenCL)
  - rewriting the binary executable
- **Measurement**:
  - profiling or tracing
  - direct instrumentation
  - sampling
  - throttling
- **Analysis:**
  - visualization of profiles and traces
  - 3D visualization
  - trace conversion tools

TAU Instrumentation

3 methods for **tracking the performance** of applications:

- **Dynamic instrumentation** through library pre-loading, e.g.:
  - `srun -n 4  tau_exec -T mpi ./a.out`
  - `mpirun -np 4 tau_exec -io ./a.out`
  - `tau_exec -io ./a.out`
- **Scripted compilation**, e.g.:
  - For C: compiler replaced with `tau_cc.sh`
  - For C++: compiler replaced with `tau_cxx.sh`
  - For Fortran: compiler replaced with `tau_f90.sh` or `tau_f77.sh`
- **Source transformation** using PDT (Program Database Toolkit): also needs to be installed

TAU Analysis

- TAU creates **one profile file per node** in a single location, **profile outputs** are named:

  `profile.0.0.0, profile.1.0.0, ...`
- **Text summary** of profile data with:

  `$ pprof`
- **2D and 3D visualization** of profile data with:

  `$ paraprof`
- TAU can also create **traces**:

  `tautrace.0.0.0.trc, tautrace.1.0.0.trc, ...`
- traces can be **converted and displayed** with external visualizers:

  `jumpshot` (included in TAU installation), `vampir` and others

TAU event types

TAU can measure several types of events:

- **Interval**: start-stop events (e.g.: function calls)
- **Atomic**: trigger at a single point with data (e.g.: memory allocation)
- **Context**: atomic events with executing context
- For atomic and context event types a **complete statistics of measurements** is given:

  total, samples, min/max/mean/standard deviation
- **inclusive vs. exclusive** measurements

Example 1: MPI profiling

Follow these steps on your `viz.hpc.fs.uni-lj.si` account:

- Load module and copy directory with examples:

```
$ module load tau
$ cp -r /home/leon//PTC_OpenMPI-MP_profiling /$HOME/PTC_OpenMPI-MP_profiling
```

- Go to work directory:

```
$ cd /$HOME/PTC_OpenMPI-MP_profiling/examples/openmpi/simple-work
```

- Compile the OpenMPI code (`simple.c`) with TAU script:

```
$ tau_cc.sh -tau_makefile=/opt/pkg/software/tau/2.29.1/x86_64/lib/Makefile.tau-mpi-openmp -tau_options=-optCompInst simple.c
```

- Generate profiles:

```
$ mpirun -np 4 tau_exec -io ./a.out
```

Example 1: MPI profiling (cont.)

- Use pprof **for text summary** of **profile** results:

```
$ pprof
```

Reading Profile files in profile.*

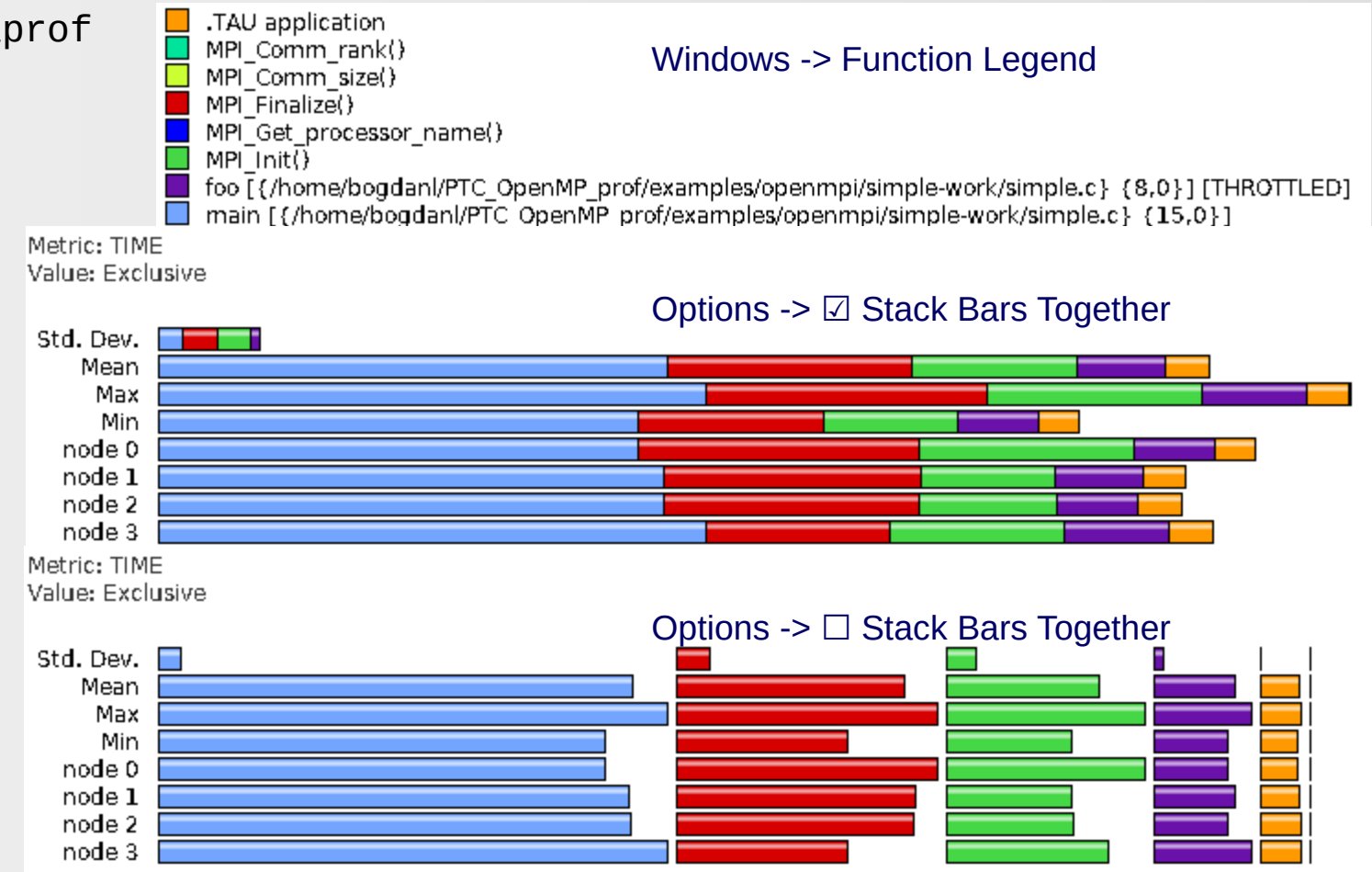
NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	9	248	1	1	248395	.TAU application
96.2	108	239	1	100006	239062	main
25.6	63	63	1	0	63669	MPI_Finalize()
19.5	48	48	1	0	48506	MPI_Init()
7.3	18	18	100001	0	0	foo [THROTTLED]
0.0	0.002	0.002	1	0	2	MPI_Get_processor_name()
0.0	0.001	0.001	1	0	1	MPI_Comm_size()
0.0	0	0	1	0	0	MPI_Comm_rank()

Example 1: MPI profiling (cont.)

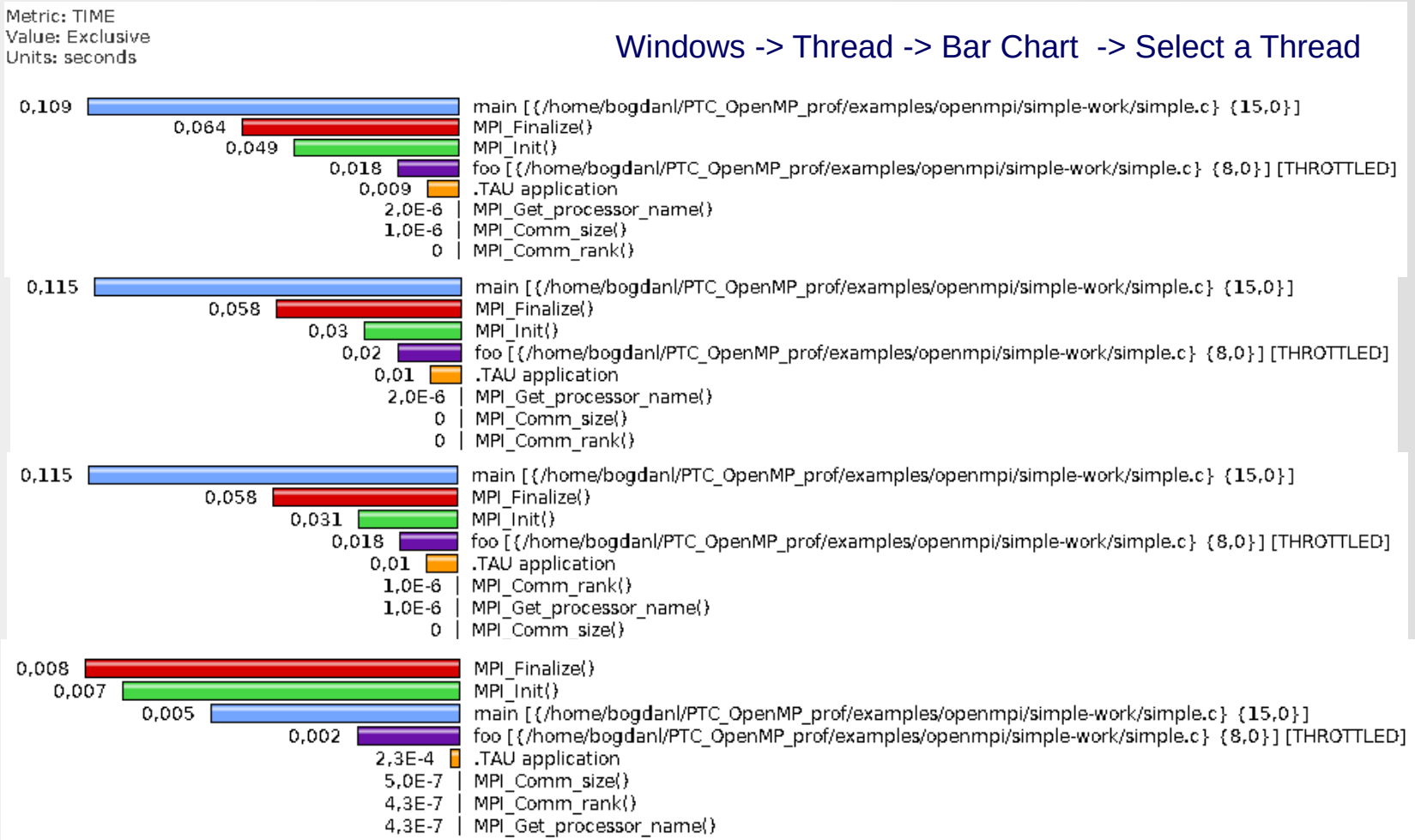
- Use paraprof for **visualizing profiles**:

```
$ paraprof
```



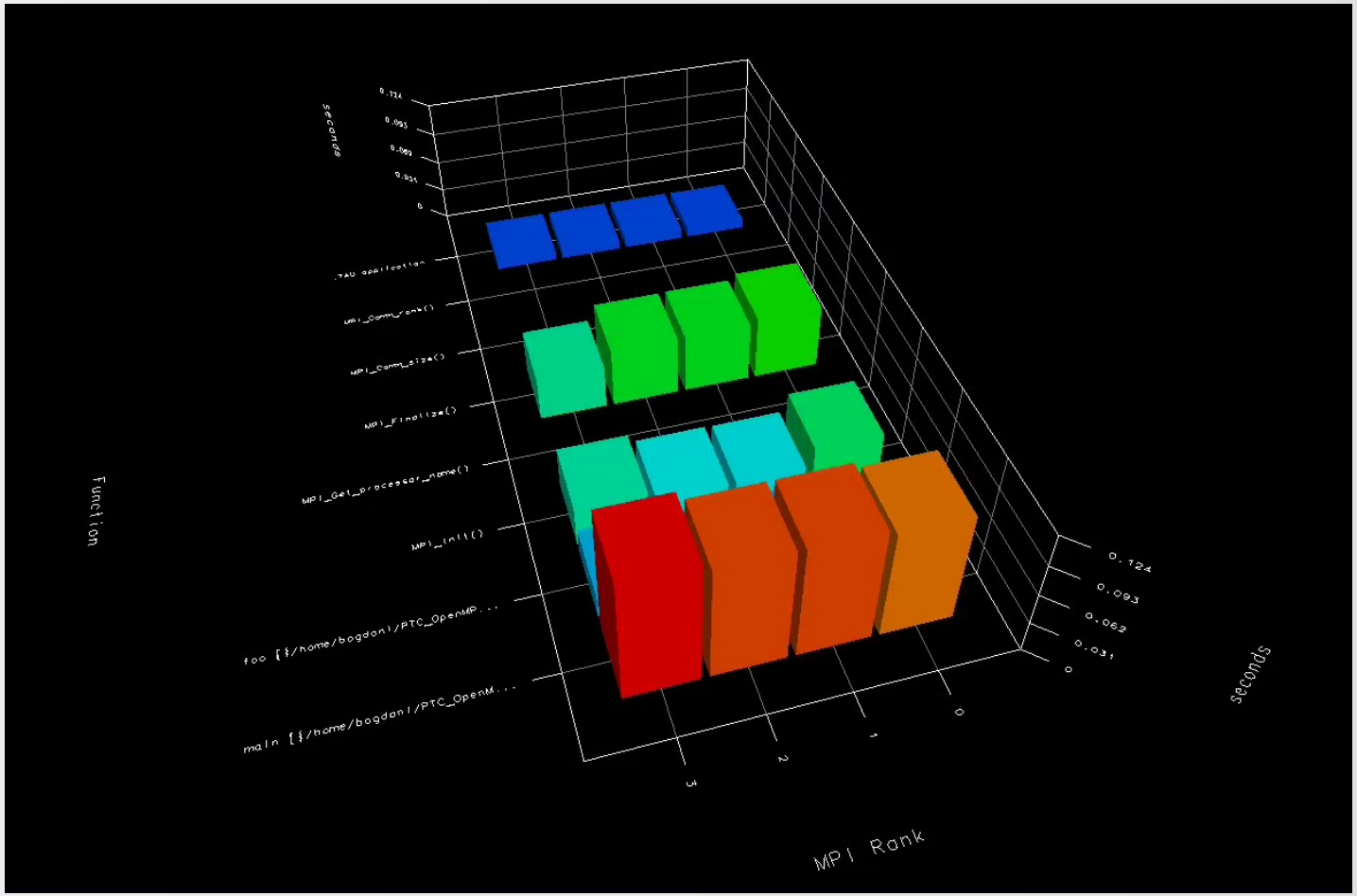
Example 1: MPI profiling (cont.)

Nodes (0, 1, 2 and 3) analysis:



Example 1: MPI profiling (cont.)

3D visualization of profiles in `paraprof`:



Example 2: Tracing MPI codes

Follow these steps on your `viz.hpc.fs.uni-lj.si` account:

- Go to work directory:

```
$ cd /$HOME/PTC_OpenMPI-MP_profiling/examples/openmpi/simple-work
```

- For the previous compiled example of the OpenMPI code (`simple.c`) generate traces with:

```
$ TAU_TRACE=1 mpirun -np 4 tau_exec -io ./a.out
```

- Merge traces and convert them to slog2 format:

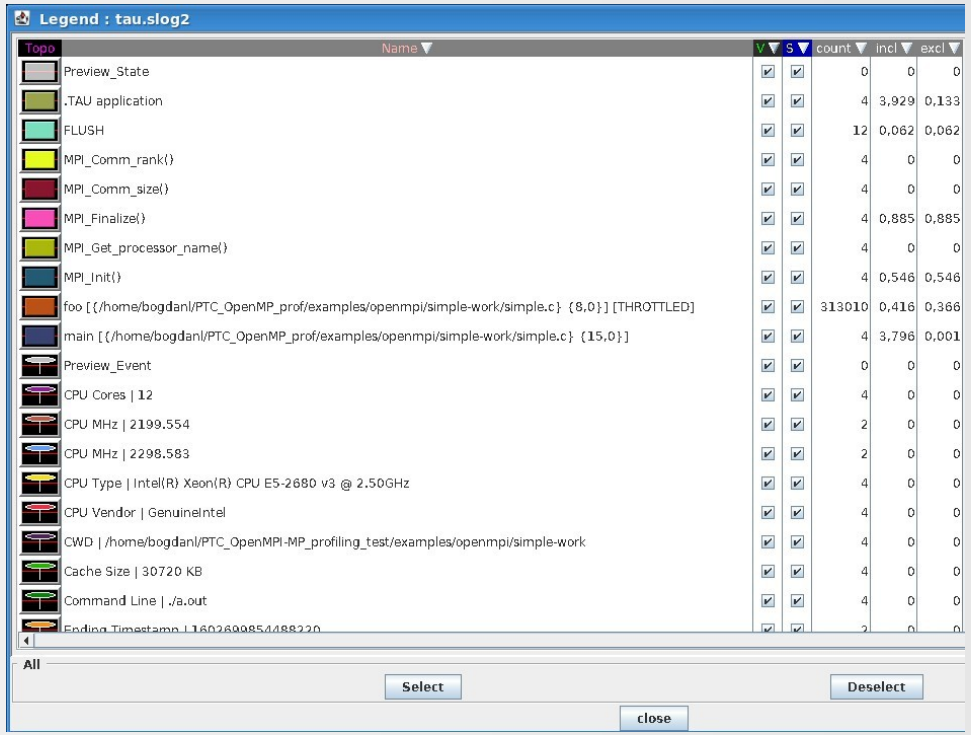
```
$ tau_treemerge.pl
$ tau2slog2 tau.trc tau.edf -o tau.slog2
```

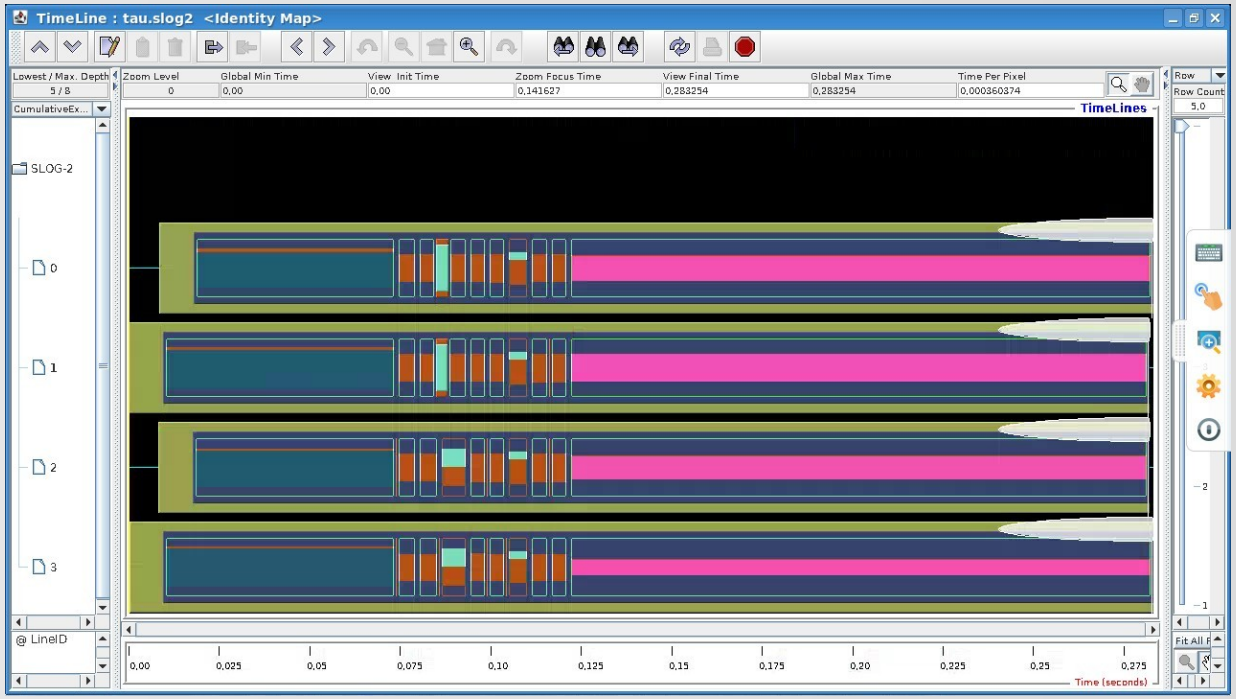
- Visualize traces:

```
$ jumpshot tau.slog2
```

Example 2: Tracing MPI codes (cont.)

Traces visualized with `jumpshot`





Example 3: Profiling OpenMP codes

Follow these steps on your `viz.hpc.fs.uni-lj.si` account:

- Go to work directory:

```
$ cd /$HOME/PTC_OpenMPI-MP_profiling/examples/openmp/pi_work
```

- Open `pi-openmp_profile.c` with text editor:

```
$ gedit pi-openmp_profile.c
```

Example 3: Profiling OpenMP codes (cont.)

Add TAU instrumentation code (in red) to the original code:

```
#include <stdio.h>
#include <math.h>
#include <Profile/Profiler.h>
extern void mytimer_(int *);

#define N 1000000

int main()
{
  int numpe;
  double area = 0.0;

  TAU_PROFILE_TIMER(mt, "main()", "int (int, char **)",
          TAU_DEFAULT);
  TAU_PROFILE_SET_NODE(0);
  TAU_PROFILE_START(mt);

  numpe = 1;
  mytimer_(0);

#pragma omp parallel for reduction(+:area)
  for(int i = 0; i < N; i++)
  {
    numpe = omp_get_num_threads();
    TAU_PROFILE_TIMER(fl, "For loop", " " , TAU_DEFAULT);
    TAU_PROFILE_START(fl);

    double x = (i+0.5)/N;
    area += sqrt(1.0 - x*x);

    TAU_PROFILE_STOP(fl);
  }

  mytimer_(&numpe);
  printf("Pi : %14lf\n", 4.0*area/N);
  TAU_PROFILE_STOP(mt);

  return 0;
}
```

Example 3: Profiling OpenMP codes (cont.)

- Build executable with TAU instrumentation and run it to get profiles:

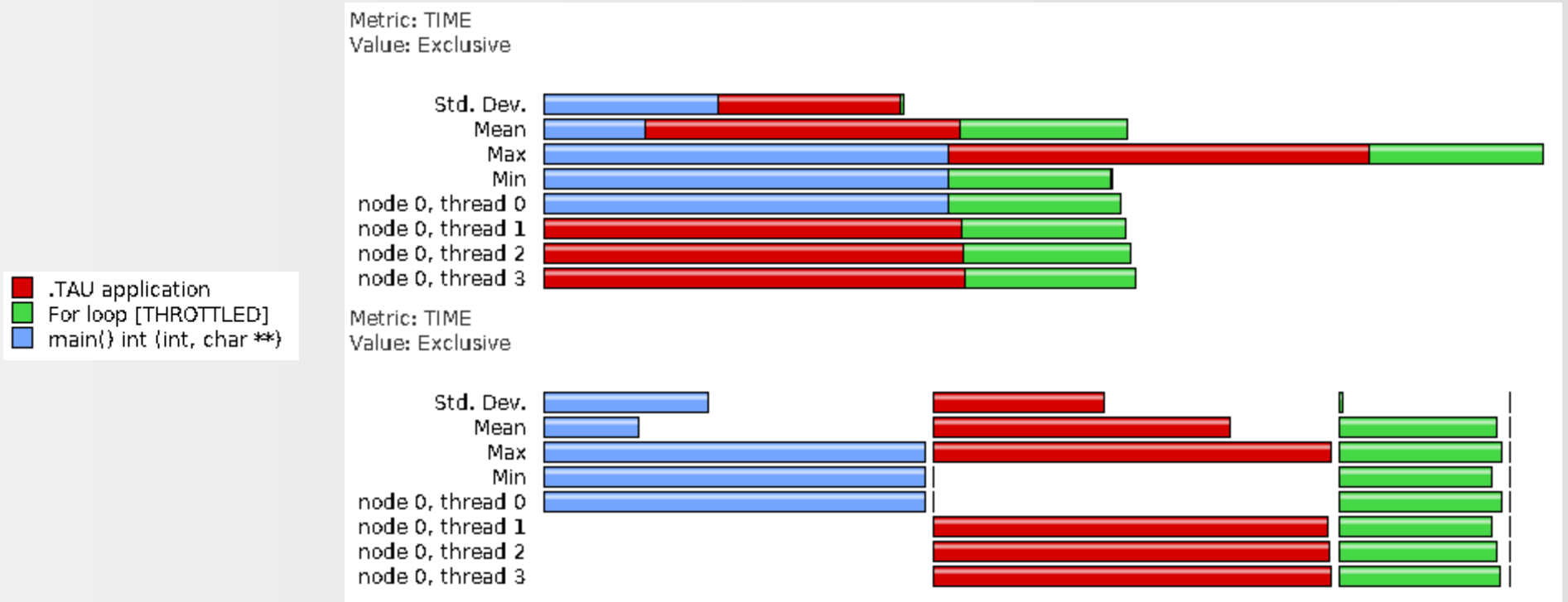
```
$ make
$ TAU_SET_NODE=0 OMP_NUM_THREADS=4 tau_exec -T openmp ./pi-openmp_profile
```

- Use `pprof` and `paraprof` to see the profiling results:

```
$ pprof
$ paraprof
```

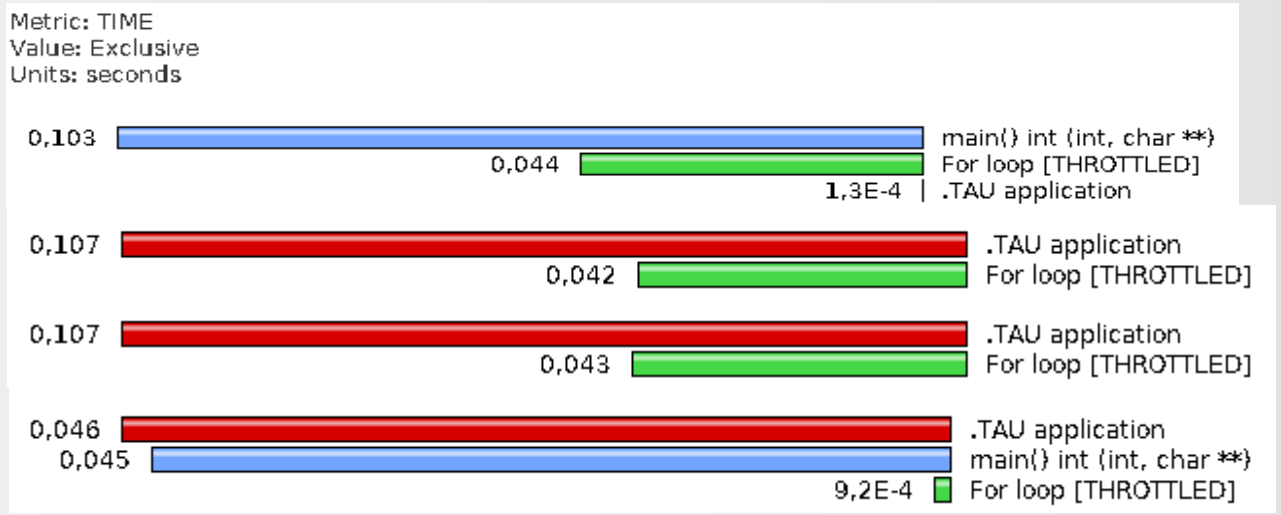
Example 3: Profiling OpenMP codes (cont.)

Profiling results with paraprof:

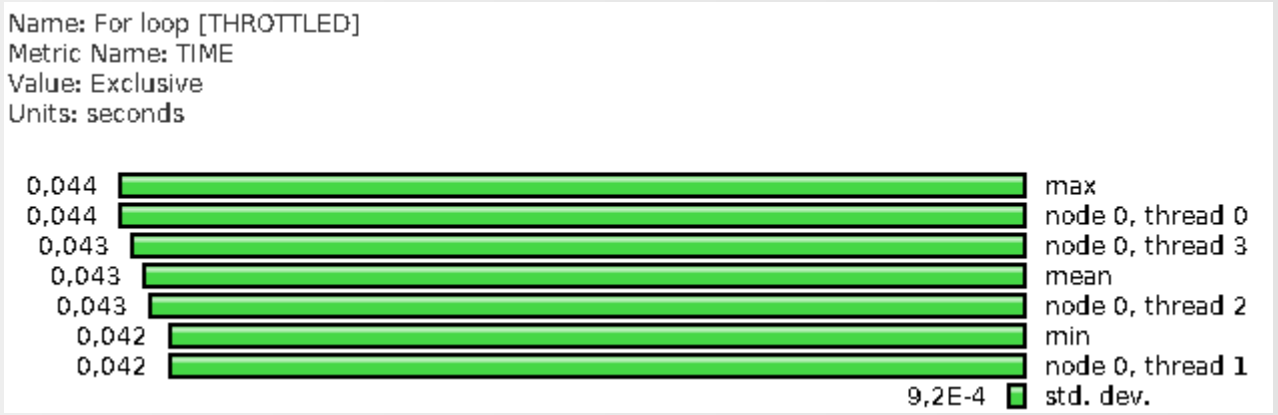


Example 3: Profiling OpenMP codes (cont.)

- Threads (0, 1, 2 and 3) analysis:



- Function (for loop) analysis:



Bonus example 1: CUDA profiling

Follow these steps on your `viz.hpc.fs.uni-lj.si` account:

- Load modules and start interactive session on GPU node:

```
$ module purge
$ module load tau/2.29.1-CUDA
$ module load jre
$ env --unset=LD_PRELOAD TMOUT=600 srun --time=1:0:0 --partition=gpu --x11 --
pty bash -i
```

- Go to work directory:

```
$ cd $HOME/PTC_OpenMPI-MP_profiling/examples/gpu/cuda/matmult
```

- Compile CUDA code (`matmult.cu`):

```
$ make
```

- Run profiling with TAU:

```
$ tau_exec -T serial -cupti ./matmult
```

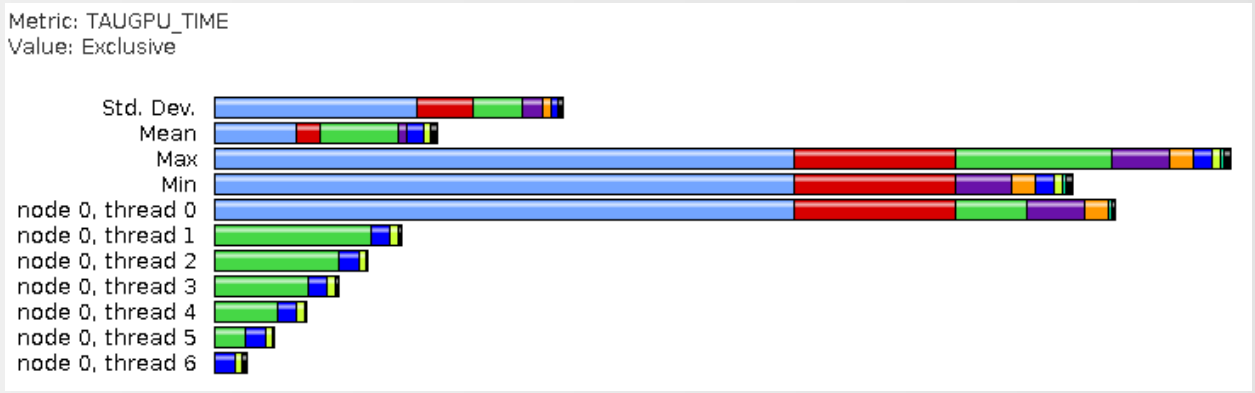
- Use pprof and paraprof to see the profiling results:

```
$ pprof
$ paraprof
```

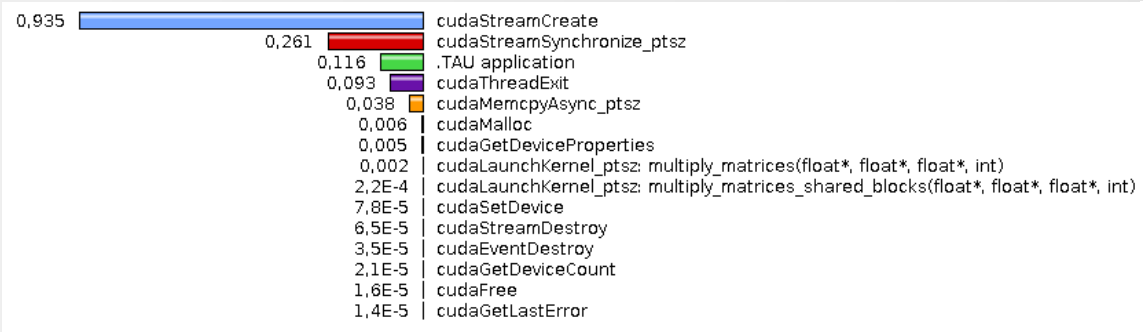
Bonus example 1: CUDA profiling (cont.)

Profiling results with `paraprof`:

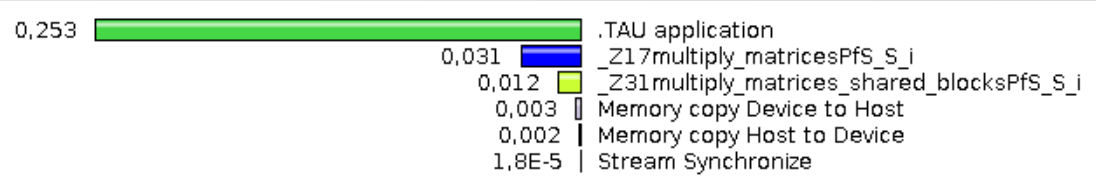
- Threads (1-6) represent GPUs in the node (6 in total)



- Thread 0 (CUDA API calls):



- Thread 1 (GPU 0 in the GPU node):



Bonus example 2: OpenCL profiling

Follow these steps on your `viz.hpc.fs.uni-lj.si` account:

- Go to work directory:

```
$ cd $HOME/PTC_OpenMPI-MP_profiling/examples/gpu/opencl
```

- Compile OpenCL code (`matmult.cpp`):

```
$ make
```

- Run profiling with TAU:

```
$ tau_exec -T serial -opencl ./matmult
```

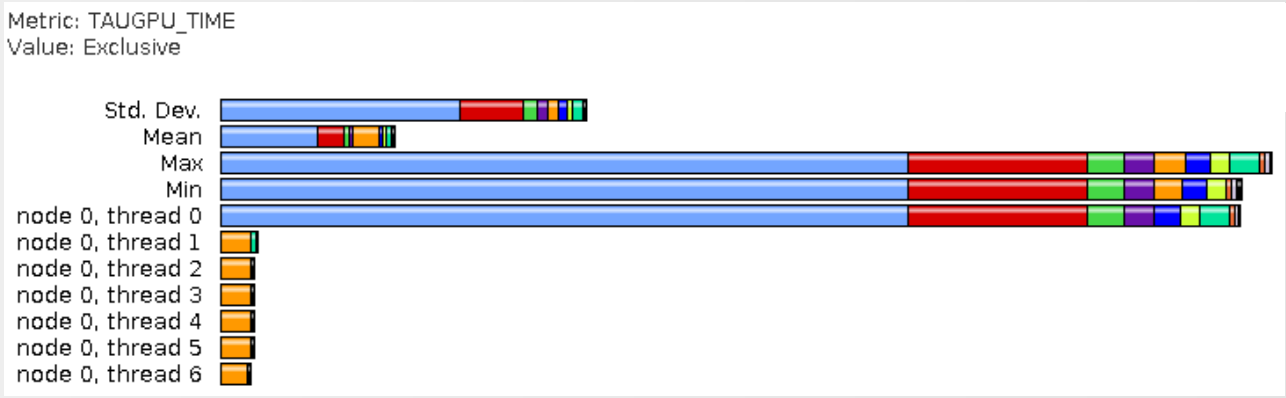
- Use pprof and paraprof to see the profiling results:

```
$ pprof
$ paraprof
```

Bonus example 2: OpenCL profiling (cont.)

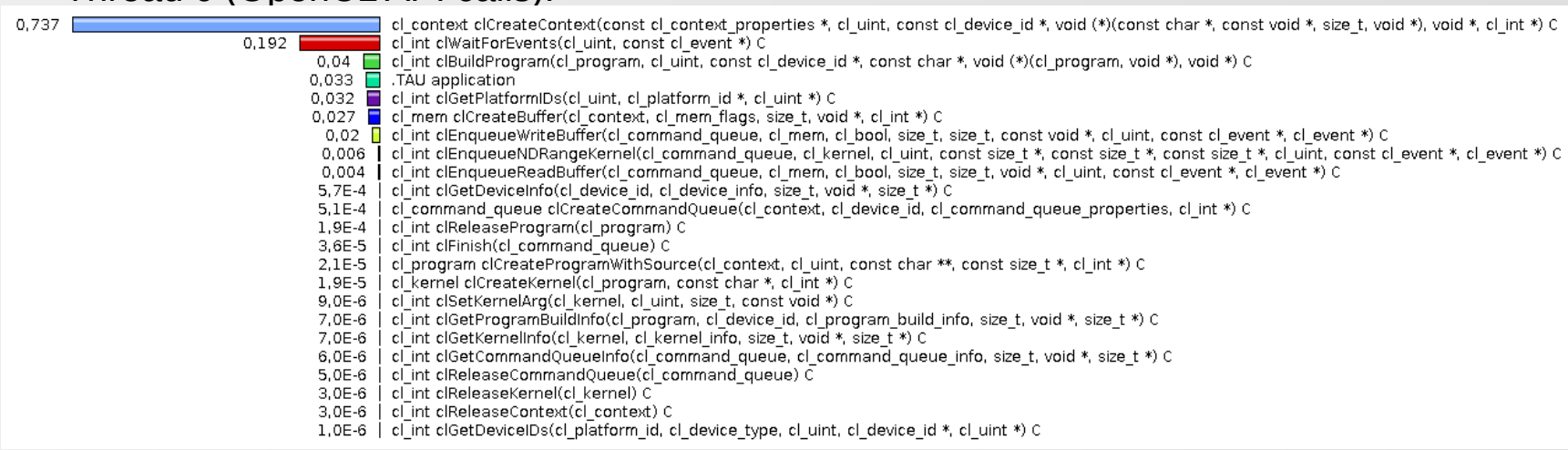
Profiling results with `paraprof`:

- Threads (1-6) represent GPUs in the node (6 in total)



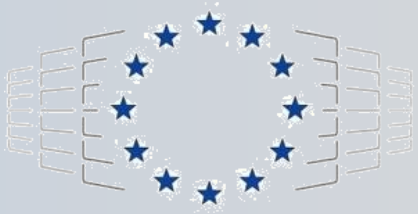
- Thread 1 (GPU 0 in the GPU node):

- Thread 0 (OpenCL API calls):



SLING EURO

Thanks!



EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro