HPC Programming

Profiling

Peter-Bernd Otte, 29.1.2019

Post-Mortem Analysis

Recap
Status: enabled on nodes, login22 with 250MB limit

Process does segmentation fault etc.

1. In bash: “ulimit -c unlimited” (check with ulimit –a and look for “core file size”)
2. Build your app with -O0 and –g and run
3. Test: ”kill -s SEGV <PID>”
4. Core file will be generated in same directory
5. Analyse with
   “totalview executable coreFileName”
   (or “gdb executable coreFileName”)

- Currently not allowed on Himster2, only backtrace (this will change)
- Hint: With “gcore <pid> -o <filename>“ a core dump is being generated and program remains running.

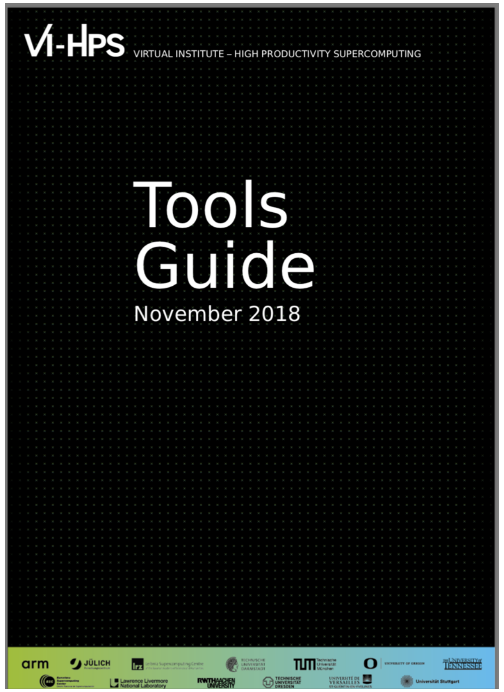
Profiling (and Tracing)

1. Overview
2. Profiling
3. Tracing (only briefly)

Possible Tools

- MUST
  - MPI usage correctness checking
- PAPI
  - Interfacing to hardware (CPU) performance counters
- Periscope Tuning Framework
  - Automativ analysis and tuning
- Scalasca
  - Large-scale parallel performance analysis
- TAU
  - Integrated parallel performance system
- Vampir
  - Interactive graphical trace visualisation & analysis
- Score-P
  - Community-developed instrumentation & measurement infrastructure

- Nice overview:
  - VI HPS Tools Guide
  - http://www.vi-hps.org/tools



Typical Workflow

1. Programming
2. Execution
3. Debugging (TotalView, gdb)
4. Analysis
    1. Hardware monitoring (Cache usage and PAPI)
    2. Profile and trace analysis (TAU, Score-P, Periscope, Scalasca, Vampir)
5. Apply changes and start again from (2.)

Tuning basics

- For success, think before about
  - right algorithms and libraries
  - compiler flags
  - Further optimisation possible in parallel file system, networking and other involved components (advanced)
- Measure your progress (todays topic)
  - To judge different optimisations
  - Test for bottlenecks
  - (your measurement always affects your runtime)

The 80/20 rule

- “If you optimize everything, you will always be unhappy.”
  Donald Knuth
  https://www.brainyquote.com/quotes/donald_knuth_181636
- Programmers spend 20% of their time to get 80% of the possible speedup.
  → Leave the remaining 20% for later…
- Find out the important parts of your code.

Possible Metrics

- Measurable:
  - Counting (call of a user-function, MPI-function, etc.)
  - Duration (e.g. time in these calls)
    - Inclusive: Timings include time spent in all timings of subroutines
    - Exclusive: Time spent in that routine, without subroutines.
  - Sizes (eg. bytes transferred, written to disc)
  - and function(counts, duration, sizes)

Example (inclusive / exclusive)

```
int func_a() {
  int c;
  c = 1*1;
  func_b();
  c += 1;
  return c;
}
```

inclusive

- Hints:
  - Execution is non-deterministic (throttling, other threads in OS, software and hardware bugs, etc.)
  - Run several times!

Instrumentation techniques

- Static instrumentation
  - Prepared before execution
- Dynamic instrumentation
  - At runtime

- Both change (minimal) the principal timing of the application
  - But also possible to change the memory access pattern
  - Accuracy of timers and counters will change
  - Measurement itself needs performance

Profiling Applications with TAU

- Next 9 slides: curtesy to Dalibor
- universal tool for single core, multi thread and multiple process applications
- Available on HIMster2, Modules:
  ```
  module load toolchain/gompi/2017a
  module load profile/TAU/gompi_2017a_2.27.1
  ```
- Application is instrumented in source code automatically by replacing CC with tau_cc.sh, i.e.

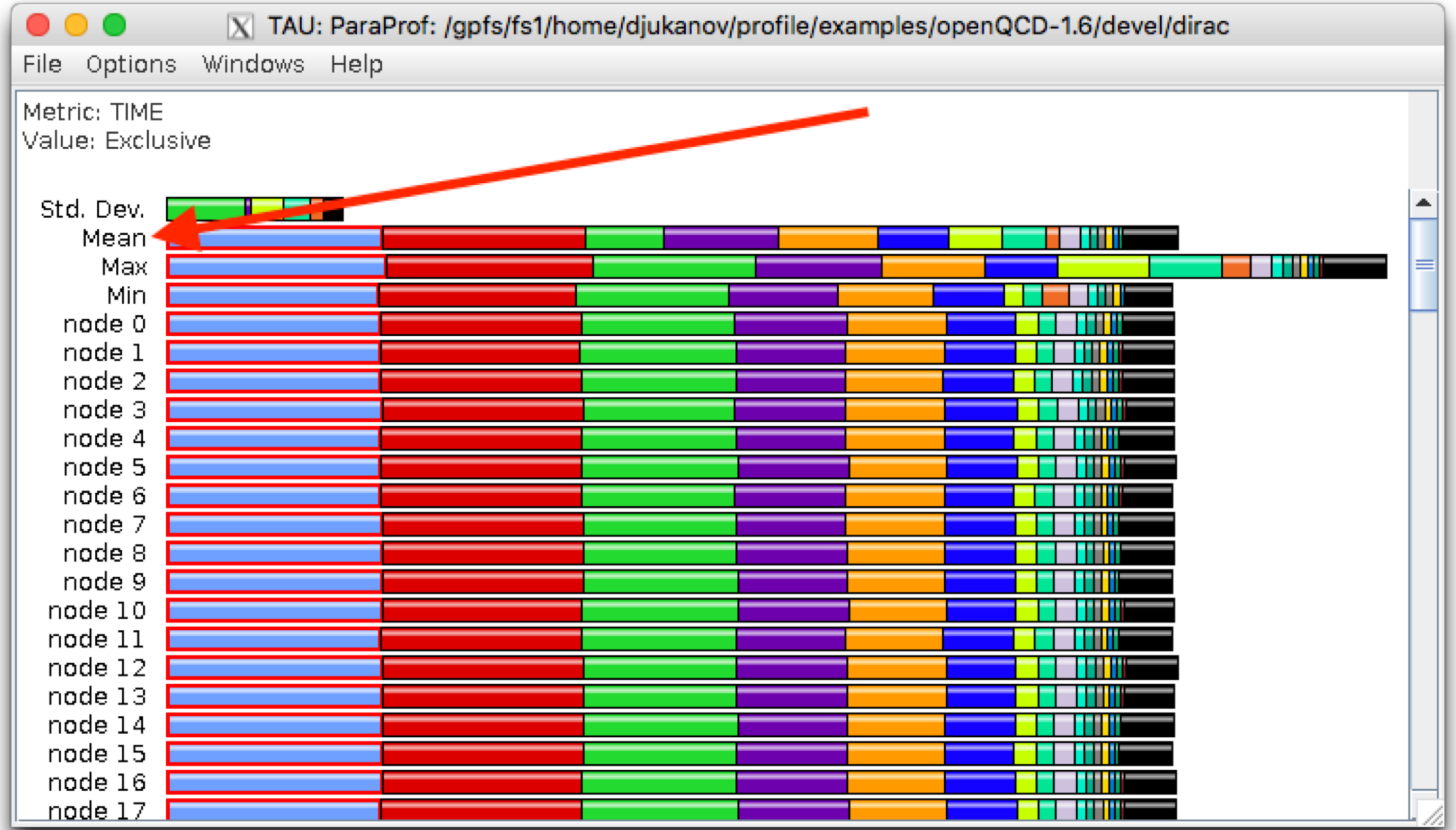
  ```
  CC=tau_cc.sh -optTauSelectFile=./select.file
  ```
- One can give a lot of extra options for more details, see for example
  man taucc
  or
  http://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf

Paraprof for Visualization

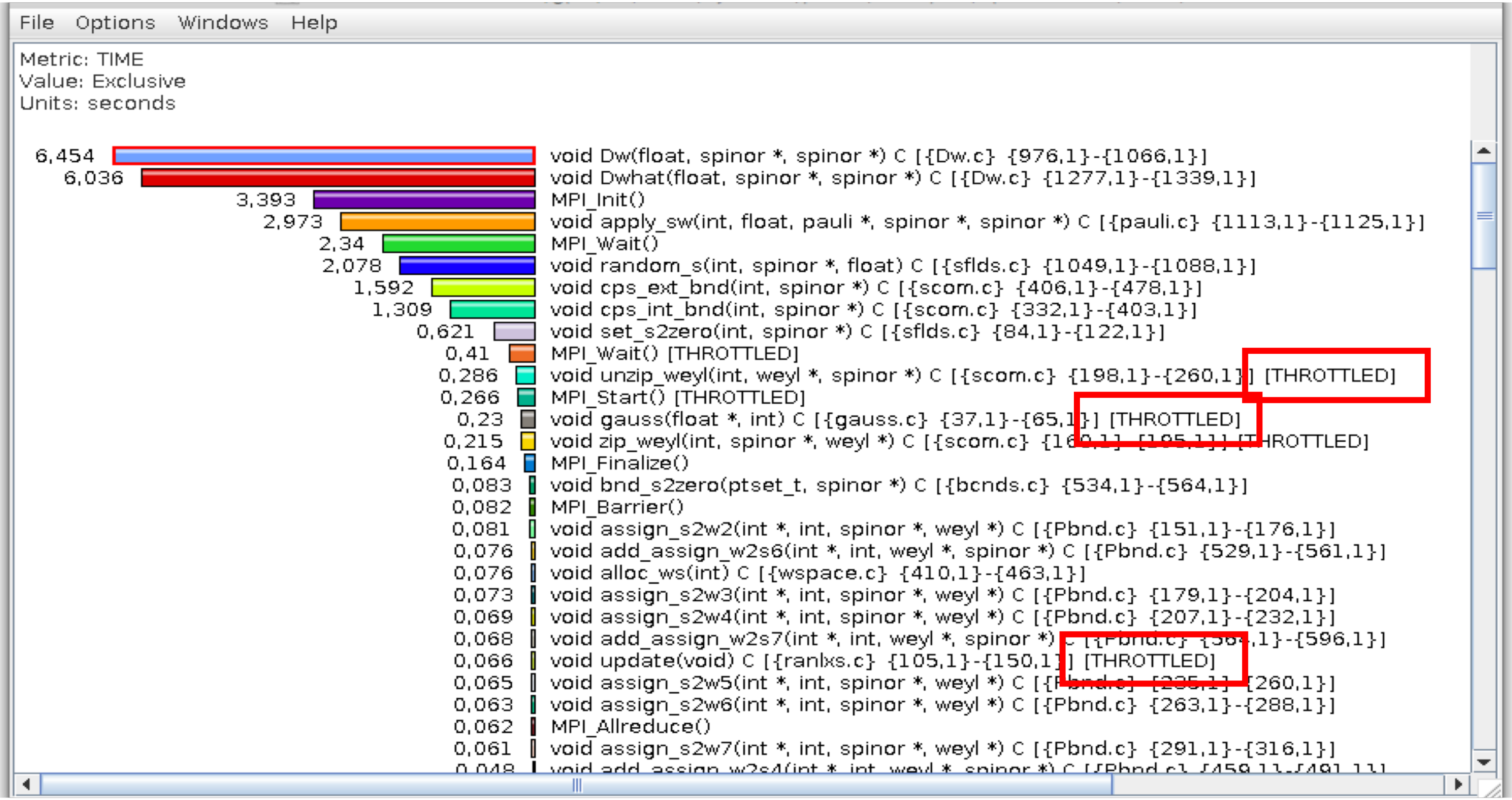
- If you want to use paraprof need:
  module load lang/Java/1.8.0_121 (this is automtically load in the profile module)
- run
  $ paraprof
  and it takes the profiles in current directory
- If you want MPI Matrix plots issue:
  export TAU_COMM_MATRIX=1

Profile breakdown

- There is inclusive and exclusive timings (check for load imbalances and barriers!)
- In the standard screen click on Mean to get a summary



Profile Breakdown – Exclusive mean

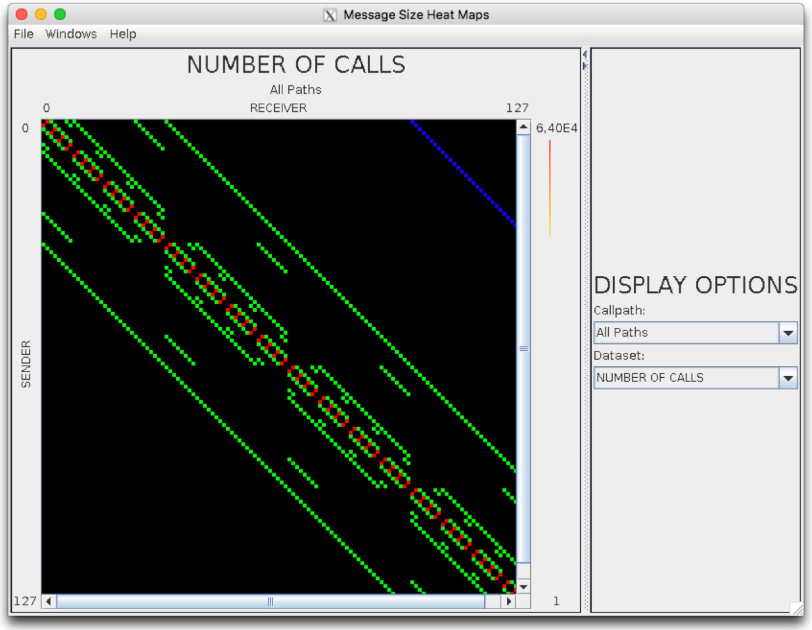


Profile Breakdown

- This will give you a first impression of where time is spent
- Note that some lines read Throttled!
  - For the timing to not drastically impact runtime a mechanism called throttling is introduced in TAU.
  - If a function that takes
    <10 µs/call and
    is called >100k
    → it is no longer profiled, and times are attached to the calling process
- These parameters can be set using:
  TAU_THROTTLE_NUMCALLS
  TAU_THROTTLE_PERCALL
- One can disable this feature with
  TAU_THROTTLE=0

MPI Communication Matrix

MPI Communication Matrix

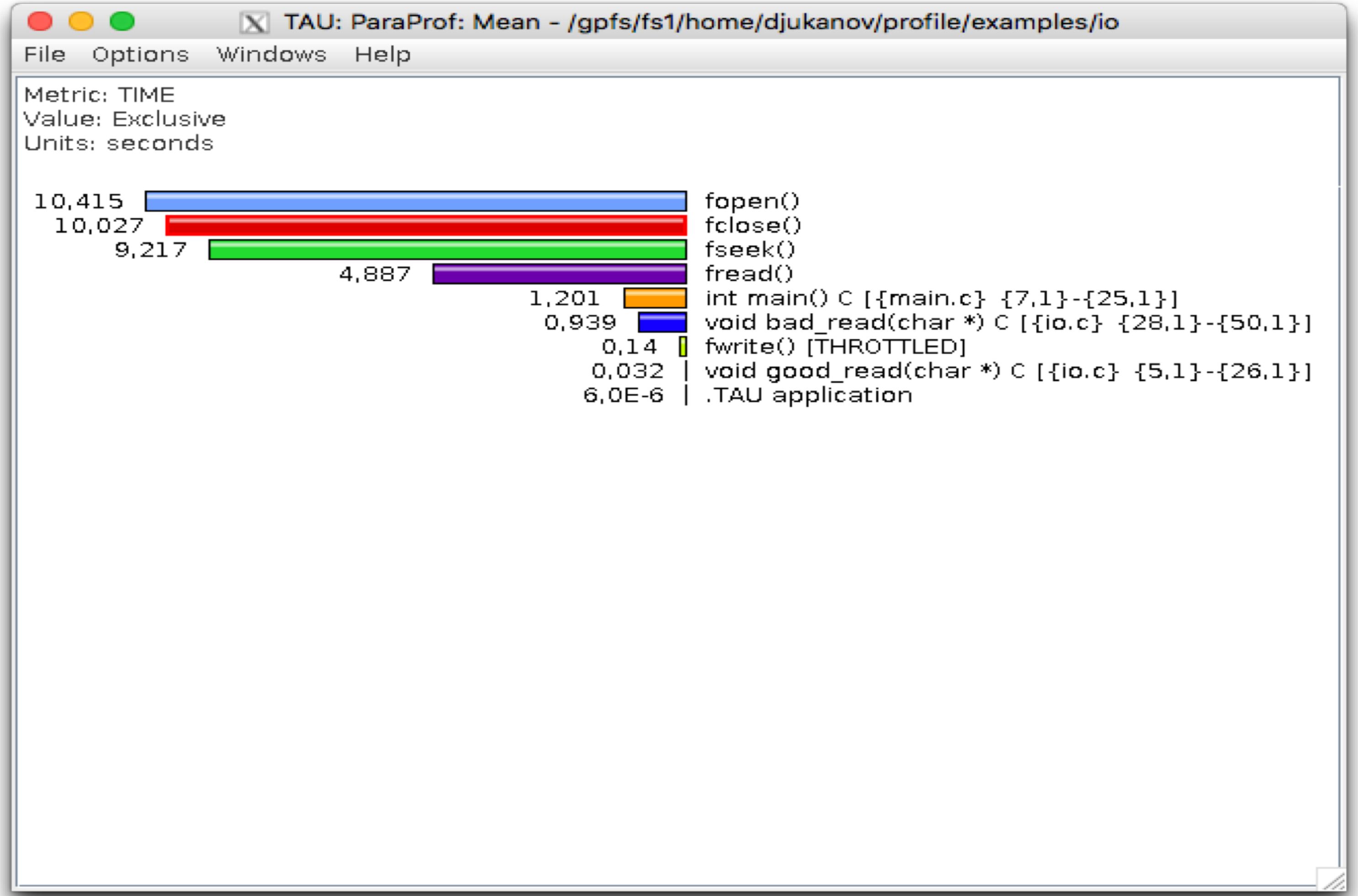


NOTE: This is only available if you have TAU_COMM_MATRX=1 in
Your submission script.
Matrix can be shown for all paths, only for subroutines,
where the number of calls can be displayed, or max messag size etc.

Non-MPI & Track IO

- If you want to profile non-mpi code use:
  export TAU_MAKEFILE=/cluster/him/tau/toolchain/gompi/2017a/2.27.1/x86_64/lib/Makefile.tau-pdt
- Track IO using:
  -optTrackIO
  as compile option via
  export TAU_OPTIONS='-optTrackIO -optVerbose‘

IO


TAU: ParaProf: Mean - /gpfs/fs1/home/djukanov/profile/examples/io
File
Options
Windows
Help
Metric: TIME
Value: Exclusive
Units: seconds
10,415
fopen()
10,027
fclose()
9,217
fseek()
4,887
fread()
1,201
int main() C [{main.c} {7,1}-{25,1}]
0,939
void bad_read(char *) C [{io.c} {28,1}-{50,1}]
0,14
fwrite() [THROTTLED]
0,032
void good_read(char *) C [{io.c} {5,1}-{26,1}]
6,0E-6
.TAU application

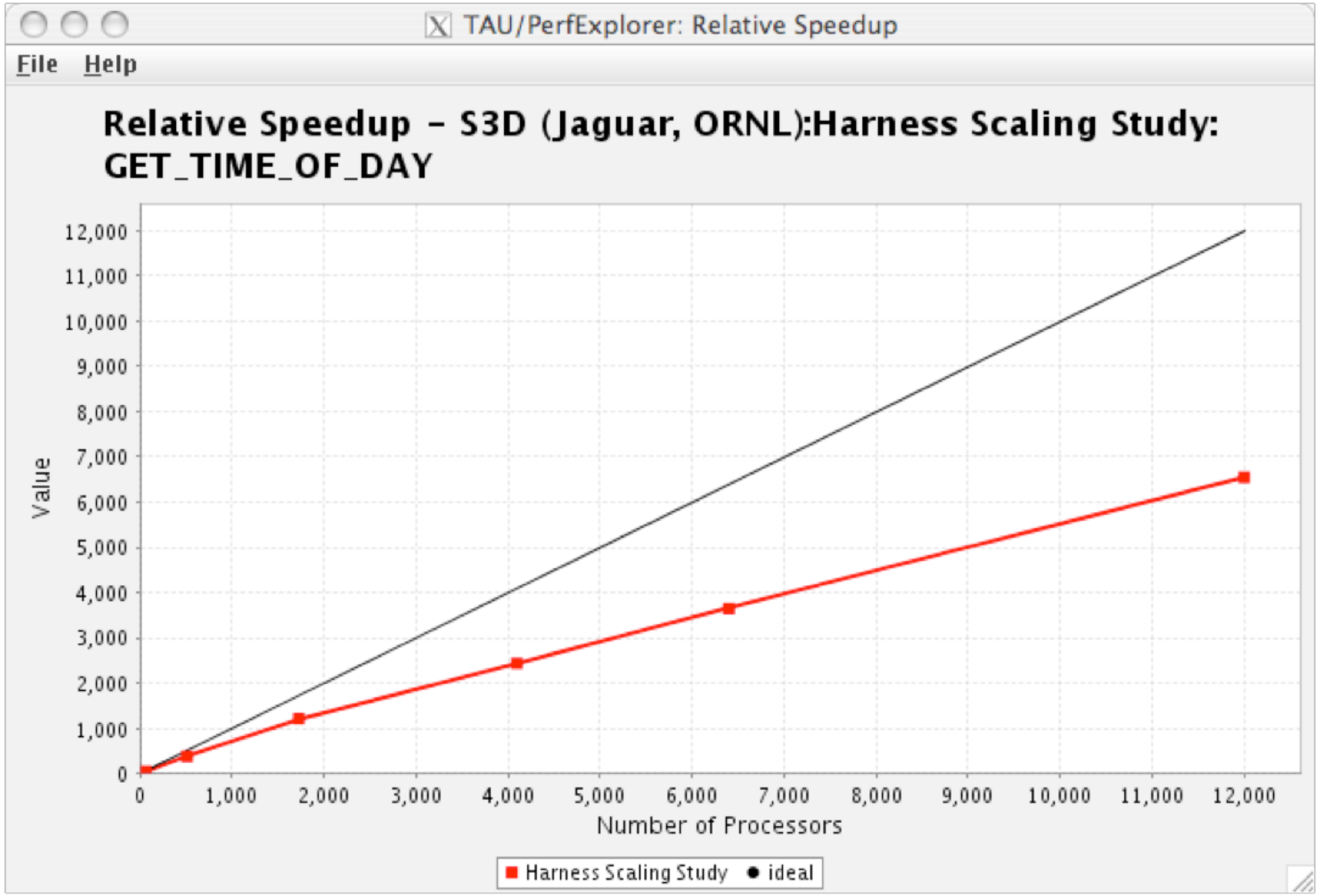
IO

TAU: ParaProf: Context Events for: node 0 - /gpfs/fs1/home/djukanov/profile/examples/io

File Options Windows Help

Name △	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
Bytes Read	40.000	8	8	8	0
Bytes Read <file=./test_io.data>	40.000	8	8	8	0
Bytes Written	640.000	8	8	8	0
Bytes Written <file=./test_io.data>	640.000	8	8	8	0
Read Bandwidth (MB/s)	20.951	8	0,014	0,399	1,657
Read Bandwidth (MB/s) <file=./test_io.data>	20.951	8	0,014	0,399	1,657
Write Bandwidth (MB/s)	38.941	8	0,021	7,987	0,244
Write Bandwidth (MB/s) <file=./test_io.data>	38.941	8	0,021	7,987	0,244
int main() C [{main.c} {7,1}-{25,1}]					
Bytes Written <file=./test_io.data>	539.999	8	8	8	0
Bytes Written	539.999	8	8	8	0
Write Bandwidth (MB/s) <file=./test_io.data>	33.979	8	0,021	7,995	0,163
Write Bandwidth (MB/s)	33.979	8	0,021	7,995	0,163
fwrite()					
Write Bandwidth (MB/s) <file=./test_io.data>	4.962	8	1,333	7,932	0,533
Write Bandwidth (MB/s)	4.962	8	1,333	7,932	0,533
Bytes Written <file=./test_io.data>	100.001	8	8	8	0
Bytes Written	100.001	8	8	8	0
void bad_read(char *) C [{io.c} {28,1}-{50,1}]					
fread()					
Read Bandwidth (MB/s) <file=./test_io.data>	20.000	0,056	0,014	0,037	0,011
Read Bandwidth (MB/s)	20.000	0,056	0,014	0,037	0,011
Bytes Read <file=./test_io.data>	20.000	8	8	8	0
Bytes Read	20.000	8	8	8	0
void good_read(char *) C [{io.c} {5,1}-{26,1}]					
fread()					
Read Bandwidth (MB/s) <file=./test_io.data>	951	8	4	7,996	0,13
Read Bandwidth (MB/s)	951	8	4	7,996	0,13
Bytes Read <file=./test_io.data>	20.000	8	8	8	0
Bytes Read	20.000	8	8	8	0

Scalability chart



Profiling with Scalasca on HIMSter2

- Since 29.1.2019 13:30 (too close for todays lecture):
  Scalasca 2.4 is available
  module load perf/Scalasca/2.4-gompi-2018°

- Will be topic of a additional workshop

Set up your workbench

- Connect via SSH to Mogon2 / HIMster2 and work on the head node
  - Load necessary software:
    module load toolchain/gompi/2017a
    module load profile/TAU/gompi_2017a_2.27.1

Exercise 5:

Learning objectives:

- What routines account for the most time? How much?

Steps:

1. Download the MPI ring solution (MPI exercise 4) from lecture webpage:
   - wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/MPI-04-solution.zip
     and unzip
2. Compile and run with
   tau_cxx.sh ring.c -o ring
   mpirun -n 2 ./ring
   paraprof
3. Click on the mean and individual nodes and check the gives times for the routines.

Exercise 6:

Learning objectives:

- Show Call graph

Steps:

1. Use same example as before and compile and run with
   TAU_CALLPATH=1
   TAU_CALLPATH_DEPTH=100
   export TAU_CALLPATH
   export TAU_CALLPATH_DEPTH
   tau_cxx.sh ring.c -o ring
   mpirun -n 2 ./ring
   paraprof
2. Click: Windows -> Thead -> Call Graph