ALCF INCITE GPU Hackathon May 20-22, 2025

# TAU Performance System®

Sameer Shende, University of Oregon and ParaTools, Inc.

http://tau.uoregon.edu/TAU_ALCF25.pdf

# Using TAU on Aurora

```
tar xf /soft/perftools/tau/tar/workshop.tgz
cd workshop; cat README
qsub -I  -l walltime=0:29:00 -lfilesystems=home:flare -A gpu_hack -q gpu_hack_prio
-l select=1 -X
module use /soft/modulefiles
module load tau
module load tau/tau2
module load frameworks

mpiexec -n 4 ./a.out
mpiexec -n 4 tau_exec -l0 -ebs ./a.out

mpiexec -n 4 tau_exec -l0 -ebs python ./foo.py
pprof -a  | more
paraprof --pack foo.ppk
On your desktop: % paraprof foo.ppk
```

# Motivation and Challenges

- With growing hardware complexity, it is getting harder to accurately measure and optimize the performance of our HPC and AI/ML workloads.
- TAU Performance System®:
  - Deliver a scalable, portable, performance evaluation toolkit for HPC and AI/ML workloads.
  - http://tau.uoregon.edu

# Motivation: Improving Productivity

- TAU Performance System®:
  - Deliver a scalable, portable, performance evaluation toolkit for HPC and AI/ML workloads
  - http://tau.uoregon.edu

# TAU Performance System®

- Versatile profiling and tracing toolkit that supports:
  - MPI, DPC++/SYCL (Level Zero), OpenCL, and OpenMP (OpenMP Tools Interface for Target Offload)
- Scalable, portable, performance evaluation toolkit for HPC and AI/ML workloads that supports:
  - C++/C/DPC++, Fortran, Python
- Supports PAPI, Likwid for hardware performance counter information
- Instrumentation includes support for Kokkos, MPI, pthread, event-based sampling, GPU runtimes
- A single tool (tau_exec) is used to launch un-instrumented, un-modified binaries
- TAU's paraprof, pprof, perfexplorer for profile analysis; Vampir, Jumpshot, Perfetto.dev for traces
- http://tau.uoregon.edu

Tuning and Analysis Utilities

UNIVERSITY OF OREGON

ParaTools

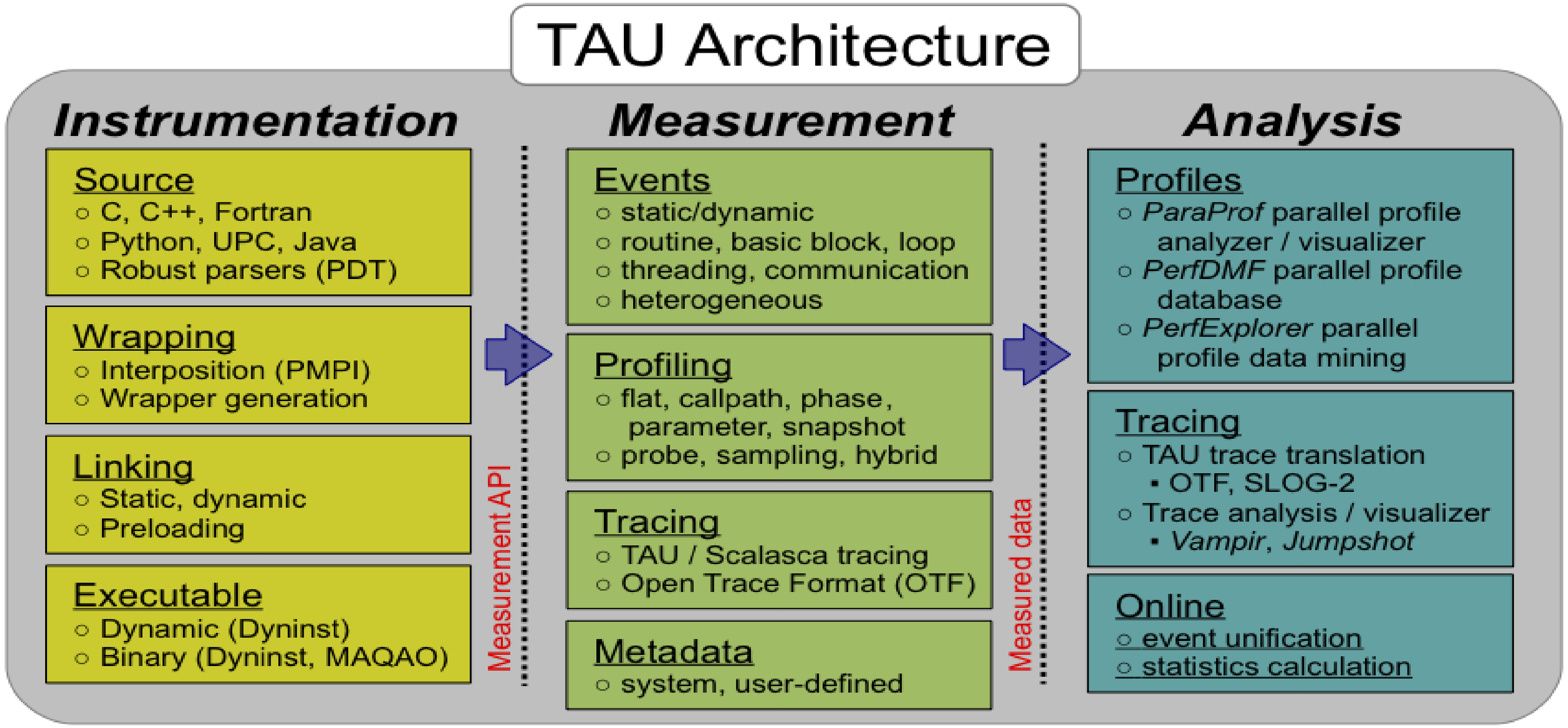
# Application Performance Engineering using TAU

- How much time is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*? What is the time spent in OpenMP loops? In kernels on GPUs.
- How many instructions are executed in these code regions?
  Floating point, Level 1 and 2 *data cache misses*, hits, branches taken? What is the extent of vectorization for loops?
- How much time did my application spend waiting at a barrier in MPI collective operations?
- What is the memory usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks? What is the memory footprint of the application? What is the memory high water mark?
- How much energy does the application use in Joules? What is the peak power usage?
- What are the I/O characteristics of the code?  What is the peak read and write *bandwidth* of individual calls, total volume?
- How does the application *scale*? What is the efficiency, runtime breakdown of performance across different core counts?

# TAU Performance System®

## Parallel performance framework and toolkit

Supports all HPC platforms, compilers, runtime system
Provides portable instrumentation, measurement, analysis



# TAU Performance System®

Instrumentation

- Fortran, C++, C, UPC, Java, Python, Chapel, Spark
- Automatic instrumentation

Measurement and analysis support

- MPI (MVAPICH2, Intel MPI), OpenSHMEM, ARMCI, PGAS, DMAPP
- Supports Intel oneAPI compilers
- pthreads, OpenMP, OMPT interface, hybrid, other thread models
- GPU: OpenCL, oneAPI DPC++/SYCL (Level Zero), OpenACC, Kokkos, RAJA
- Parallel profiling and tracing

Analysis

- Parallel profile analysis (ParaProf), data mining (PerfExplorer)
- Performance database technology (TAUdb)
- 3D profile browser

# Instrumentation

## Add hooks in the code to perform measurements

- **Source instrumentation using a preprocessor**
  - Add timer start/stop calls in a copy of the source code.
  - Use Program Database Toolkit (PDT) for parsing source code.
  - Requires recompiling the code using TAU shell scripts (tau_cc.sh, tau_f90.sh)
  - Selective instrumentation (filter file) can reduce runtime overhead and narrow instrumentation focus.
- **Compiler-based instrumentation**
  - Use system compiler to add a special flag to insert hooks at routine entry/exit.
  - Requires recompiling using TAU compiler scripts (tau_cc.sh, tau_f90.sh…)
- **Runtime preloading of TAU's Dynamic Shared Object (DSO)**
  - No need to recompile code! Use **mpiexec tau_exec ./app** with options.
  - It will preload TAU's DSO and intercept MPI calls and generate profile/trace data.

# Configurations of TAU installed on Aurora

```
module use /soft/modulefiles;

module load tau;

module load tau/tau2

module load frameworks


ls $TAU/Makefile*

/soft/perftools/tau/tau2/x86_64/lib/
Makefile.tau-level_zero-icpx-papi-ompt-mpi-pthread-python-pdt-openmp

/soft/perftools/tau/tau2/x86_64/lib/
Makefile.tau-level_zero-icpx-papi-ompt-pthread-python-pdt-openmp

mpiexec -n 12 tau_exec -l0 -ebs ./a.out will choose the MPI
configuration by default. For non MPI jobs, please use -T serial. E.g.,

tau_exec -T serial -ebs -l0 python ./foo.py and will LD_PRELOAD:
/soft/perftools/tau/tau2/x86_64/lib/shared-level_zero-icpx-papi-ompt-
pthread-python-pdt-openmp/libTAU.so
```

# Using TAU for source instrumentation

- TAU supports several measurement and thread options
  - Phase profiling, profiling with hardware counters, MPI library, Python…
  - Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it
- To instrument source code automatically using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:

```
% module use /soft/modulefiles; module load tau; load tau/tau2
% export TAU_MAKEFILE=/soft/perftools/tau/tau2/x86_64/lib/
Makefile.tau-level_zero-icpx-papi-ompt-mpi-pthread-python-pdt-openmp

% export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh )
% export PATH=$TAUDIR/x86_64/bin:$PATH
Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, C++, UPC, or C compilers respectively:
% mpif90 foo.f90        changes to
% tau_f90.sh foo.f90
```

- Set runtime environment variables, execute application and analyze performance data:
  - **% pprof (for text based profile display)**
  - **% paraprof (for GUI)**

# Configurations of TAU installed on Polaris

```
module use /soft/modulefiles; module load tau; ls $TAU/Makefile*

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-papi-mpi-cupti-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-papi-mpi-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-papi-mpi-pthread-cupti-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-tbb-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-nvidia-papi-mpi-cupti-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-nvidia-papi-mpi-pdt

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-nvidia-pdt


aprun –n 16 tau_exec –T gnu-papi-mpi-pthread-cupti-pdt –ebs ./a.out
will choose a configuration represented by:

/soft/perftools/tau/tau-2.33.2/craycnl/lib/Makefile.tau-gnu-papi-mpi-pthread-cupti-pdt
```

# TAU's Support for Runtime Systems

- *MPI*
  - PMPI profiling interface
  - MPI_T tools interface using performance and control variables
- *Pthread*
  - Captures time spent in routines per thread of execution
- *OpenMP*
  - OMPT tools interface to track salient OpenMP runtime events
  - Opari source rewriter
  - Preloading wrapper OpenMP runtime library when OMPT is not supported
- *OpenACC*
  - OpenACC instrumentation API
  - Track data transfers between host and device (per-variable)
  - Track time spent in kernels

# TAU's Support for Runtime Systems (contd.)

- *OpenCL*
  - OpenCL profiling interface
  - Track timings of kernels
- *Intel® OneAPI*
  - Level Zero
  - Track time spent in kernels executing on GPU
  - Track time spent in OneAPI runtime calls
- *Kokkos*
  - Kokkos profiling API
  - Push/pop interface for region, kernel execution interface
- *Python*
  - Python interpreter instrumentation API
  - Tracks Python routine transitions as well as Python to C transitions

# Examples of Multi-Level Instrumentation

- *MPI + OpenMP*
  - MPI_T + PMPI + OMPT may be used to track MPI and OpenMP
- *MPI + pthread*
  - PMPI + pthread interfaces
- *MPI + Intel ® oneAPI DPC++/SYCL*
- PMPI + Level Zero interfaces
- *OpenCL + Python*
  - OpenCL + Python instrumentation interfaces
- *Kokkos + OpenMP*
  - Kokkos profiling API + OMPT to transparently track events
- *Kokkos + pthread + MPI*
  - Kokkos + pthread wrapper interposition library + PMPI layer
- *MPI + OpenCL*
  - PMPI + OpenCL profiling interfaces

# Binary instrumentation of libraries: Work in progress

```
% tau_run a.out -o a.inst
```

instruments a binary. Other flags –T <tags>, -f <selective instrumentation file>

```
% tau_run -l /path/to/libhdf5.so.310 -o libhdf5.so.310
```

instruments a DSO

```
% tau_exec ./a.out
```

executes the uninstrumented application with the instrumented shared object.

To use with DyninstAPI 13 on x86_64:

1. Load spack: source spack/share/spack/setup-env.sh
2. Install dyninst: spack install dyninst@13 %gcc@11
3. Configure tau with dyninst:
   3.1 spack find -p dyninst boost tbb elfutils
   3.2 Copy the paths for each package into the configure line

3.3 ./configure -bfd=download -dyninst=<dir> -tbb=<dir> -boost=<dir> -elf=<dir>; <set paths>; make install

# Binary instrumentation of libraries: HDF5

STEP SOFTWARE TOOLS ECOSYSTEM PROJECT

```
$ pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.272           68           1           1      68245 .TAU application
 99.6            1           67           1          26      67973 taupreload_main
 65.8        0.008           44           6           1       7484 H5open
 65.8            6           44           2          14      22448 H5_init_library
 36.0            4           24           1          12      24563 H5VL_init_phase2
 27.8            1           18           1         319      18943 H5T_init
 19.8        0.193           13         179         179         76 H5T__register_int
 19.5        0.302           13         179         310         74 H5T__register
 19.0            4           12         155        2555         84 H5T__path_find_real
 13.0            2            8           1          79       8857 H5P_init_phase1
 12.7        0.663            8           2          51       4349 H5F_open
 11.2        0.348            7           1           6       7610 H5Fcreate
 10.5        0.386            7           1           6       7138 H5F__create_api_common
  9.8        0.406            6           1           2       6707 H5VL_file_create
  9.2        0.005            6           1           1       6299 H5VL__native_file_create
  7.1            1            4         488         976         10 H5T_copy
  6.5            1            4           1         363       4452 H5E_init
  5.6        0.013            3           4          12        956 H5I_dec_app_ref
  5.6        0.013            3           2          10       1896 H5Fclose
  5.5        0.009            3           2           4       1878 H5F__close_cb
  5.5         0.01            3           2           6       1868 H5VL_file_close
  5.4        0.013            3           2           4       1852 H5VL__native_file_close
  5.4        0.019            3           4           8        924 H5F_try_close.localalias
```

# Using TAU's Runtime Preloading Tool: tau_exec

Preload a wrapper that intercepts the runtime system call and substitutes with another

- **MPI**
- **OpenMP**
- **POSIX I/O**
- **Memory allocation/deallocation routines**
- **Wrapper library for an external package**

No modification to the binary executable!

Enable other TAU options (communication matrix, OTF2, event-based sampling)

# TAU Execution Command (tau_exec)

Uninstrumented execution

% mpirun -np 256 ./a.out

Track GPU operations

% mpirun –np 256 tau_exec –l0 ./a.out

% mpirun –np 256 tau_exec –cupti ./a.out

% mpirun –np 256 tau_exec –rocm ./a.out

% mpirun –np 256 tau_exec –openacc ./a.out

Track MPI performance

% mpirun -np 256 tau_exec ./a.out

Track I/O, and MPI performance (MPI enabled by default)

% mpirun -np 256 tau_exec -io ./a.out

Track OpenMP and MPI execution (using OMPT for Intel v19+ or Clang 8+)

% export TAU_OMPT_SUPPORT_LEVEL=full;

% mpirun –np 256 tau_exec –T ompt,mpi -ompt ./a.out

Track memory operations

% export TAU_TRACK_MEMORY_LEAKS=1

% mpirun –np 256 tau_exec –memory_debug ./a.out (bounds check)

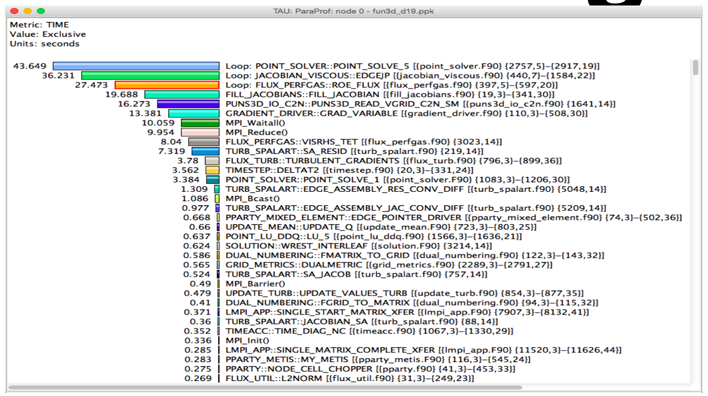
Use event based sampling (compile with –g)

% mpirun –np 256 tau_exec –ebs ./a.out

Also export TAU_METRICS=TIME,PAPI_L1_DCM… -ebs_resolution=<file | function | line>

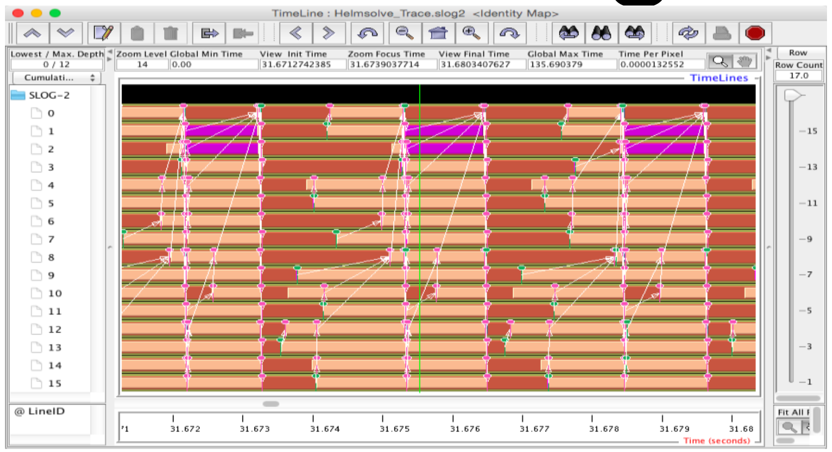
# Profiling and Tracing

## Profiling




- **Profiling** shows you **how much** (total) time was spent in each routine

## Tracing




- Tracing shows you when the events take place on a timeline

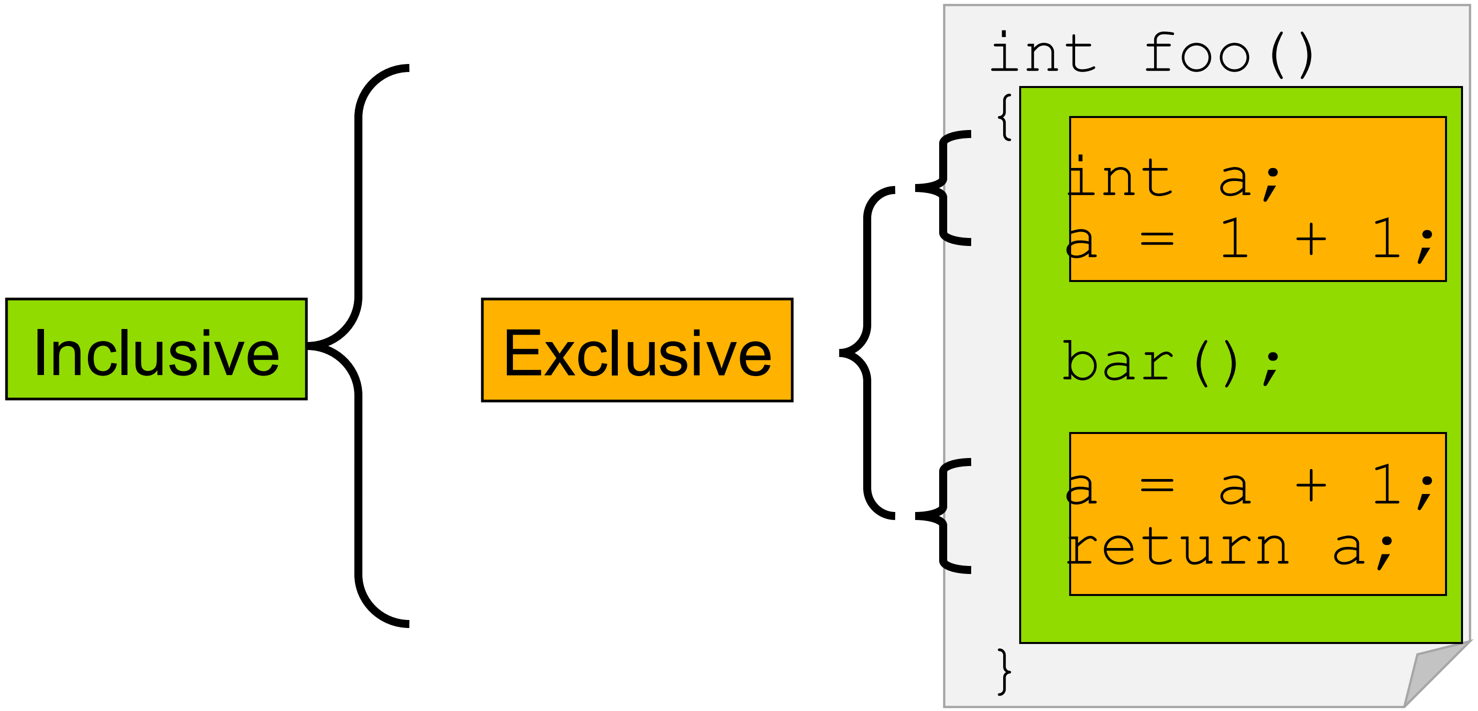
- Profiling and tracing

  **Profiling** shows you **how much** (total) time was spent in each routine

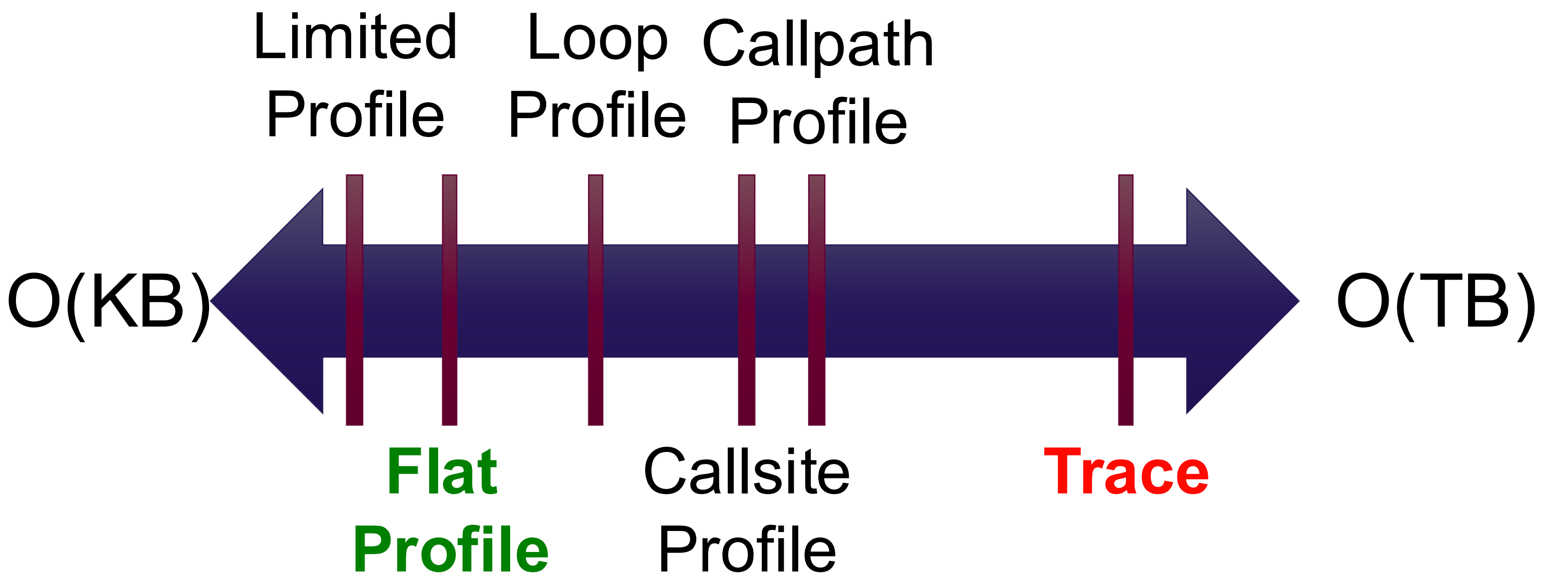
  **Tracing** shows you **when** the events take place on a timeline

# Inclusive vs. Exclusive values

- Inclusive
  - Information of all sub-elements aggregated into single value
- Exclusive
  - Information cannot be subdivided further



# How much data do you want?



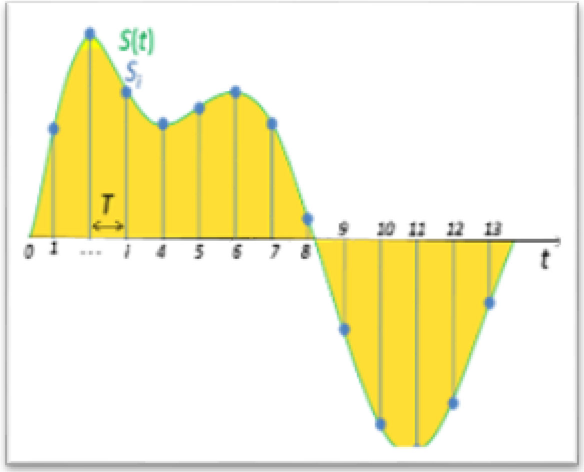
# Performance Data Measurement

## Direct via Probes

```
Call
START('potential')
// code
Call
STOP('potential')
```

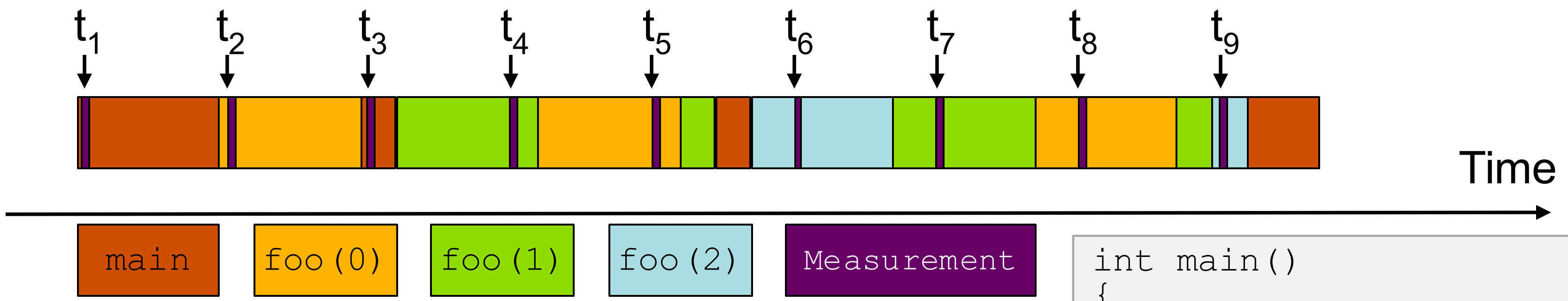
- Exact measurement
- Fine-grain control
- Calls inserted into code

## Indirect via Sampling




- No code modification
- Minimal effort
- Relies on debug symbols (**-g**)

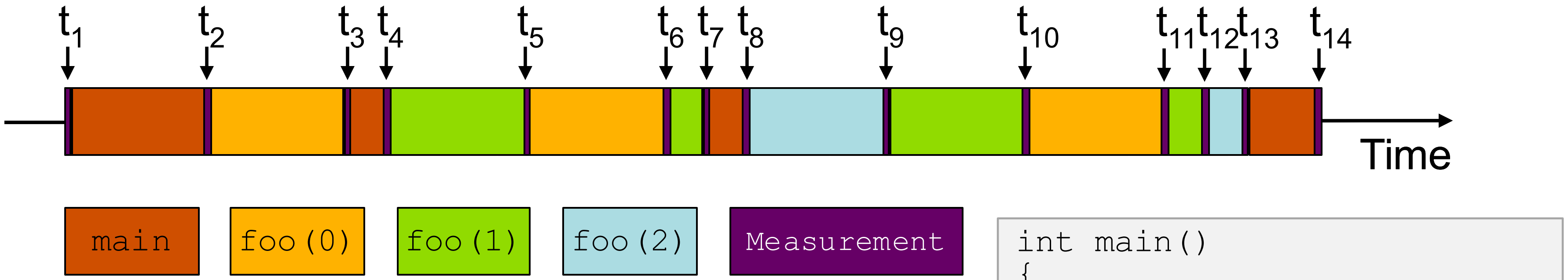
# Sampling




- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

# Instrumentation



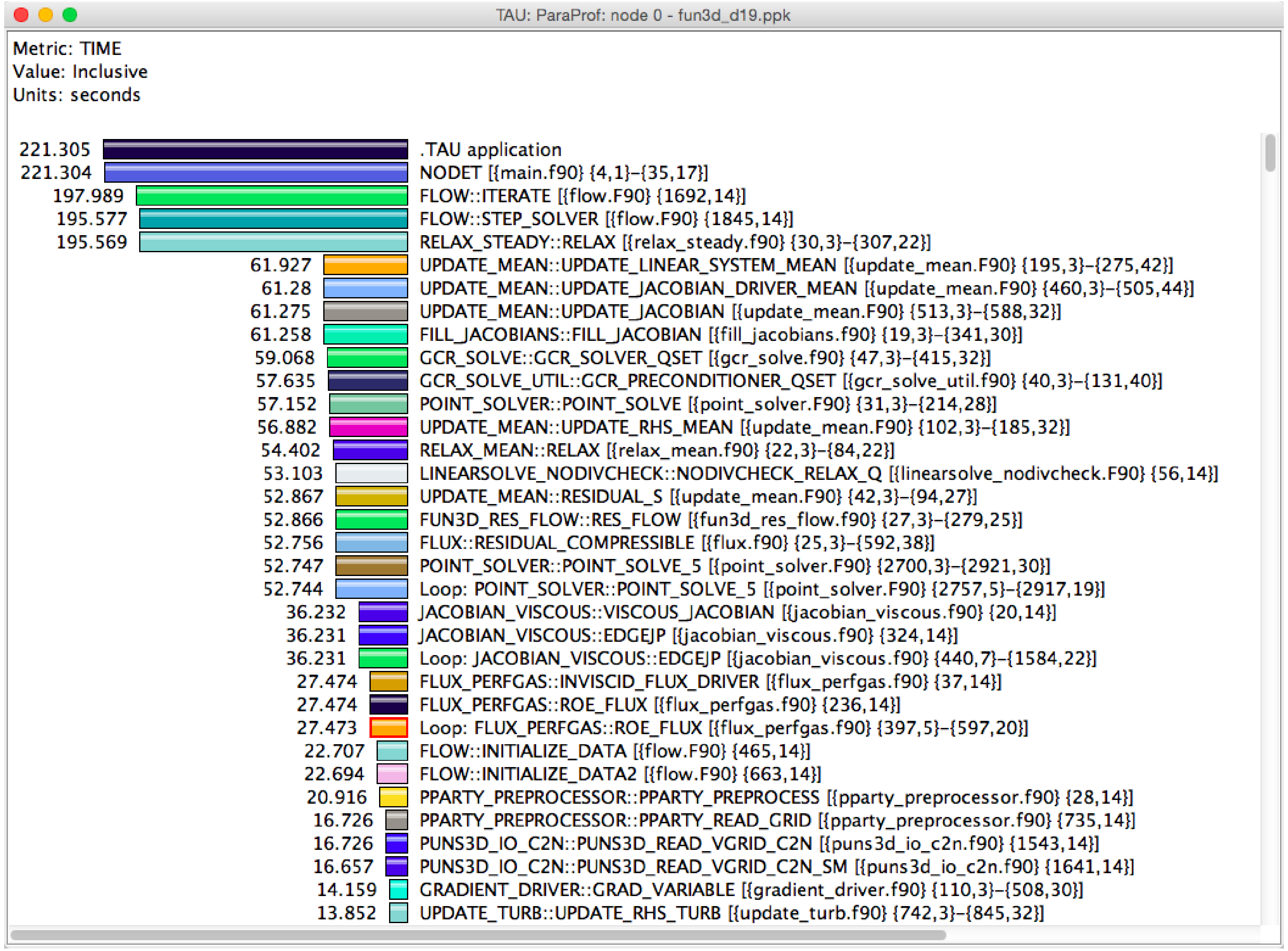

Measurement code is inserted such that every event of interest is captured directly
- Can be done in various ways

Advantage:
- Much more detailed information

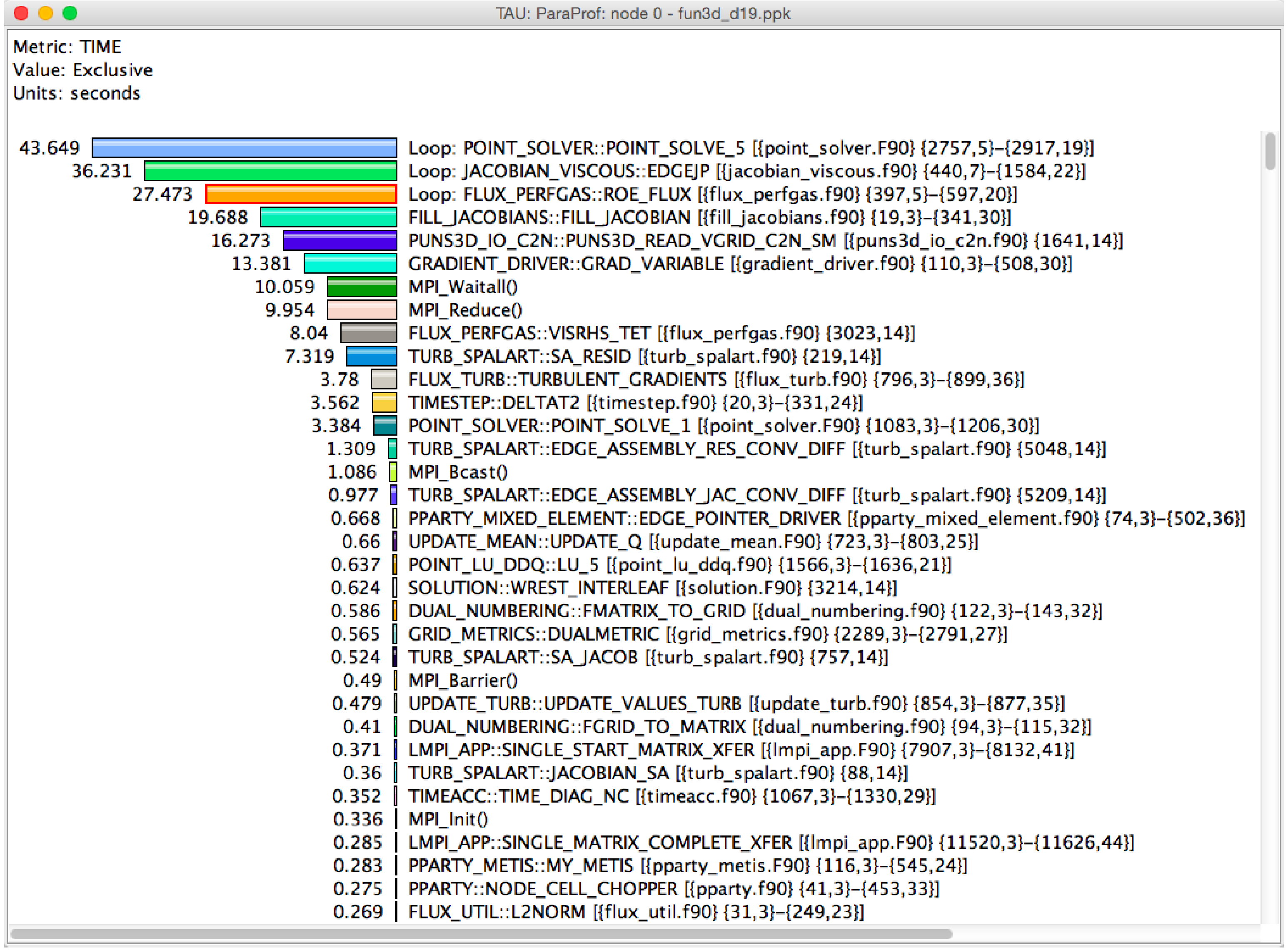
Disadvantage:
- Processing of source-code / executable necessary
- Large relative overheads for small functions

```
int main()
{
  int i;
  TAU_START("main");
  for (i=0; i < 3; i++)
    foo(i);
  TAU_STOP("main");
  return 0;
}

void foo(int i)
{
  TAU_START("foo");
  if (i > 0)
    foo(i - 1);
  TAU_STOP("foo");

}
```

# Inclusive Measurements



# Exclusive Time



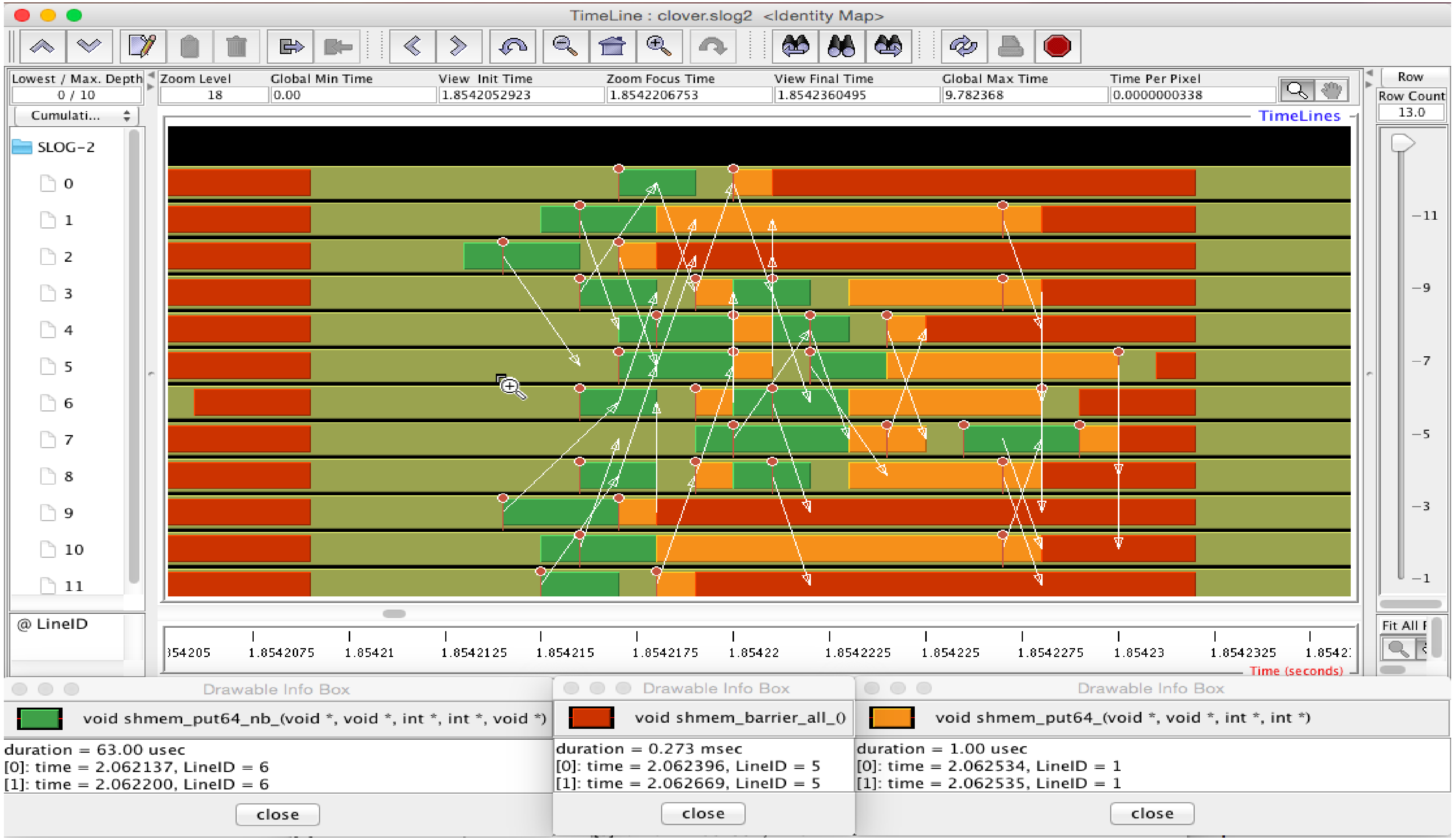


Argonne NATIONAL LABORATORY

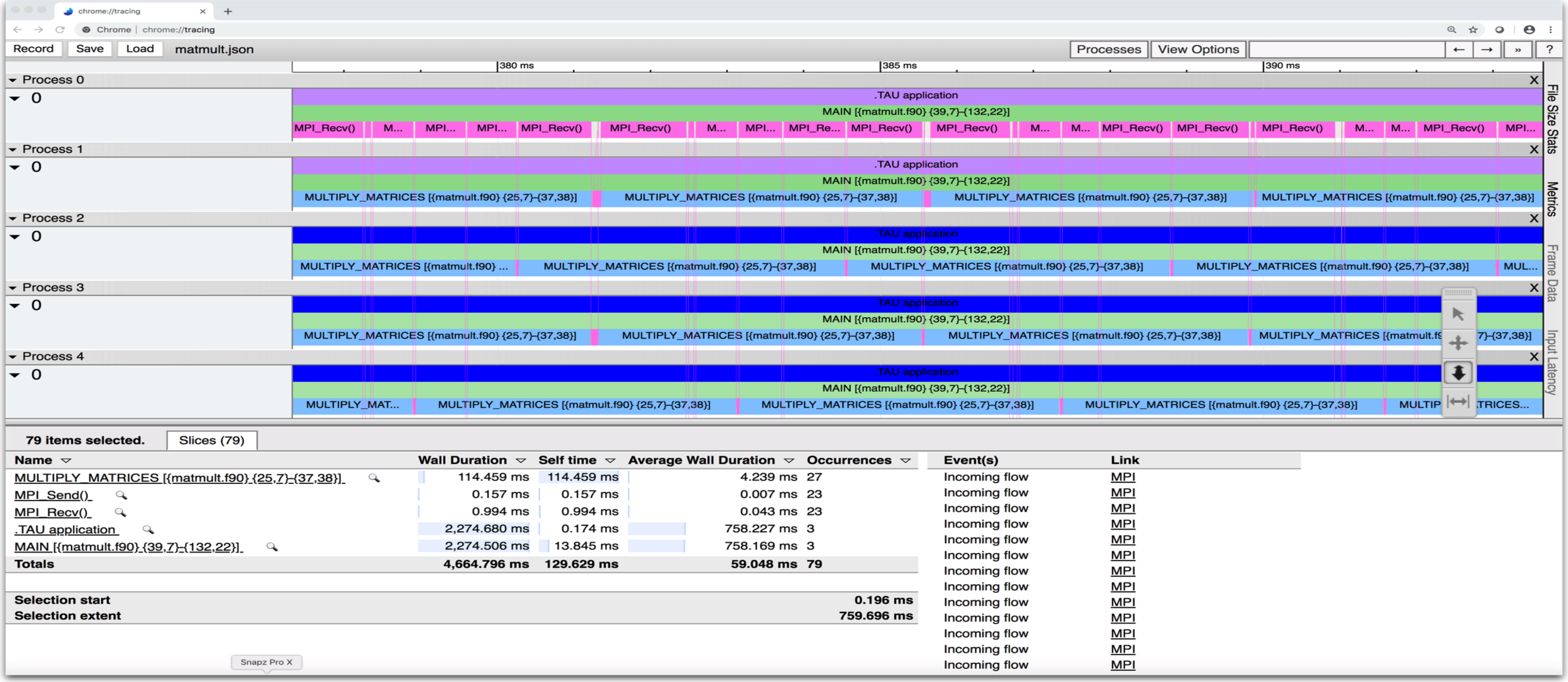
# TAU's Runtime Environment Variables

| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACE | 0 | Setting to 1 turns on tracing |
| TAU_CALLPATH | 0 | Setting to 1 turns on callpath profiling |
| TAU_TRACK_MEMORY_FOOTPRINT | 0 | Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high-water mark of memory usage |
| TAU_TRACK_POWER | 0 | Tracks power usage by sampling periodically. |
| TAU_CALLPATH_DEPTH | 2 | Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo) |
| TAU_SAMPLING | 1 | Setting to 1 enables event-based sampling. |
| TAU_TRACK_SIGNALS | 0 | Setting to 1 generate debugging callstack info when a program crashes |
| TAU_COMM_MATRIX | 0 | Setting to 1 generates communication matrix display using context events |
| TAU_THROTTLE | 1 | Setting to 0 turns off throttling. Throttles instrumentation in lightweight routines that are called frequently |
| TAU_THROTTLE_NUMCALLS | 100000 | Specifies the number of calls before testing for throttling |
| TAU_THROTTLE_PERCALL | 10 | Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call |
| TAU_CALLSITE | 0 | Setting to 1 enables callsite profiling that shows where an instrumented function was called. Also compatible with tracing. |
| TAU_PROFILE_FORMAT | Profile | Setting to "merged" generates a single file. "snapshot" generates xml format |
| TAU_METRICS | TIME | Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>) |

# Tracing: Jumpshot [ANL] (ships with TAU)



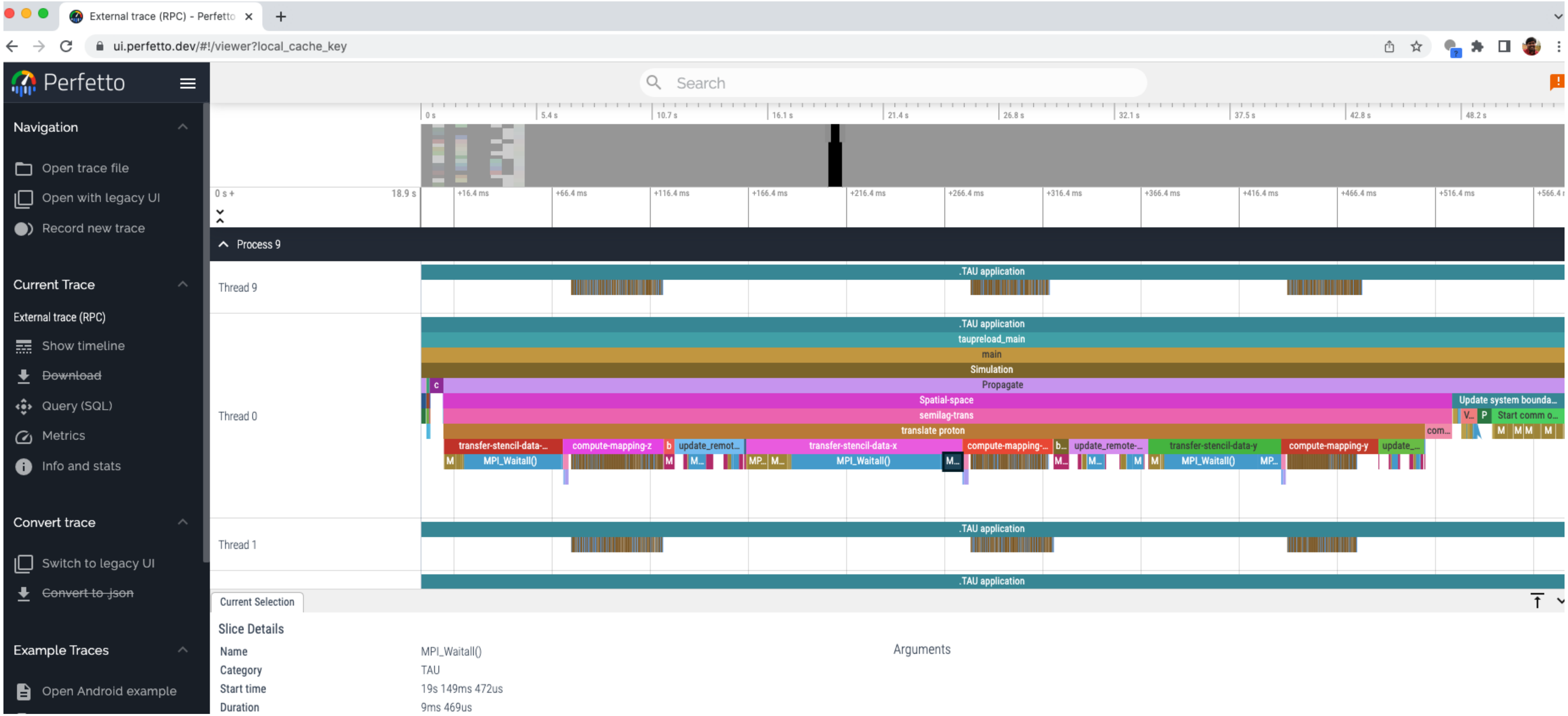
# Tracing: Chrome Browser



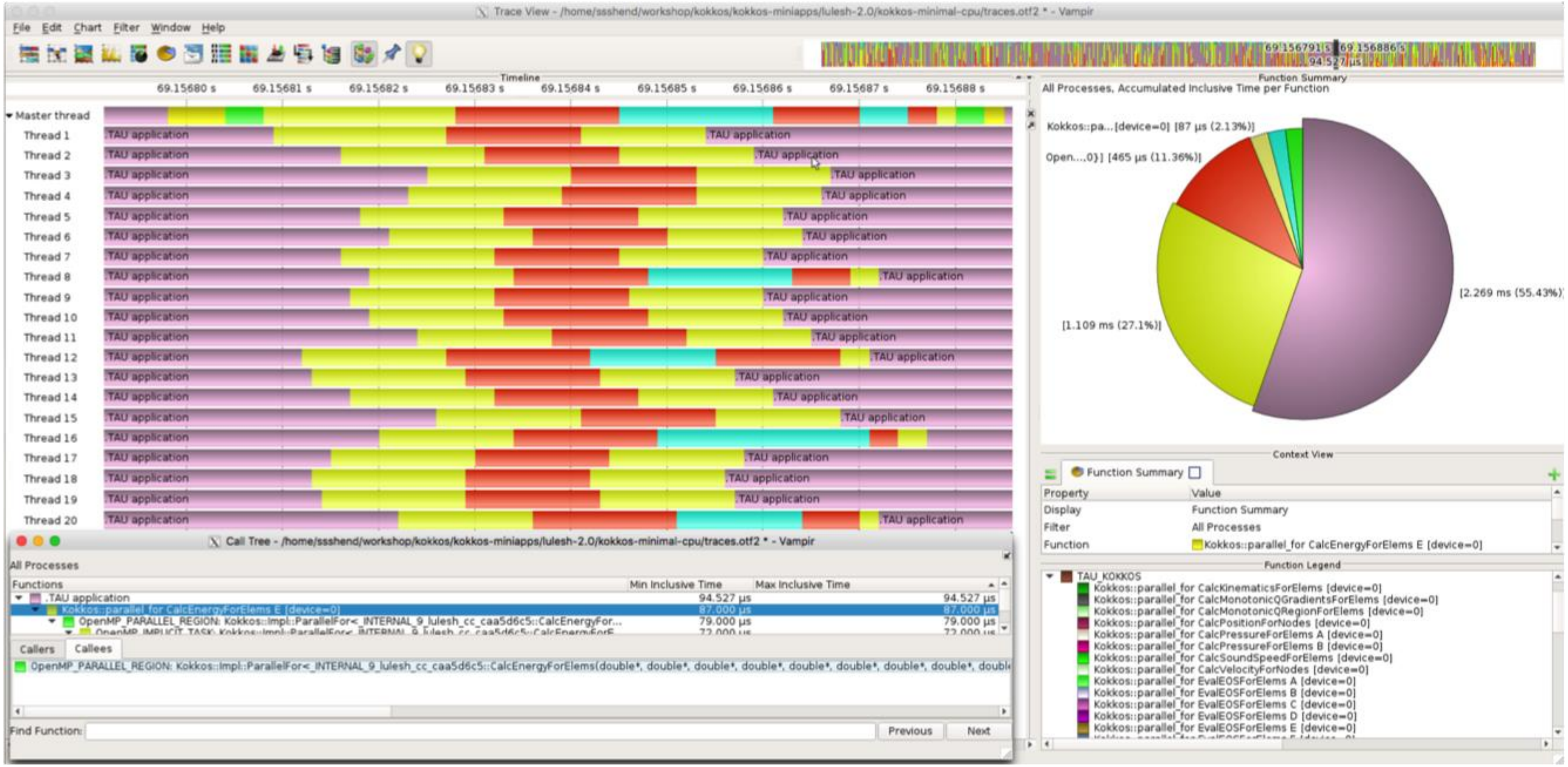

**% export TAU_TRACE=1**
**% mpirun –np 256 tau_exec ./a.out**
**% tau_treemerge.pl; tau_trace2json tau.trc tau.edf –chrome –ignoreatomic –o app.json**

**Chrome browser: chrome://tracing (Load -> app.json) OR use https://perfetto.dev**

# Perfetto.dev



# Vampir [TU Dresden] Timeline: Kokkos




% export TAU_TRACE=1; export TAU_TRACE_FORMAT=otf2

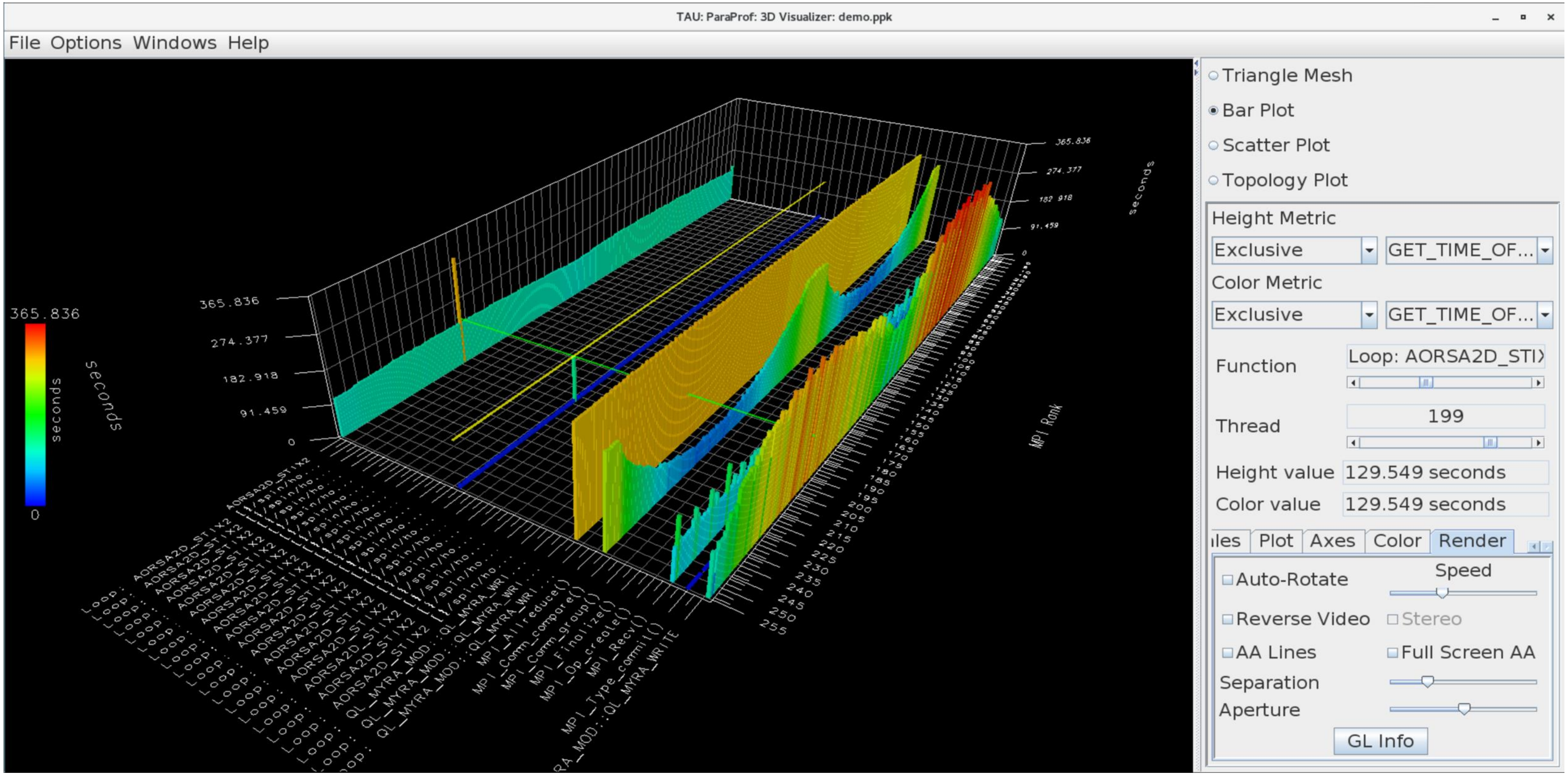
% tau_exec -T ompt –ompt ./a.out

% vampir traces.otf2 &

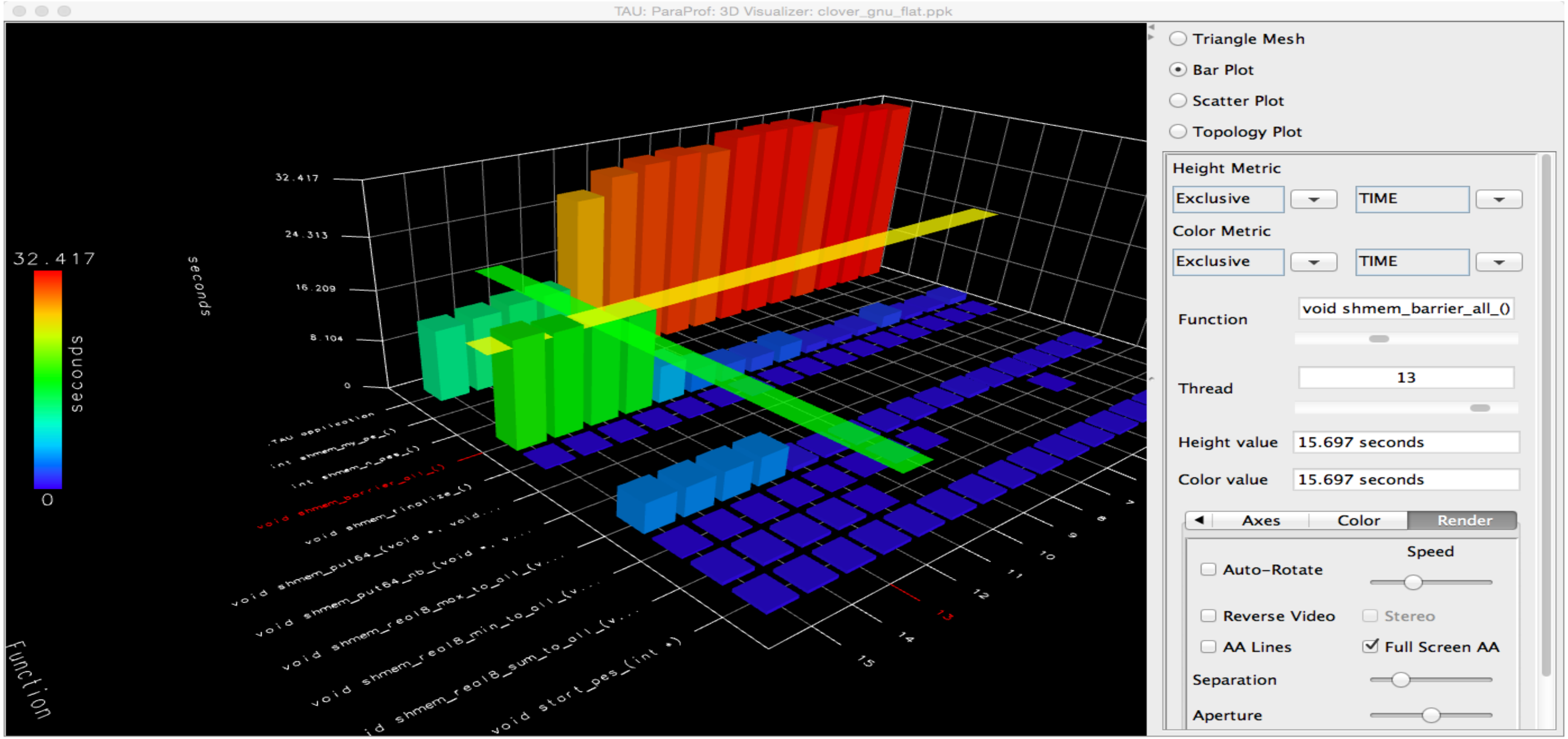
# ParaProf Profile Browser


```
% paraprof
```

# ParaProf 3D Profile Browser

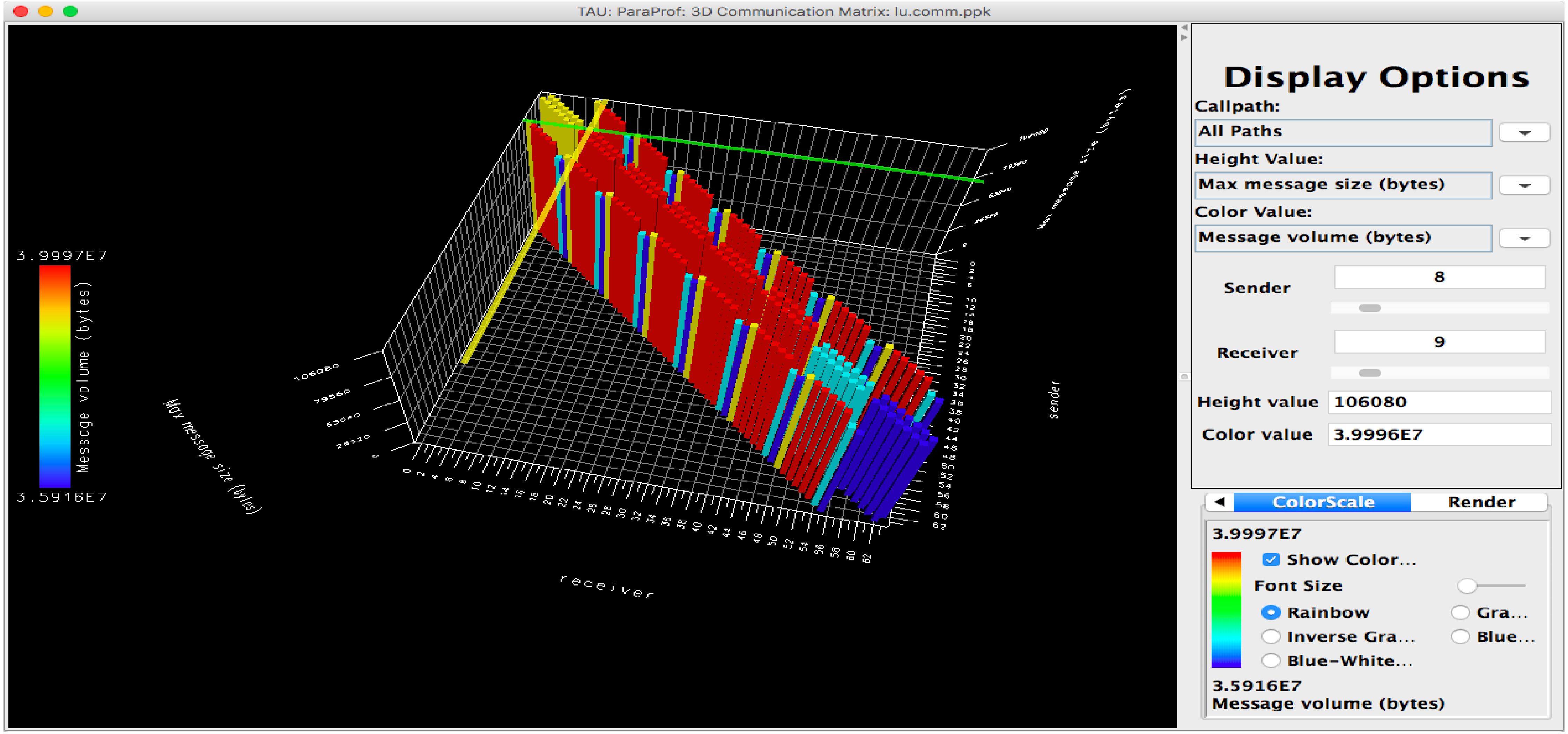


# TAU – ParaProf 3D Visualization




% paraprof app.ppk
Windows -> 3D Visualization -> Bar Plot (right pane)

# TAU – 3D Communication Window



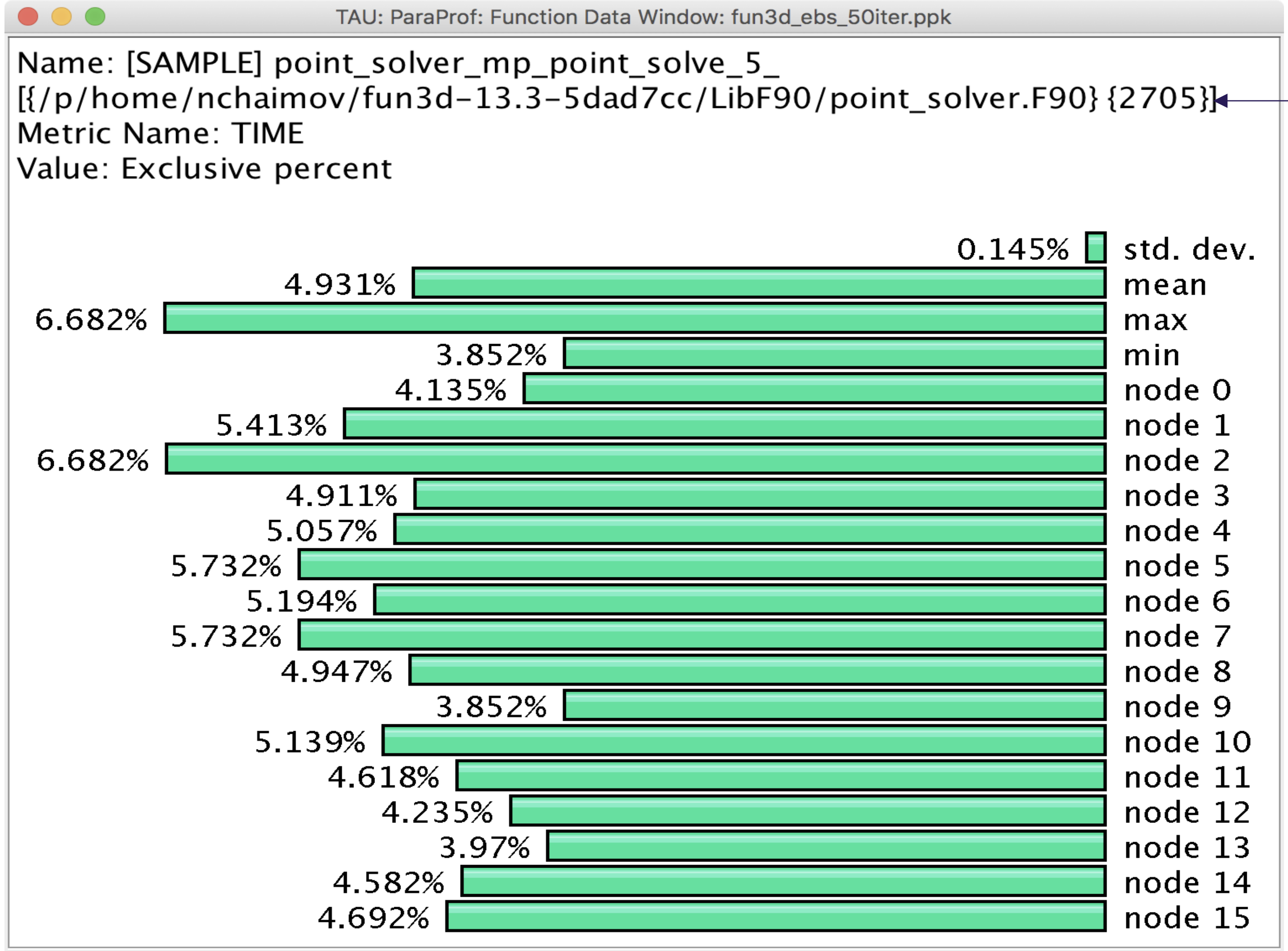

% export TAU_COMM_MATRIX=1; mpirun … tau_exec ./a.out

% paraprof ; Windows -> 3D Communication Matrix



Argonne NATIONAL LABORATORY

# Event Based Sampling (EBS)




File: point_solver.F90
Line: 2705

Uninstrumented!

```
% mpirun -n 16 tau_exec -ebs a.out
```

# TAU's Runtime Environment Variables

| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACE | 0 | Setting to 1 turns on tracing |
| TAU_CALLPATH | 0 | Setting to 1 turns on callpath profiling |
| TAU_TRACK_MEMORY_FOOTPRINT | 0 | Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high water mark of memory usage |
| TAU_TRACK_POWER | 0 | Tracks power usage by sampling periodically. |
| TAU_CALLPATH_DEPTH | 2 | Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo) |
| TAU_SAMPLING | 1 | Setting to 1 enables event-based sampling. |
| TAU_TRACK_SIGNALS | 0 | Setting to 1 generate debugging callstack info when a program crashes |
| TAU_COMM_MATRIX | 0 | Setting to 1 generates communication matrix display using context events |
| TAU_THROTTLE | 1 | Setting to 0 turns off throttling. Throttles instrumentation in lightweight routines that are called frequently |
| TAU_THROTTLE_NUMCALLS | 100000 | Specifies the number of calls before testing for throttling |
| TAU_THROTTLE_PERCALL | 10 | Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call |
| TAU_CALLSITE | 0 | Setting to 1 enables callsite profiling that shows where an instrumented function was called. Also compatible with tracing. |
| TAU_PROFILE_FORMAT | Profile | Setting to “merged” generates a single file. “snapshot” generates xml format |
| TAU_METRICS | TIME | Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>) |

# Runtime Environment Variables

| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACE | 0 | Setting to 1 turns on tracing |
| TAU_TRACE_FORMAT | Default | Setting to “otf2” turns on TAU’s native OTF2 trace generation (configure with –otf=download) |
| TAU_EBS_UNWIND | 0 | Setting to 1 turns on unwinding the callstack during sampling (use with tau_exec –ebs or TAU_SAMPLING=1) |
| TAU_EBS_RESOLUTION | line | Setting to “function” or “file” changes the sampling resolution to function or file level respectively. |
| TAU_TRACK_LOAD | 0 | Setting to 1 tracks system load on the node |
| TAU_SELECT_FILE | Default | Setting to a file name, enables selective instrumentation based on exclude/include lists specified in the file. |
| TAU_OMPT_SUPPORT_LEVEL | basic | Setting to “full” improves resolution of OMPT TR6 regions on threads 1.. N-1. Also, “lowoverhead” option is available. |
| TAU_OMPT_RESOLVE_ADDRESS_EAGERLY | 1 | Setting to 1 is necessary for event based sampling to resolve addresses with OMPT. Setting to 0 allows the user to do offline address translation. |

# Runtime Environment Variables

| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACK_MEMORY_LEAKS | 0 | Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory) |
| TAU_EBS_SOURCE | TIME | Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1) |
| TAU_EBS_PERIOD | 100000 | Specifies the overflow count for interrupts |
| TAU_MEMDBG_ALLOC_MIN/MAX | 0 | Byte size minimum and maximum subject to bounds checking (used with TAU_MEMDBG_PROTECT_*) |
| TAU_MEMDBG_OVERHEAD | 0 | Specifies the number of bytes for TAU's memory overhead for memory debugging. |
| TAU_MEMDBG_PROTECT_BELOW/ABOVE | 0 | Setting to 1 enables tracking runtime bounds checking below or above the array bounds (requires –optMemDbg while building or tau_exec –memory) |
| TAU_MEMDBG_ZERO_MALLOC | 0 | Setting to 1 enables tracking zero byte allocations as invalid memory allocations. |
| TAU_MEMDBG_PROTECT_FREE | 0 | Setting to 1 detects invalid accesses to deallocated memory that should not be referenced until it is reallocated (requires –optMemDbg or tau_exec –memory) |
| TAU_MEMDBG_ATTEMPT_CONTINUE | 0 | Setting to 1 allows TAU to record and continue execution when a memory error occurs at runtime. |
| TAU_MEMDBG_FILL_GAP | Undefined | Initial value for gap bytes |
| TAU_MEMDBG_ALINGMENT | Sizeof(int) | Byte alignment for memory allocations |
| TAU_EVENT_THRESHOLD | 0.5 | Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max |

# Download TAU from U. Oregon

Tuning and Analysis Utilities

**http://tau.uoregon.edu**

**https://e4s.io [TAU in Docker/Singularity containers]**

**for more information**

**Free download, open source, BSD license**

# Reference

# Installing and Configuring TAU

- Installing PDT:
  - wget tau.uoregon.edu/pdt_lite.tgz
  - ./configure –prefix=<dir>; make ; make install
- Installing TAU:
  - wget tau.uoregon.edu/tau.tgz;    tar zxf tau.tgz; cd tau-2.<ver>
  - wget http://tau.uoregon.edu/ext.tgz ; tar xf ext.tgz
  - ./configure -bfd=download -pdt=<dir> -papi=<dir> -mpi
    –pthread –c++=mpicxx –cc=mpicc –fortran=mpif90
    –dwarf=download –unwind=download –otf=download
    –iowrapper –papi=<dir>
  - make install
- Using TAU for source instrumentation (not needed with tau_exec):
  - export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-<TAGS>
  - make CC=tau_cc.sh   CXX=tau_cxx.sh   F90=tau_f90.sh

# Compile-Time Options

Optional parameters for the TAU_OPTIONS environment variable:

% tau_compiler.sh

| | |
|---|---|
| -optVerbose | Turn on verbose debugging messages |
| -optCompInst | Use compiler based instrumentation |
| -optNoCompInst | Do not revert to compiler instrumentation if source instrumentation fails. |
| -optTrackIO | Wrap POSIX I/O call and calculates vol/bw of I/O operations (Requires TAU to be configured with *–iowrapper*) |
| -optTrackGOMP | Enable tracking GNU OpenMP runtime layer (used without –opari) |
| -optMemDbg | Enable runtime bounds checking (see TAU_MEMDBG_* env vars) |
| -optKeepFiles | Does not remove intermediate .pdb and .inst.* files |
| -optPreProcess | Preprocess sources (OpenMP, Fortran) before instrumentation |
| -optTauSelectFile=”<file>" | Specify selective instrumentation file for *tau_instrumentor* |
| -optTauWrapFile=”<file>" | Specify path to *link_options.tau* generated by *tau_gen_wrapper* |
| -optHeaderInst | Enable Instrumentation of headers |
| -optTrackUPCR | Track UPC runtime layer routines (used with tau_upc.sh) |
| -optLinking="" | Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS) |
| -optCompile="" | Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS) |
| -optPdtF95Opts="" | Add options for Fortran parser in PDT (f95parse/gfparse) … |

# Compile-Time Options (contd.)

Optional parameters for the TAU_OPTIONS environment variable:

% tau_compiler.sh

| | |
|---|---|
| -optShared | Use TAU's shared library (libTAU.so) instead of static library (default) |
| -optPdtCxxOpts="" | Options for C++ parser in PDT (cxxparse). |
| -optPdtF90Parser="" | Specify a different Fortran parser |
| -optPdtCleanscapeParser | Specify the Cleanscape Fortran parser instead of GNU gfparser |
| -optTau="" | Specify options to the tau_instrumentor |
| -optTrackDMAPP | Enable instrumentation of low-level DMAPP API calls on Cray |
| -optTrackPthread | Enable instrumentation of pthread calls |

See tau_compiler.sh for a full list of TAU_OPTIONS.

…

# TAU's Runtime Environment Variables

| Environment Variable | Default | Description |
| --- | --- | --- |
| TAU_TRACE | 0 | Setting to 1 turns on tracing |
| TAU_CALLPATH | 0 | Setting to 1 turns on callpath profiling |
| TAU_TRACK_MEMORY_FOOTPRINT | 0 | Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high water mark of memory usage |
| TAU_TRACK_POWER | 0 | Tracks power usage by sampling periodically. |
| TAU_CALLPATH_DEPTH | 2 | Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo) |
| TAU_SAMPLING | 1 | Setting to 1 enables event-based sampling. |
| TAU_TRACK_SIGNALS | 0 | Setting to 1 generate debugging callstack info when a program crashes |
| TAU_COMM_MATRIX | 0 | Setting to 1 generates communication matrix display using context events |
| TAU_THROTTLE | 1 | Setting to 0 turns off throttling. Throttles instrumentation in lightweight routines that are called frequently |
| TAU_THROTTLE_NUMCALLS | 100000 | Specifies the number of calls before testing for throttling |
| TAU_THROTTLE_PERCALL | 10 | Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call |
| TAU_CALLSITE | 0 | Setting to 1 enables callsite profiling that shows where an instrumented function was called. Also compatible with tracing. |
| TAU_PROFILE_FORMAT | Profile | Setting to "merged" generates a single file. "snapshot" generates xml format |
| TAU_METRICS | TIME | Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>) |

# Runtime Environment Variables

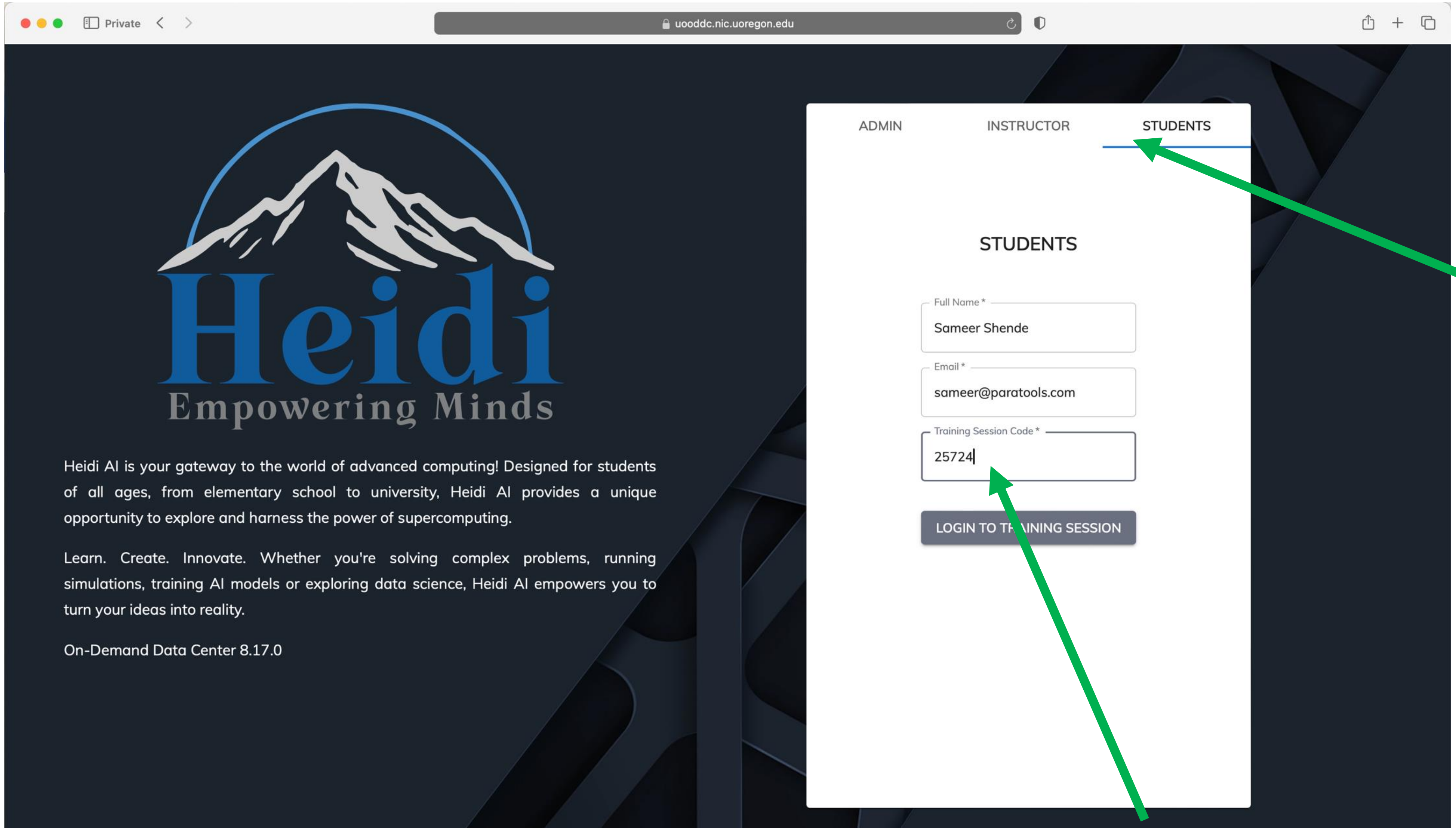
| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACE | 0 | Setting to 1 turns on tracing |
| TAU_TRACE_FORMAT | Default | Setting to “otf2” turns on TAU’s native OTF2 trace generation (configure with –otf=download) |
| TAU_EBS_UNWIND | 0 | Setting to 1 turns on unwinding the callstack during sampling (use with tau_exec –ebs or TAU_SAMPLING=1) |
| TAU_EBS_RESOLUTION | line | Setting to “function” or “file” changes the sampling resolution to function or file level respectively. |
| TAU_TRACK_LOAD | 0 | Setting to 1 tracks system load on the node |
| TAU_SELECT_FILE | Default | Setting to a file name, enables selective instrumentation based on exclude/include lists specified in the file. |
| TAU_OMPT_SUPPORT_LEVEL | basic | Setting to “full” improves resolution of OMPT TR6 regions on threads 1.. N-1. Also, “lowoverhead” option is available. |
| TAU_OMPT_RESOLVE_ADDRESS_EAGERLY | 1 | Setting to 1 is necessary for event based sampling to resolve addresses with OMPT. Setting to 0 allows the user to do offline address translation. |

# Runtime Environment Variables

| Environment Variable | Default | Description |
|---|---|---|
| TAU_TRACK_MEMORY_LEAKS | 0 | Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory) |
| TAU_EBS_SOURCE | TIME | Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1) |
| TAU_EBS_PERIOD | 100000 | Specifies the overflow count for interrupts |
| TAU_MEMDBG_ALLOC_MIN/MAX | 0 | Byte size minimum and maximum subject to bounds checking (used with TAU_MEMDBG_PROTECT_*) |
| TAU_MEMDBG_OVERHEAD | 0 | Specifies the number of bytes for TAU's memory overhead for memory debugging. |
| TAU_MEMDBG_PROTECT_BELOW/ABOVE | 0 | Setting to 1 enables tracking runtime bounds checking below or above the array bounds (requires –optMemDbg while building or tau_exec –memory) |
| TAU_MEMDBG_ZERO_MALLOC | 0 | Setting to 1 enables tracking zero byte allocations as invalid memory allocations. |
| TAU_MEMDBG_PROTECT_FREE | 0 | Setting to 1 detects invalid accesses to deallocated memory that should not be referenced until it is reallocated (requires –optMemDbg or tau_exec –memory) |
| TAU_MEMDBG_ATTEMPT_CONTINUE | 0 | Setting to 1 allows TAU to record and continue execution when a memory error occurs at runtime. |
| TAU_MEMDBG_FILL_GAP | Undefined | Initial value for gap bytes |
| TAU_MEMDBG_ALINGMENT | Sizeof(int) | Byte alignment for memory allocations |
| TAU_EVENT_THRESHOLD | 0.5 | Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max |

# Hands on examples on Aurora: TAU

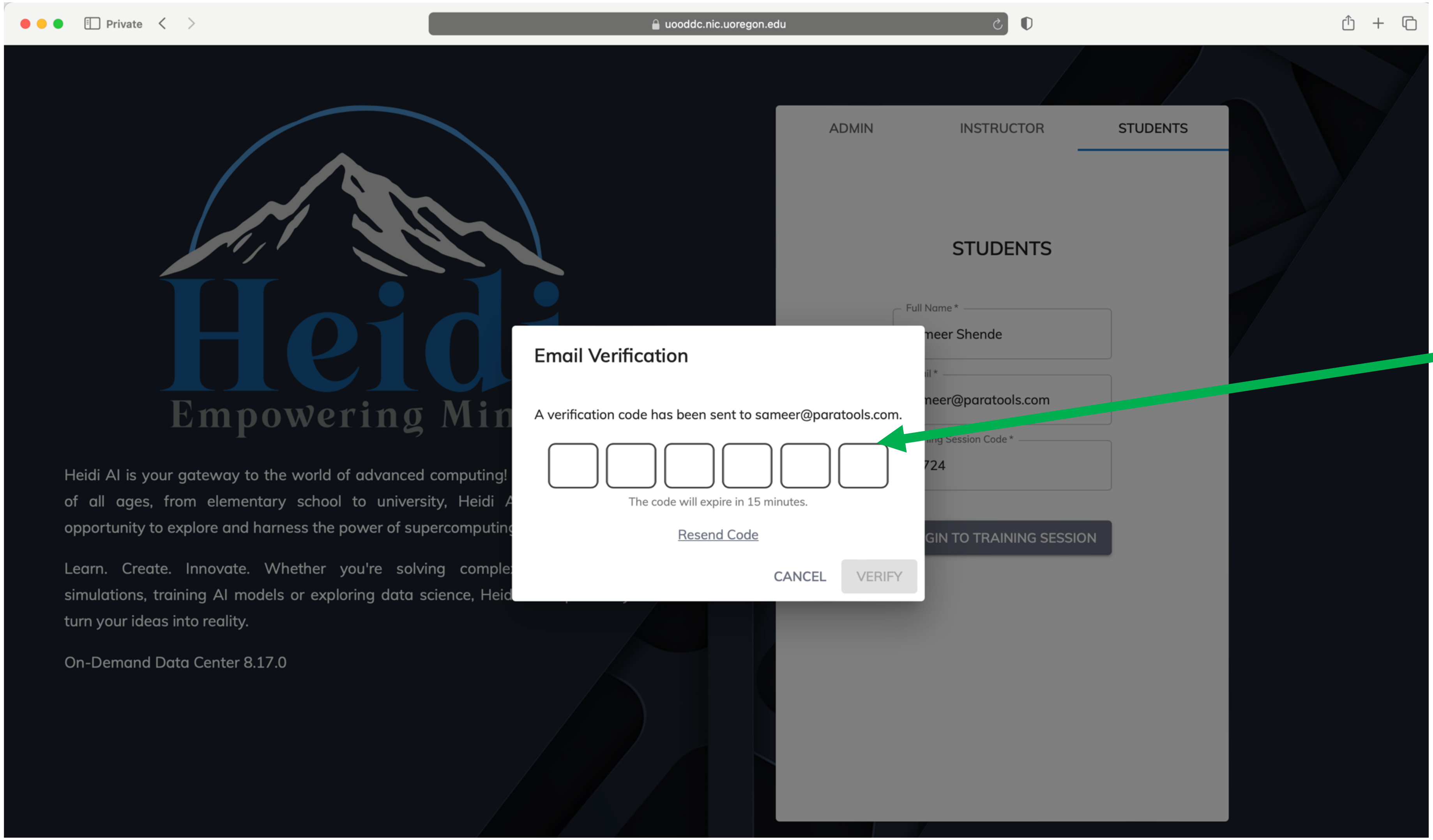
# Logging in using Heidi at https://uooddc.nic.uoregon.edu




Click here

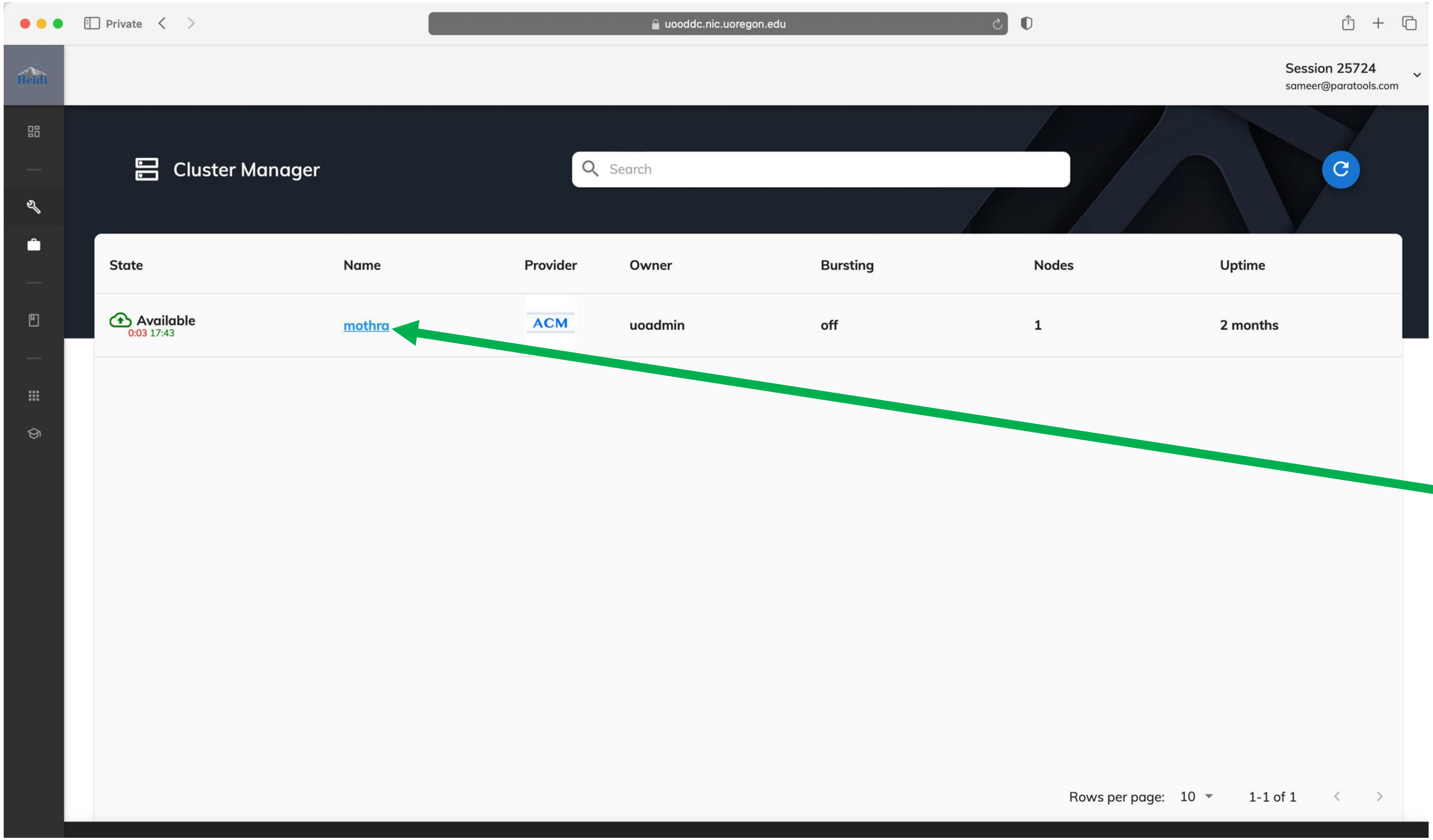
https://uooddc.nic.uoregon.edu and enter training session code **25724** and login

# 2FA: Check your email for a six digit code




Enter
code here

# Adaptive Computing’s Heidi: Cluster Manager

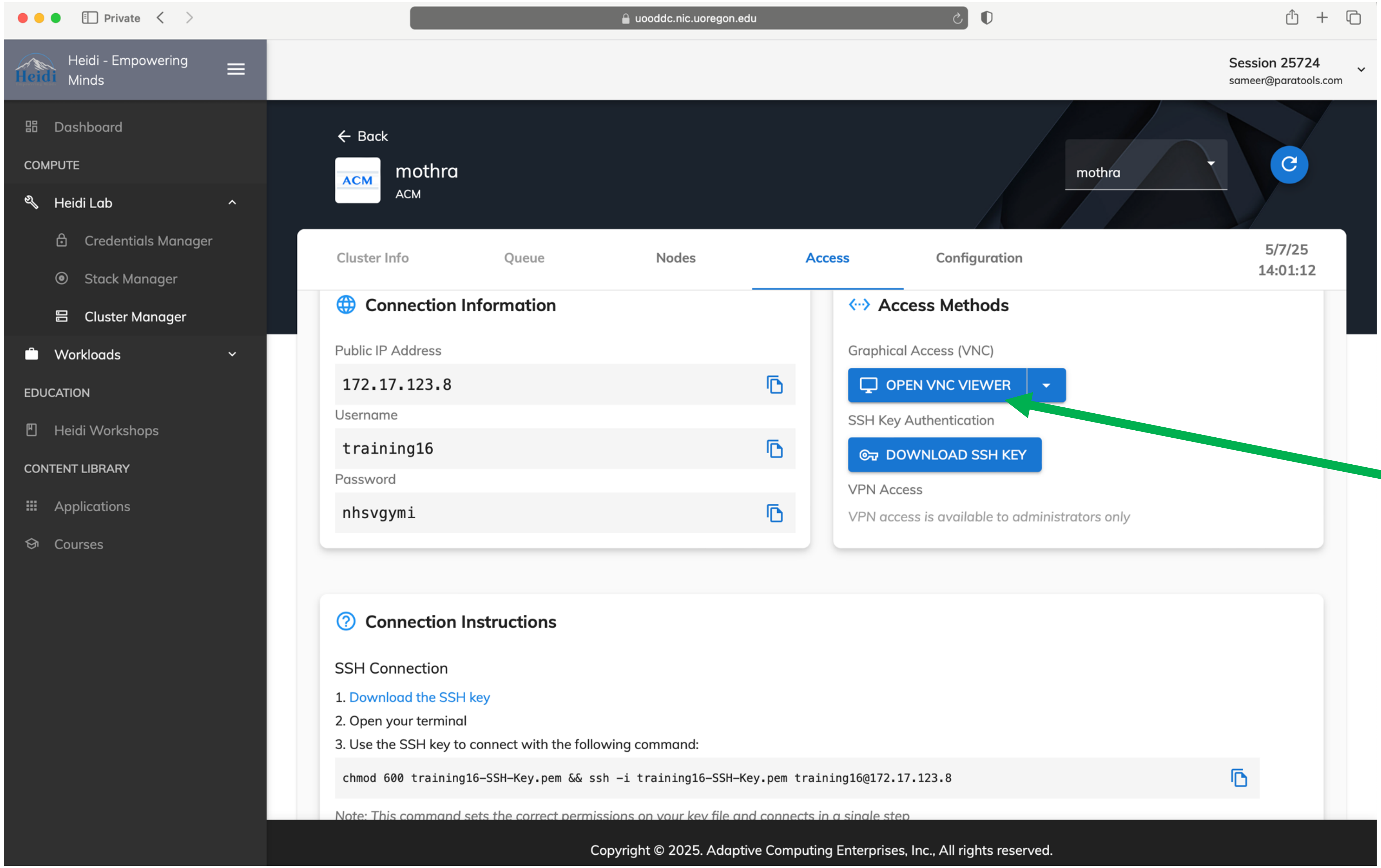



Click here



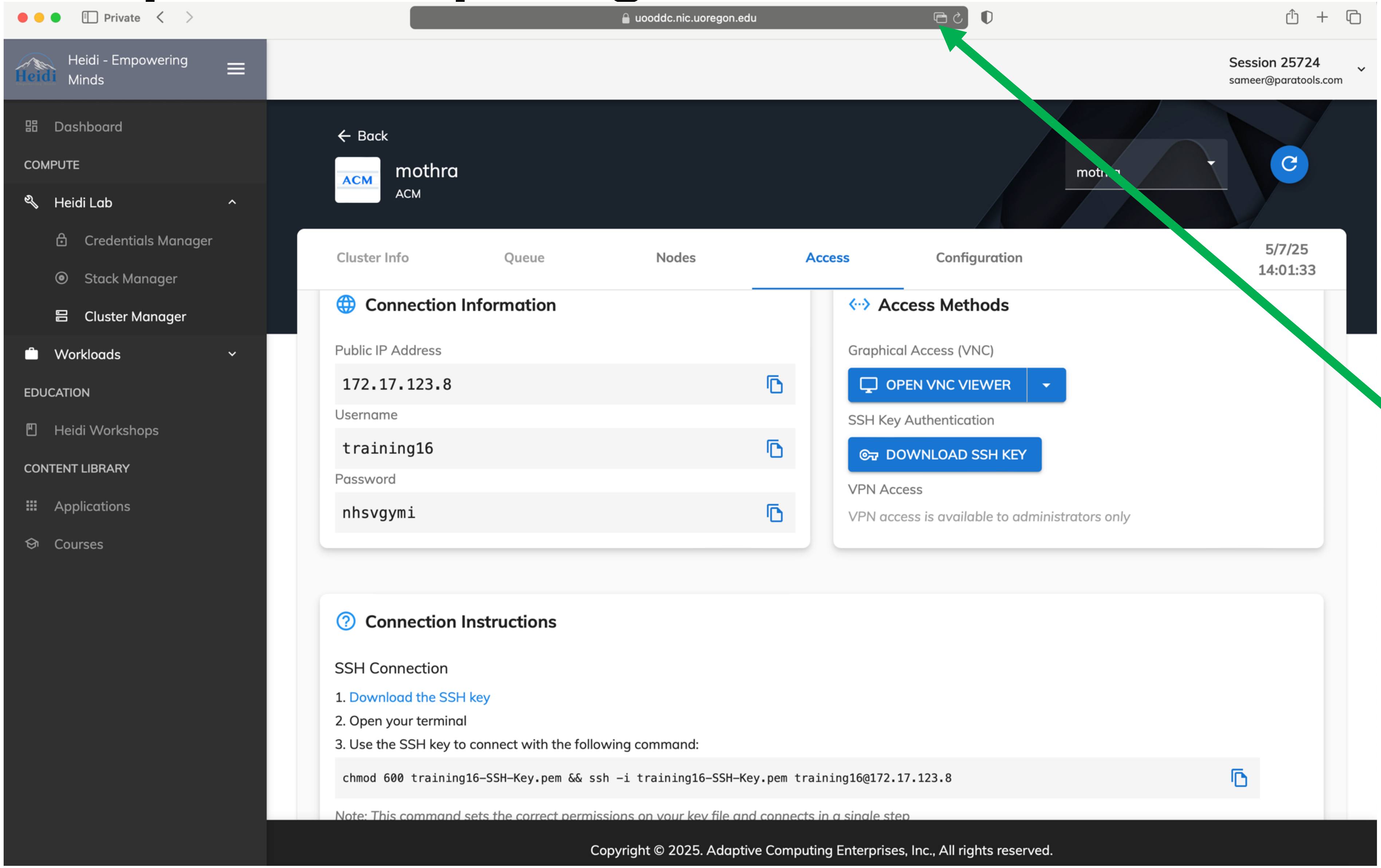
Argonne NATIONAL LABORATORY

# Adaptive Computing's Heidi: Launch VNC Viewer



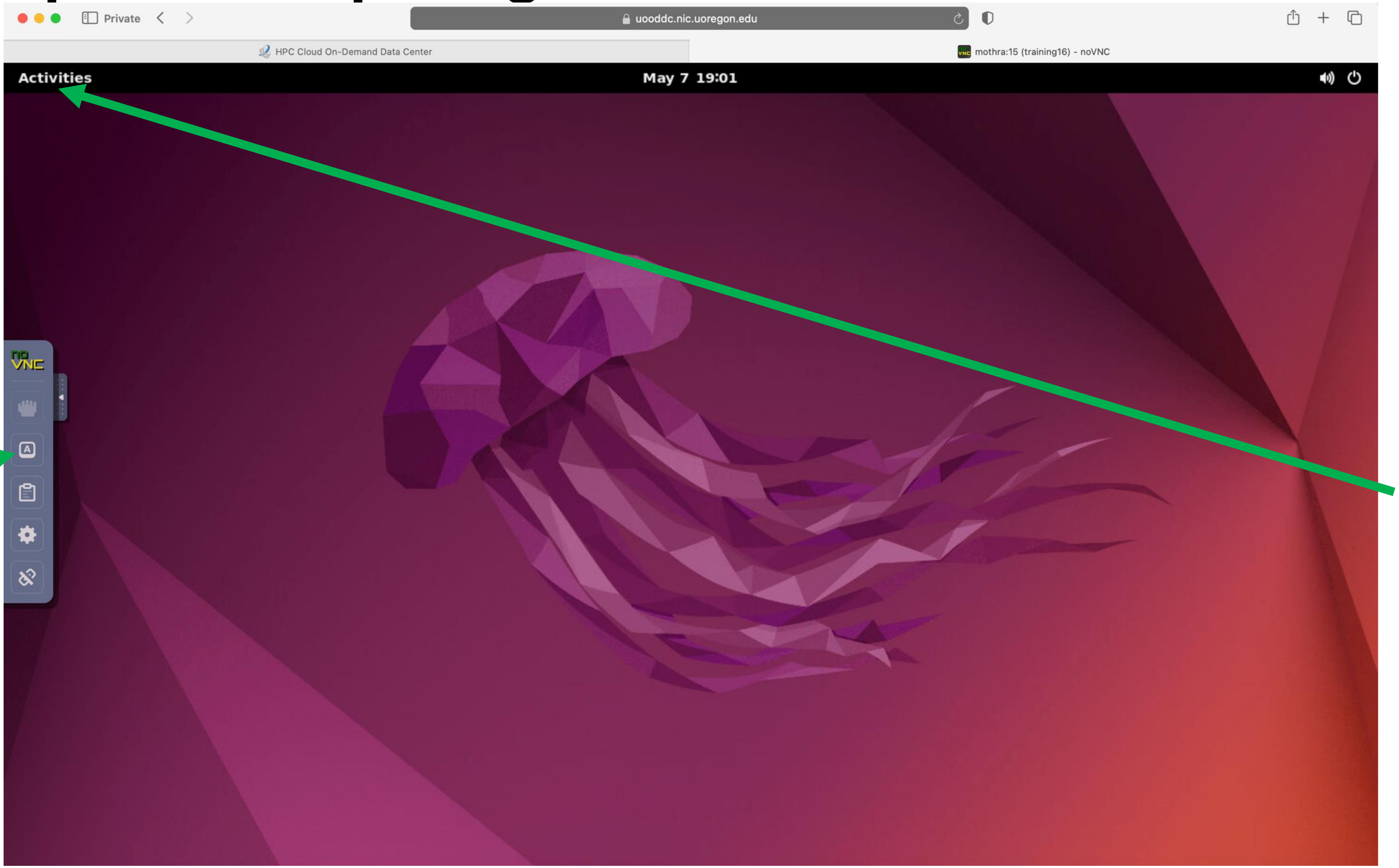

Click here

# Adaptive Computing's Heidi: Launch VNC Viewer




Popup blocked
Click here and launch popup window

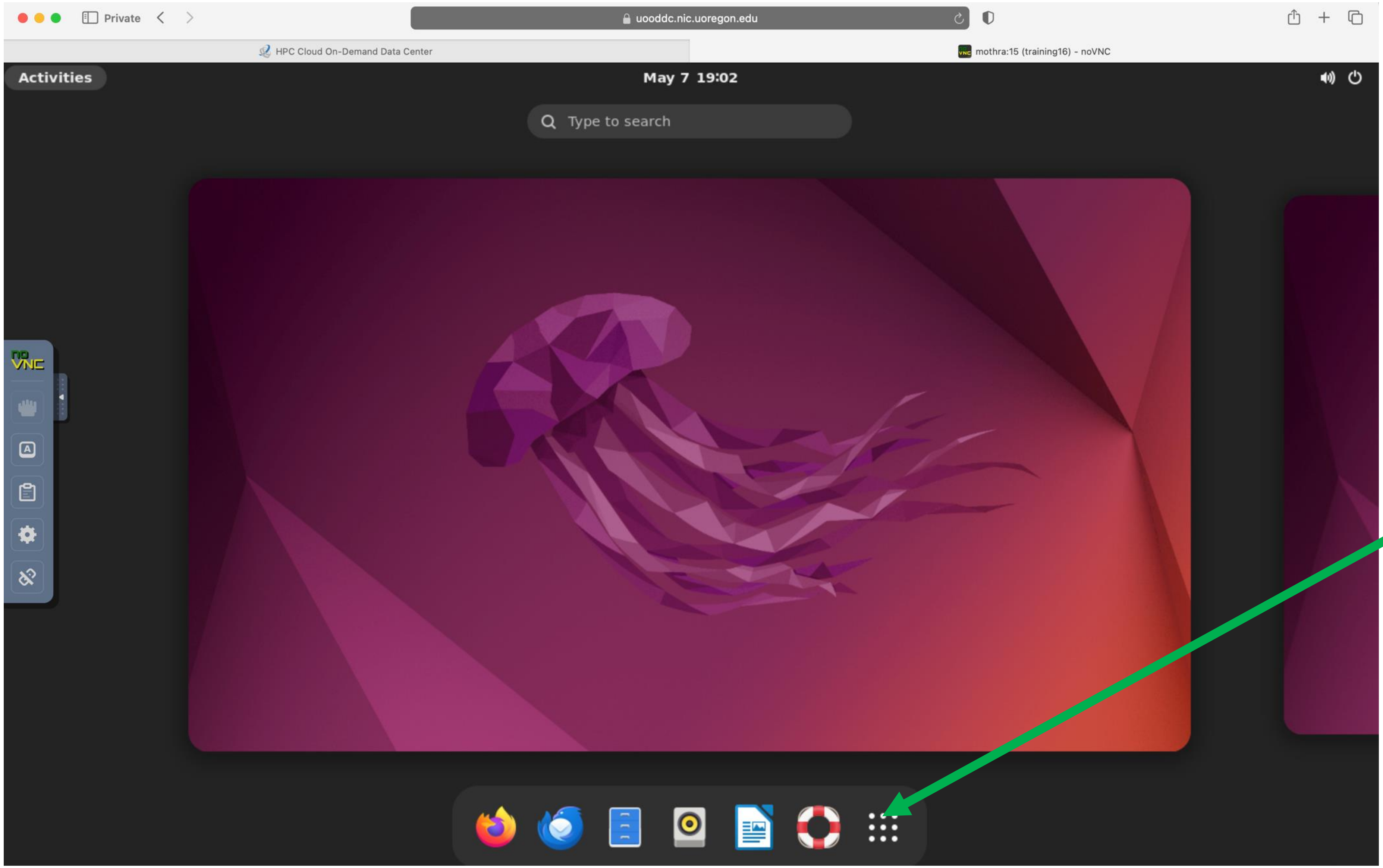
# Adaptive Computing's Heidi: Launch VNC Viewer




Clipboard to copy to/from host

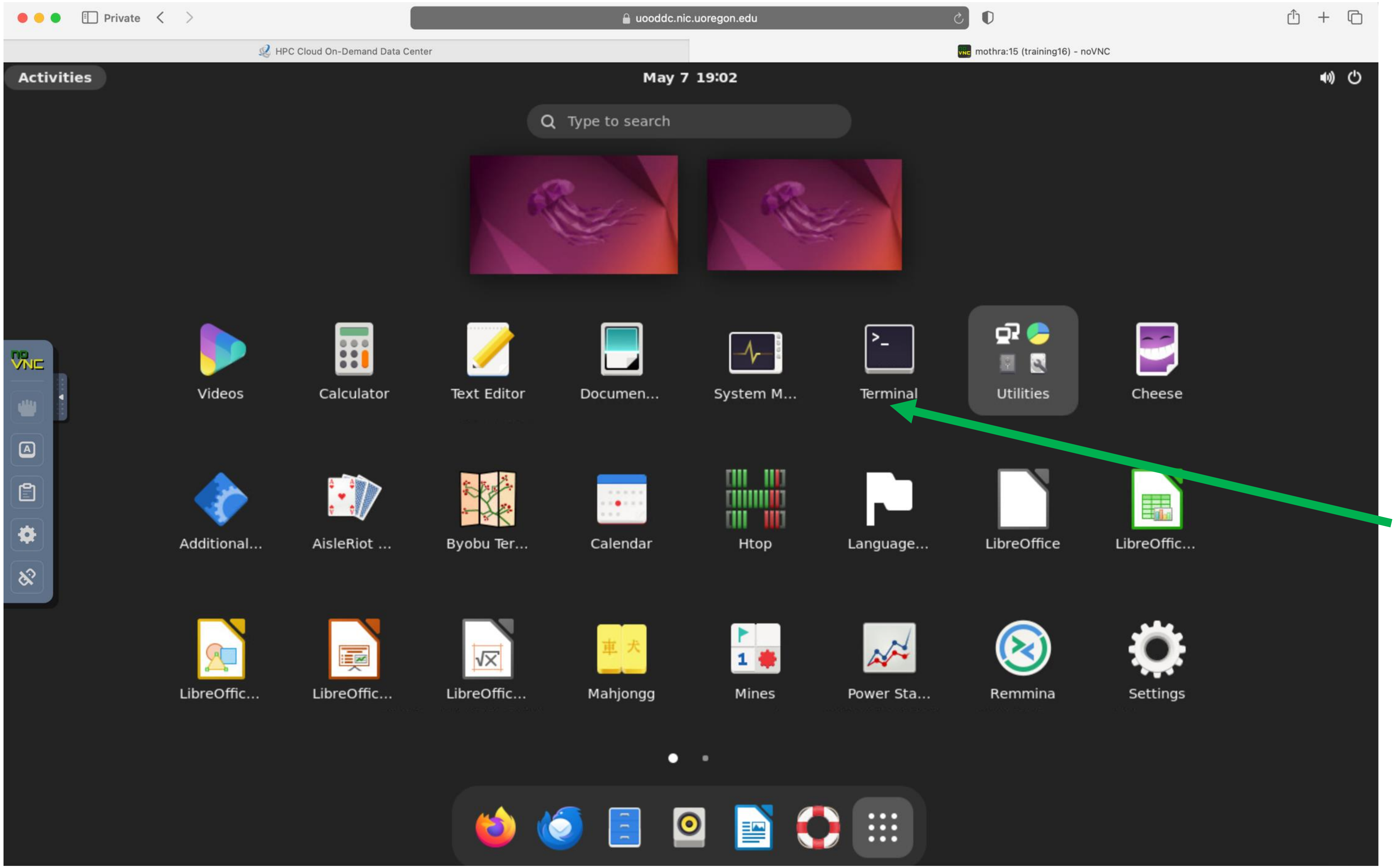
Click Activities

# Adaptive Computing’s Heidi: remote desktop



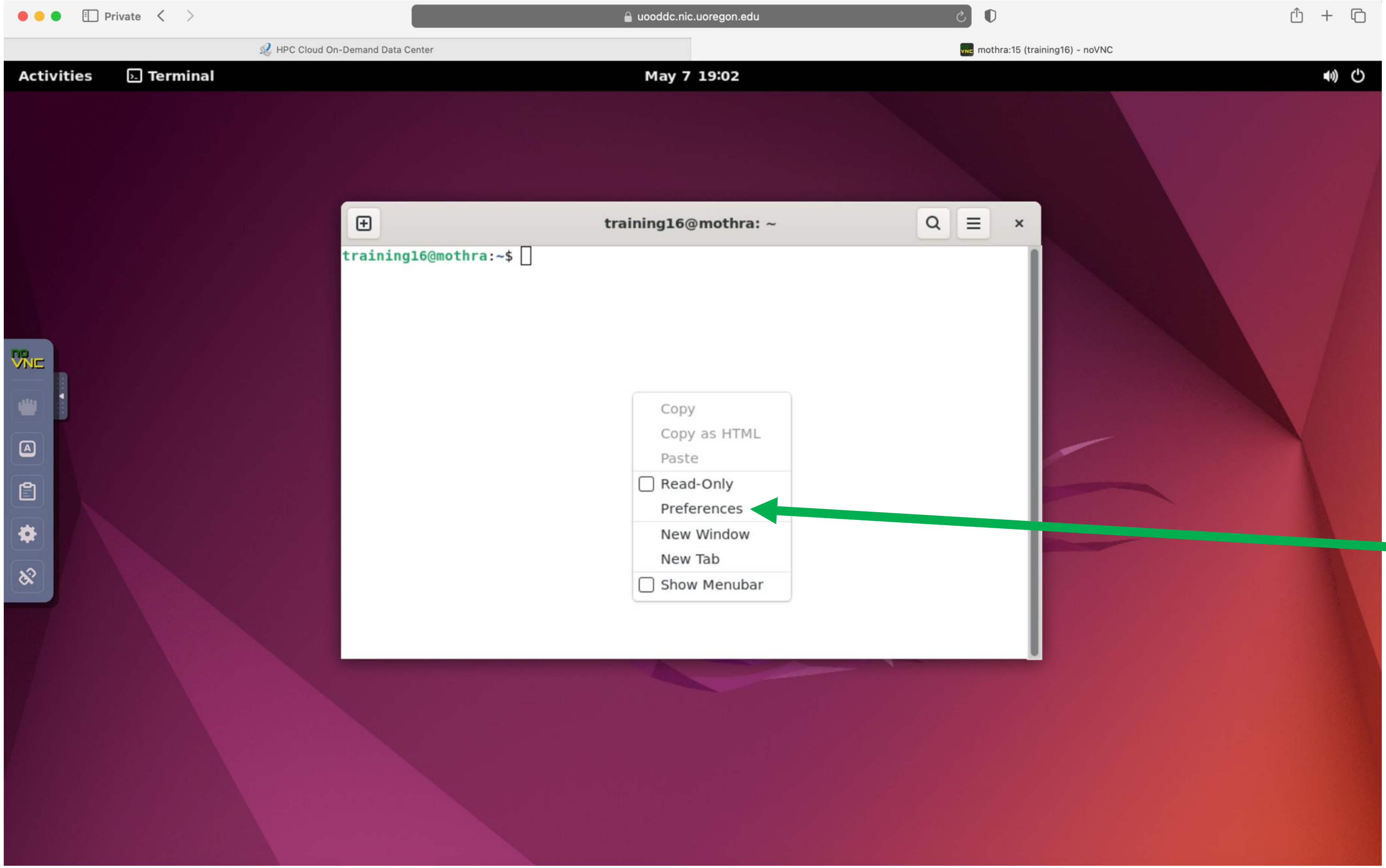

Click nine dots
to launch terminal

# Adaptive Computing's Heidi: Launch Terminal application



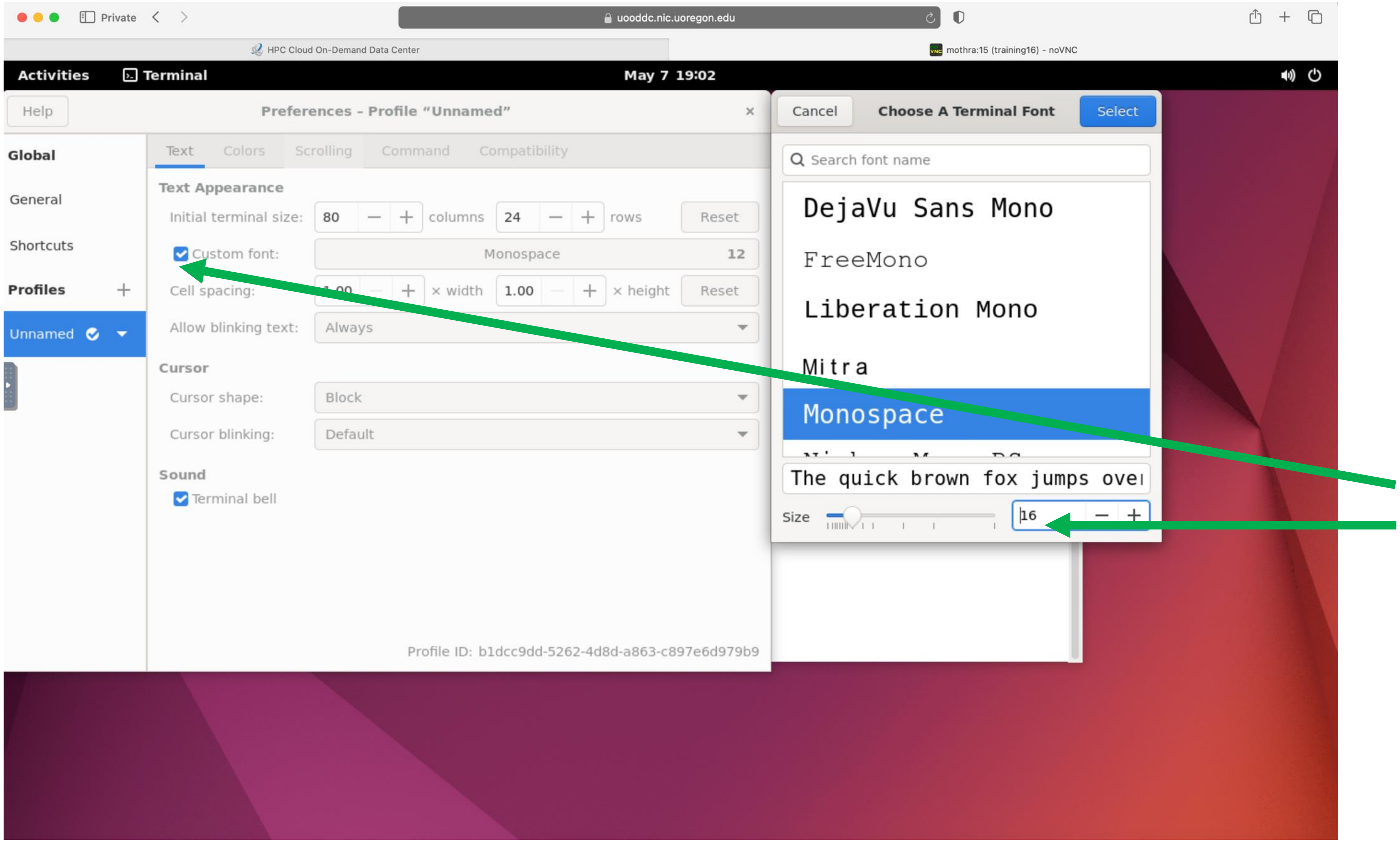

Double click and launch Terminal application

# Adaptive Computing's Heidi: Launch Terminal application



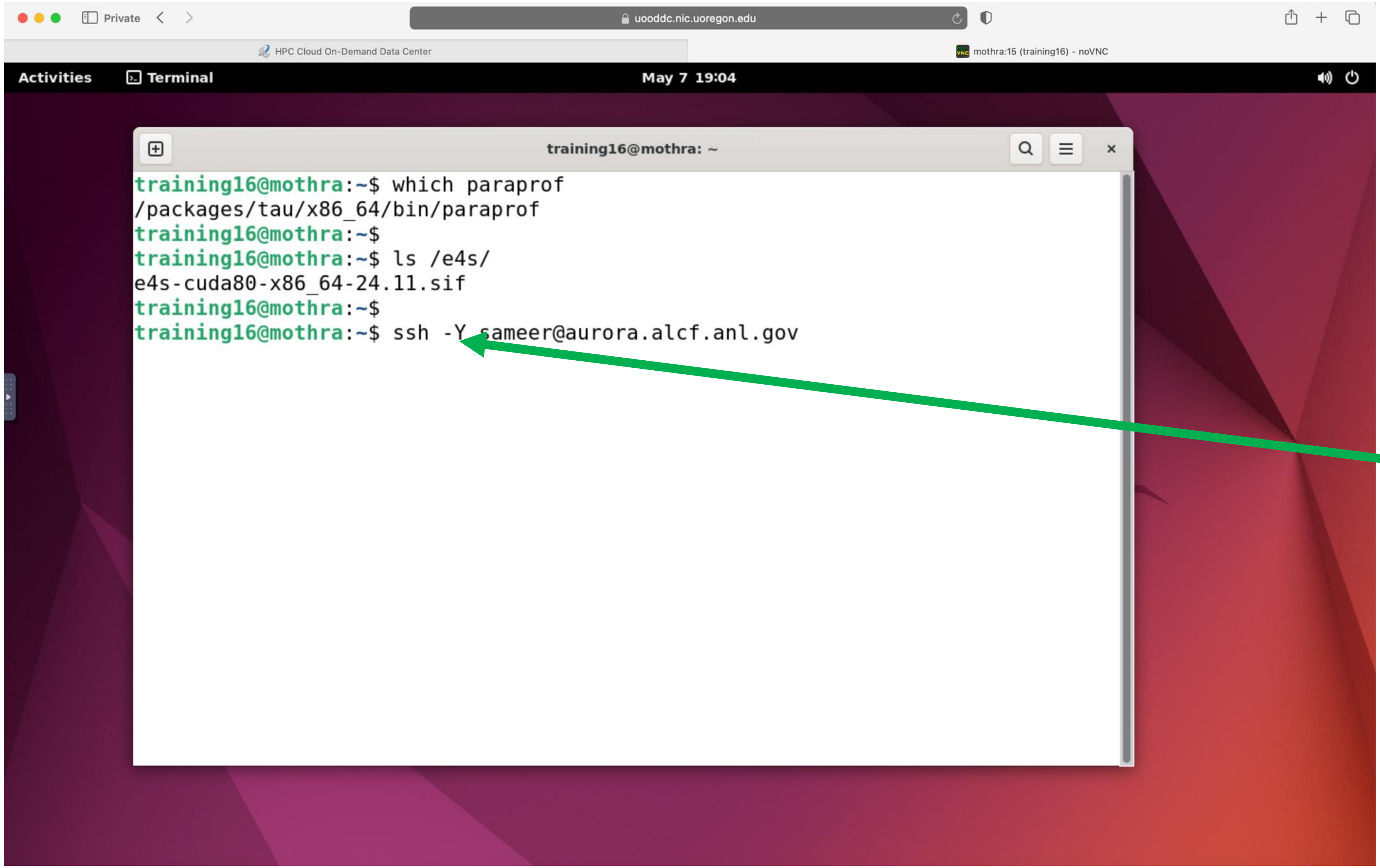

Change font size? Right click in the terminal and choose Preferences

# Adaptive Computing's Heidi: Launch Terminal application



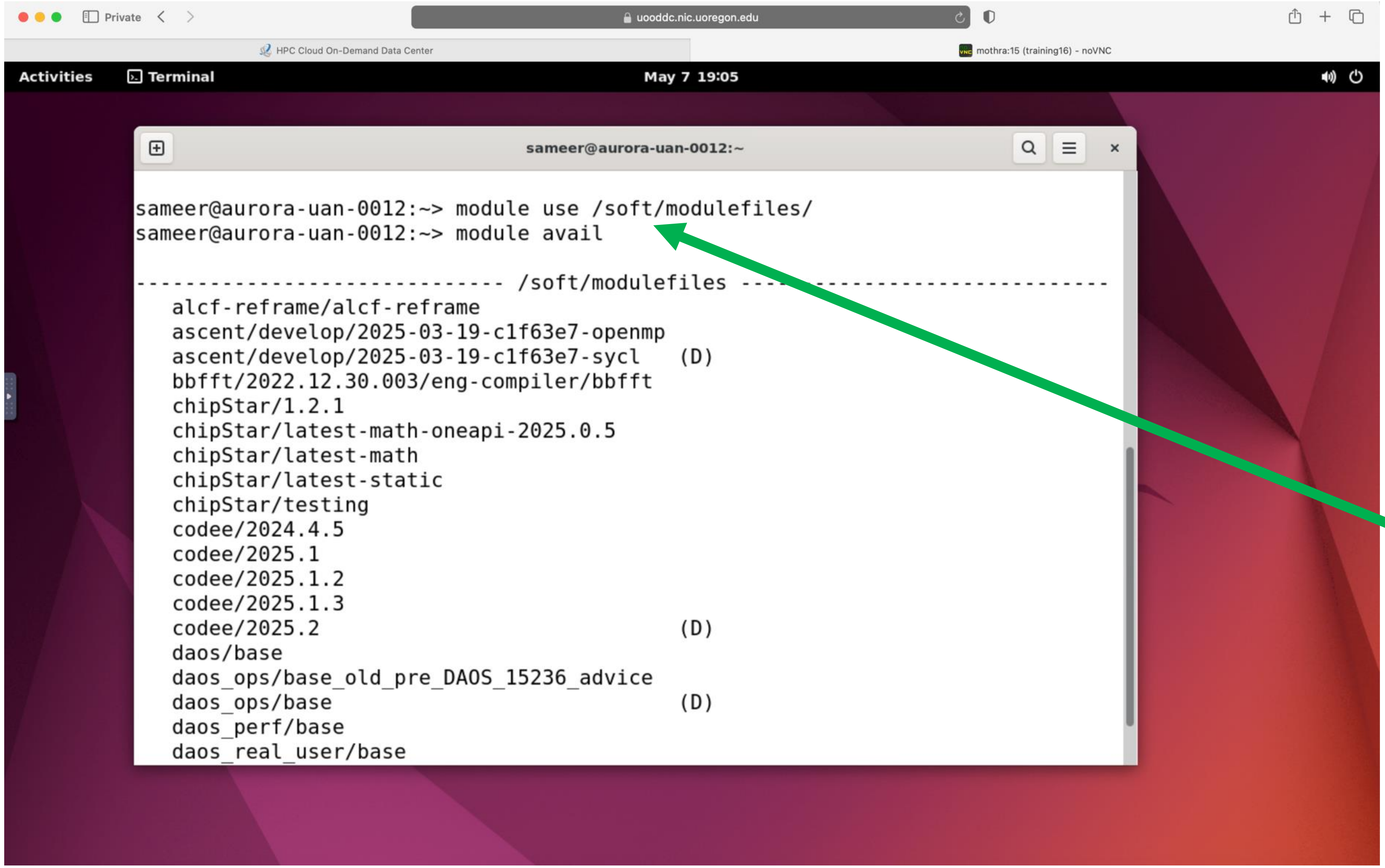

Click Custom font and choose size

# Adaptive Computing's Heidi: Logging into Aurora




ssh using -Y

# Aurora: Add a directory to the default module path

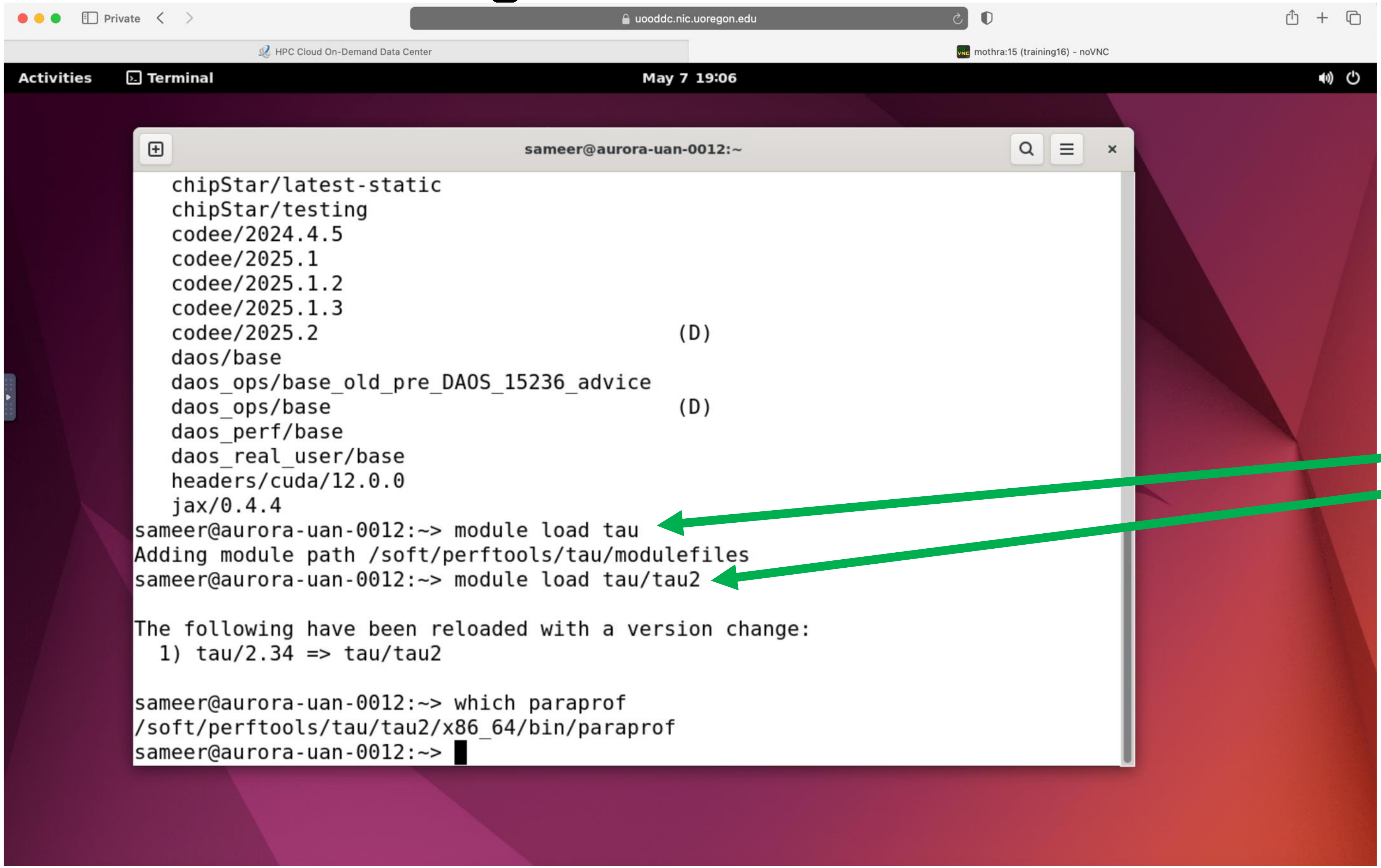



module use /soft/modulefiles



Argonne NATIONAL LABORATORY

# Aurora: Loading TAU

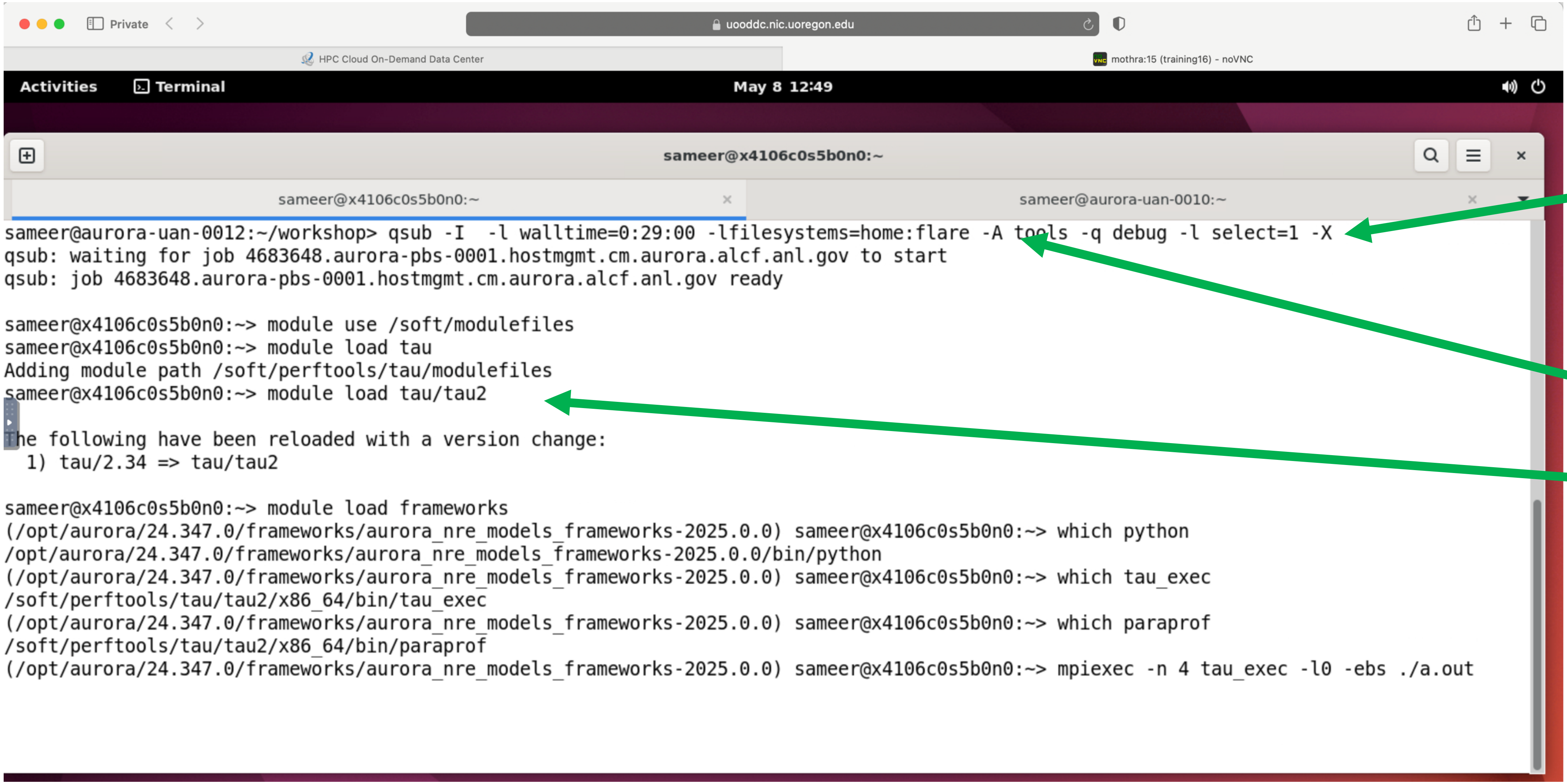



Load TAU using
module load tau
module load tau/tau2



Argonne NATIONAL LABORATORY

# Aurora: Setting up workshop examples




Use -X for GUIs

Use your allocation

Load TAU on compute nodes



Argonne NATIONAL LABORATORY

# Using TAU on Aurora

```
tar xf /soft/perftools/tau/tar/workshop.tgz
cd workshop; cat README
qsub -I  -l walltime=0:29:00 -lfilesystems=home:flare -A gpu_hack -q gpu_hack_prio
-l select=1 -X
module use /soft/modulefiles
module load tau
module load tau/tau2                    # Uses the latest git head version
module load frameworks

mpiexec -n 4 ./a.out
mpiexec -n 4 tau_exec -l0 -ebs ./a.out

mpiexec -n 4 tau_exec -l0 -ebs python ./foo.py
pprof -a  | more
paraprof
For large scale runs: export TAU_PROFILE_FORMAT="merged" and use
paraprof tauprofile.xml &
```

# Setup: Installing TAU on Laptops

Prerequisites: Java in your path

- Microsoft Windows
  - Install Java from Oracle.com
    - http://tau.uoregon.edu/tau.exe
    - Install, click on a ppk file to launch paraprof
  - macOS (arm64, Apple Silicon M series)
    - http://tau.uoregon.edu/java_arm64.dmg
    - http://tau.uoregon.edu/tau_arm64.dmg
- macOS (x86_64)
  - Install Java 11.0.3:
    - Download and install http://tau.uoregon.edu/java.dmg
    - If you have multiple Java installations, add to your ~/.zshrc (or ~/.bashrc as appropriate):
    - export PATH=/Library/Java/JavaVirtualMachines/jdk-11.0.3.jdk/Contents/Home/bin:$PATH
  - Download and install TAU (copy to /Applications from dmg):
    - http://tau.uoregon.edu/tau.dmg
    - export PATH=/Applications/TAU/tau/apple/bin:$PATH
    - paraprof  app.ppk &
  - Linux (http://tau.uoregon.edu/tau.tgz)
    - ./configure; make install; export PATH=<taudir>/x86_64/bin:$PATH; paraprof app.ppk &

# TAU: Quickstart Guide for aurora.alcf.anl.gov

```
% tar xf /soft/perftools/tau/tar/workshop.tgz; cd workshop; cat README
% qsub -I  -l walltime=0:29:00 -lfilesystems=home:flare –A gpu_hack –q gpu_hack_prio -l select=1 -X
% module use /soft/modulefiles; module load tau; module load tau/tau2
```

- Un-instrumented run with MPI          % mpiexec –np N ./a.out
- Profiling an un-instrumented application (use tau_exec –ebs with any of the following for event-based sampling):
- MPI without GPUs:                % mpiexec -np N tau_exec -ebs ./a.out
- DPC++/SYCL with MPI:             % mpiexec -np N tau_exec –T level_zero  -l0 ./a.out
- DPC++/SYCL without MPI:          % tau_exec –T serial –l0 ./a.out
- OpenMP: export TAU_OMPT_SUPPORT_LEVEL=full
- OpenMP with MPI:                 % mpiexec -np N tau_exec –T ompt  –ompt ./a.out
- OpenMP without MPI:              % tau_exec –T serial -ompt./a.out

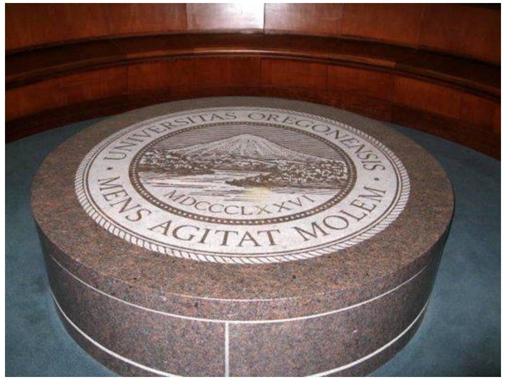
Analysis:                          % pprof –a –m | more;  % paraprof (GUI)

Tracing:

```
Perfetto.dev:   % export TAU_TRACE=1; mpiexec –np N tau_exec ./a.out; tau_treemerge.pl;
                % tau_trace2json tau.trc tau.edf –chrome –ignoreatomic –o app.json
   Chrome browser: chrome://tracing   (Load -> app.json) or https://Perfetto.dev
```

- Jumpshot:     % export TAU_TRACE=1; mpiexec –np N tau_exec [Options]./a.out;
          % tau_treemerge.pl; tau2slog2 tau.trc tau.edf –o app.slog2; jumpshot app.slog2 &

# Performance Research Laboratory, University of Oregon, Eugene




www.uoregon.edu

# Support Acknowledgements

- US Department of Energy (DOE)
  - — ANL
  - — Office of Science contracts, ECP
  - — SciDAC, LBL contracts
  - — LLNL-LANL-SNL ASC/NNSA contract
  - — Battelle, PNNL and ORNL contract
- Department of Defense (DoD)
  - — PETTT, HPCMP
- National Science Foundation (NSF)
  - — SI2-SSI, Glassbox, E4S Workshop
- NASA
- Intel
- CEA, France
- Partners:
  - —University of Oregon
  - —The Ohio State University
  - —ParaTools, Inc.
  - —University of Tennessee, Knoxville
  - —T.U. Dresden, GWT
  - —Jülich Supercomputing Center

# Acknowledgment


- *This work was supported by the U.S. Department of Energy, Office of Science, Advanced Computing Research, through the Next-Generation Scientific Software Technologies (NGSST) under contract DE-AC02-AC05-00OR22725 and DOE SBIR DE-SC0022502.*


U.S. DEPARTMENT OF ENERGY | Office of Science

- https://science.osti.gov/ascr
- https://pesoproject.org
- https://ascr-step.org
- https://hpsf.io
- https://www.energy.gov/technologytransitions/sbirsttr

STEP SOFTWARE TOOLS ECOSYSTEM PROJECT

PESO



Argonne NATIONAL LABORATORY

# Thank you

https://www.exascaleproject.org

*This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.*

EXASCALE COMPUTING PROJECT

**Thank you** to all collaborators in the ECP and broader computational science communities. The work discussed in this presentation represents creative contributions of many people who are passionately working toward next-generation computational science.