Performance Analysis of HPX using APEX and the TAU Performance System®

Sameer Shende, Kevin Huck

sameer@cs.uoregon.edu, khuck@cs.uoregon.edu

http://tau.uoregon.edu

Download slides from: http://tau.uoregon.edu/APEX.pdf

UNIVERSITY OF OREGON

Outline

- Introduction to TAU
- Instrumentation: PDT, MPI, OpenMP, **tau_exec**
- I/O and Memory evaluation
- PAPI
- Demonstration of analysis tools: ParaProf, TAUdb and PerfExplorer, Vampir and Jumpshot
- Introduction to APEX
- Building HPX with APEX & TAU
- Program Examples

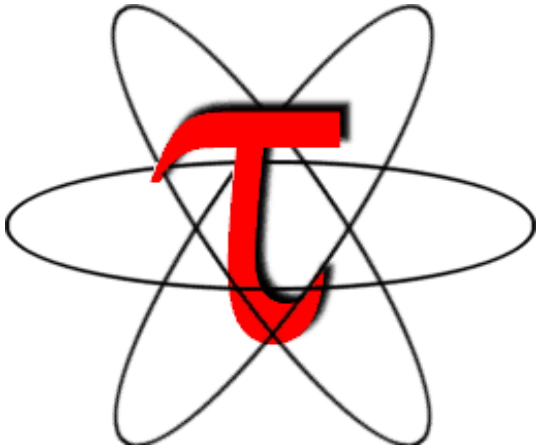
Tutorial Goals

This tutorial is an introduction to portable performance evaluation tools. You should leave here with a better understanding of…

- Concepts and steps involved in performance evaluation
- Understanding key concepts in understanding code performance
- How to collect and analyze data from hardware performance counters (PAPI)
- How to instrument your programs with TAU
- Measurement options provided by TAU
- Environment variables used for choosing metrics, generating performance data
- How to use ParaProf, TAU's profile browser
- General familiarity with TAU use for Fortran, C++, C, and mixed language
- How to generate trace data in different formats

TAU Performance System®

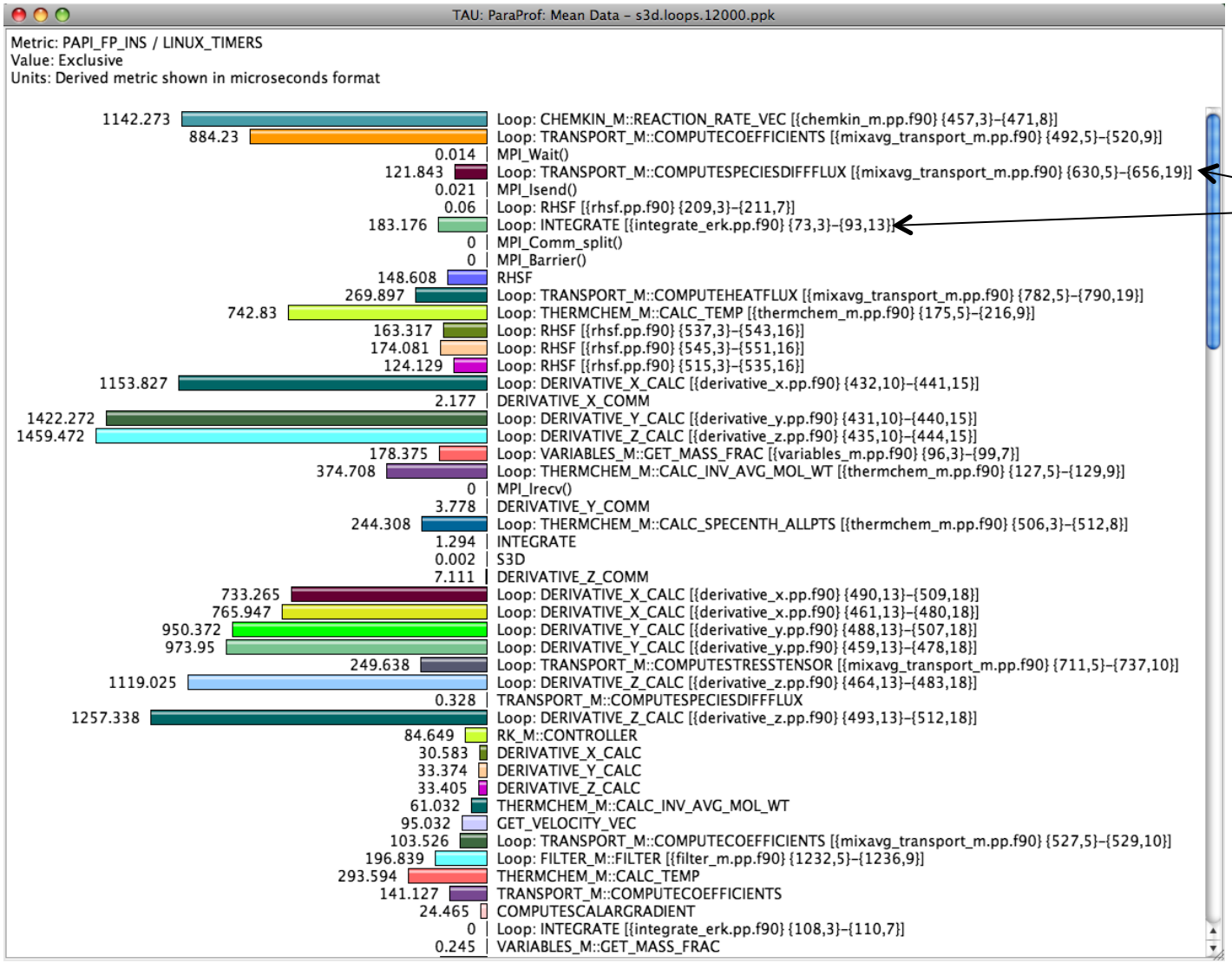
- **Tuning and Analysis Utilities (20+ year project)**
- **Comprehensive performance profiling and tracing**
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- **Integrated performance toolkit**
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- **Integrates with application frameworks**



Understanding Application Performance using TAU

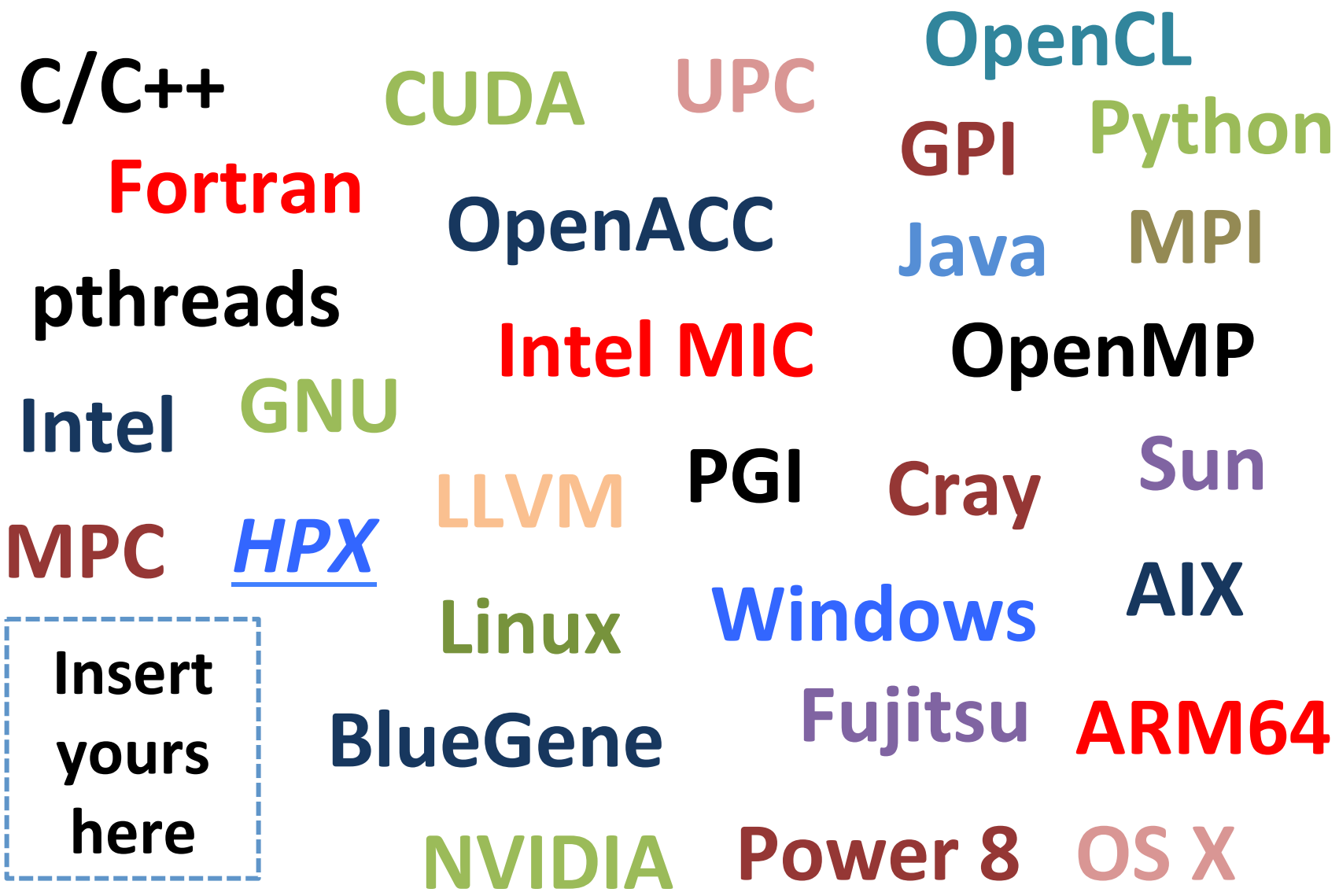
- **How much time** is spent in each application routine and outer loops? Within loops, what is the contribution of each *statement*?
- **How many instructions** are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?
- **What is the memory usage** of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
- **What are the I/O characteristics** of the code? What is the peak read and write *bandwidth* of individual calls, total volume?
- **What is the contribution of each *phase*** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- **How does the application *scale*?** What is the efficiency, runtime breakdown of performance across different core counts?

Identifying Potential Bottlenecks



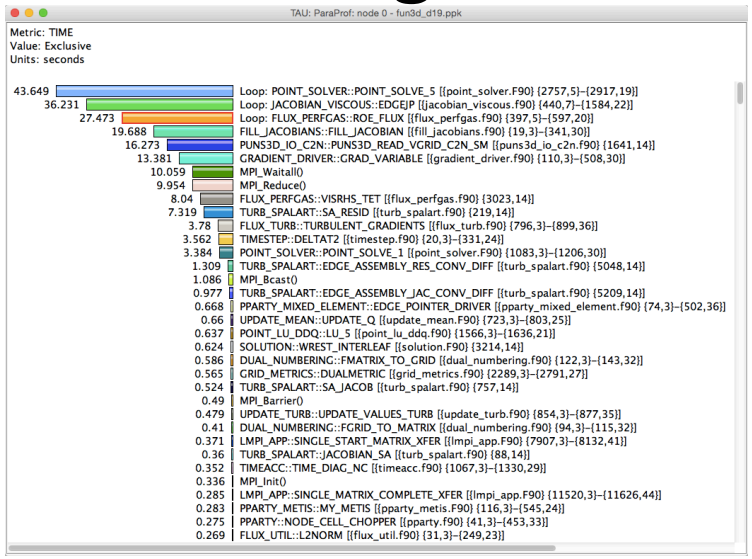
low mflops in loops?

What does TAU support?

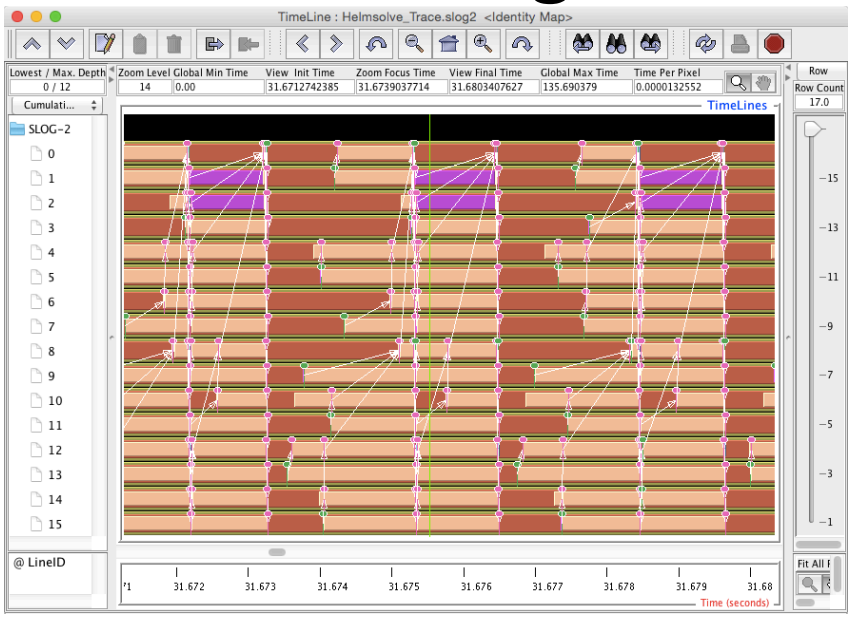


Profiling and Tracing

Profiling



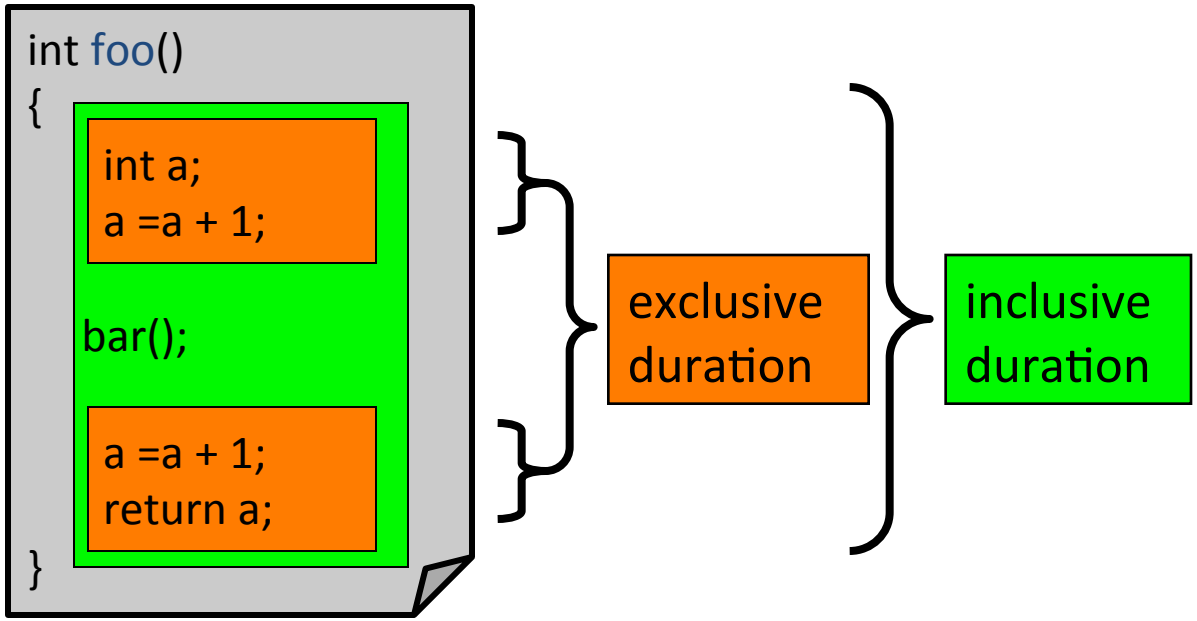
Tracing



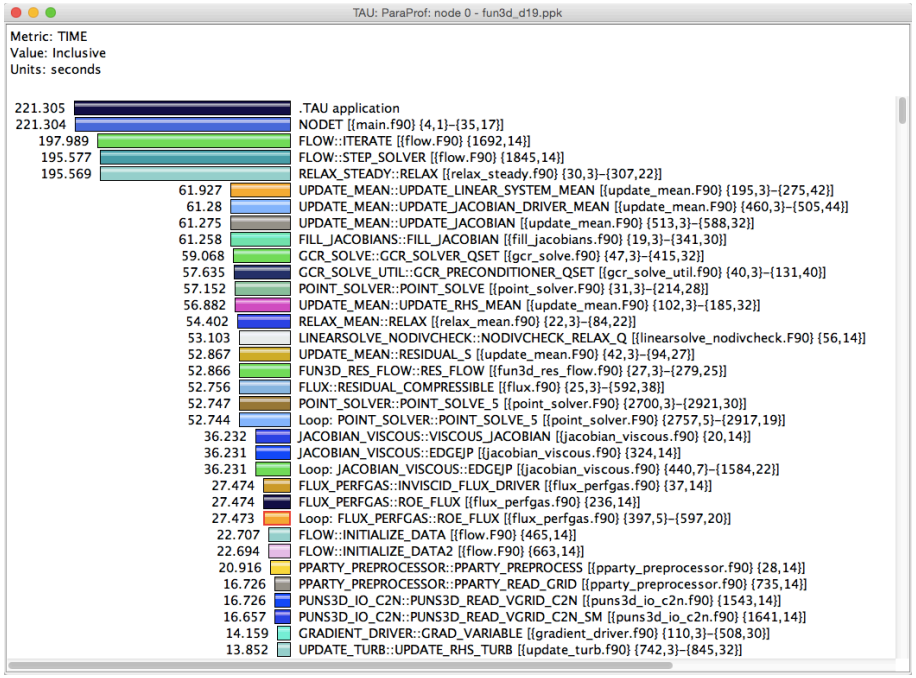
- Profiling and tracing
  - **Profiling** shows you **how much** (total) time was spent in each routine
  - **Tracing** shows you **when** the events take place on a timeline

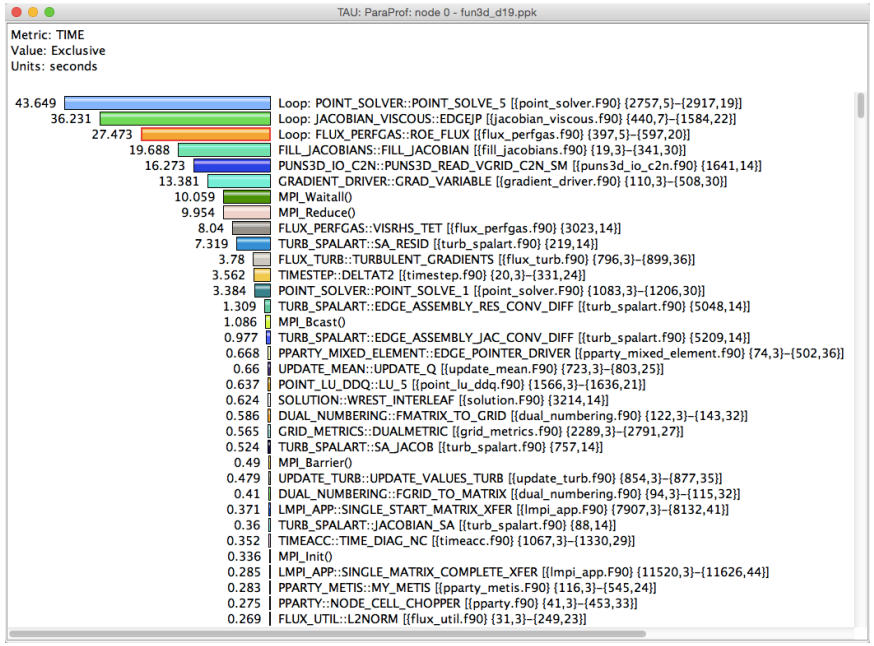
Inclusive vs. Exclusive Measurements

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



Inclusive vs. Exclusive Measurements

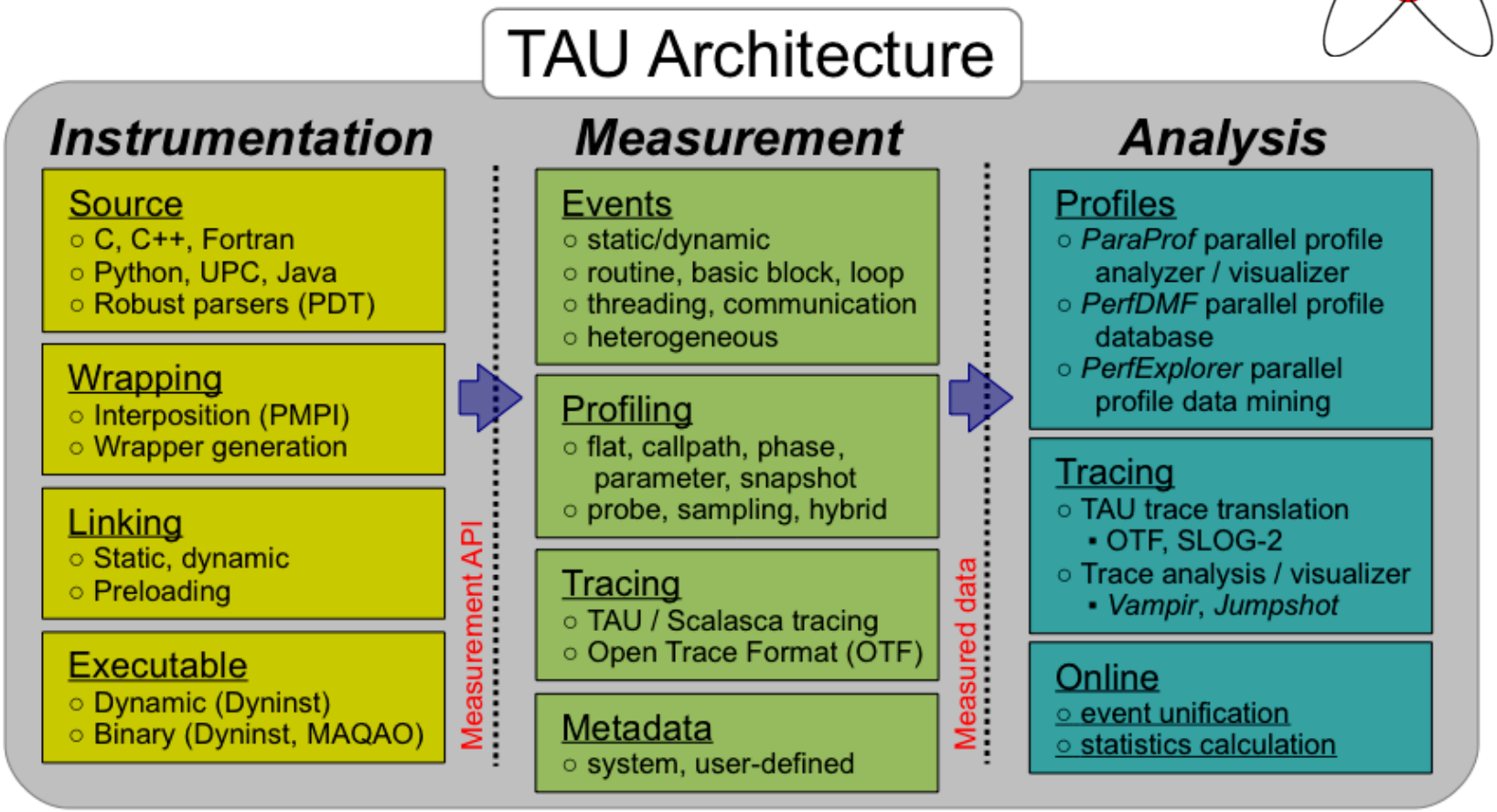




Inclusive time

Exclusive time

TAU Architecture and Workflow



TAU Architecture and Workflow

- **Instrumentation: Add probes to perform measurements**
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable

- **Measurement: Profiling or tracing using various metrics**
  - Direct instrumentation (Interval events measure exclusive or inclusive duration)
  - Indirect instrumentation (Sampling measures statement level contribution)
  - Throttling and runtime control of low-level events that execute frequently
  - Per-thread storage of performance data
  - Interface with external packages (e.g. PAPI hw performance counter library)

- **Analysis: Visualization of profiles and traces**
  - 3D visualization of profile data in paraprof or perfexplorer tools
  - Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)

Instrumentation

- **Direct and indirect performance observation**
  - Instrumentation invokes performance measurement
  - Direct measurement with *probes*
  - Indirect measurement with periodic sampling or hardware performance counter overflow interrupts
  - Events measure performance data, metadata, context, etc.

- **User-defined events**
  - **Interval** (start/stop) events to measure exclusive & inclusive duration
  - **Atomic events** take measurements at a single point
    - Measures total, samples, min/max/mean/std. deviation statistics
  - **Context events** are atomic events with executing context
    - Measures above statistics for a given calling path

Direct Observation Events

- **Interval events (begin/end events)**
  - Measures exclusive & inclusive durations between events
  - Metrics monotonically increase
  - Example: Wall-clock timer
- **Atomic events (trigger with data value)**
  - Used to capture performance data state
  - Shows extent of variation of triggered values (min/max/mean)
  - Example: heap memory consumed at a particular point
- **Code events**
  - Routines, classes, templates
  - Statement-level blocks, loops
  - Example: for-loop begin/end

Direct Instrumentation Options in TAU

- **Source Code Instrumentation**
  - Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  - Compiler generates instrumented object code
  - Manual instrumentation
- **Library Level Instrumentation**
  - Statically or dynamically linked wrapper libraries
  - MPI, I/O, memory, etc.
  - Wrapping external libraries where source is not available
- **Runtime pre-loading and interception of library calls**
- **Binary Code instrumentation**
  - Rewrite the binary, runtime instrumentation
- **Virtual Machine, Interpreter, OS level instrumentation**

TAU Execution Command (tau_exec)

Uninstrumented execution

- % mpirun -np 256 ./a.out

Track GPU operations

- % mpirun –np 256 tau_exec –cupti ./a.out
- % mpirun –np 256 tau_exec –cupti -um ./a.out (for Unified Memory)
- % mpirun –np 256 tau_exec –opencl ./a.out
- % mpirun –np 256 tau_exec –openacc ./a.out

Track MPI performance

- % mpirun -np 256 tau_exec ./a.out
- Track OpenMP, I/O, and MPI performance (MPI enabled by default)
- % mpirun -np 256 tau_exec –ompt -io ./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 –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>

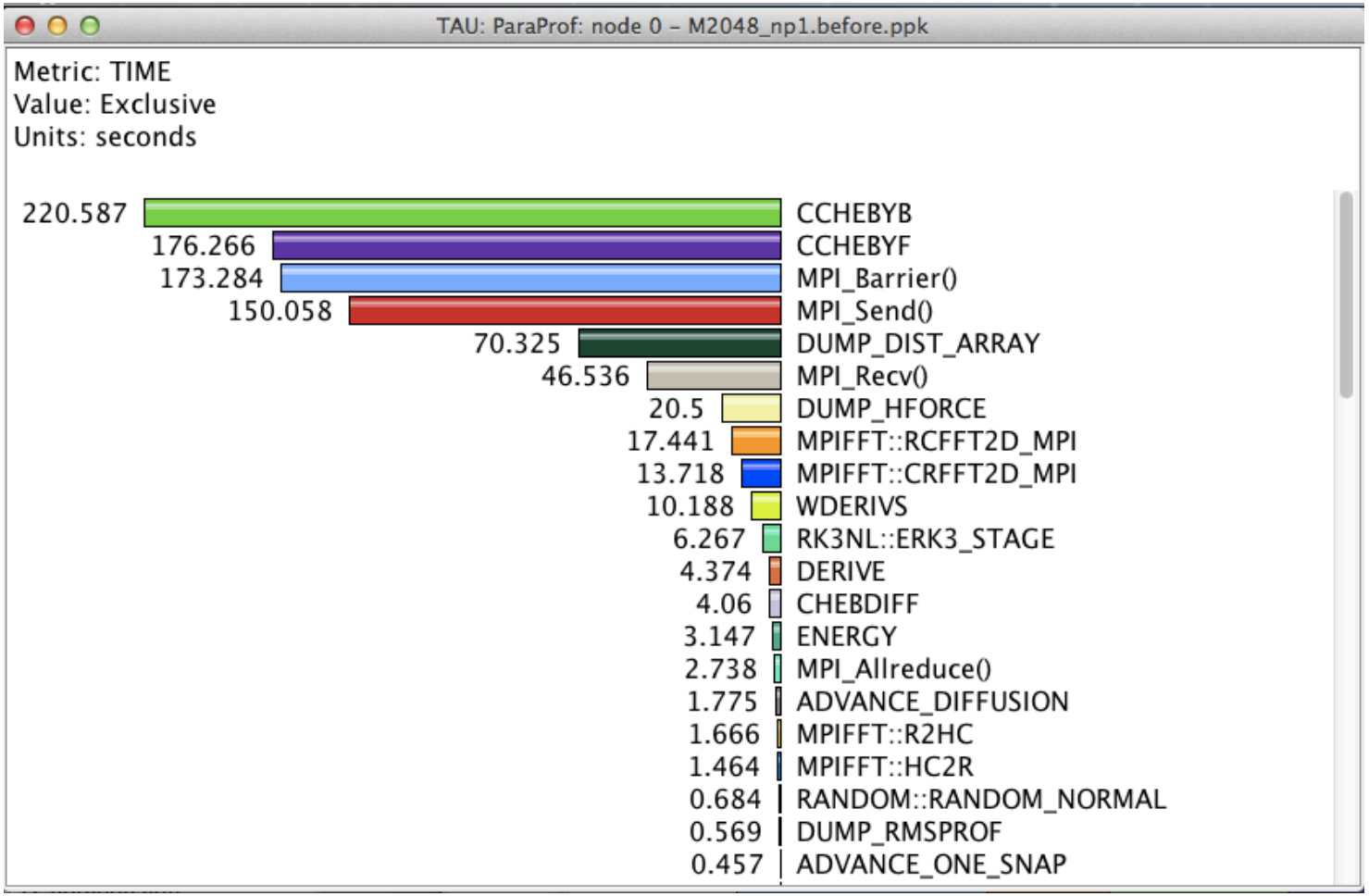
Configuration Tags for tau_exec

```
% ./configure -pdt=<dir> -mpi -papi=<dir>; make install
```
Creates in $TAU:
```
Makefile.tau-papi-mpi-pdt(Configuration parameters in stub makefile)
shared-papi-mpi-pdt/libTAU.so

% ./configure -pdt=<dir> -mpi; make install
```
creates
```
Makefile.tau-mpi-pdt
shared-mpi-pdt/libTAU.so
```
To explicitly choose preloading of shared-<options>/libTAU.so change:
```
% mpirun -np 256 ./a.out      to
% mpirun -np 256  tau_exec -T <comma_separated_options> ./a.out

% mpirun -np 256  tau_exec -T papi,mpi,pdt ./a.out
```
Preloads `$TAU/shared-papi-mpi-pdt/libTAU.so`
```
% mpirun -np 256  tau_exec -T papi ./a.out
```
Preloads `$TAU/shared-papi-mpi-pdt/libTAU.so by matching.`
```
% mpirun -np 256 tau_exec -T papi,mpi,pdt -s ./a.out
```
Does not execute the program. Just displays the library that it will preload if executed without the –s option.

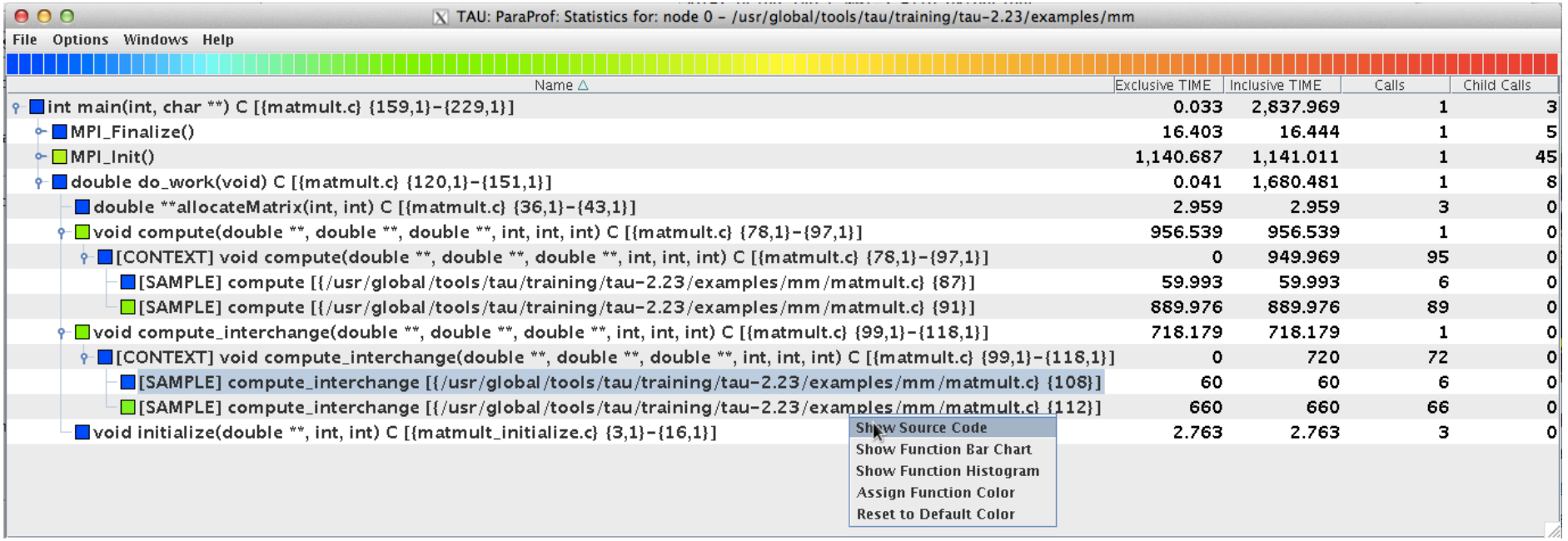
NOTE: -mpi configuration is selected by default. Use –T serial for Sequential programs.

Routine Level Profile

How much time is spent in each application routine?

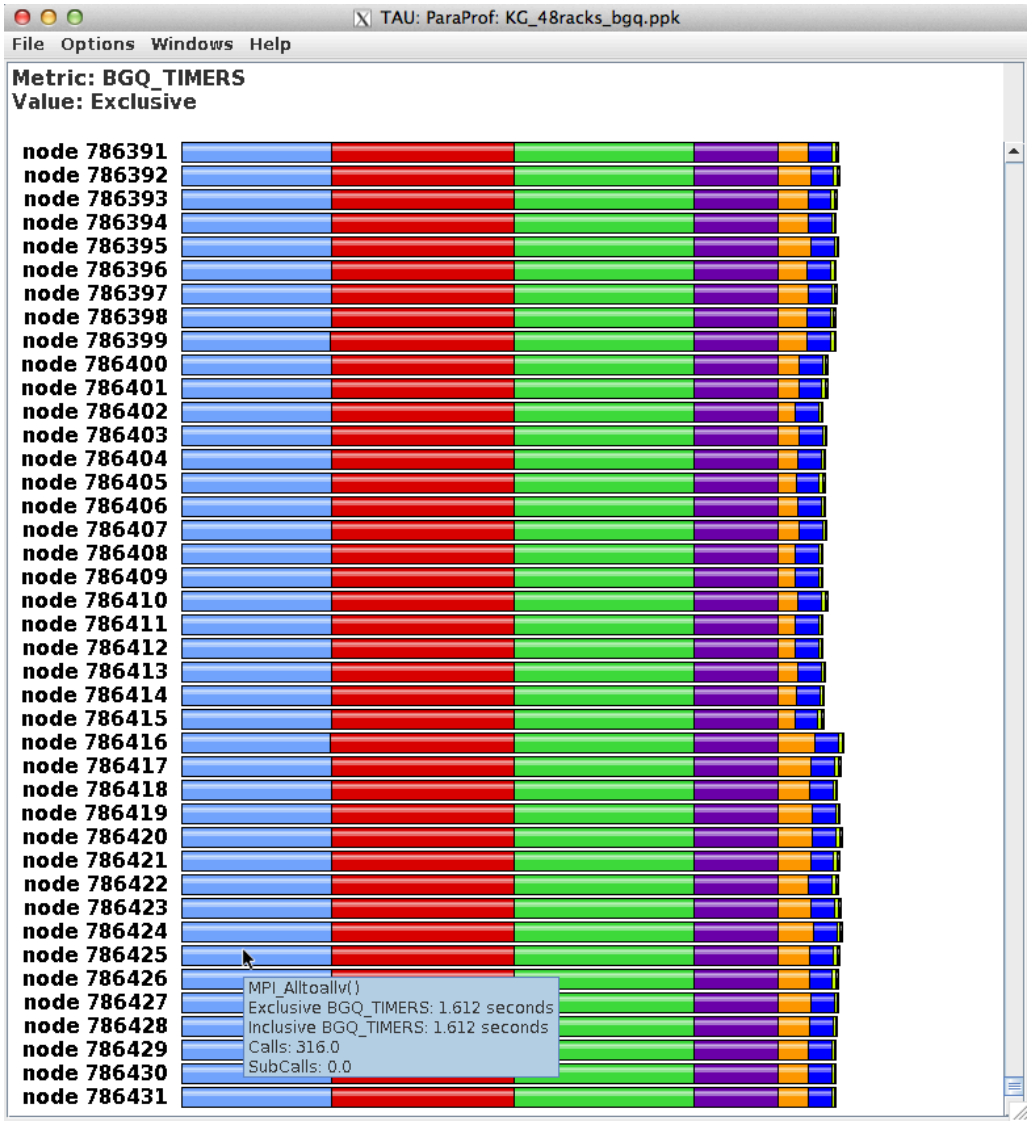


Event Based Sampling in TAU

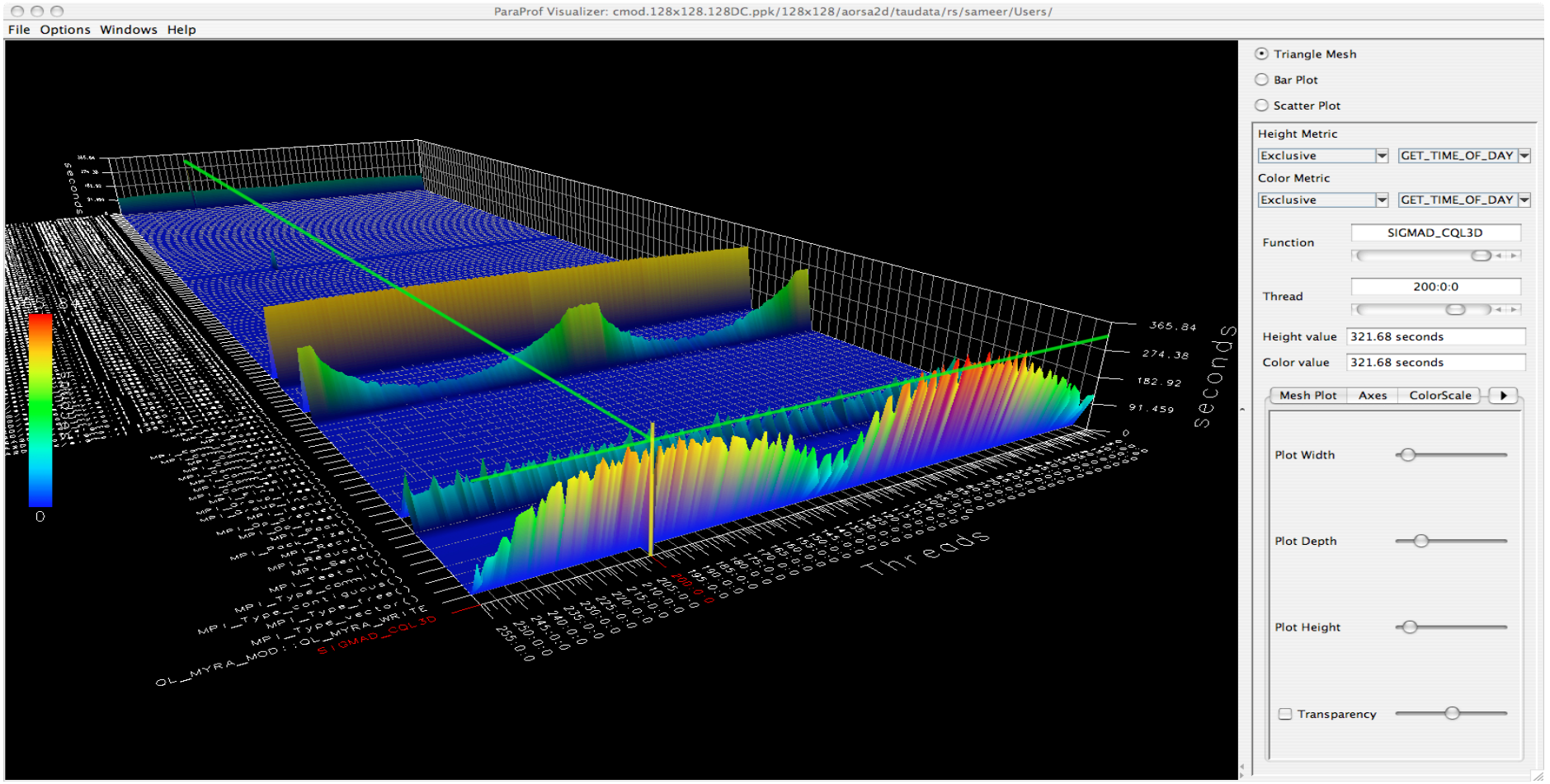



```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make CC=tau_cc.sh CXX=tau_cxx.sh
% export TAU_SAMPLING=1
% mpirun –np 256 ./a.out
% paraprof
```

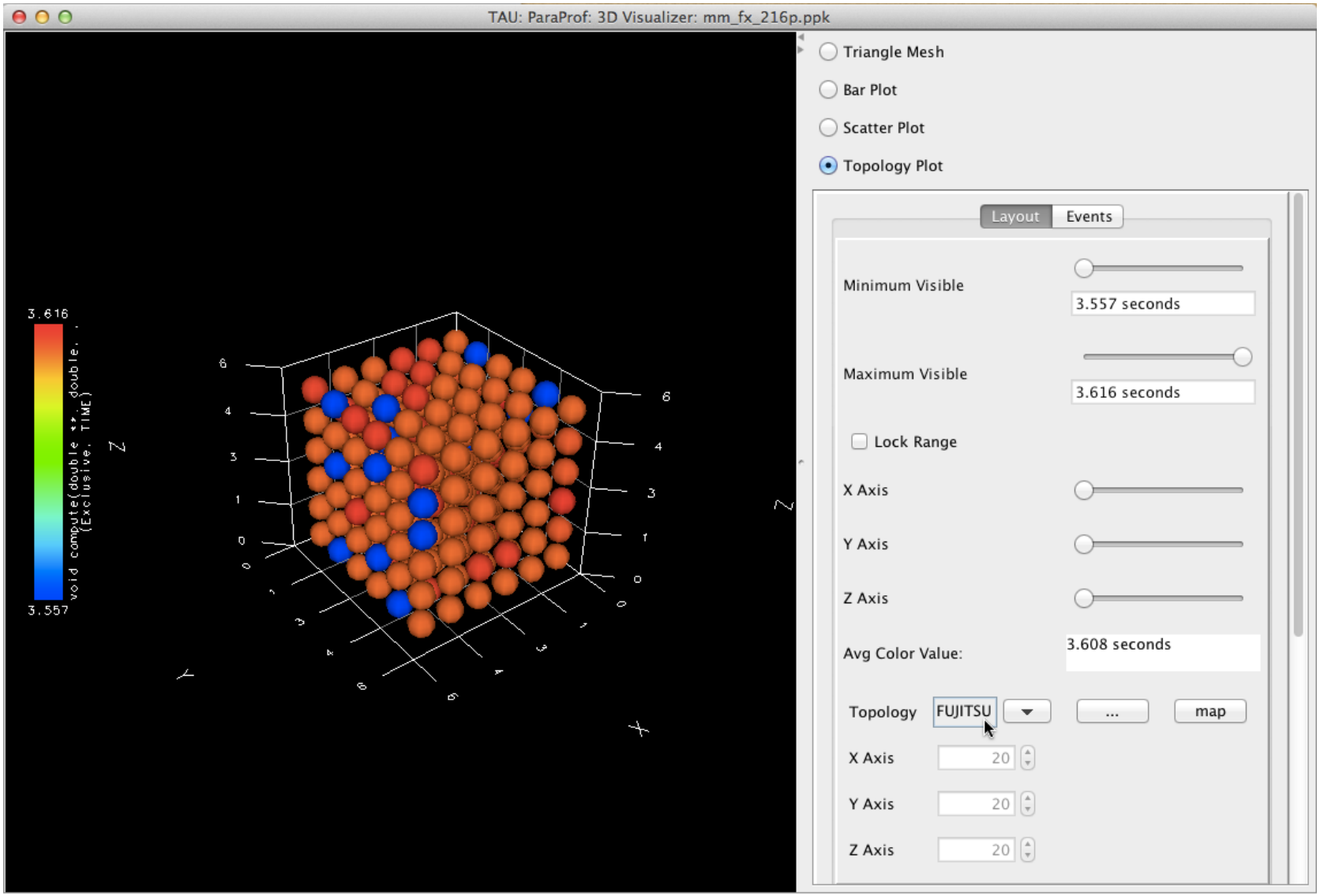
ParaProf Profile Browser



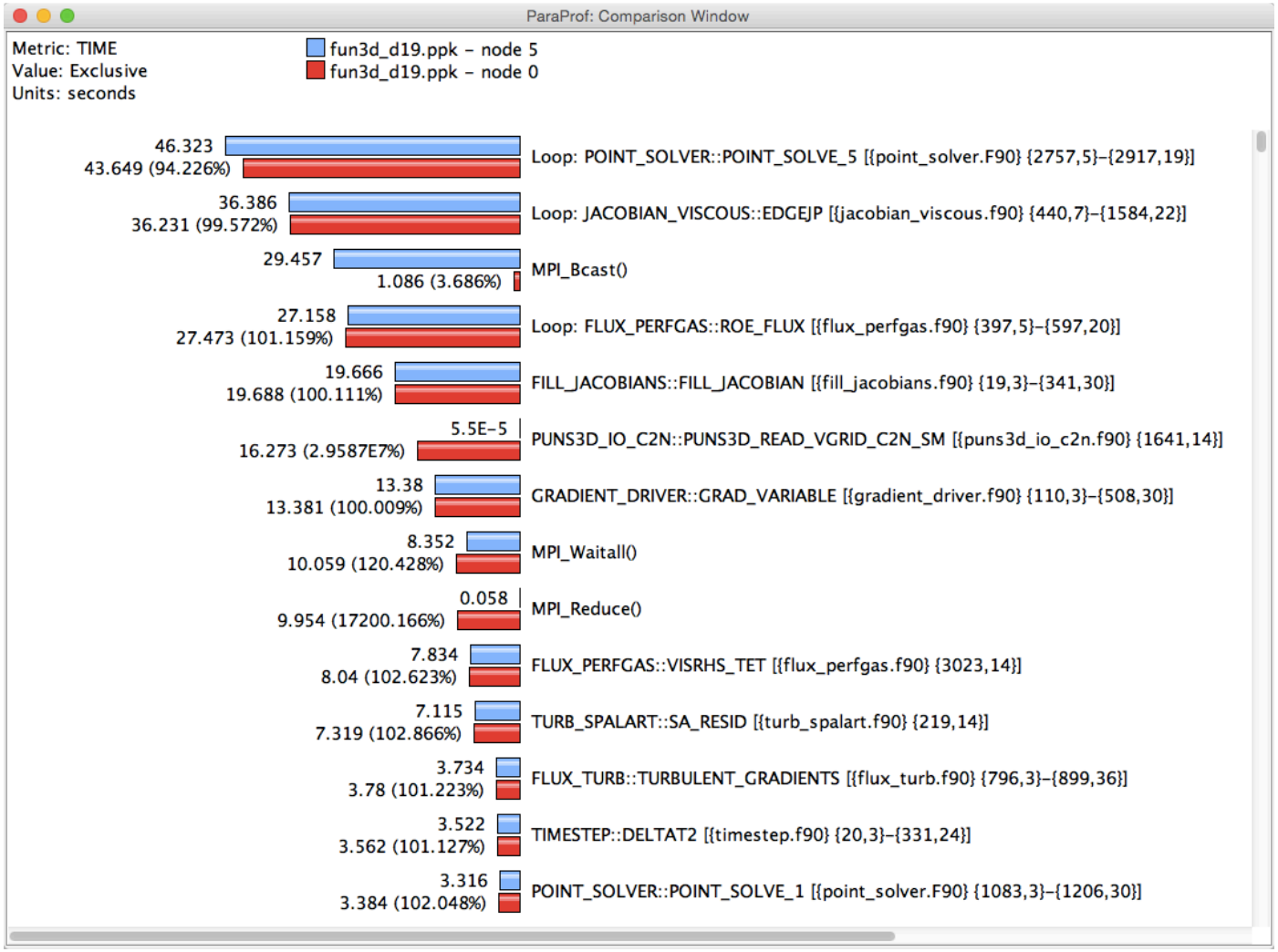
ParaProf 3D Profile Browser



ParaProf Topology Display



ParaProf Comparison Window



Comparing Rank 0 with 5 : Right click on “node 5” -> Add node to comparison window

Profiling with Multiple Counters

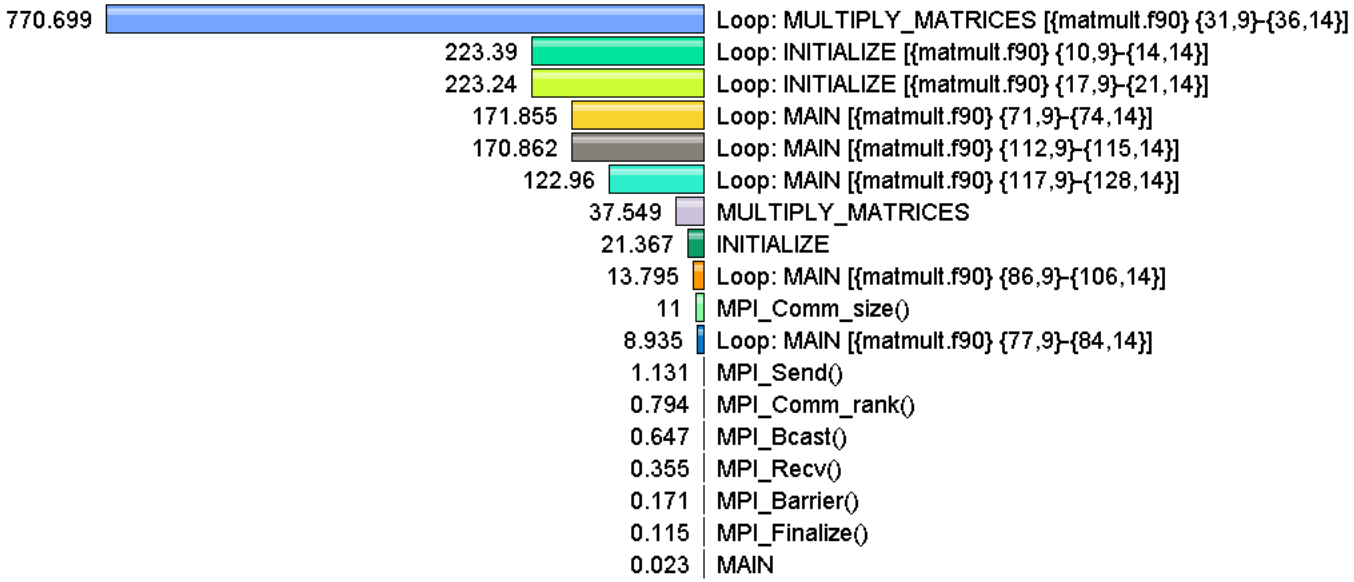
```
% source tau.bashrc
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION
% make F90=tau_f90.sh
% mxterm 1 16 40
% export TAU_METRICS=TIME,PAPI_FP_INS,PAPI_L1_DCM
% mpirun -np 4  ./matmult
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> Click PAPI_FP_INS,
 Click "/",  Click TIME, Apply, Choose new metric by double
clicking.
```

Computing FLOPS

- Goal: What is the execution rate of my loops in MFLOPS?

- Flat profile with PAPI_FP_INS and time with loop instrumentation:

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format

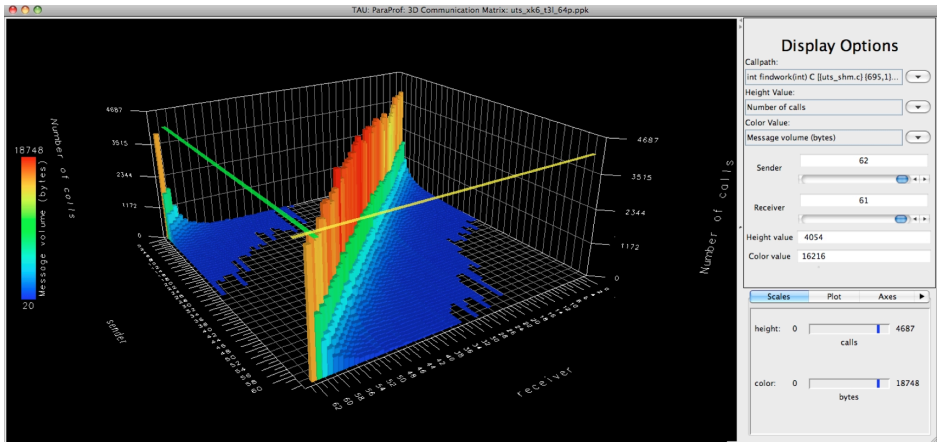


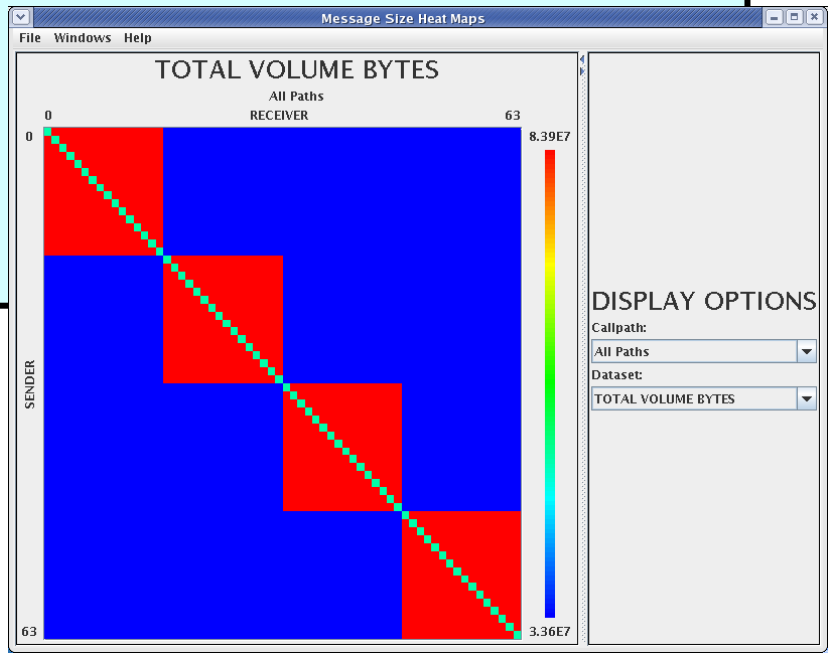
Callpath Profiles

```
% source tau.bashrc
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% mxterm 1 16 40
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)
% mpirun -np 4   ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

Comunication Matrix

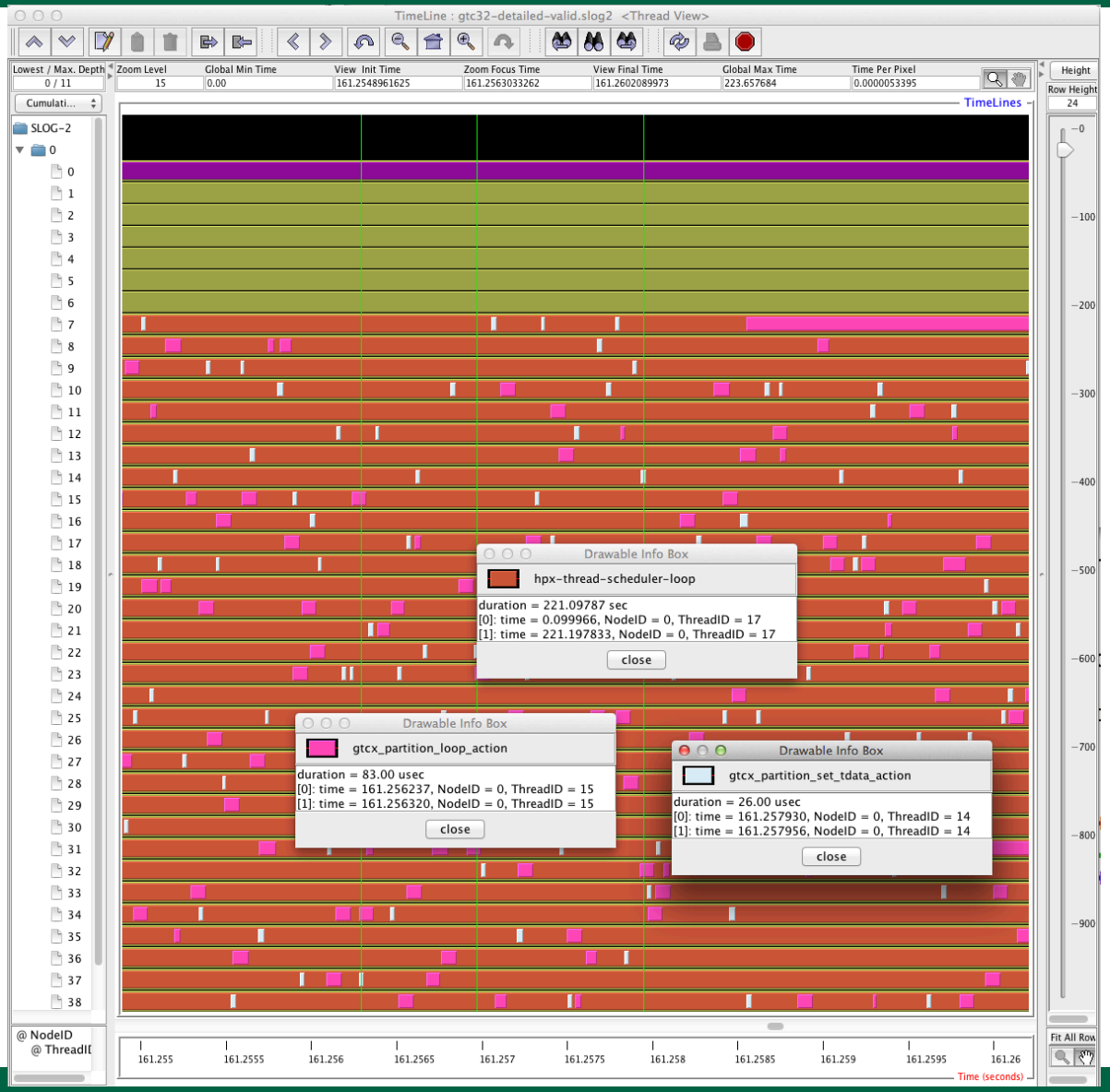
```
% source tau.bashrc
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% mxterm 1 16 40
•    % export TAU_COMM_MATRIX=1
% mpirun -np 4    ./a.out

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
```





Goal: What is the volume of inter-process communication? Along which calling path?

Jumpshot Trace Visualizer



Vampir Trace Visualizer

APEX

Autonomic Performance Environment for eXascale

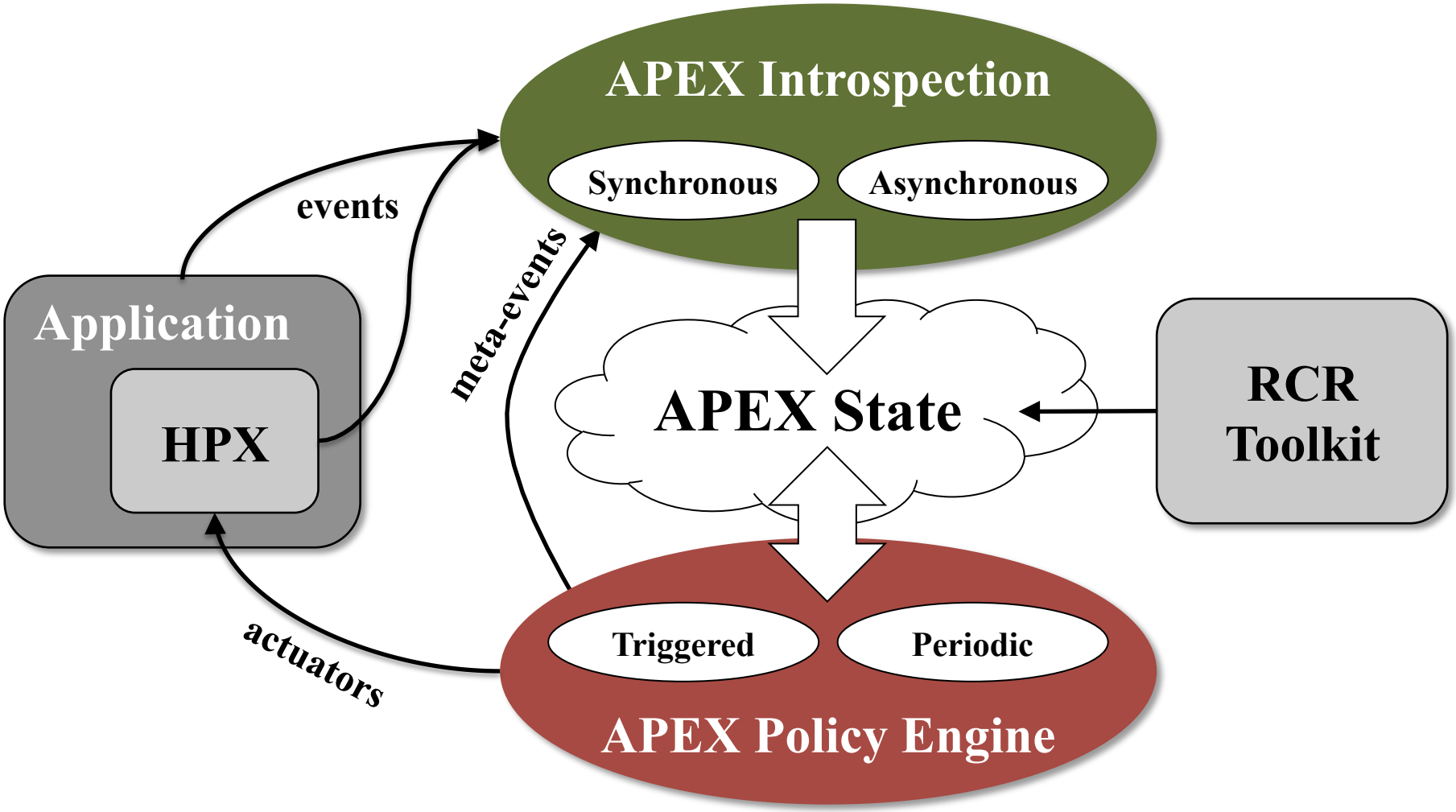
APEX Runtime Adaptation : Motivation

- **Controlling concurrency**
  - Energy efficiency
  - Performance
- **Parametric variability**
  - Granularity for this machine / dataset?
- **Load Balancing**
  - When to perform AGAS migration?
- **Parallel Algorithms (for_each…)**
  - Separate *what* from *how*
- **Address the "SLOW(ER)" performance model**

Introduction to APEX

- **Performance awareness and performance adaptation**
- **Top down** and **bottom up** performance mapping / feedback
  - Make node-wide resource utilization data and analysis, energy consumption, and health information available in real time
  - Associate performance state with policy for feedback control
- **APEX introspection**
  - OS: track system resources, utilization, job contention, overhead
  - Runtime (HPX): track threads, queues, concurrency, remote operations, parcels, memory management
  - Application timer / counter observation

APEX architecture



APEX Introspection

- **APEX collects data through "inspectors"**
  - *Synchronous* uses an event API and event "listeners"
    - Initialize, terminate, new thread – added to HPX runtime
    - Timer start, stop, yield*, resume* - added to HPX task scheduler
    - Sampled value (counters from HPX-5, HPX-3)
    - Custom events (meta-events)
  - *Asynchonous* do not rely on events, but occur periodically
- **APEX exploits access to performance data from lower stack components**
  - Reading from the RCR blackboard (i.e., power, energy)
  - "Health" data through other interfaces (/proc/stat, cpuinfo, meminfo, net/dev, self/status, lm_sensors, power*, etc.)

APEX Event Listeners

- **Profiling listener**
  - Start event: input name/address, get timestamp, return profiler handle
  - Stop event: get timestamp, put profiler object in a queue for back-end processing, return
  - Sample event: put the name & value in the queue
  - Asynchronous consumer thread: process profiler objects and samples to build statistical profile (in HPX-3, processed/scheduled as a thread/task)
- **TAU Listener (postmortem analysis)**
  - Synchronously passes all measurement events to TAU to build an offline profile
- **Concurrency listener (postmortem analysis)**
  - Start event: push timer ID on stack
  - Stop event: pop timer ID off stack
  - Asynchronous consumer thread: periodically log current timer for each thread, output report at termination

APEX Policy Listener

- ***Policies*** are rules that decide on outcomes based on observed state
  - *Triggered* policies are invoked by introspection API events
  - *Periodic* policies are run periodically on asynchronous thread
- Polices are registered with the Policy Engine
  - Applications, runtimes, and/or OS register callback functions
- Callback functions define the policy rules
  - "If x < y then…"
- Enables runtime adaptation using introspection data
  - Engages actuators across stack layers
  - Could also be used to involve online auto-tuning support*

Building the tools…

- Building TAU pre-requisites
  - PDT
- Building TAU
- Building APEX pre-requisites
  - ActiveHarmony
- Building HPX with TAU+APEX support

Building PDT for use with TAU

```
$ wget http://tau.uoregon.edu/pdt_lite.tgz
$ tar –xzf pdt_lite.tgz
$ cd pdtoolkit-3.21
$ ./configure -GNU
    -prefix=$HOME/install/pdtoolkit/3.21
$ make
$ make install
```

Optional step – TAU with HPX doesn't require instrumentation

Building TAU for use with HPX

```
$ wget http://tau.uoregon.edu/tau.tgz
$ tar -xvzf tau.tgz
$ cd tau-2.25
$ wget http://tau.uoregon.edu/ext.tgz
$ tar -xvzf ext.tgz
# build for the host
$ ./configure -pthread -bfd=download
        -unwind=download -pdt=$HOME/install/pdtoolkit/3.21
$ make -j install
# build for the device
$ ./configure -pthread -cc=icc -c++=icpc -fortran=intel
        -arch=mic_linux -bfd=download -unwind=download
        -pdt=$HOME/install/pdtoolkit/3.21 -pdt_c++=g++
$ make -j install
```

Italics = optional – TAU with HPX doesn't require instrumentation, binutils or unwinding

Building ActiveHarmony for APEX

```
$ wget
http://www.dyninst.org/sites/default/files/downloads/
harmony/ah-4.5.tar.gz
$ tar -xvzf ah-4.5.tar.gz
$ cd activeharmony-4.5
$ make
$ make install PREFIX=$HOME/local/harmony-4.5
# for device:
$ make MPICC=mpicc_disabled CC=icc CXX=icpc
    CFLAGS="-mmic" CXXFLAGS="-mmic"
    LDFLAGS="-mmic"
$ make install PREFIX=$HOME/local/harmony-4.5-mic
```

Optional step – APEX in HPX will build its own version of ActiveHarmony if it isn't found

Building HPX with APEX & TAU

$ cmake <usual HPX settings>…

-DHPX_WITH_APEX=TRUE \

-DAPEX_WITH_ACTIVEHARMONY=TRUE \

-DACTIVEHARMONY_ROOT=<path-to-harmony> \

-DHPX_WITH_TAU=TRUE \

-DTAU_ROOT=<path-to-TAU> \

-DTAU_ARCH=<x86_64, craycnl, mic_linux…>

-DTAU_OPTIONS=<-icpc-pthread, -intel-pthread, …>

..

$ make (as usual) i.e. “make –j 8 core examples tests”

Runtime Environment Variables - APEX

Environment Variable	Default	Description
APEX_TAU	**0**	**Enable TAU profiling**
APEX_POLICY	1	Execute APEX policies
APEX_MEASURE_CONCURRENCY	0	Periodically sample thread activity
APEX_MEASURE_CONCURRENCY_PERIOD	1000000	Thread sampling period, microseconds
APEX_SCREEN_OUTPUT	**0**	**Output APEX summary at exit**
APEX_PROC_CPUINFO	0	Periodically read data from /proc/cpuinfo
APEX_PROC_MEMINFO	0	Periodically read data from /proc/meminfo
APEX_PROC_NET_DEV	0	Periodically read data from /proc/net/dev
APEX_PROC_SELF_STATUS	0	Periodically read data from /proc/self/status
APEX_PROC_STAT	**1**	**Periodically read data from /proc/stat**
APEX_THROTTLE_CONCURRENCY	**0**	**Deactivate/Activate threads for policies**
APEX_THROTTLING_MAX_THREADS	**48**	**Max threads allowed**
APEX_THROTTLING_MIN_THREADS	**1**	**Min threads allowed**
APEX_THROTTLE_ENERGY	0	Enable energy throttling
APEX_THROTTLING_MIN_WATTS	150	Minimum Watt threshold
APEX_THROTTLING_MAX_WATTS	300	Maximum Watt threshold

Runtime Environment Variables - TAU

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 instantaneous 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	**0**	**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. Enabled by default to remove 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_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
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., TIME,ENERGY,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>)

Runtime Environment Variables - TAU

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

Note about environment variables…

- mpirun.mic does not pass environment variables to the application without considerable effort (to me, anyway)
- TAU and APEX will read environment variables from $PWD/tau.conf or $PWD/apex.conf files, respectively
- When running on Babbage, this is sometimes necessary

Example 1: HPX+APEX

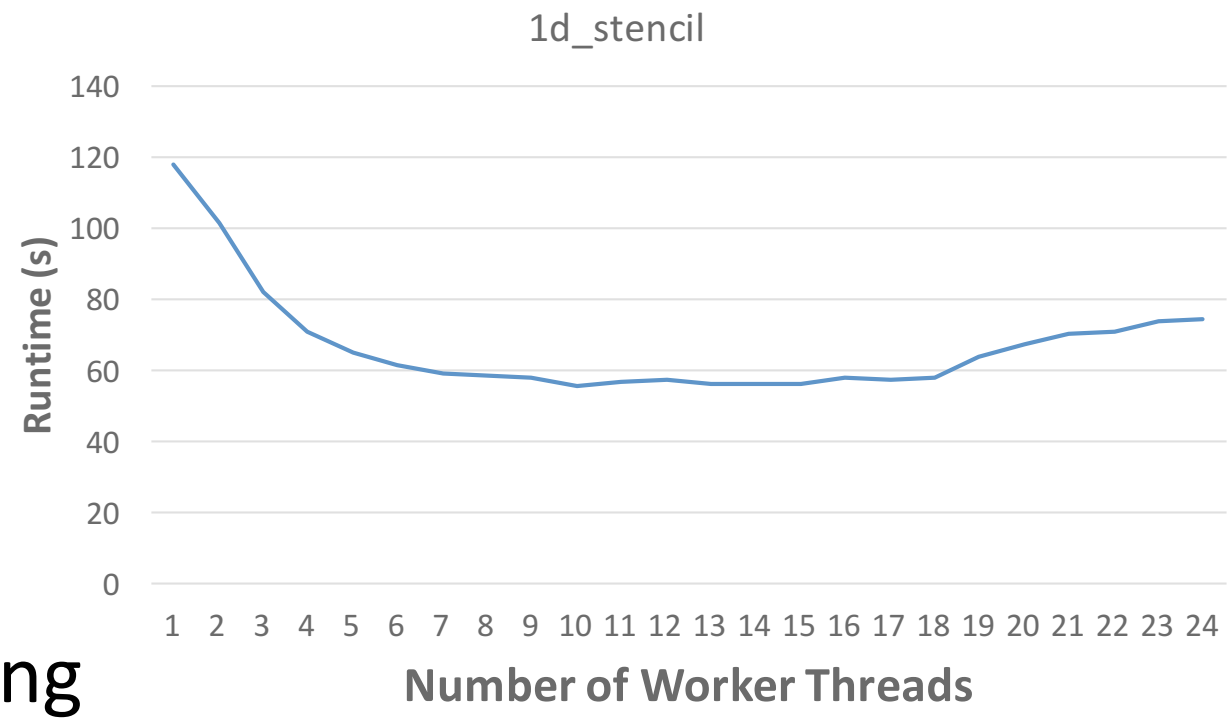
- Follow instructions found at http://github.com/khuck/SC15_APEX_tutorial

```
$ bash
$ source /usr/share/Modules/init/bash
$ source /project/projectdirs/training/SC15/
  HPX-SC15/hpx_install/env.sh
$ module load hpx/mic-0.9.11
$ ./scripts/configure-mic.sh
$ salloc --reservation=SC_Reservation -N 1 -p
  debug
# after the allocation is granted:
$ ./scripts/run_1d_stencil-mic.sh
```

…example will run and spit lots of verbose APEX output to confirm it is working

Example2 : HPX+APEX+TAU

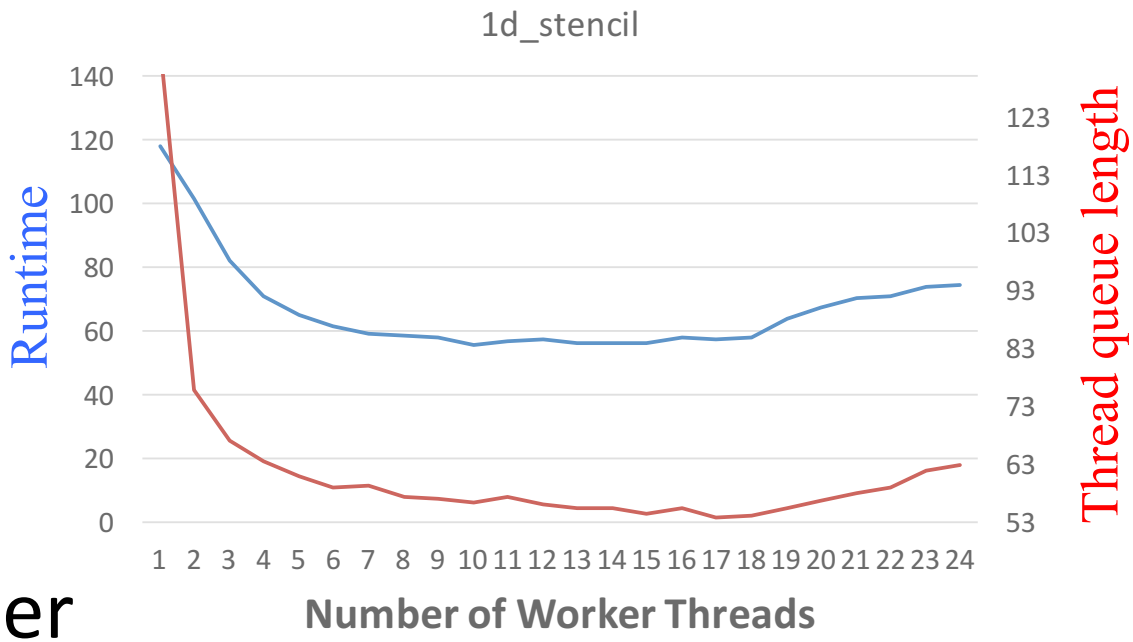
- Heat diffusion
- 1D stencil code
- Data array partitioned into chunks
- 1 node with no hyperthreading



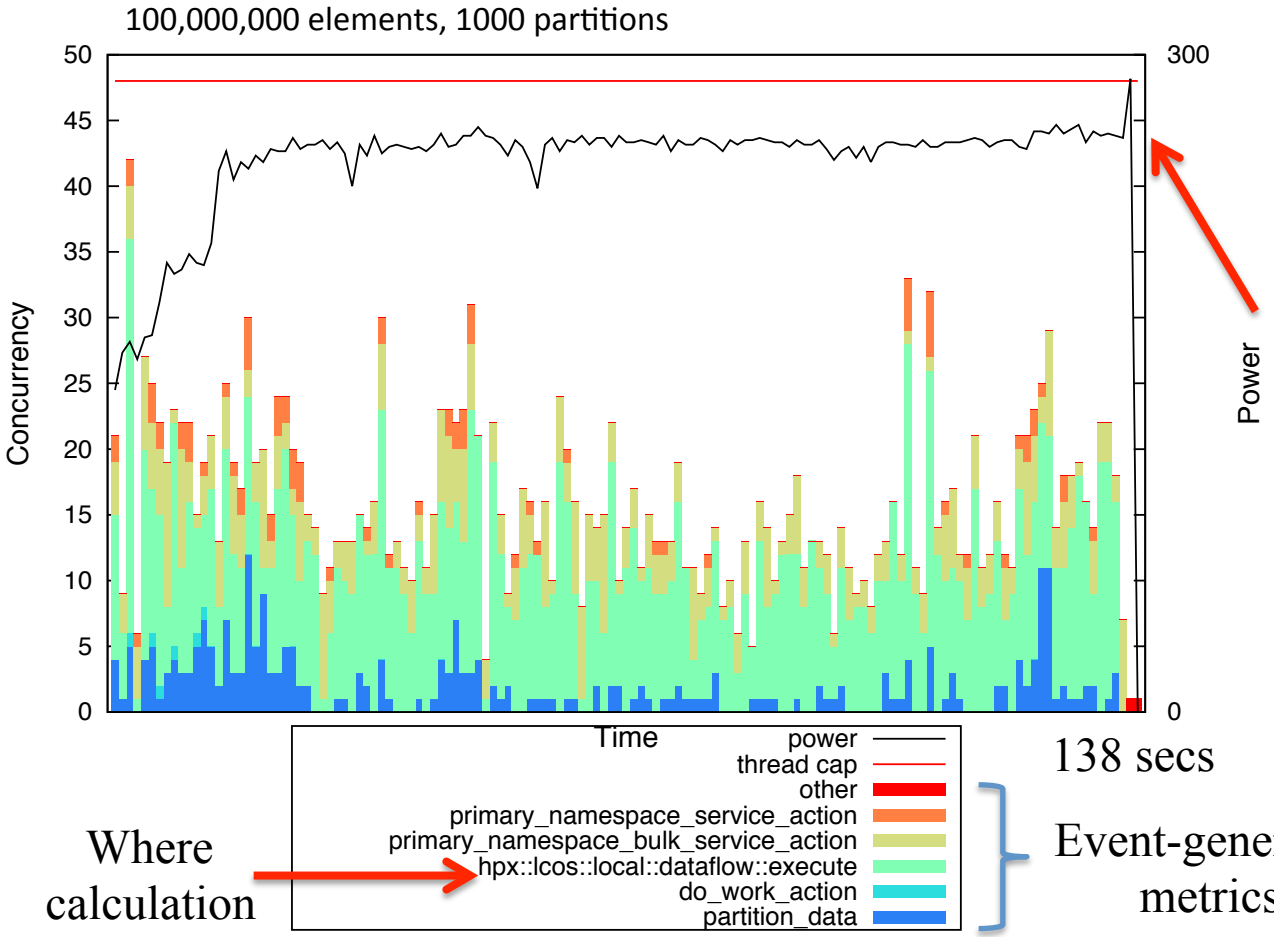
- Performance increases to a point with increasing worker threads, then decreases

Concurrency & Performance

- Region of maximum performance correlates with thread queue length runtime performance counter
  - Represents # tasks currently waiting to execute
- Could do introspection on this to control concurrency throttling policy (see example 3)

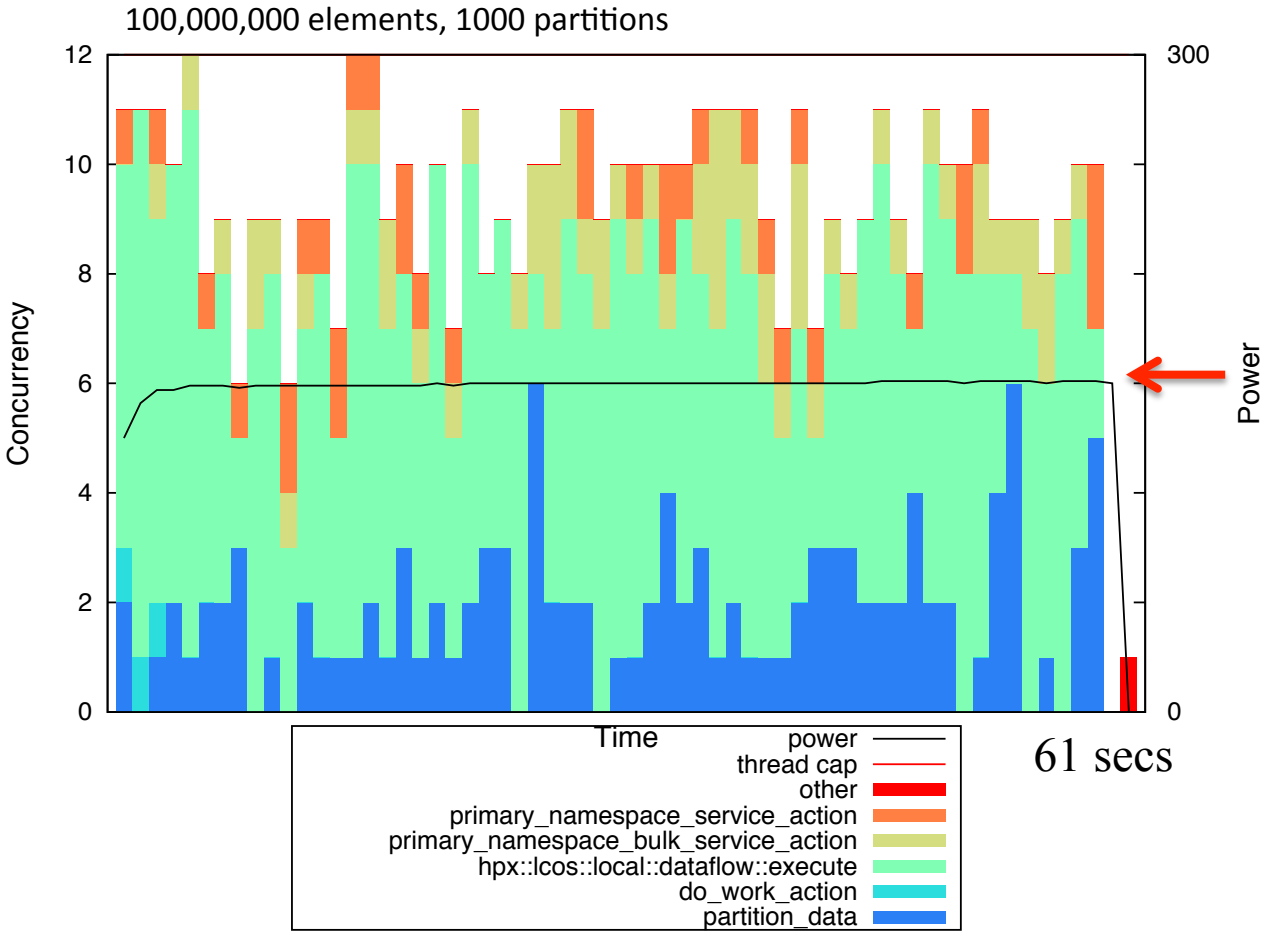


1d_stencil_4 Baseline



- 48 worker threads (with hyperthreading, on Edison)
- Actual concurrency much lower
  - Implementation is memory bound
- Large variation in concurrency over time
  - Tasks waiting on prior tasks to complete

1d_stencil w/optimal # of Threads

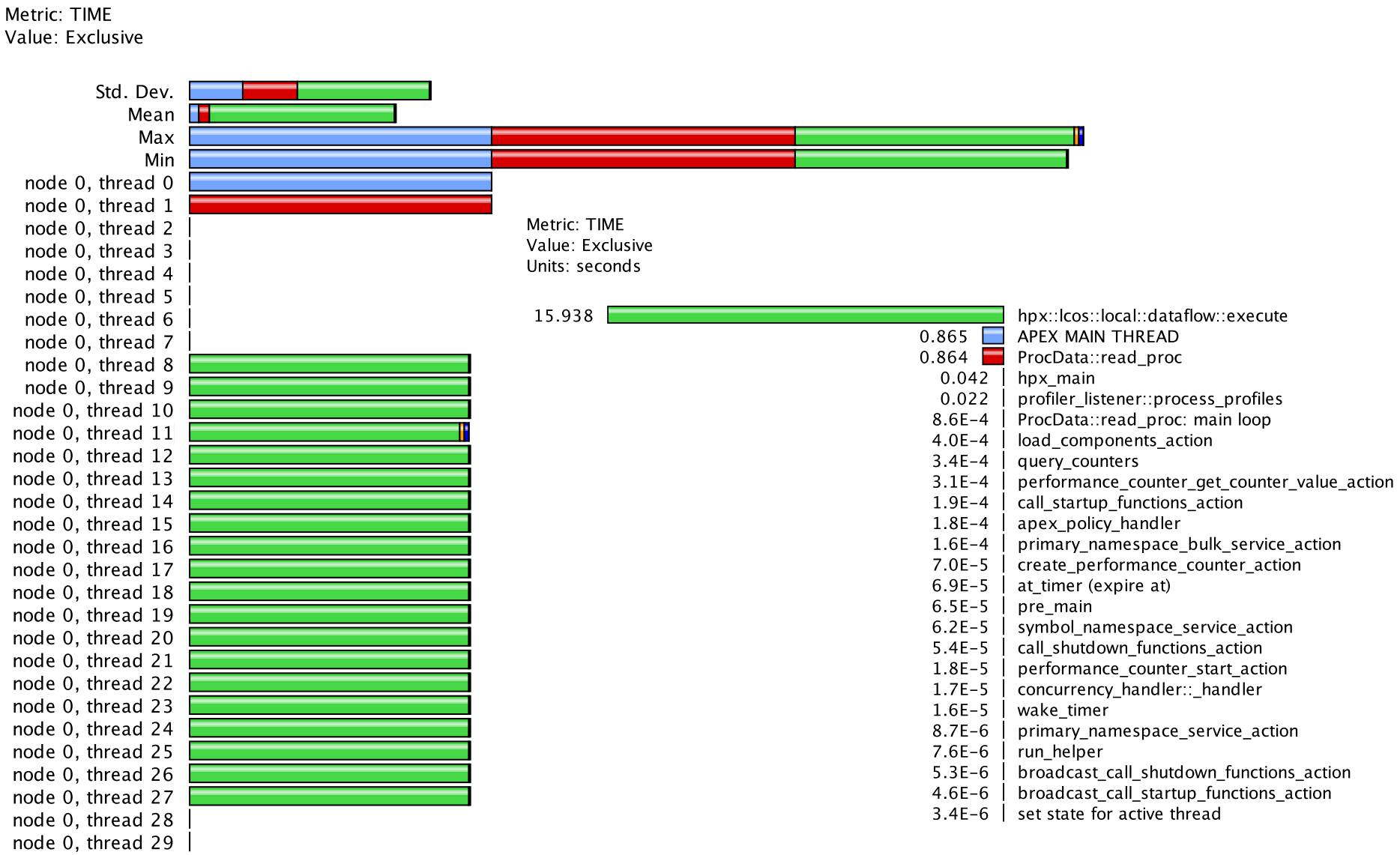


- 12 worker threads on Edison
- Greater proportion of threads kept busy
  - Less interference between active threads and threads waiting for memory
- Much faster
  - 61 sec. vs 138 sec.

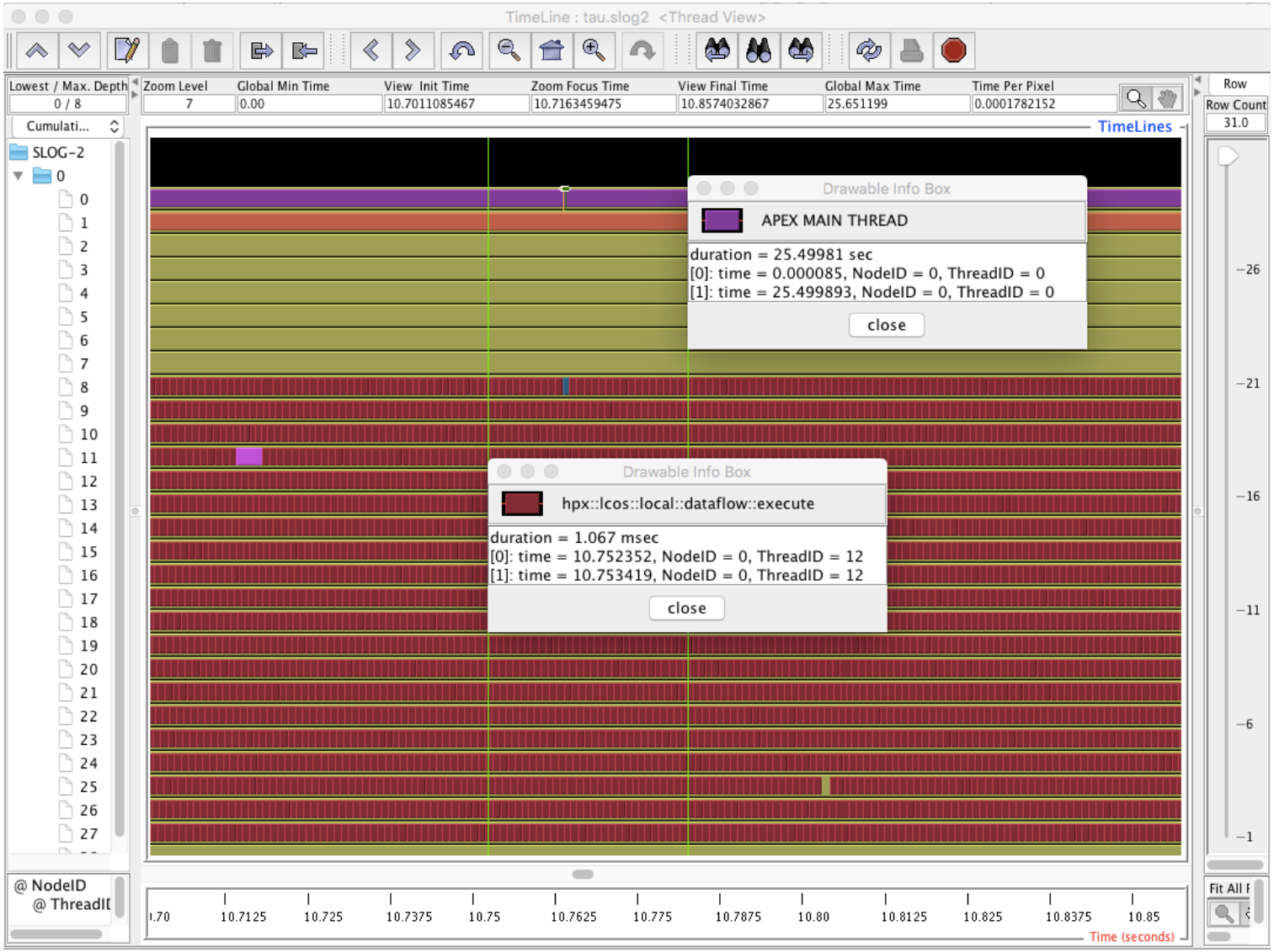
Run the example #2

```
$ ./scripts/run_1d_stencil-mic.sh
# which does:
$ export APEX_TAU=1
$ export TAU_TRACE=1 (to collect a trace, optional)
$ export APEX_PROFILE=1
# to run on device (babbage):
$ get_micfile ; srun mpirun.mic -n 1 -hostfile
    micfile.$SLURM_JOB_ID -ppn 1 ./build-mic/
    apex_examples/1d_stencil_4 --nx 100000 --np 1000
    --nt 45 --hpx:threads 60
# to view summary of TAU profiles:
$ pprof –s
# to merge trace:
$ tau_multimerge && tau2slog2 tau.trc tau.edf -o
    tau.slog2
```

Paraprof view of profile



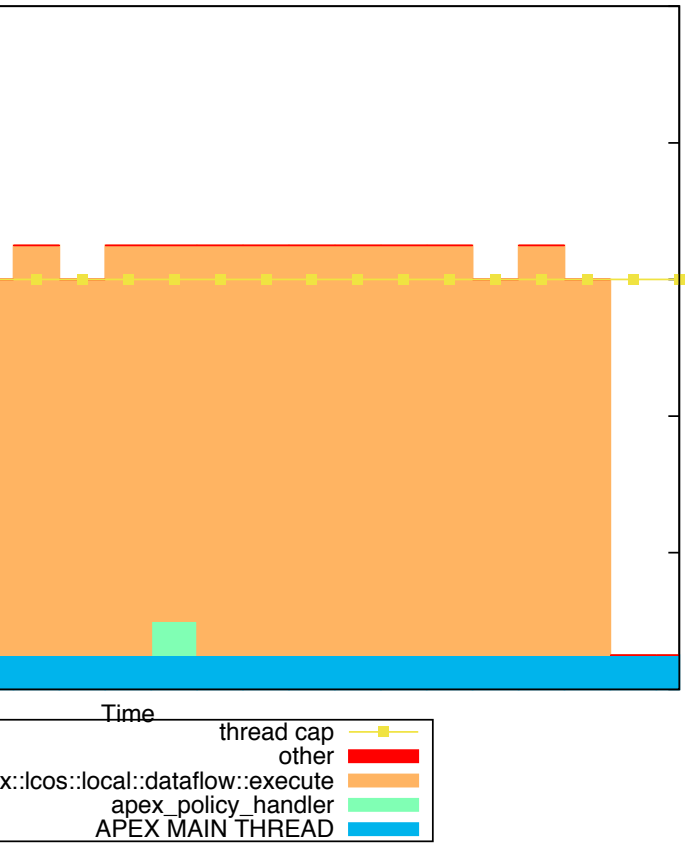
Jumpshot view of trace



Run the example #3:

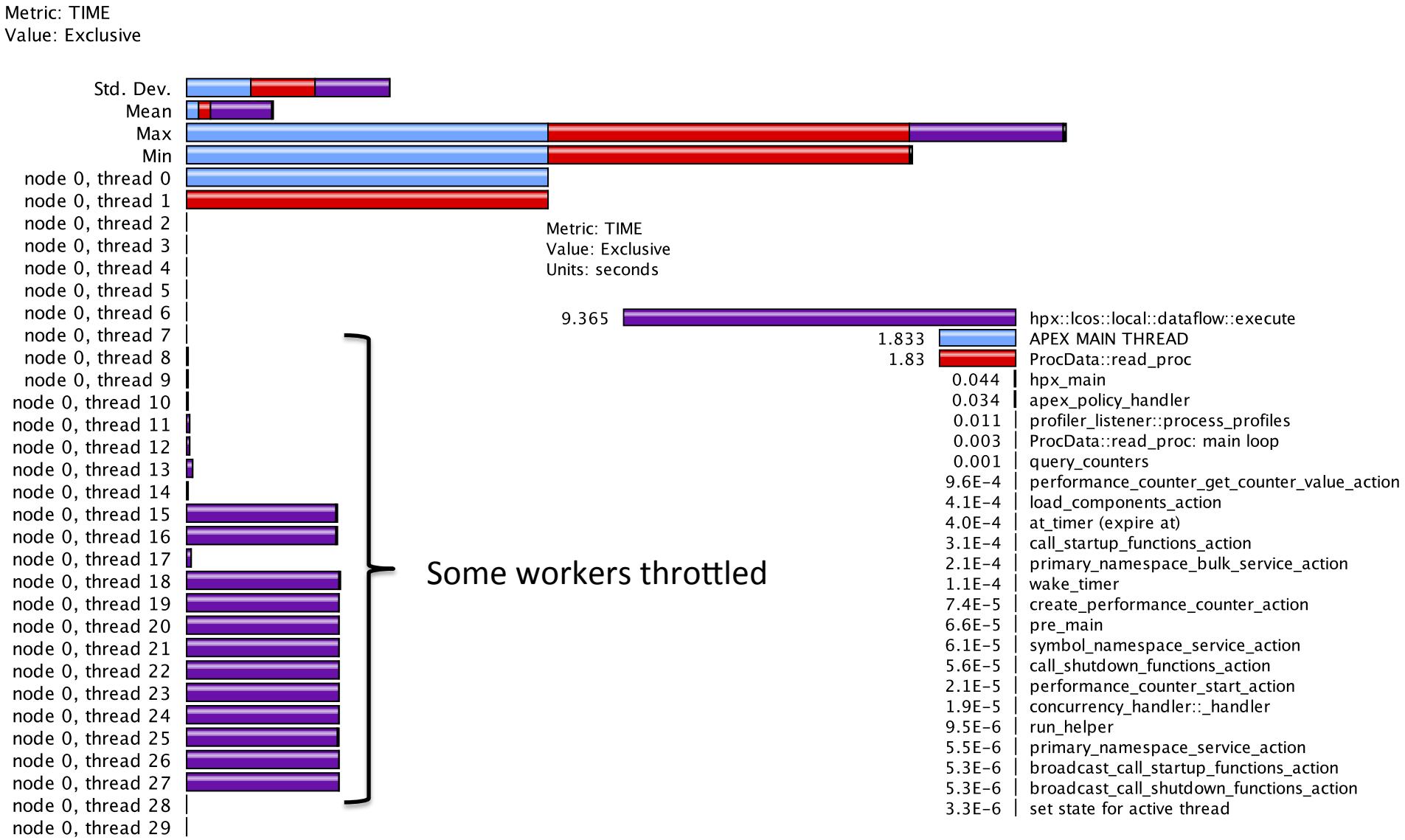
```
$ ./scripts/run_1d_stencil-host.sh
# which will:
$ export APEX_TAU=1
$ export TAU_TRACE=1
$ export TAU_PROFILE=1
$ export TAU_PROFILE_FORMAT=merged
$ export APEX_POLICY=1
$ export APEX_THROTTLE_CONCURRENCY=1
$ export APEX_THROTTLING_MIN_THREADS=8
$ export APEX_THROTTLING_MAX_THREADS=32
# note: nx is number of cells per partition, np is
    number of partitions, nt is number of timesteps
$ ./build-host/apex_examples/1d_stencil_4_throttle
    --nx 100000 --np 1000 --nt 450 --
    hpx:queuing=throttle --hpx:threads 32
```

1d_stencil Adaptation with APEX

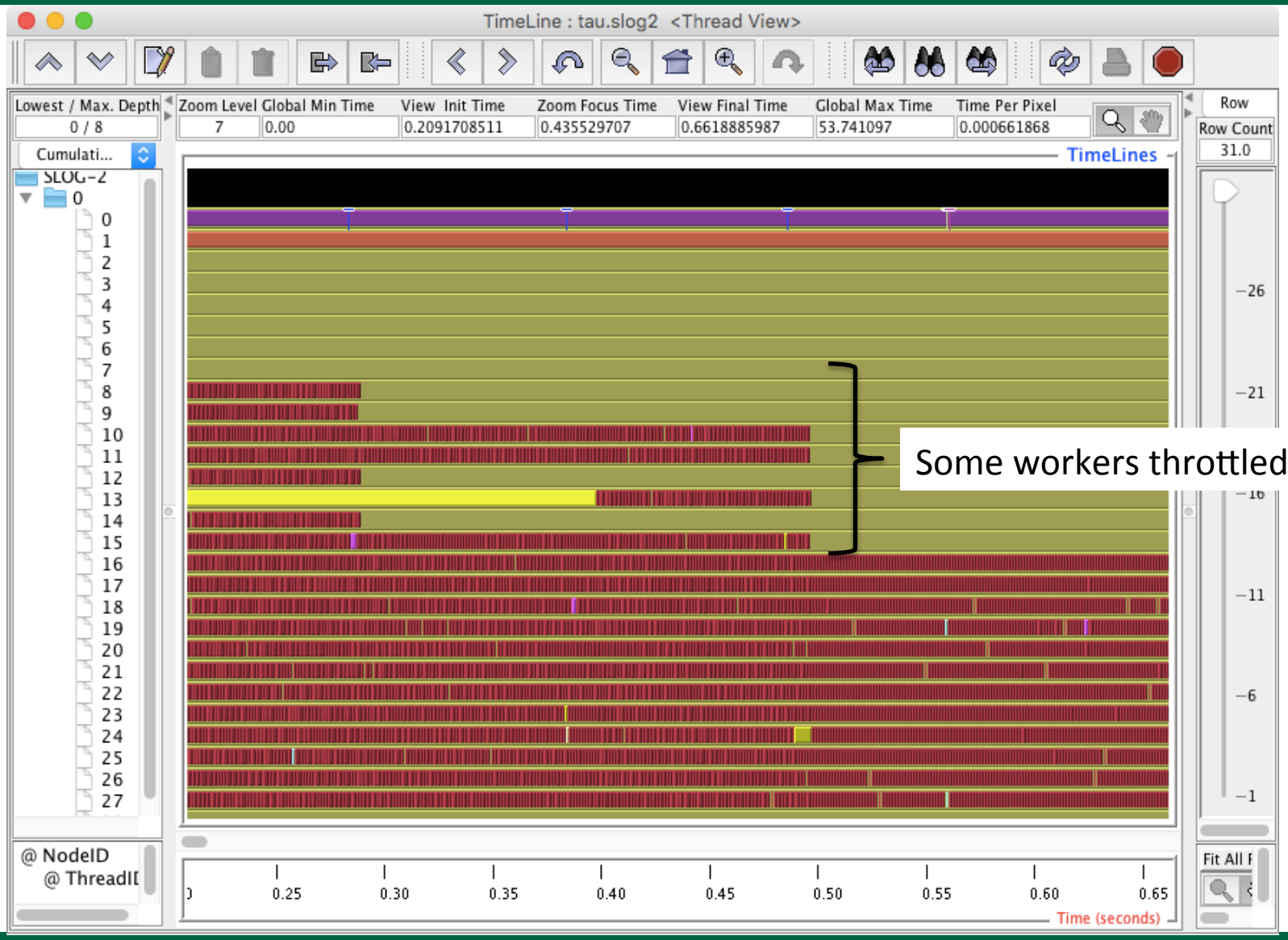


- Initially 32 worker threads
- ActiveHarmony searches for minimal thread queue length
- Quickly converges on 20

Profile of throttled execution



Trace of throttled execution



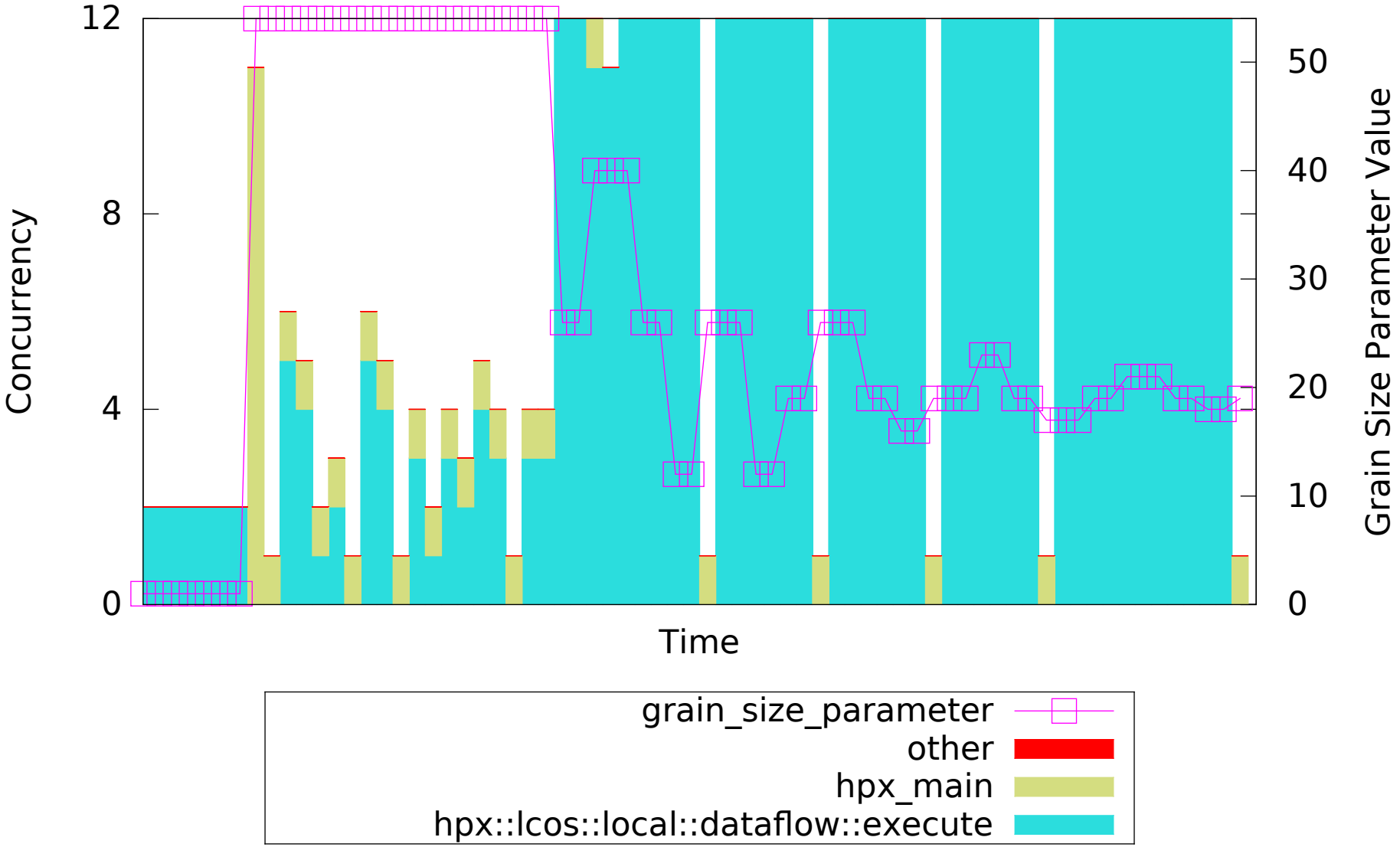
Adapting Block Size

- Is 1000 partitions of 100000 cells the best partitioning?
- Parametric studies say “no”.
- Can we modify the example to repartition as necessary to find better performance?

Run the example #4:

```
$ ./scripts/run_1d_stencil_repart-
    host.sh
# which will:
$ export APEX_POLICY=1
# note: nx is TOTAL elements, nt is
    timesteps between repetitions, nr
    is number of repetitions
$ ./build-host/apex_examples/
    1d_stencil_4_repart –nx 10000000
    –nr 50 --nt 50 --hpx:threads 12
```

1d_stencil: adapting block size



Wrap-up…

- Introduced TAU
- Instrumentation: PDT, MPI, OpenMP, **tau_exec**
- I/O and Memory evaluation
- PAPI
- Demonstration of analysis tools: ParaProf, TAUdb and PerfExplorer, Vampir and Jumpshot
- Introduced APEX
- Built HPX with APEX & TAU
- Ran Program Examples

Support Acknowledgements

- Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/ Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE- FG02-11ER26050 and DE-SC0006925.
- Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. DOE's National Nuclear Security Administration under contract DE- AC04-94AL85000.
- This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.