TAU Performance System

Chandan Basu

Application expert meeting

C3SE, Göteborg

September 19-20, 2012

TAU Performance System

- TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, and Python.
  - Profiling shows how much time was spent in a routine / loop / phase
  - Tracing shows when the events take place in each process along a timeline
- TAU runs on various HPC platforms and it is free (BSD style license)
  - Linux Cluster / Cray / Blue-Gene
  - Wide variety of applications
  - Very wide jobs
- TAU has instrumentation, measurement and analysis tools
- Tau project is more than 15 years and is quite active
- http://tau.uoregon.edu

Understanding Application Performance using TAU

- How much time is spent in subroutines / loops?
- How many instructions are executed in these code regions? Floating point rate?
- What is the memory usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
- How much time does the application spend performing I/O? What is the peak read and write bandwidth of individual calls, total volume?
- What is the contribution of different phases of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- How much time is spent in different MPI routines? What is the communication pattern?
- Which rank is slow and why?

How Does TAU Work?

- Instrumentation: Probes to perform measurements
  - Automatic source code instrumentation
  - Pre-loading external libraries (MPI, I/O, CUDA, OpenCL)
  - Rewriting the binary executable

- Measurement:
  - Direct instrumentation / sampling
  - Throttling
  - Per-thread storage of performance data
  - Interface with external packages (PAPI, Scalasca, Score-P, VampirTrace)

- Analysis:
  - Visualization of profiles and traces with paraprof, jumpshot etc
  - Trace conversion tools

TAU Instrumentation

Method	Requires recompil-ing	Shows MPI events	Routine level events	Low level events (loops, phases etc)	Throttling to reduce overhead	Ability for selective instrume ntation
Runtime		Yes			Yes	
Compiler	yes	yes	yes		yes	yes
PDT	yes	yes	yes	yes	yes	yes

- External libraries added at link time are not instrumented
- It is possible to wrap pre-built libraries by tau
  - tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a
  - Link the tau wrapped libraries instead of original libraries

Runtime instrumentation
Library Preloading

- Runtime instrumentation is achieved through library pre-loading.
  - MPI, io, memory, cuda, opencl
  - MPI instrumentation is included by default the others are enabled by command-line options
  - Substitutes I/O, MPI and memory allocation/ deallocation routines with instrumented calls
- Works on dynamic executables
- The command is tau_exec
  - place this command before the application executable when running the application.
- Usage example
  - tau_exec ./a.out; mpirun -np 4 tau_exec -io ./a.out
  - export TAU_TRACK_MEMORY_LEAKS=1; mpirun -np 4 tau_exec -memory ./a.out

Runtime instrumentation
Binary Rewriting

- Dynamic Instrumentation using DyninstAPI
  - U. Wisconsin, Madison, and U. Maryland
- Binary re-writing
  - Support for both static and dynamic executables
- Specify the list of routines to instrument/exclude from instrumentation
- Specify the TAU measurement library to be injected
- To instrument:
  tau_run a.out -o a.inst
  mpirun -np 4 ./a.inst
  paraprof

Automatic Source code Instrumentation

TAU provides compiler wrappers

- tau_f90.sh, tau_cc.sh, and tau_cxx.sh
- Know the native compiler
- Know the MPI library

  mpif90 foo.f90 → tau_f90.sh foo.f90
- The compiler wrapper does the automatic instrumentation
  - Compiler based instrumentation
  - PDT based instrumentation

Automatic Source code Instrumentation
Compiler based

- TAU Compiler wrapper can use compilers native instrumentation:

  export TAU_OPTIONS="-optCompInst"

  tau_f90.sh foo.f90

- Easy & safe
- Our tests showed that there is large performance penalty

Automatic Source code Instrumentation PDT based

- PDT: Program Database Toolkit
  - Documentation of program components
  - Creation of graphic program browsers that show class hierarchies, function call graphs, and template instantiations
  - Insertion of instrumentation calls
- TAU compiler wrapper calls PDT to instrument source code
- A typical compilation of foo.f90 with tau_f90.sh looks like:
  - Preprocessing : foo.f90 → foo.pp.f90
  - Parsing with PDT Parser: (foo.pp.f90) → foo.pp.pdb
  - Instrumenting with TAU: (foo.pp.f90, foo.pp.pdb) → foo.pp.inst.f90
  - Compiling with Instrumented Code: (ifort, foo.pp.inst.f90) → foo.o
- Intermediate files are removed by default
  - Can be preserved for viewing

Automatic Source code Instrumentation PDT based

- The Source code instrumentation works !
  - Tried different large applications, e.g., NWChem, EC-EARTH, VASP etc.
  - For profiling very little performance impact
    - TAU keeps data in memory and writes at the end of the program
    - This works well for profiling but not for tracing
- Failures occur when instrumentation calls are placed in wrong place. The failure rates are **small**
  - In VASP only in one subroutine
  - In EC-EARTH in 2 files
  - NWChem none
- Failures are not catastrophic, there are fall-back options
  - Exclude the files / routines
  - Use compiler based instrumentation for those files
- User support is active in debugging

Using TAU

- Declare the appropriate environment variables for compilation:

  export TAU_MAKEFILE=/nobackup/global/cbasu/tau/x86_64/lib/Makefile.tau-callpath-icpc-mpi-pdt-profile-trace

  export TAU_OPTIONS=”-optTrackIO -optTauSelectFile=select.tau” etc.

- Declare the compiler for makefile, e.g.:

  export MPIF90=tau_f90.sh; make

- Execute, set different runtime options, e.g.:

  export TAU_TRACE=0 ; export TAU_PROFILE=1

  export TAU_COMM_MATRIX=1; export TAU_SAMPLING=0

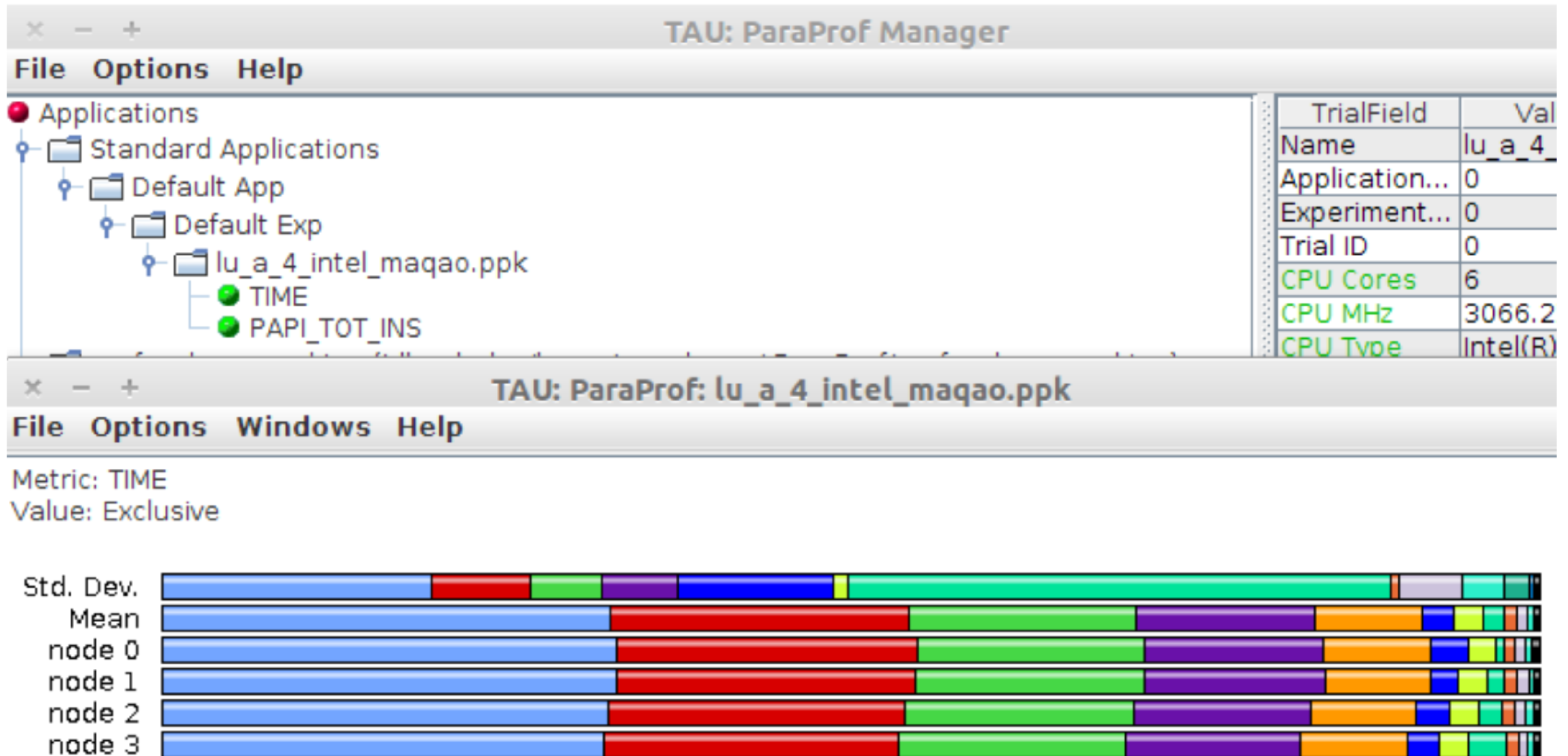
  mpirun -np 4 ./a.out

- Analyze profile data using paraprof

  paraprof --pack app.ppk ## packing of data

  paraprof app.ppk ## visualization

Visualization with Paraprof



TAU Loop level profile

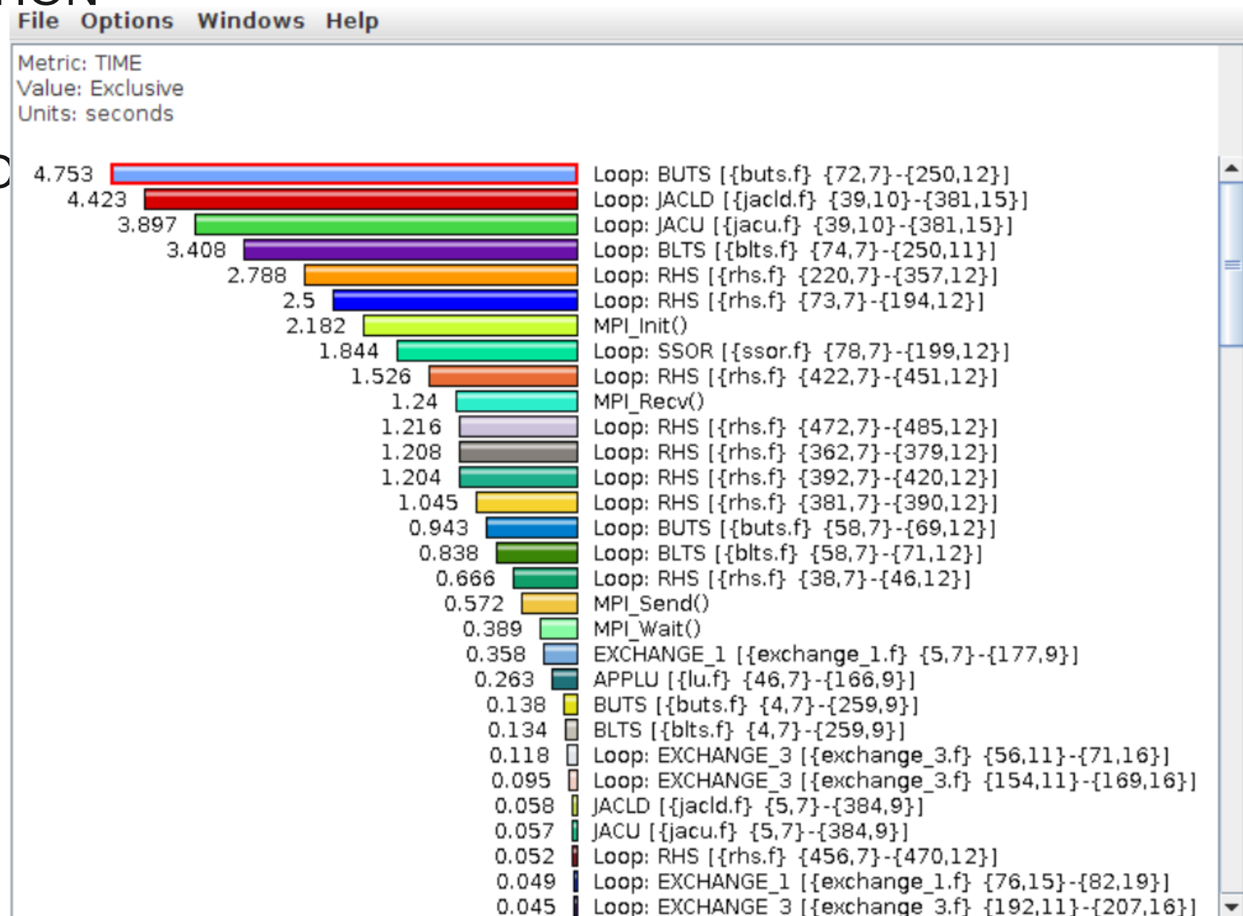
export TAU_OPTIONS='-optTauSelectFile=select.tau'

cat select.tau

BEGIN_INSTRUMENT_SECTION

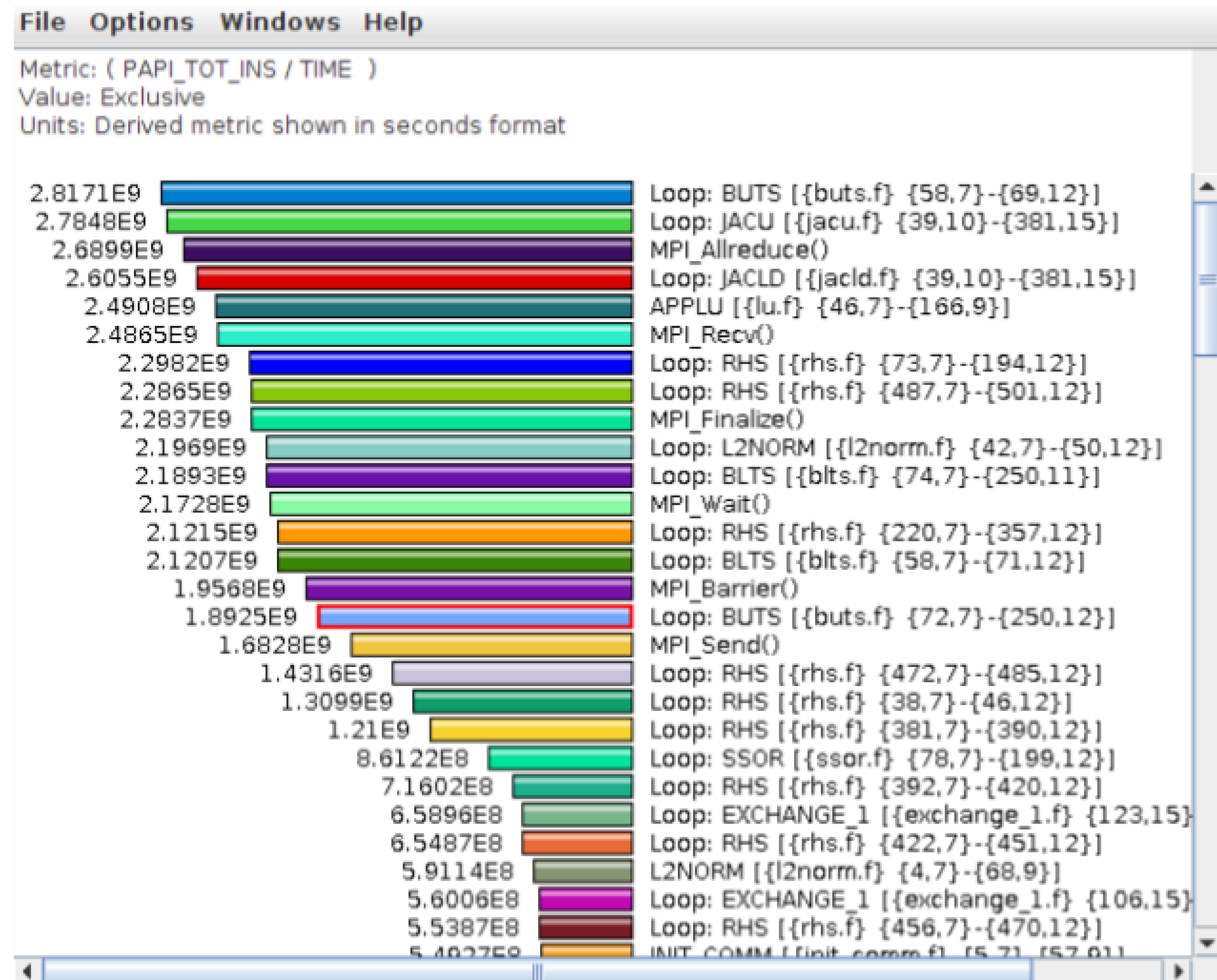
loops routine="#"

END_INSTRUMENT_SECTIO



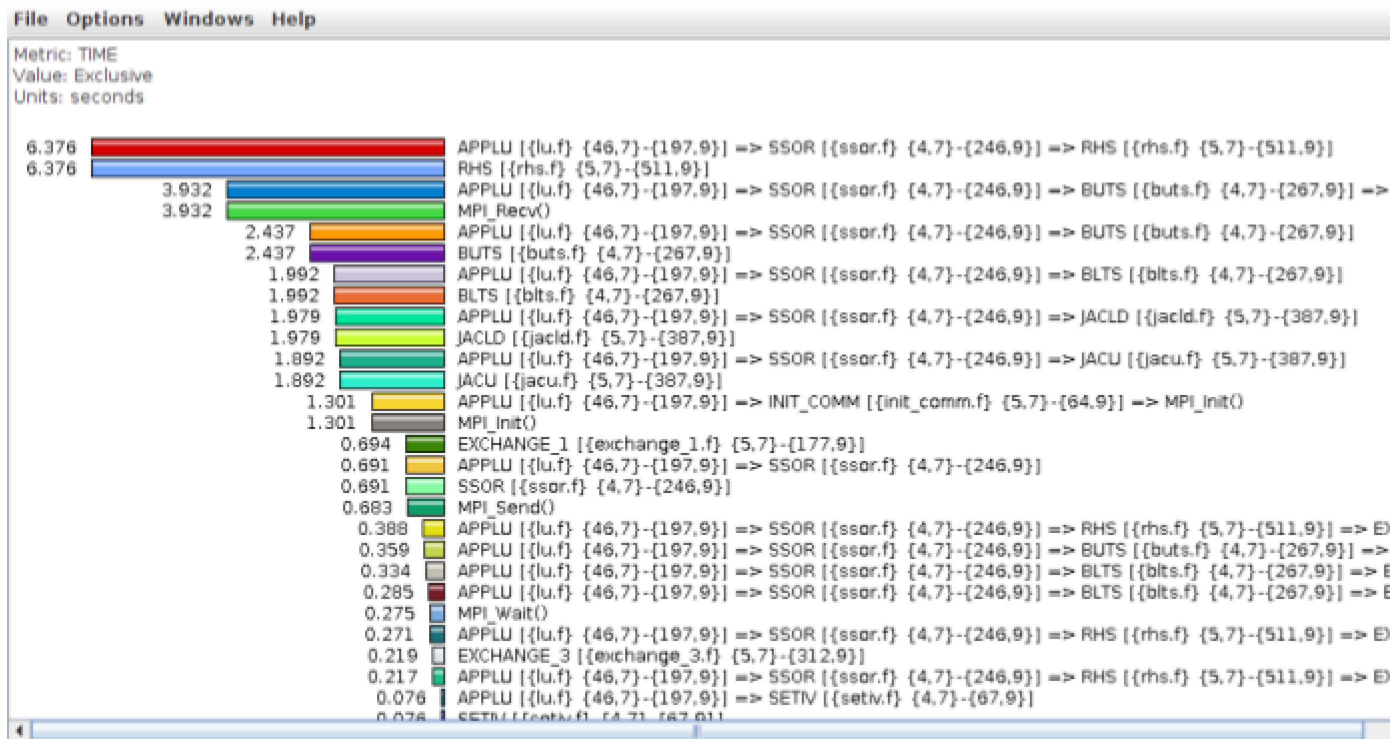
TAU with PAPI

- export TAU_METRICS=TIME:PAPI_FP_OPS:PAPI_TOT_INS
- run

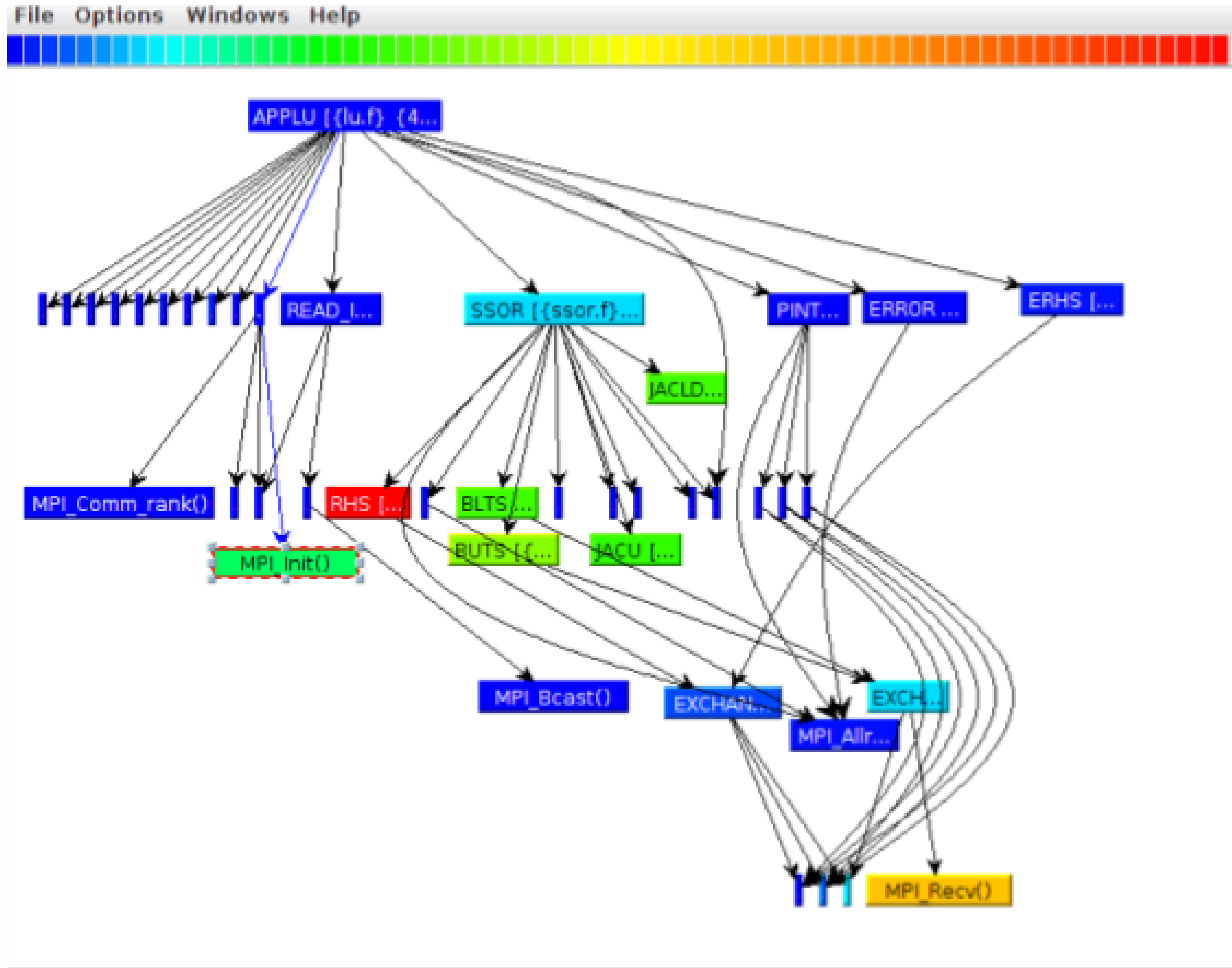


Callpath Profile

- export TAU_CALLPATH=1
- export TAU_CALLPATH_DEPTH=10
- run

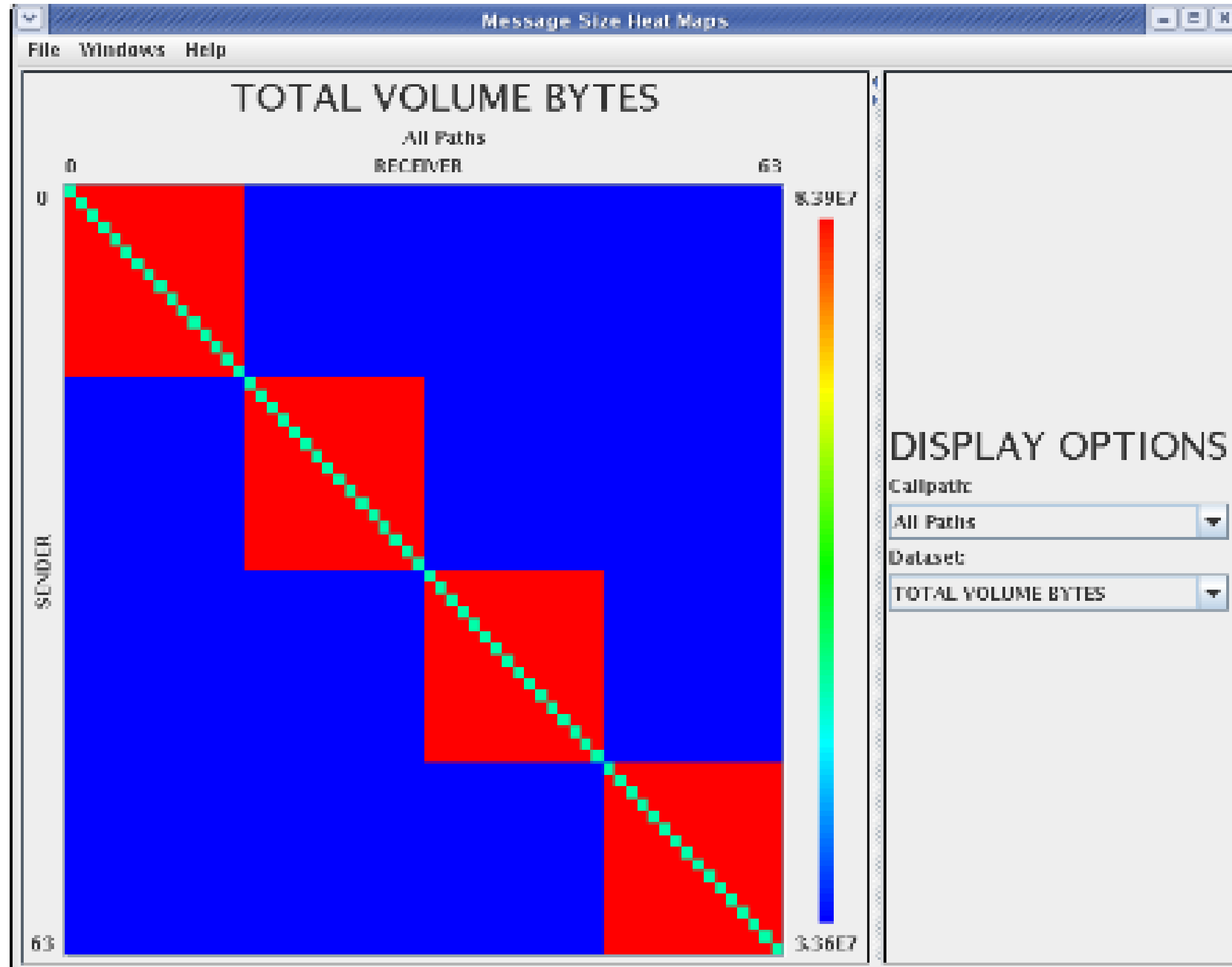


Call Graph



Communication Matrix Display

- export TAU_COMM_MATRIX=1
- run



Detect I/O, Memory Usage

- export TAU_OPTIONS="-optDetectMemoryLeaks -optTrackIO"
- Compile, run

TAU: ParaProf: Context Events for thread: n,c,t, 0,0,0 – samarc_obe_4p_iomem_cp.ppk

Name △	Total	MeanValue	NumSamples	MaxValue	MinValue	Std. Dev.
▼ .TAU application						
▼ MPI_Finalize()						
free size	23,901,253	22,719.822	1,052	2,099,200	2	186,920.948
malloc size	5,013,902	65,972.395	76	5,000,000	2	569,732.815
MEMORY LEAK!	5,000,264	500,026.4	10	5,000,000	3	1,499,991.2
▼ read()						
Bytes Read	4	4	1	4	4	0
READ Bandwidth (MB/s) <file="pipe">		0.308	1	0.308	0.308	0
Bytes Read <file="pipe">	4	4	1	4	4	0
READ Bandwidth (MB/s)		0.308	1	0.308	0.308	0
▼ write()						
WRITE Bandwidth (MB/s)		0.635	102	12	0	1.472
Bytes Written <file="/dev/infiniband/rdma_cm">	24	24	1	24	24	0
Bytes Written	1,456	14.275	102	28	4	5.149
WRITE Bandwidth (MB/s) <file="/dev/infiniband/uverbs0">		0.528	97	12	0.089	1.32
Bytes Written <file="pipe">	64	16	4	28	4	12
WRITE Bandwidth (MB/s) <file="/dev/infiniband/rdma_cm">		1.714	1	1.714	1.714	0
Bytes Written <file="/dev/infiniband/uverbs0">	1,368	14.103	97	24	12	4.562
WRITE Bandwidth (MB/s) <file="pipe">		2.967	4	5.6	0	2.644
▼ writev()						
WRITE Bandwidth (MB/s)		4.108	2	7.667	0.549	3.559
Bytes Written	297	148.5	2	230	67	81.5
WRITE Bandwidth (MB/s) <file="socket">		4.108	2	7.667	0.549	3.559
Bytes Written <file="socket">	297	148.5	2	230	67	81.5
▼ readv()						
Bytes Read	112	28	4	36	20	8
READ Bandwidth (MB/s) <file="socket">		25.5	4	36	10	11.079
Bytes Read <file="socket">	112	28	4	36	20	8
READ Bandwidth (MB/s)		25.5	4	36	10	11.079
▼ MPI_Comm_free()						
free size	10,952	195.571	56	1,024	48	255.353
▶ read()						
▶ MPI_Type_free()						
▶ MPI_Init()						
▼ fopen64()						
free size	231,314	263.456	878	568	35	221.272
MEMORY LEAK!	1,105,956	1,868.169	592	7,200	32	3,078.574
malloc size	1,358,286	901.318	1,507	7,200	32	2,087.737
▶ OurMain()						
▶ fclose()						

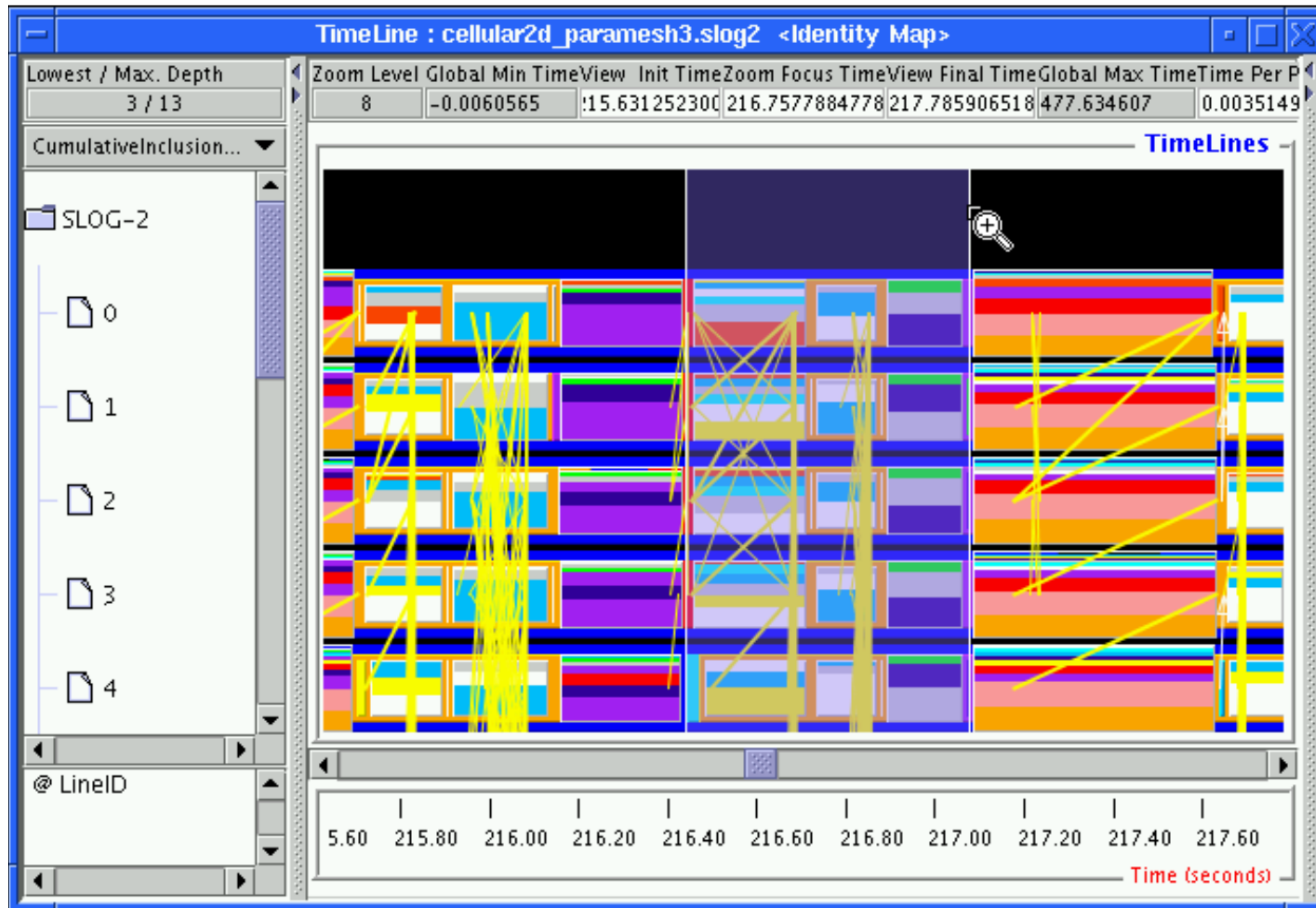
Generating Event Traces

- What happens in a code at a given time?
  - export TAU_TRACE=1
  - export TRACEDIR=/nobackup/global/cbasu/trace
  - Run
  - tau_treemerge.pl
- For Jumpshot:
  - tau2slog2 tau.trc tau.edf -o app.slog2; jumpshot app.slog2
- For Vampir (OTF):
  - tau2otf tau.trc tau.edf app.otf; vampir app.otf
- For ParaVer:
  - tau_convert -paraver tau.trc tau.edf app.prv; paraver app.prv

TAU trace With Jumpshot

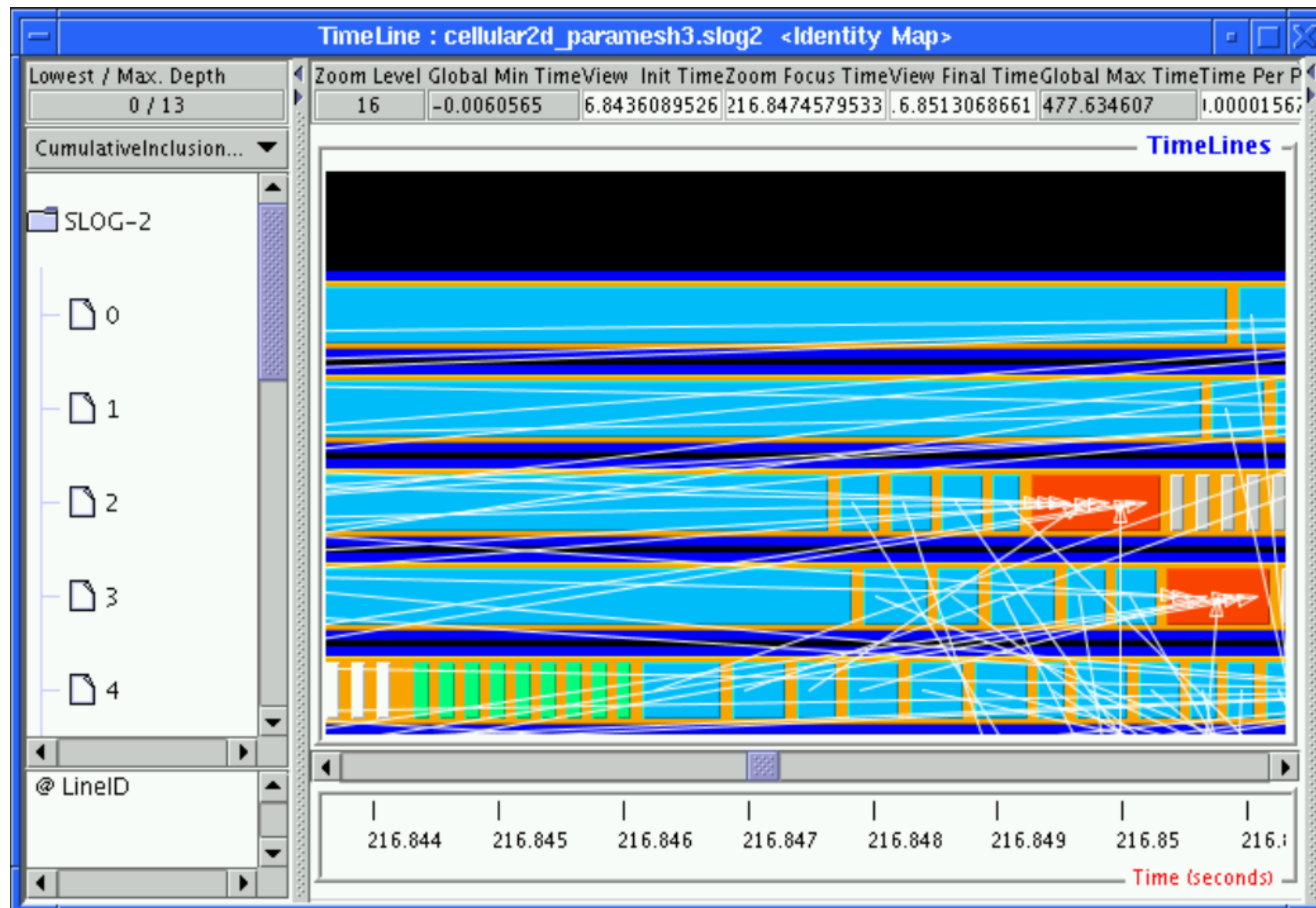
- The timeline canvas is a timeline vs time plot
- Each point on the canvas is identified by two numbers: a timestamp and a timeline ID
- State, arrow, and event : types of objects
- Objects are in preview state when zoomed out
  - Each thick line represents a collection of arrows between its two ending timelines
  - The rectangle that has horizontal strips of colors is the preview state
  - The different colors inside a preview state represent the various categories of real states that are amalgamated within the time range of the preview state

TAU trace Jumpshot



.

TAU trace Jumpshot



TAU trace Jumpshot