TAU Potpourri and Working with Open Components, Interfaces, and Environments

Scott Biersdorff, Chee Wai Lee, Allen D. Malony, Sameer Shende, Wyatt Spear

{scottb,cheelee,malony,shende,wspear}@cs.uoregon.edu

Dept. Computer and Information Science

Performance Research Laboratory

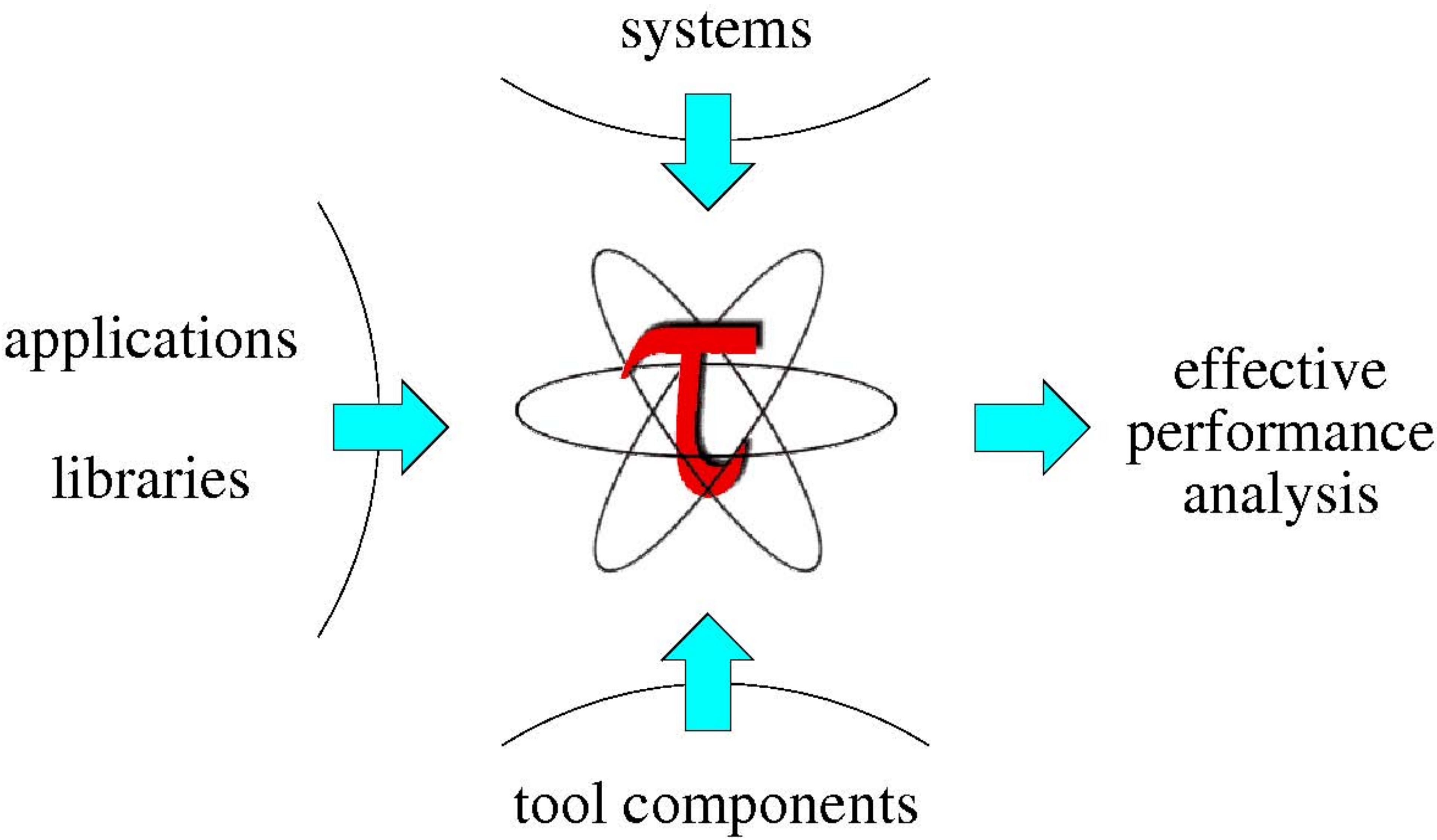
University of Oregon

Potpourri: a mixture of dried petals and spices placed in a bowl, origin 17^{th} century, from French, literally 'rotten pot'

Petal and Spices

- Binary instrumentation: DyninstAPI and *tau_run*
- Hybrid performance measurement: TAUebs
- Library wrapping/interposition: *tau_wrap*, *tau_exec, PARMCI*
- Heterogeneous performance measurement: TAUcuda
- HPC program development and tools: Eclipse and TAU
- Monitoring running applications: TAUmon
- Potpourri smell test

The Pot


systems
applications
libraries
effective
performance
analysis
tool components

Binary Instrumentation: DyninstAPI and tau_run

- TAU and DyninstAPI are mature technologies for performance instrumentation, measurement and analysis
- TAU has been a long-time user of DyninstAPI
- Using DyninstAPI's recent binary re-writing capabilities, created a binary re-writer tool for TAU (*tau_run*)
  - Supports TAU's performance instrumentation
  - Works with TAU instrumentation selection
    - files and routines based on exclude/include lists
  - TAU's measurement library (DSO) is loaded by *tau_run*
- Runtime (pre-execution) and binary re-writing are both supported
- Simplifies code instrumenation and tool usage greatly!
- Included on POINT LiveDVD (tau.uoregon.edu/point.iso)

tau_run with NAS PBS

livetau@paratools01:~

```
/home/livetau% cd ~/tutorial
/home/livetau/tutorial% # Build an uninstrumented bt NAS Parallel Benchmark
/home/livetau/tutorial% make bt CLASS=W NPROCS=4
/home/livetau/tutorial% cd bin
/home/livetau/tutorial/bin% # Run the instrumented code
/home/livetau/tutorial/bin% mpirun -np 4 ./bt_W.4
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Instrument the executable using TAU with DyninstAPI
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run ./bt_W.4 -o ./bt.i
/home/livetau/tutorial/bin% rm -rf profile.* MULT*
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.i
/home/livetau/tutorial/bin% paraprof
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Choose a different TAU configuration
/home/livetau/tutorial/bin% ls $TAU/libTAUsh
libTAUsh-depthlimit-mpi-pdt.so*                libTAUsh-papi-pdt.so*
libTAUsh-mpi-pdt.so*                           libTAUsh-papi-pthread-pdt.so*
libTAUsh-mpi-pdt-upc.so*                       libTAUsh-param-mpi-pdt.so*
libTAUsh-mpi-python-pdt.so*                    libTAUsh-pdt.so*
libTAUsh-papi-mpi-pdt.so*                      libTAUsh-pdt-trace.so*
libTAUsh-papi-mpi-pdt-upc.so*                  libTAUsh-phase-papi-mpi-pdt.so*
libTAUsh-papi-mpi-pdt-upc-udp.so*              libTAUsh-pthread-pdt.so*
libTAUsh-papi-mpi-pdt-vampirtrace-trace.so*    libTAUsh-python-pdt.so*
libTAUsh-papi-mpi-python-pdt.so*
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run -XrunTAUsh-papi-mpi-pdt-vampirtrace-trace bt_W.4 -o bt.vpt
/home/livetau/tutorial/bin% setenv VT_METRICS PAPI_FP_INS:PAPI_L1_DCM
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.vpt
/home/livetau/tutorial/bin% vampir bt.vpt.otf &
```

Going Forward

- Currently, *tau_run* only supports dynamic executables (v6.1)
- Would like support for static binary rewriting
- Would like support for rewriting shared objects
- Validation for compilers other than gcc
  - XLC, PathScale, Cray CCE, Intel, PGI,…
- Availability for more platforms
  - Apple Mac OS X, Windows, IBM BG/P, AIX, …
- Instrumentation at the loop level
- Interaction with generic binary instrumentation

Hybrid Performance Measurement: TAUebs

- Integrate sampling-based and probe-based measurement
- TAUebs combines TAU, PerfSuite, and HPCToolkit
  - TAU for probe-based instrumentation and measurement
  - PerfSuite technology for timer-based sampling
  - HPCToolkit for call stack unwinding on fully-optimized codes
    - problems with StackWalkerAPI at the time ... will retry
- Foundation is TAU with linked SBM capabilities
  - "Context" linking between event stack and call stack
  - Augment PBM with SBM performance views
- TAUebs measurement
- Capture a trace of EBS samples, each containing:
  - Timestamp, TAUkey, PCkey, hardware counters, delta time

TAUebs Data Analysis (Profile)

- Process EBS trace in two ways: profile, trace
- Merged profile analysis with ParaProf
  - Augments TAU profile with PC call stack information
  - Merge stacks for each sample and update TAU profile
  - For all samples that match on TAUkey:
    - distribute TAU inclusive time across PC locations
  - Intermediate routine parent nodes will be inserted in profile
    - only compute inclusive time
  - Can aggregate callsites or show explicitly
- Instrumentation spectrum
  - Top-level on (main) then get profile entirely from EBS
  - All routines then get PC locations merged in profile

TAUebs Data Analysis (Trace)

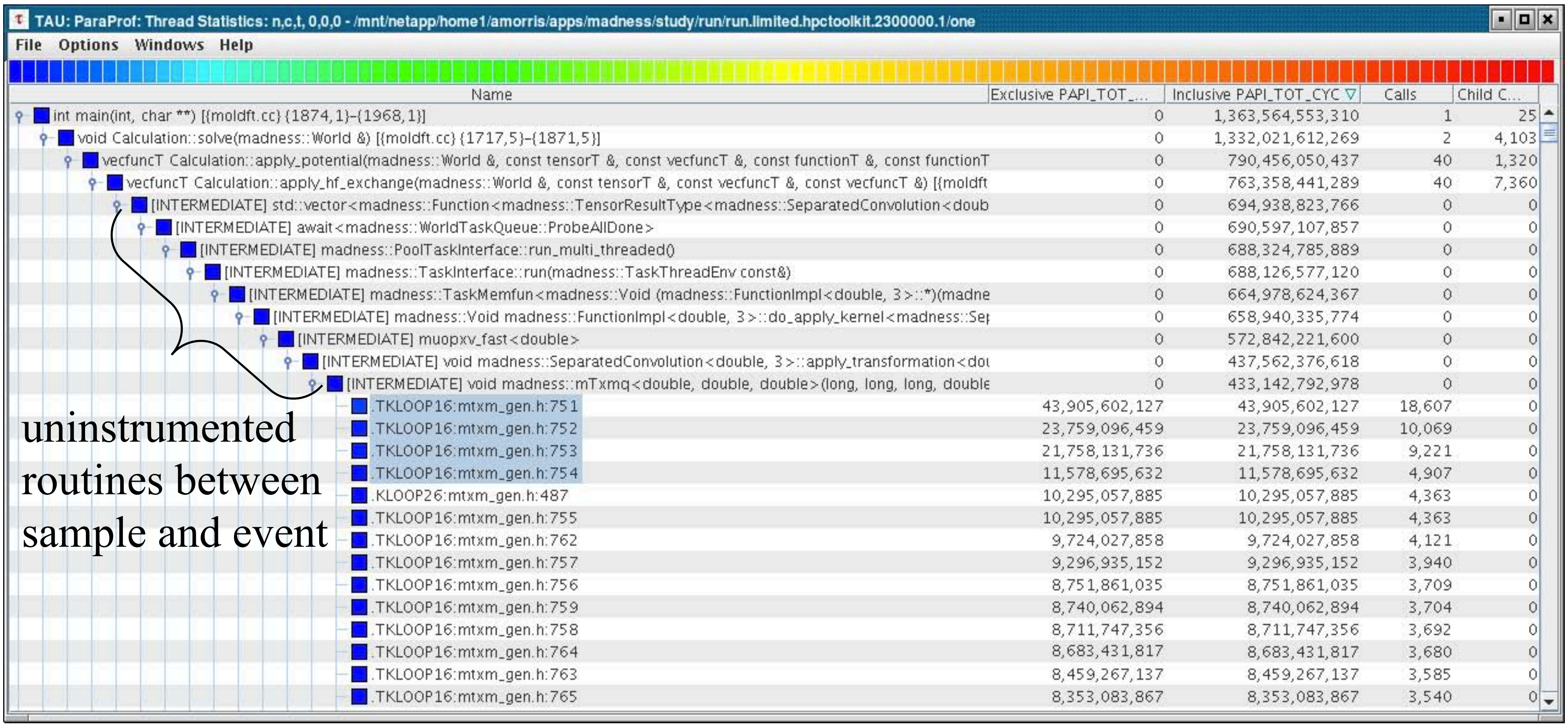
- EBS to OTF trace converter
- Analyze EBS trace with powerful trace analysis tools
- For each sample
  - Place timestamp in trace record
  - Merge TAU event stack and PC call stack into merged call path
  - Create event ID for merged call path and put in trace record
  - Put collected PAPI metrics in trace record
  - Can store PC locations in trace record

Real World Examples

- MADNESS (quantum chemistry application)
  - Heavy use of C++ templates and new features
  - Assembly regions/files and lots of code in header files
  - Makes source instrumentation a challenge
    - TAU source instrumenter could handle a fair amount
  - Instrumentation overhead kicks TAU's butt
    - GNU compiler instrumentation saw 2901% overhead
      - many small routines (getter/setter)
    - TAU source instrumentation with selection (<6%)
      - introduces potential blind spots
- GPAW
- FLASH

TAUebs Profile for MADNESS

- 11 minute run on 8 threads produces 67 MB per thread
- Significant time in .TKLOOP16 which is an assembly file
- Profile for each thread

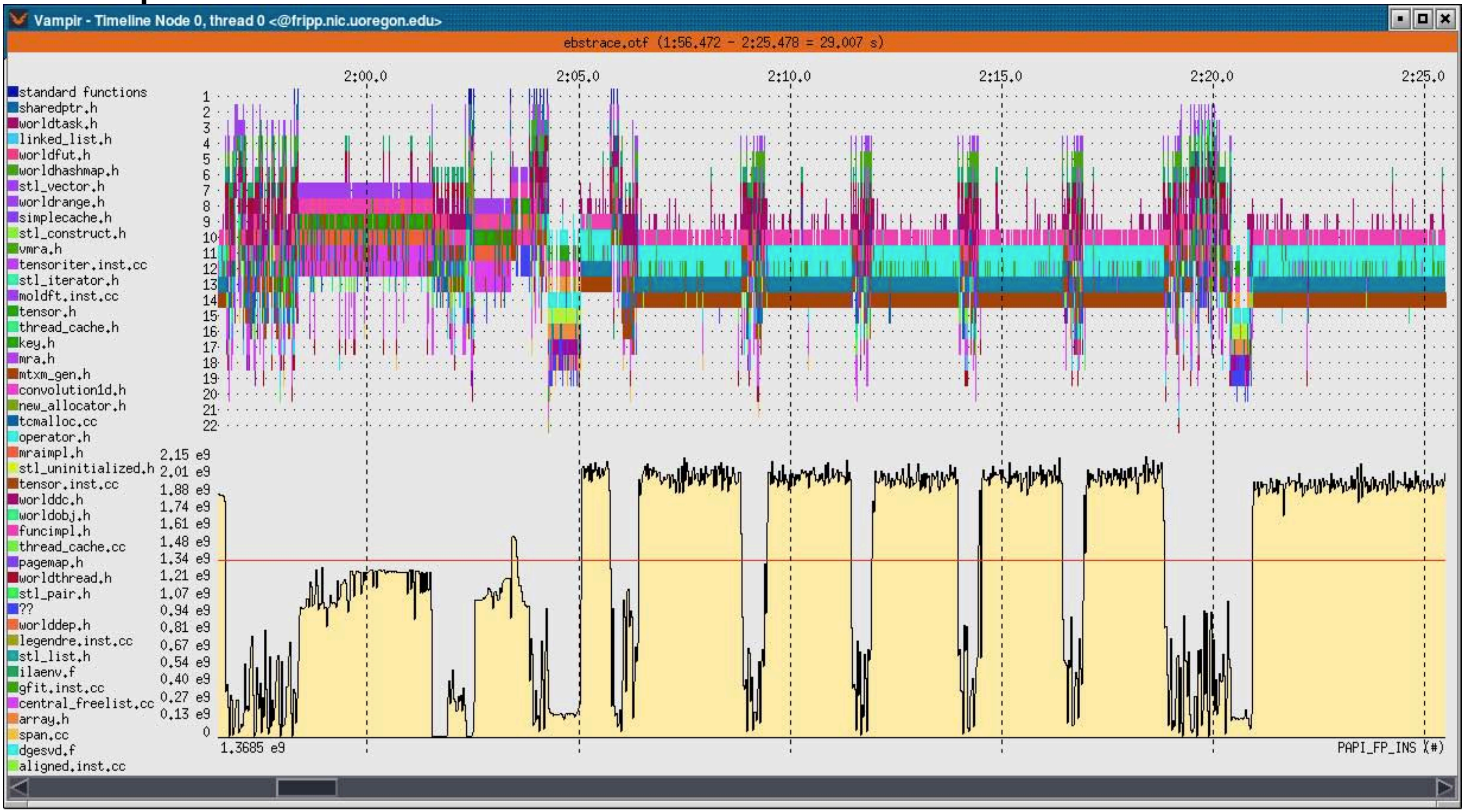


Name	Exclusive PAPI_TOT_...	Inclusive PAPI_TOT_CYC ▽	Calls	Child C...
int main(int, char **) [{moldft.cc} {1874,1}-{1968,1}]	0	1,363,564,553,310	1	25
void Calculation::solve(madness::World &) [{moldft.cc} {1717,5}-{1871,5}]	0	1,332,021,612,269	2	4,103
vecfuncT Calculation::apply_potential(madness::World &, const tensorT &, const vecfuncT &, const functionT &, const functionT	0	790,456,050,437	40	1,320
vecfuncT Calculation::apply_hf_exchange(madness::World &, const tensorT &, const vecfuncT &, const vecfuncT &) [{moldft	0	763,358,441,289	40	7,360
[INTERMEDIATE] std::vector<madness::Function<madness::TensorResultType<madness::SeparatedConvolution<doub	0	694,938,823,766	0	0
[INTERMEDIATE] await<madness::WorldTaskQueue::ProbeAllDone>	0	690,597,107,857	0	0
[INTERMEDIATE] madness::PoolTaskInterface::run_multi_threaded()	0	688,324,785,889	0	0
[INTERMEDIATE] madness::TaskInterface::run(madness::TaskThreadEnv const&)	0	688,126,577,120	0	0
[INTERMEDIATE] madness::TaskMemfun<madness::Void (madness::FunctionImpl<double, 3>::*)(madne	0	664,978,624,367	0	0
[INTERMEDIATE] madness::Void madness::FunctionImpl<double, 3>::do_apply_kernel<madness::Se|	0	658,940,335,774	0	0
[INTERMEDIATE] muopxv_fast<double>	0	572,842,221,600	0	0
[INTERMEDIATE] void madness::SeparatedConvolution<double, 3>::apply_transformation<dou	0	437,562,376,618	0	0
[INTERMEDIATE] void madness::mTxmq<double, double, double>(long, long, long, double	0	433,142,792,978	0	0
.TKLOOP16:mtxm_gen.h:751	43,905,602,127	43,905,602,127	18,607	0
.TKLOOP16:mtxm_gen.h:752	23,759,096,459	23,759,096,459	10,069	0
.TKLOOP16:mtxm_gen.h:753	21,758,131,736	21,758,131,736	9,221	0
.TKLOOP16:mtxm_gen.h:754	11,578,695,632	11,578,695,632	4,907	0
.KLOOP26:mtxm_gen.h:487	10,295,057,885	10,295,057,885	4,363	0
.TKLOOP16:mtxm_gen.h:755	10,295,057,885	10,295,057,885	4,363	0
.TKLOOP16:mtxm_gen.h:762	9,724,027,858	9,724,027,858	4,121	0
.TKLOOP16:mtxm_gen.h:757	9,296,935,152	9,296,935,152	3,940	0
.TKLOOP16:mtxm_gen.h:756	8,751,861,035	8,751,861,035	3,709	0
.TKLOOP16:mtxm_gen.h:759	8,740,062,894	8,740,062,894	3,704	0
.TKLOOP16:mtxm_gen.h:758	8,711,747,356	8,711,747,356	3,692	0
.TKLOOP16:mtxm_gen.h:764	8,683,431,817	8,683,431,817	3,680	0
.TKLOOP16:mtxm_gen.h:763	8,459,267,137	8,459,267,137	3,585	0
.TKLOOP16:mtxm_gen.h:765	8,353,083,867	8,353,083,867	3,540	0

uninstrumented routines between sample and event

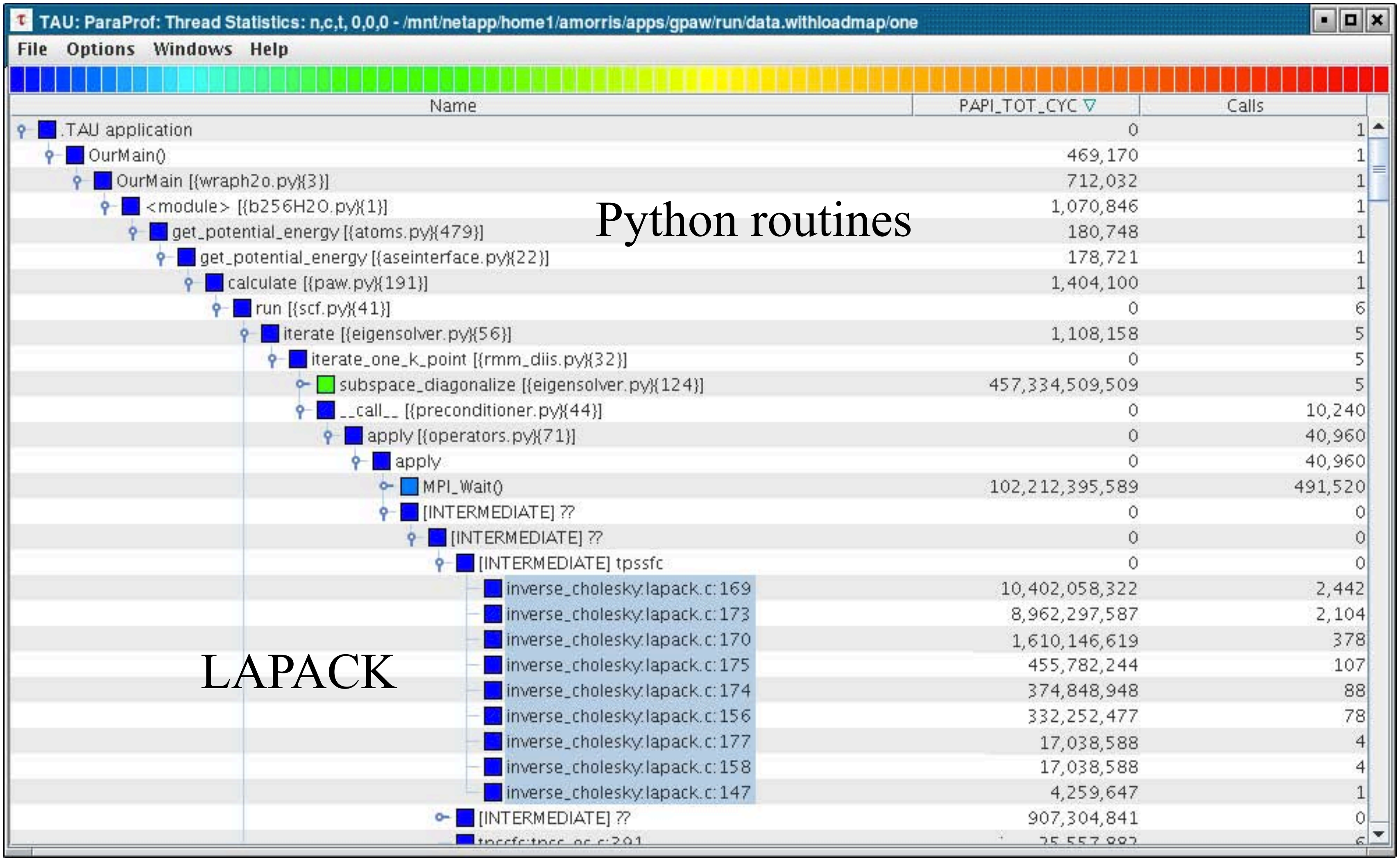
TAUebs Trace for MADNESS

- Vampir call stack color-coded by file name
- Flops rate



GPAW (Grid-Based Projector-Augmented Wave)

- Mixed Python, C, MPI run on 128 processes
- Python performance interface and LD_PRELOAD



Library interposition/wrapping: tau_exec, tau_wrap

- Performance evaluation tools such as TAU provide a wealth of options to measure the performance of an application
- Need to simplify TAU usage to easily evaluate performance properties, including I/O, memory, and communication
- Designed a new tool (*tau_exec*) that leverages runtime instrumentation by pre-loading measurement libraries
- Works on dynamic executables (default under Linux)
- Substitutes I/O, MPI, and memory allocation/deallocation routines with instrumented calls
  - Interval events (e.g., time spent in write())
  - Atomic events (e.g., how much memory was allocated)
- Measure I/O and memory usage

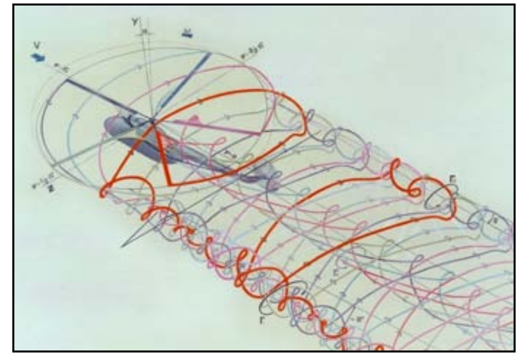
TAU Execution Command (tau_exec)

- Uninstrumented execution
  - % mpirun –np 256 ./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 memory operations
  - % setenv TAU_TRACK_MEMORY_LEAKS 1
  - % mpirun –np 256 tau_exec –memory ./a.out
- Track I/O performance and memory operations
  - % mpirun –np 256 tau_exec –io –memory ./a.out

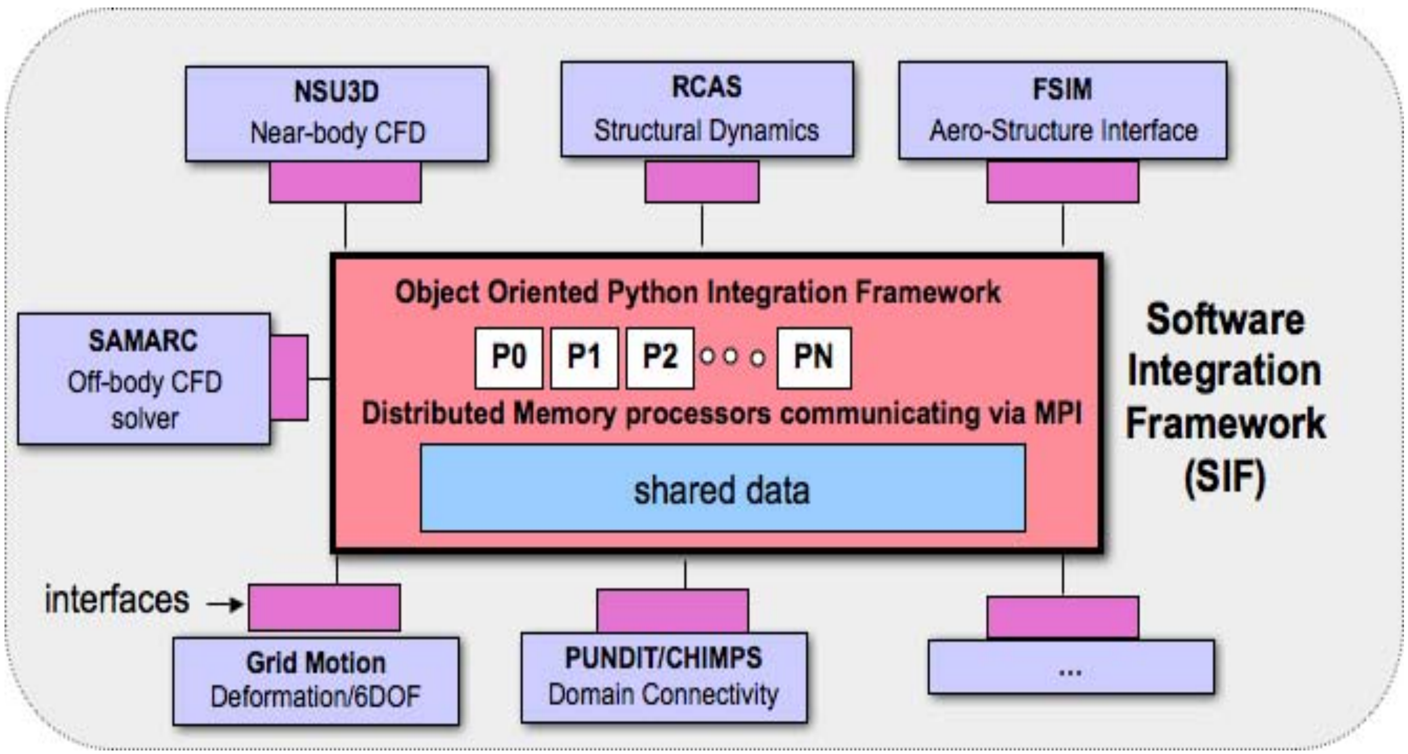
POSIX I/O Calls Supported

- Unbuffered I/O
  - *open, open64, close, read, write, readv, writev, creat, creat64*
- Buffered I/O
  - *fopen, fopen64, fdopen, freopen, fclose*
  - *fprintf, fscanf, fwrite, fread*
- Communication
  - *socket, pipe, socketpair, bind, accept, connect*
  - *recv, send, sendto, recvfrom, pclose*
- Control
  - *fcntl, rewind, lseek, lseek64, fseek, dup, dup2, mkstep, tmpfile*
- Asynchronous I/O
  - *aio_{read,write,suspend,cancel,return}, lio_listio*

HELIOS Rotorcraft Simulation



- HPC Institute for Advanced Rotorcraft Modeling and Simulation (HIARMS)
  - Andy Wissink, US Army, Aeroflight Dynamics Directorate, Ames Research
- Multi-language framework
  - Python (SIF)
  - C/C++
  - Fortran



HELIOS OBE Test

- I/O and memory measurements with Python wrapper

sameer on l2: /usr/var/tmp/sameer/HELIOS/single_block

File Edit View Terminal Tabs Help

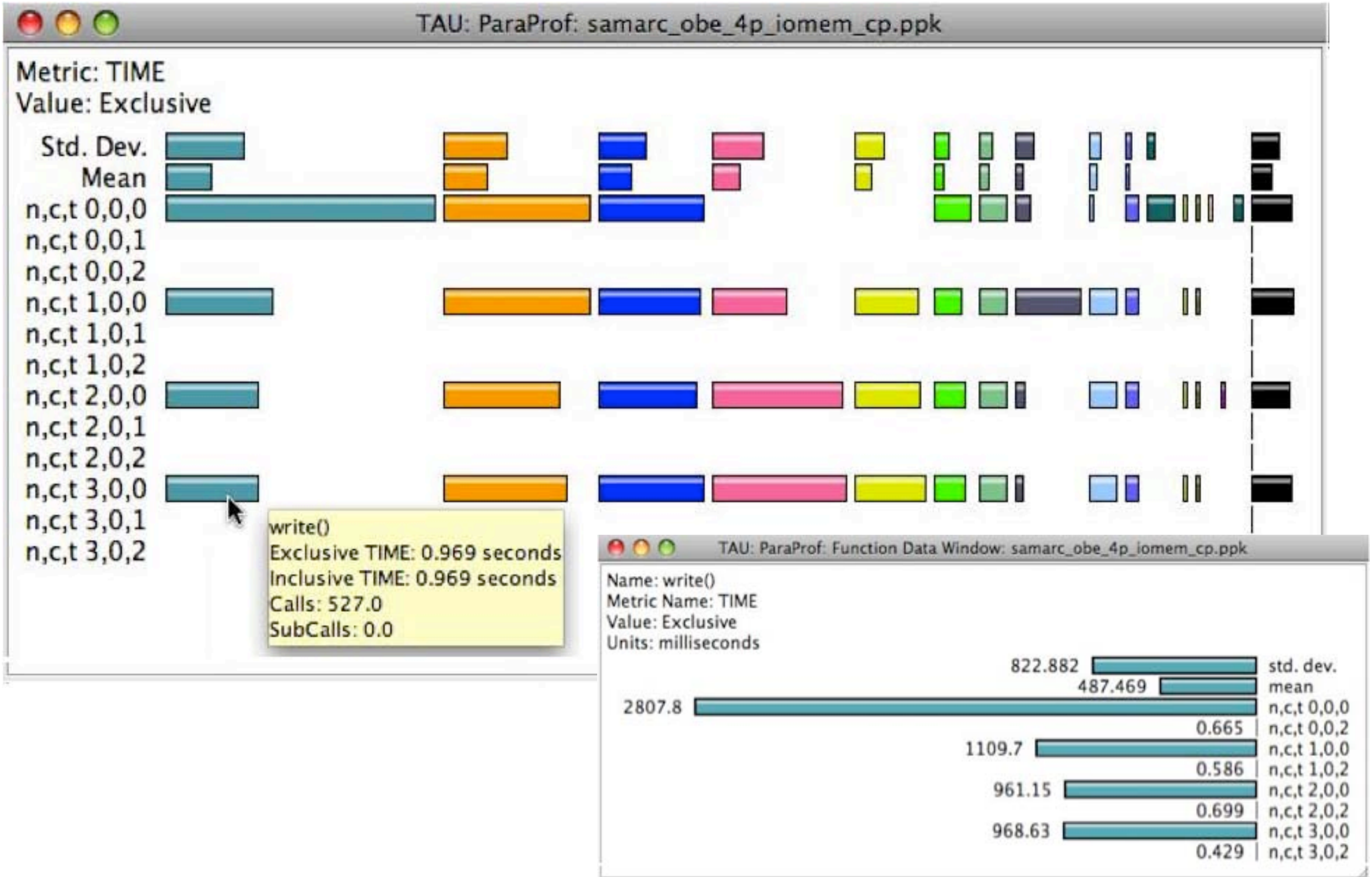
```
Uninstrumented Execution:
 > mpirun -np 4 pyMPI ./obe.py
 >
 > echo "To use with TAU, set PYTHONPATH, and PATH"
To use with TAU, set PYTHONPATH, and PATH
 > setenv PYTHONPATH $PET_HOME/pkgs/tau2/x86_64/lib/bindings-icpc-mpi-pthr
ead-python-pdt/:${PYTHONPATH}
 > set path=($PET_HOME/pkgs/tau2/x86_64/bin $path)
 > cat wrapper.py
import tau

def OurMain():
    import obe

tau.run('OurMain()')

 > mpirun -np 4 tau_exec -io -memory -T python pyMPI ./wrapper.py
 > paraprof
 > pprof
 >
```

Helios OBE Profile



Volume of I/O by File and Memory

TAU: ParaProf: Context Events for thread: n,c,t, 1,0,0 - samarc_obe_4p_iomem_cp.ppk

Name	Total	MeanValue	NumSamples	MinValue	MaxValue	Std. Dev.
▼ .TAU application						
▶ read()						
▶ fopen64()						
▶ fclose()						
▼ OurMain()						
malloc size	25,235	1,097.174	23	11	12,032	2,851.143
free size	22,707	1,746.692	13	11	12,032	3,660.642
▼ OurMain [{wrapper.py}{3}]						
▶ read()						
malloc size	3,877	323.083	12	32	981	252.72
free size	1,536	219.429	7	32	464	148.122
▶ fopen64()						
▶ fclose()						
▼ <module> [{obe.py}{8}]						
▼ writeRestartData [{samarcInterface.py}{145}]						
▼ samarcWriteRestartData						
▼ write()						
WRITE Bandwidth (MB/s) <file="samarc/restore.00002/nodes.00004/proc.00001">		74.565	117	0	2,156.889	246.386
WRITE Bandwidth (MB/s) <file="samarc/restore.00001/nodes.00004/proc.00001">		77.594	117	0	1,941.2	228.366
WRITE Bandwidth (MB/s)		76.08	234	0	2,156.889	237.551
Bytes Written <file="samarc/restore.00002/nodes.00004/proc.00001">	2,097,552	17,927.795	117	1	1,048,576	133,362.946
Bytes Written <file="samarc/restore.00001/nodes.00004/proc.00001">	2,097,552	17,927.795	117	1	1,048,576	133,362.946
Bytes Written	4,195,104	17,927.795	234	1	1,048,576	133,362.946
▶ open64()						

Memory Leaks in MPI

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()						

Library wrapping: tau_wrap

- How to instrument an external library without source?
  - Source may not be available
  - Library may be too cumbersome to build (with instrumentation)
- Build a library wrapper tools
  - Used PDT to parse header files
  - Generate new header files with instrumention files
- Application is instrumented
- Add the -I<wrapper> directory to the command line
- C pre-processor will substitute our headers
  - Redirects references at routine callsite to a wrapper call
  - Wrapper internally calls the original
  - Wrapper has TAU measurement code

HDF5 Library Wrapping

```
[sameer@zorak]$ tau_wrap hdf5.h.pdb hdf5.h -o hdf5.inst.c -f select.tau -g hdf5

Usage : tau_wrap <pdbfile> <sourcefile> [-o <outputfile>] [-r runtimelibname] [-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
```

- instrumented wrapper library source (hdf5.inst.c)
- instrumentation specification file (select.tau)
- group (hdf5)
- creates the wrapper/ directory

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.057            1           1          13       1236 int main(void) C
 70.8        0.875        0.875           1           0        875 hid_t H5Fcreate()
  9.7         0.12         0.12           1           0        120 herr_t H5Fclose()
  6.0        0.074        0.074           1           0         74 hid_t H5Dcreate()
  3.1        0.038        0.038           1           0         38 herr_t H5Dwrite()
  2.6        0.032        0.032           1           0         32 herr_t H5Dclose()
  2.1        0.026        0.026           1           0         26 herr_t H5check_version()
  0.6        0.008        0.008           1           0          8 hid_t H5Screate_simple()
  0.2        0.002        0.002           1           0          2 herr_t H5Tset_order()
  0.2        0.002        0.002           1           0          2 hid_t H5Tcopy()
  0.1        0.001        0.001           1           0          1 herr_t H5Sclose()
  0.1        0.001        0.001           2           0          0 herr_t H5open()
  0.0            0            0           1           0          0 herr_t H5Tclose()
```

NWChem and One-sided Communication

- NWChem relies on Global Arrays (GA)
  - GA is a PGAS programming model
  - provides a global view of a physically distributed array
  - one-sided access to arbitrary patches of data
  - developed as a library
  - fully interoperable with MPI
- Aggregate Remote Memory Copy Interface (ARMCI) is the
  - GA communication substrate forone-sided communication
  - portable high-performance one-sided communication library
  - rich set of remote memory access primitives
- Difficult to test representative workloads for NWChem
  - Lack of use cases for one-side programming models

NWChem Characterization

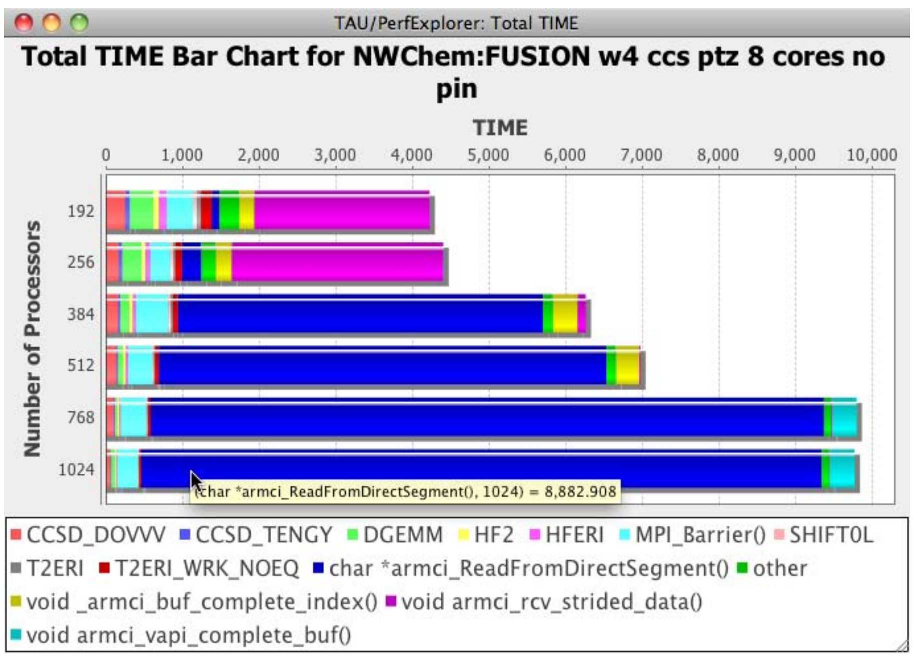
- Strong-scaling of modest problems helps to understand the behavior of larger scientifically significant problems
  - represent behavior of real calculations on future systems
- Understand interplay between data-server and compute processes as a function of scaling
  - Large numerical computation per node at small scale can obscure the cost of maintaining passive-target progress
  - Larger scale decreases numerical work per node and increases the fragmentation of data, increasing messages
  - Vary #nodes, cores-per-node, and memory buffer pinning
- Understand trade-off of core allocation
  - all to computation versus some to communication

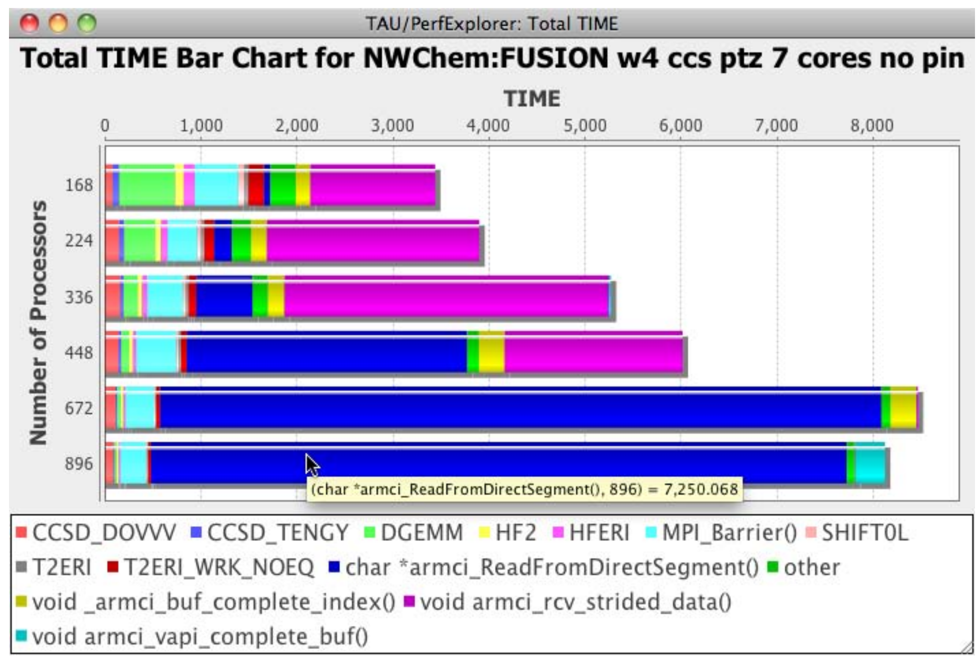
NWChem Instrumentation

- Source-base instrumentation of NWChem application routines
- Developed an ARMCI interposition library (PARMCI)
  - defines weak symbols and name-shifted PARMCI interface
  - similar to PMPI for MPI
- Developed a TAU PARMCI library
  - intervals events around interface routines
  - atomic events to capture communication size and destination
- Wrapped external libraries
  - BLAS (DGEMM)
- Need portable instrumentation to conduct cross-platform experiments

FUSION Tests with Varying Cores

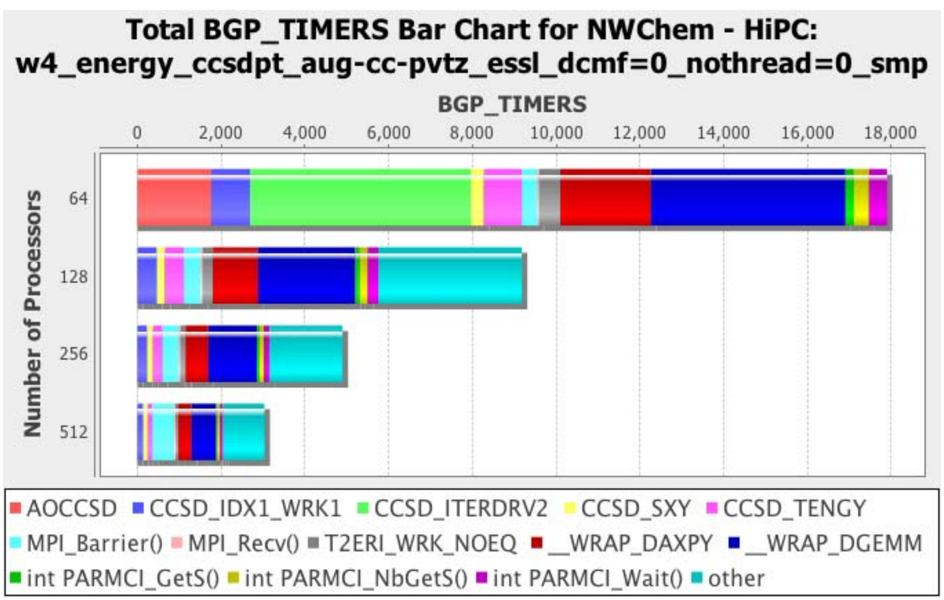
- Scaling on 24, 32, 48, 64, 96 and 128 nodes
- Test on 8 and 7 cores with pinning disabled
  - Dedicated data server with 7 cores
- Relative ARMI communication overhead increases with greater number of nodes (cores)

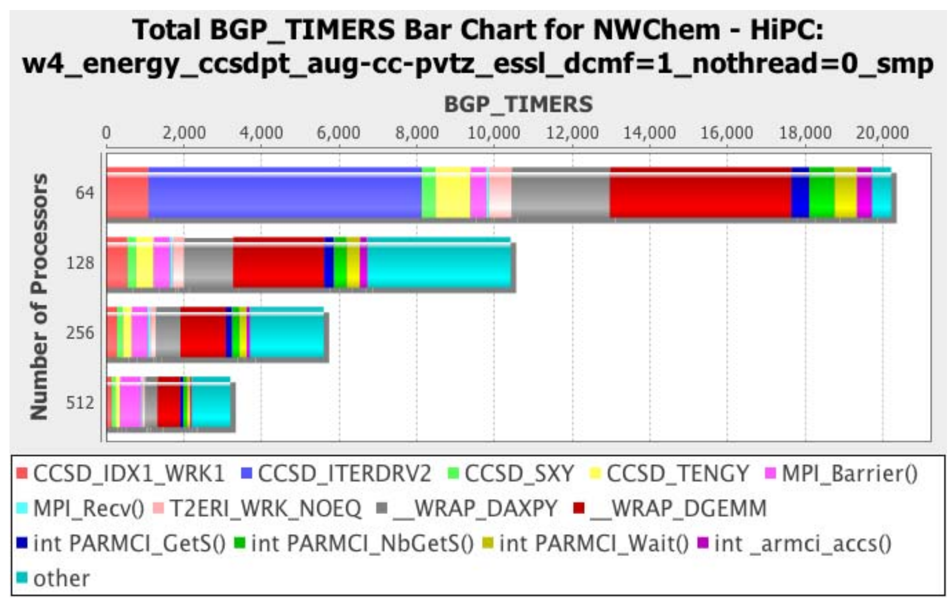




Intrepid Tests

- Scaling on 64, 128, 256 and 512 nodes
- Tests with interrupt or communication helper thread (CHT)
  - CHT requires a core to be allocated
- ARMCI calls are barely noticeable
- DAXPY calculation shows up more
- CHT performs better in both SMP and DUAL modes





Heterogeneous Systems Measurement: TAUcuda

- Want to create performance views that capture heterogeneous concurrency and execution behavior
  - Reflect interactions between heterogeneous parts
  - Capture performance semantics relative to computation model
  - Assimilate performance for all execution paths for shared view
- What perspective do we have of the parts?
  - Determines the semantics of the measurement data
  - Determines assumptions about behavior and interactions
  - Performance views may have to work with reduced data
- Need to work with heterogeneous system components
- Developed TAUcuda for CUDA performance measurement
  - TAUcuda v1 discussed at CScADS 2009

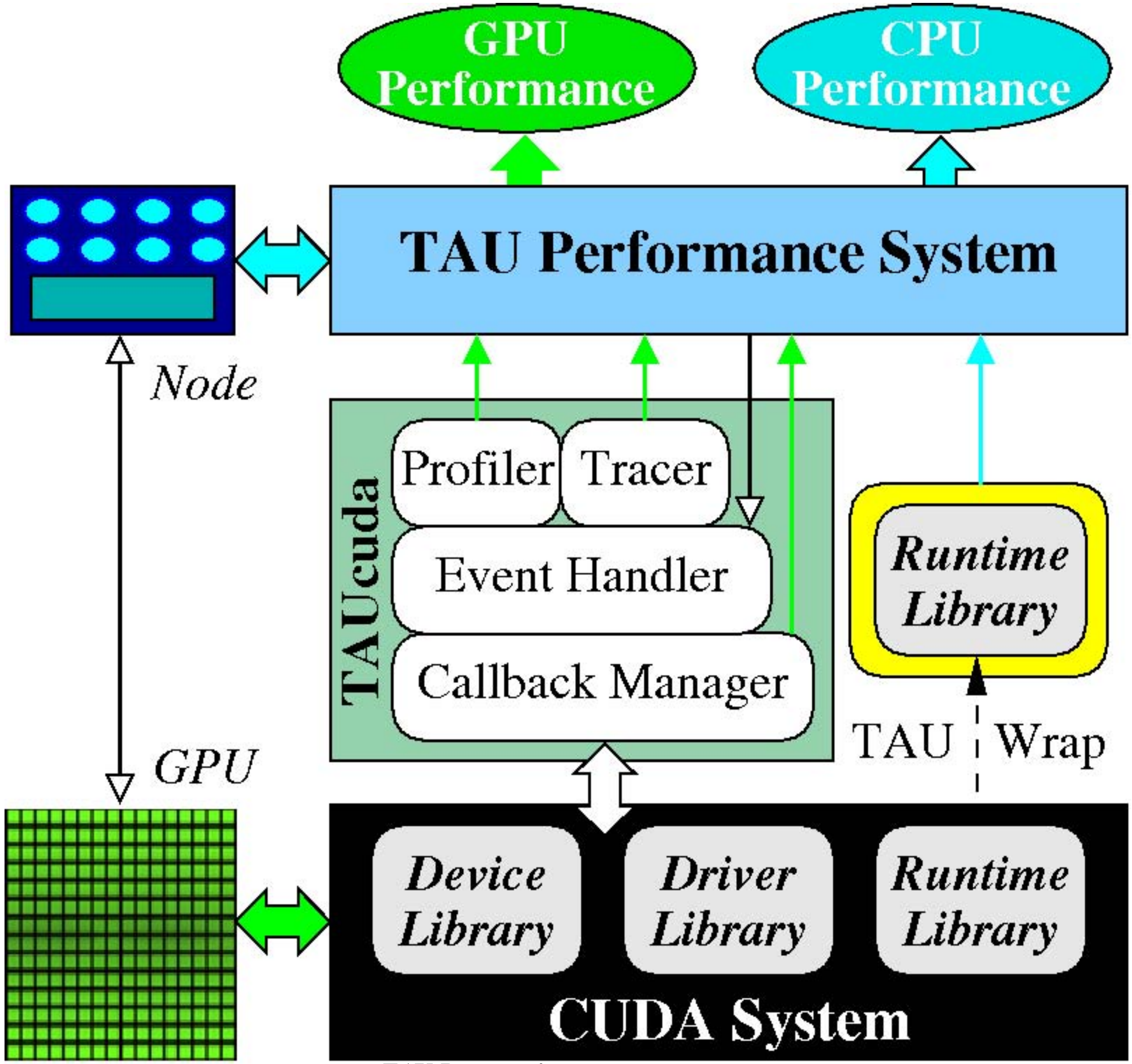
TAUcuda Performance Measurement (Version 2)

- Overcome TAUcuda (v1) deficiencies
  - Required source code instrumentation
  - Event interface only perspectives
    - could not see memory transfer or CUDA system execution
- CUDA system architecture
  - Implemented by CUDA libraries
    - driver and device (*cuXXX*) libraries
    - runtime (*cudaYYY*) library
  - Tools support (Parallel Nsight (Nexus), CUDA Profiler)
    - not intended to integrate with other HPC performance tools
- TAUcuda (v2) built on experimental Linux CUDA driver
  - Linux CUDA driver R190.86 supports a callback interface!!!

TAUcuda Architecture

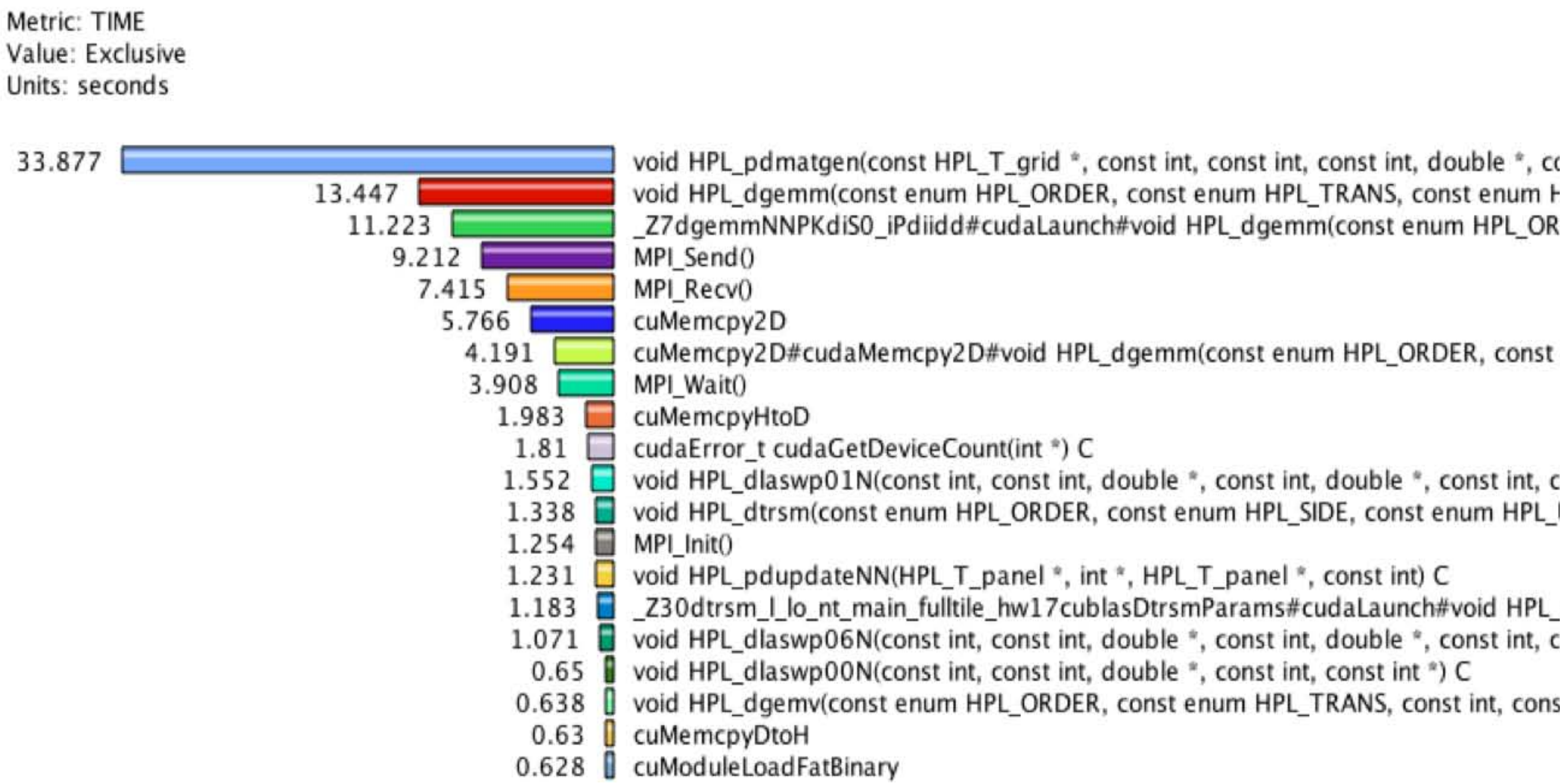
TAUcuda events

TAU events

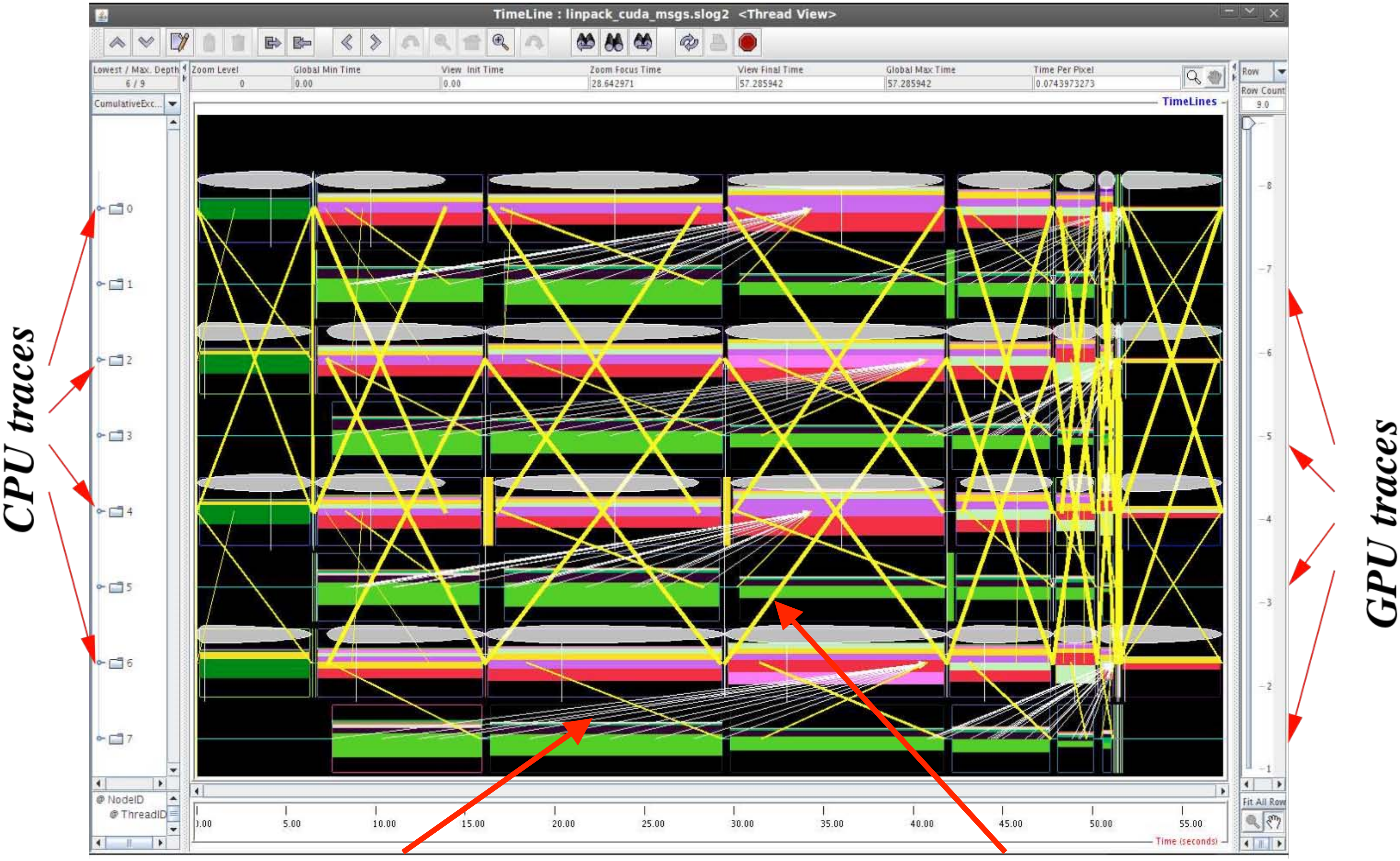


CUDA Linpack Profile (4 processes, 4 GPUs)

- Measure performance of heterogeneous parallel applications
- GPU-accelerated Linpack benchmark (M. Fatica, NVIDIA)



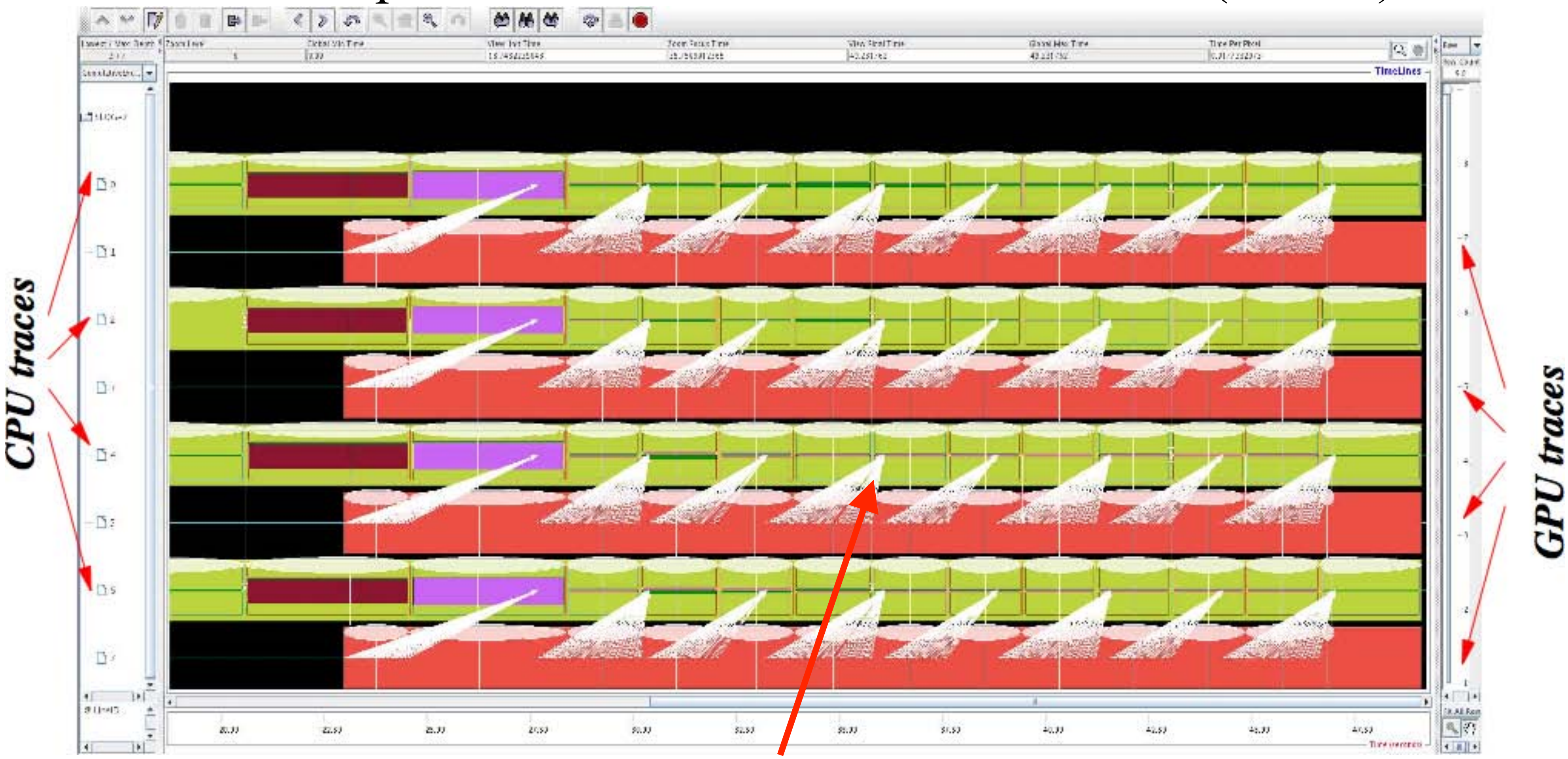
CUDA Linpack Trace



CUDA memory transfer (white) MPI communication (yellow)

SHOC Stencil2D (512 iterations, 4 CPUxGPU)

- ❐ Scalable HeterOgenerous Computing benchmark suite
  - ❍ CUDA / OpenCL kernels and microbenchmarks (ORNL)



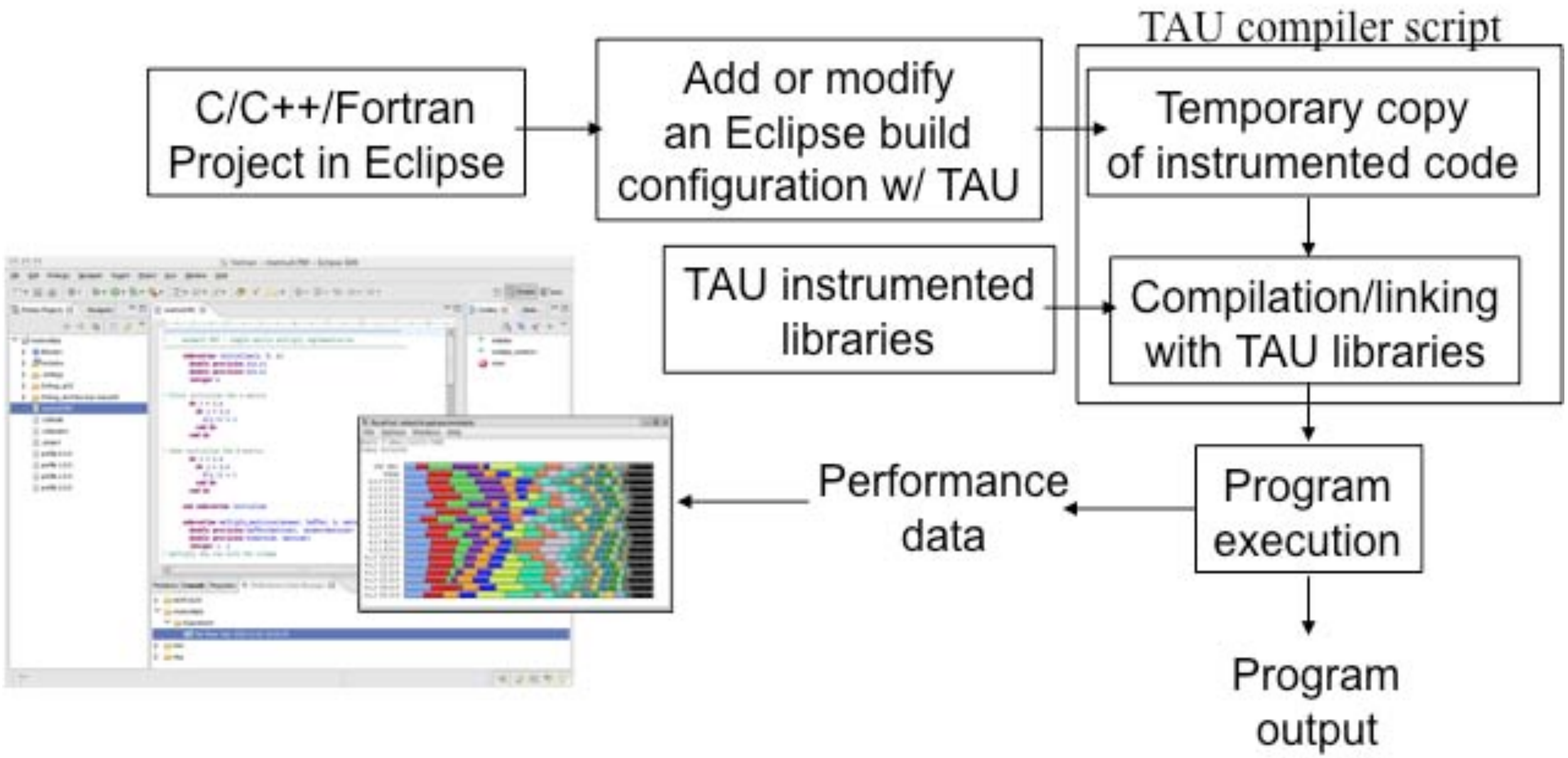
CUDA memory transfer (white)

TAU and Eclipse

- How to make performance measurement, analysis, and tuning a part of the software development cycle?
- Multi-year work with Eclipse IDE (www.eclipse.org)
  - Benefits: portable, project transition: familiar interface, supports multiple languages (Java, C/C++, Fortran, …)
  - Features: syntax highlighting, refactoring, code management
  - Modular plug-in based architecture allows for easy extension
  - Environments: JDT, CDT, PTP (*www.eclipse.org/{jdt,cdt,ptp}*)
- High-performance software development environments
  - IDE features for parallel programming + parallel tools
  - Eclipse PTP: integrate features and interface with parallel tools

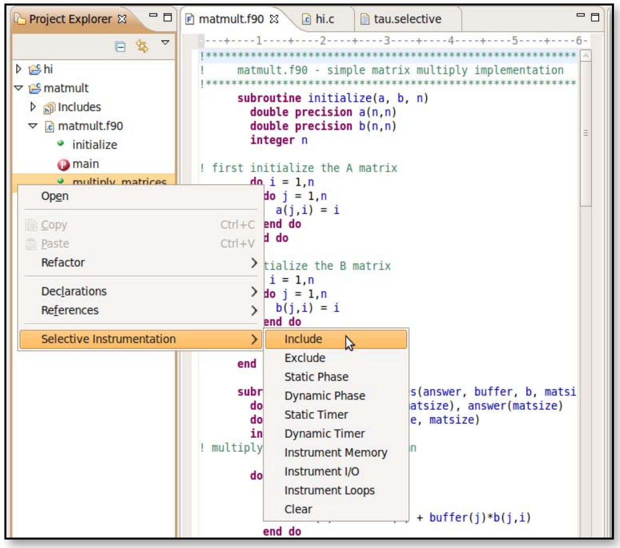
TAU and Eclipse

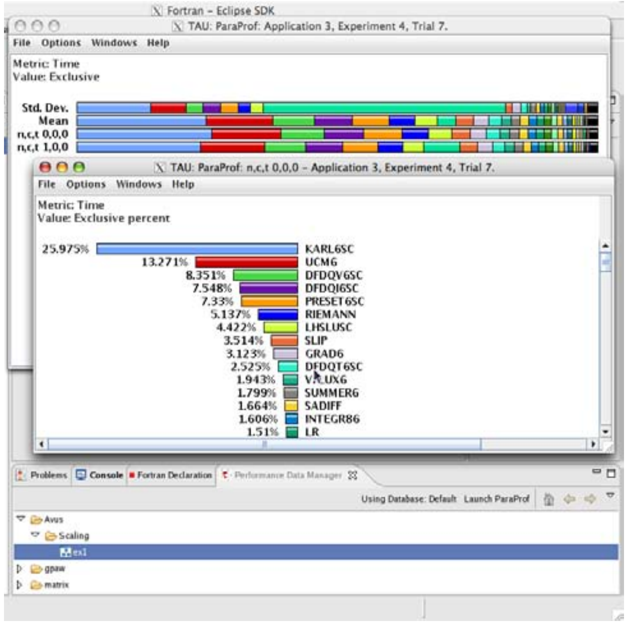
- Provide an interface for configuring TAU's automatic instrumentation within Eclipse's build system
- Manage runtime configuration settings and environment variables for execution of TAU instrumented programs



Integration Features

- Chose different TAU configurations
- Select options for control of instrumentation and compilation
- Integrated interface for generating and choosing selective instrumentation
- PAPI counter selection
- Profile data generated in Eclipse is stored in a PerfDMF database
- Performance databases can be browsed from within Eclipse
  - Trials loaded in the ParaProf
- Source callback allows quick navigation





Dynamic Tool Definitions

- Developed *External Tools Framework* (ETFw)
  - Initially to extend and generalize the TAU plug-ins
  - Considered for general tool integration in Eclipse
- TAU plug-ins' functionality was generalized to XML for:
  - Portability and ease of modification
  - Simpler alternative to Eclipse plug-in for tool integration
  - Use additionally for workflow creation
- Tools selected /configured in a launch configuration window
- ETFx adds Eclipse support for analysis tools including:
  - Valgrind, PerfSuite, Scalasca, VampirTrace
- Other tool developers are leveraging the ETFw

Monitoring running Applications: TAUmon

- Scalable access to a running application’s performance information is valuable
- Access can happen after an application completes (but before parallel teardown) or while an application is still running
- Two-way access needed for support of advanced operations
- TAUmon
  - Design as a transport-neutral application monitoring framework
  - Base on prior /existing work with various transport systems:
    - Supermon, MRNet, MPI
- Recent work by Chee Wai Lee

Overall design principles

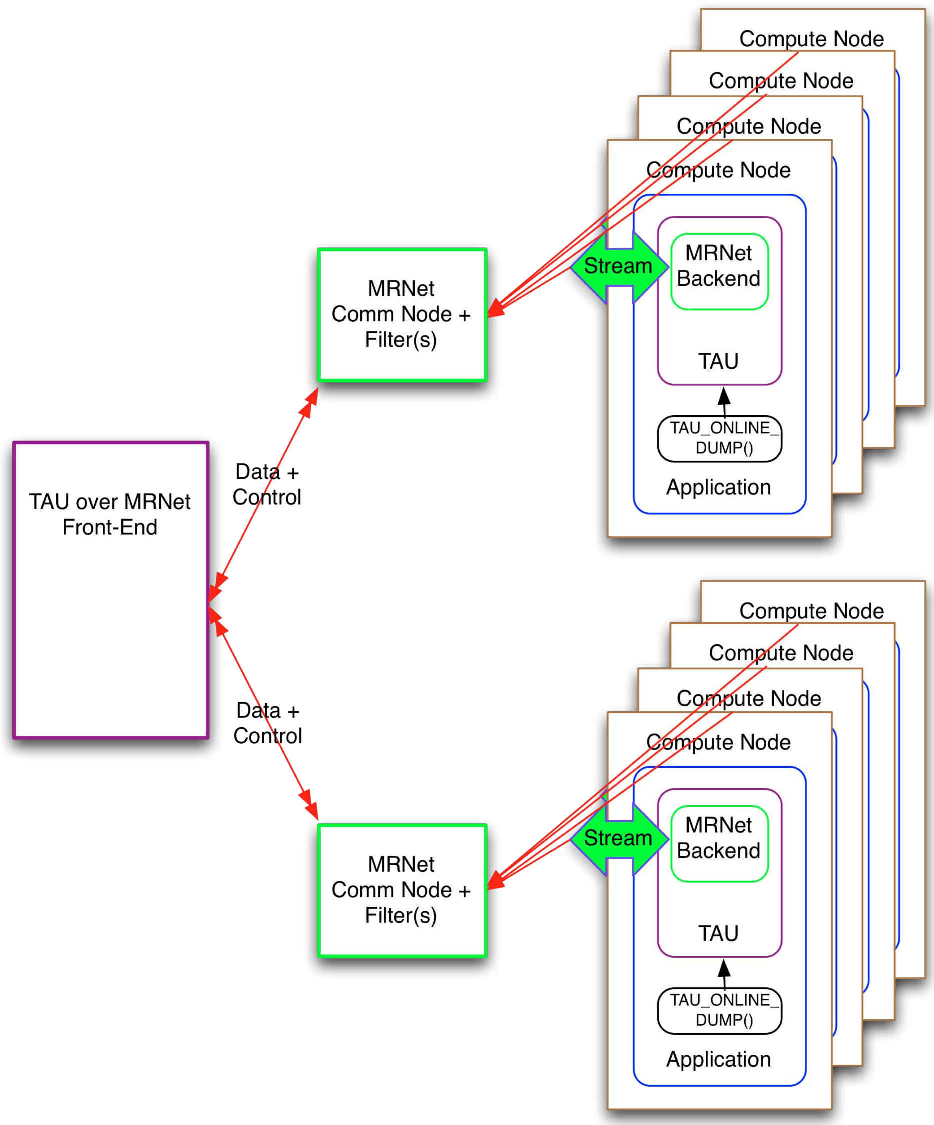
- Modular and transparent access to parallel transport systems
- Support for minimal user intervention with different system-specific launch mechanisms
- Modular support for scalable monitoring operations
  - Based on aggregation algorithms and techniques
  - Simple overall statistics: mean, min, max, standard deviation
  - Histograms
  - Clustering results (various types)
- Modular support for data delivery to output locations
  - Local or remote visualization/analysis tools
  - Local or remote storage

Current implementation and API

- *TAU_ONLINE_DUMP()* collective operations in application
  - Called by all thread / processes
  - Works with parallel profiles
- Appropriate version of TAU selected for transport system
- User instruments application with TAU support for desired monitoring transport system
- User submits instrumented application to parallel job system
- Other launch systems must be submitted along with the application to the job scheduler as needed
  - *Currently supported through different machine-specific job-submission scripts*

TAUmon and MRNet

- Overview
  - Scripts set up runtime infrastructure
  - TAU frontend coordinates gathering operations when requested
  - Application backends collectively initiate operations in a push-based approach
  - MRNet tree nodes facilitate scalable gather operations

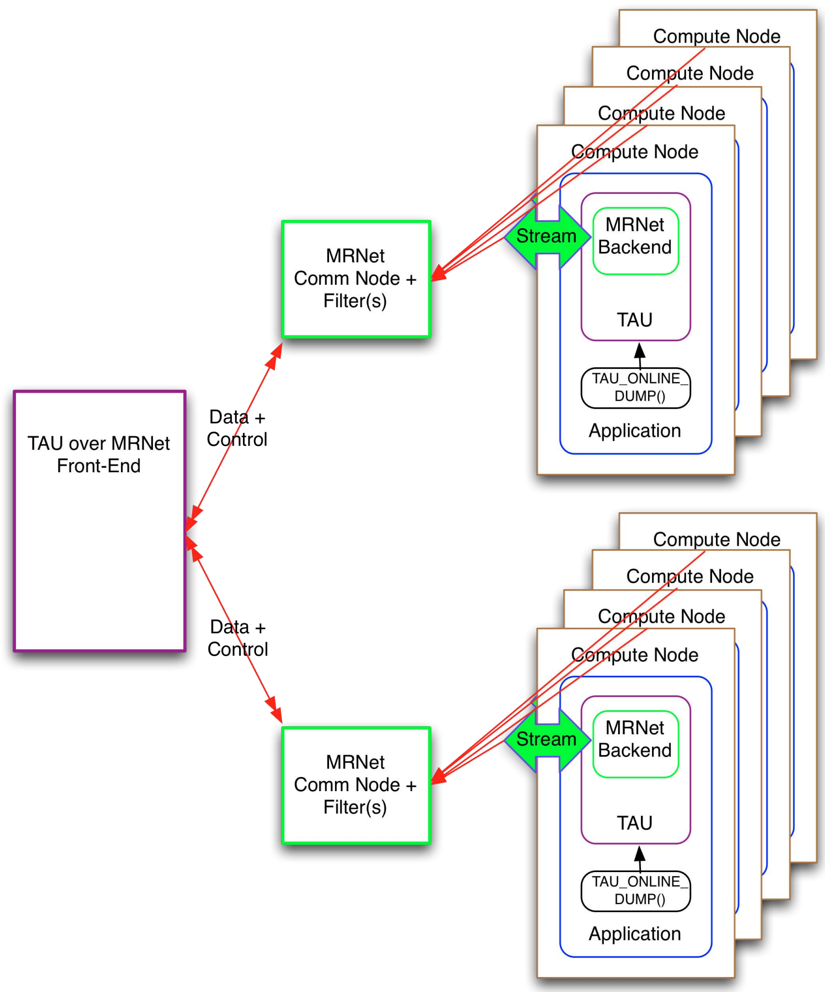


MRNet Network Configuration

- Scripts used to set up MRNet network configuration
  - Given ***P*** = *number of cores* for the application, the user can choose an appropriate ***N*** = *number of tree nodes* and ***K*** = *fanout* for deciding how to allocate sufficient computing resources for both application and MRNet
  - Number of network leaves can be computed as (***N***/***K***)*(***K***-1)
  - Probe processes discover and partition computing resources between the application and MRNet
  - ***mrnet_topgen*** utility will write a topology file given ***K*** and ***N*** and a list of processor hosts available exclusively for MRNet
  - TAU frontend reads topology file to create the MRNet tree and then write a new file to inform application how it can connect to the leaves of the tree

Monitoring Operation with MRNet

- Application collectively invokes *TAU_ONLINE_DUMP()* to start monitoring operations using current performance information
- TAU data is accessed and sent through MRNet's communication API via streams and filters
- Filters perform appropriate aggregation operations on data
- TAU frontend is responsible for collecting the data, storing it, and eventual delivery to a consumer



Experiences with MRNet - 1

- Parallel system-specific (e.g. Cray XT5 and BG/P) launch mechanisms required
- Key technical challenges:
  - Efficient data offload from application to MRNet tree
  - Support for user control of MRNet tree for performance
- Other challenges:
  - Current compiler-related incompatibility on the Cray
  - Providing uniform launch scripts across different parallel machines

Experiences with MRNet - 2

- Extra computing resources must be dedicated to MRNet tree
  - This can be viewed as an advantage or limitation
- Resources required are system-dependent:
  - On Cray systems, MRNet front-end resides on login node, intermediate tree nodes reside on dedicated (set aside by user) compute nodes, application processes (backends) reside on the remaining compute nodes
  - On BG/P systems, MRNet front-end (and possibly some tree nodes) reside on login node, intermediate tree nodes reside on IO nodes (not known a-priori until after compute nodes are launched), backends reside on compute nodes

TAUmon and MPI

- Also developing TAUmon to use MPI-based transport
  - No separate launch mechanisms required
  - Parallel gather operations implemented as a binomial heap with staged MPI point-to-point calls (Rank 0 serves as root)
- Limitations:
  - Application shares the same parallel infrastructure with monitoring transport
  - Monitoring operations may cause performance intrusion
  - Currently, no flexibility for user control of transport network configuration

TAUMon: Early results with PFLOTRAN (Cray)

- MRNet as transport
  - Only exclusive time is being monitored

XT5 Nodes (Total)	Cores (Total)	Cores (Application Only)	Mean Aggregation Time (per iteration)	Histogram Generation Time (per iteration)
374	4,488	4,104	0.2204s	0.07313s
559	6,708	6,144	0.3308s	0.1411s
746	8,952	8,196	0.4586s	0.1864s
1,118	13,416	12,288	0.6439s	0.2839s

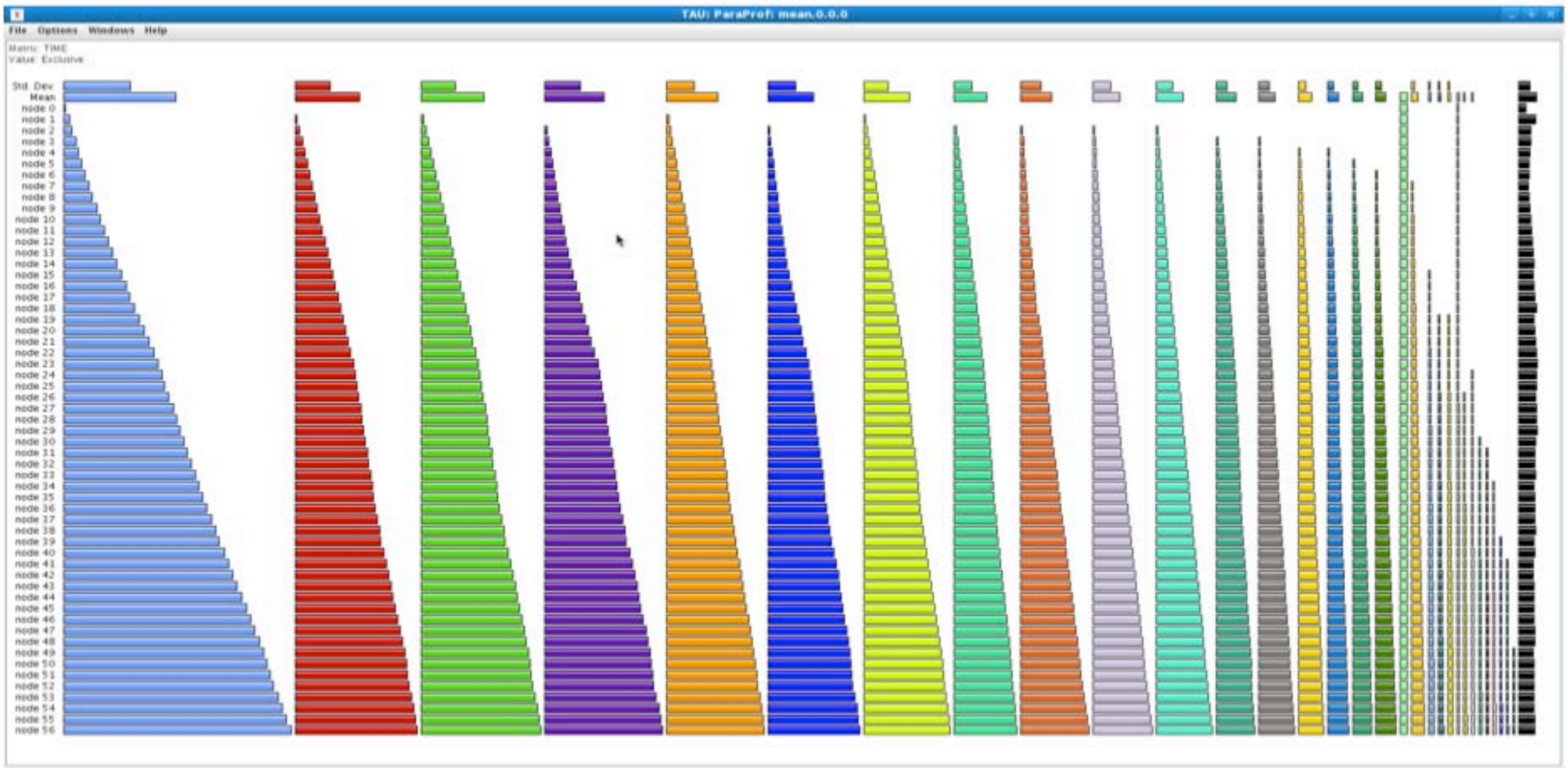
TAUMon: Early results with PFLOTRAN (Cray)

- MPI as transport
  - Only exclusive time is being monitored

XT5 Nodes	Cores	Unification Time (per iteration)	Mean Aggregation Time (per iteration)	Histogram Generation Time (per iteration)	Total Operation Time (per iteration)
342	4,104	0.02514s	0.01677s	2.339s	2.384s
512	6,144	0.02244s	0.02215s	2.06s	2.115s
683	8,196	0.04067s	0.03347s	3.651s	4.278s
1,024	12,288	0.07241s	0.02621s	0.8643s	0.9676s
1,366	16,392	0.03382s	0.01431s	1.861s	3.053s
2,048	24,576	0.02976s	0.03569s	0.6238s	0.6921s

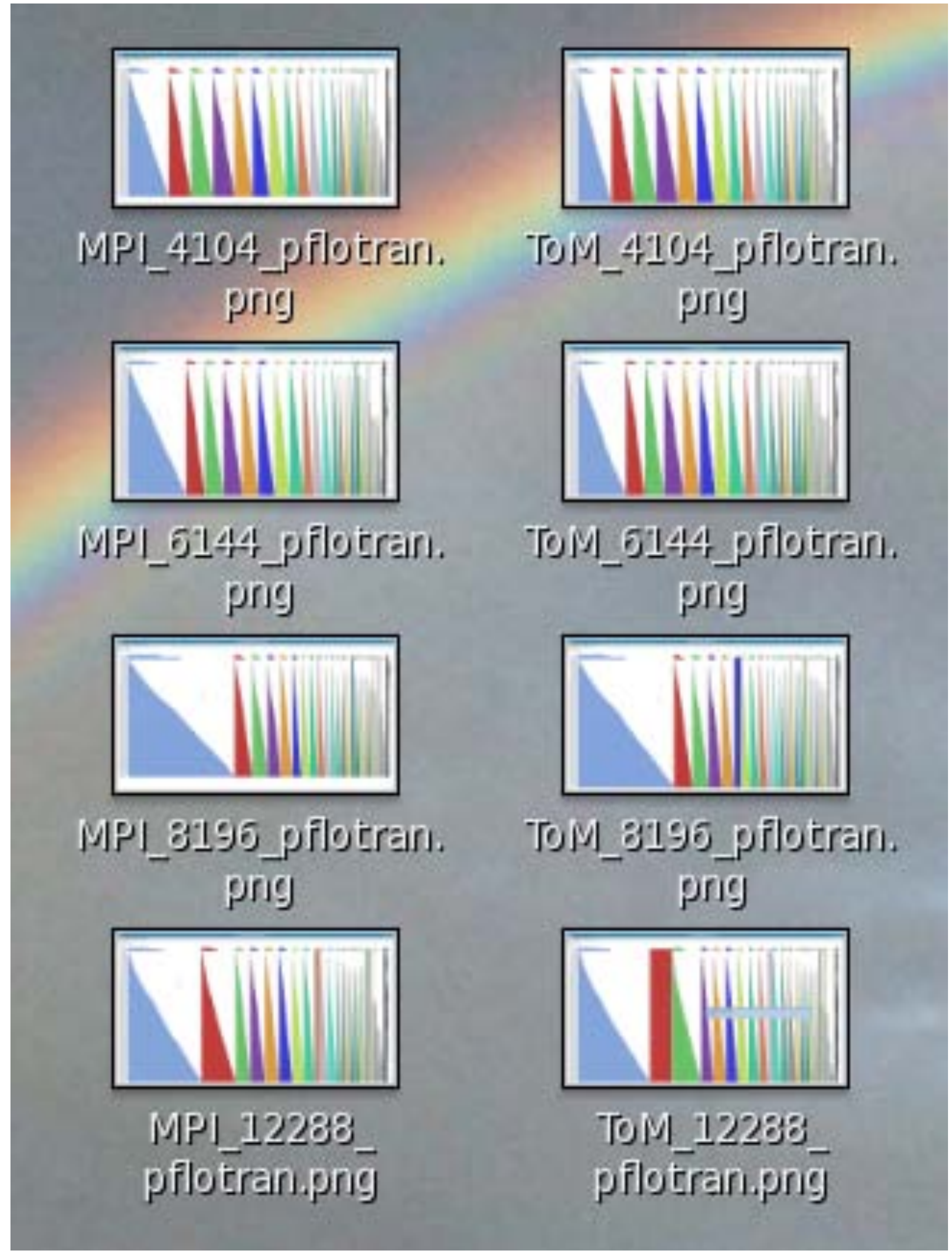
TAUMon: Early results with PFLOTRAN (Cray)

- MRNet as transport
  - 4104 cores running with 374 extra cores for MRNet transport
- Each line bar shows the mean profile of an iteration



TAUmon: Early visualization with ParaProf

- A quick side-by-side look at data monitored using MPI (left column) and MRNet (right column)
- MPI_Allreduce (blue triangle) appears to grow inordinately over time when PFLOTRAN is executed on 8,196 cores



Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science
  - ASC/NNSA
- Department of Defense (DoD)
  - HPC Modernization Office (HPCMO)
- NSF Software Development for Cyberinfrastructure (SDCI)
- Research Centre Juelich
- Argonne National Laboratory
- Technical University Dresden
- ParaTools, Inc.
- NVIDIA