

NCCS Brown Bag Series

September 4, 2012



Code Optimization Using TAU

Chongxun (Doris) Pan
doris.pan@nasa.gov



Agenda



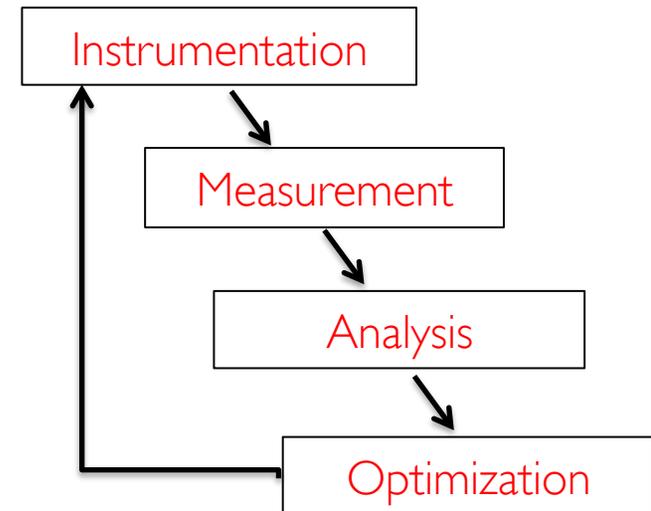
- Overview of performance analysis and available tools on Discover
- Introduction to TAU
- Demo 1: Performance tuning for an OpenMP application
- Demo 2: Performance measurement and memory leak detection for an MPI application



Typical Performance Bottlenecks



- **Your Application**
 - ❖ Synchronization, load balance, communication, memory usage, I/O usage
- **System Architecture**
 - ❖ Memory hierarchy, network latency, processor architecture, I/O system setup
- **Software**
 - ❖ Compiler options, libraries, runtime environment, communication protocols...





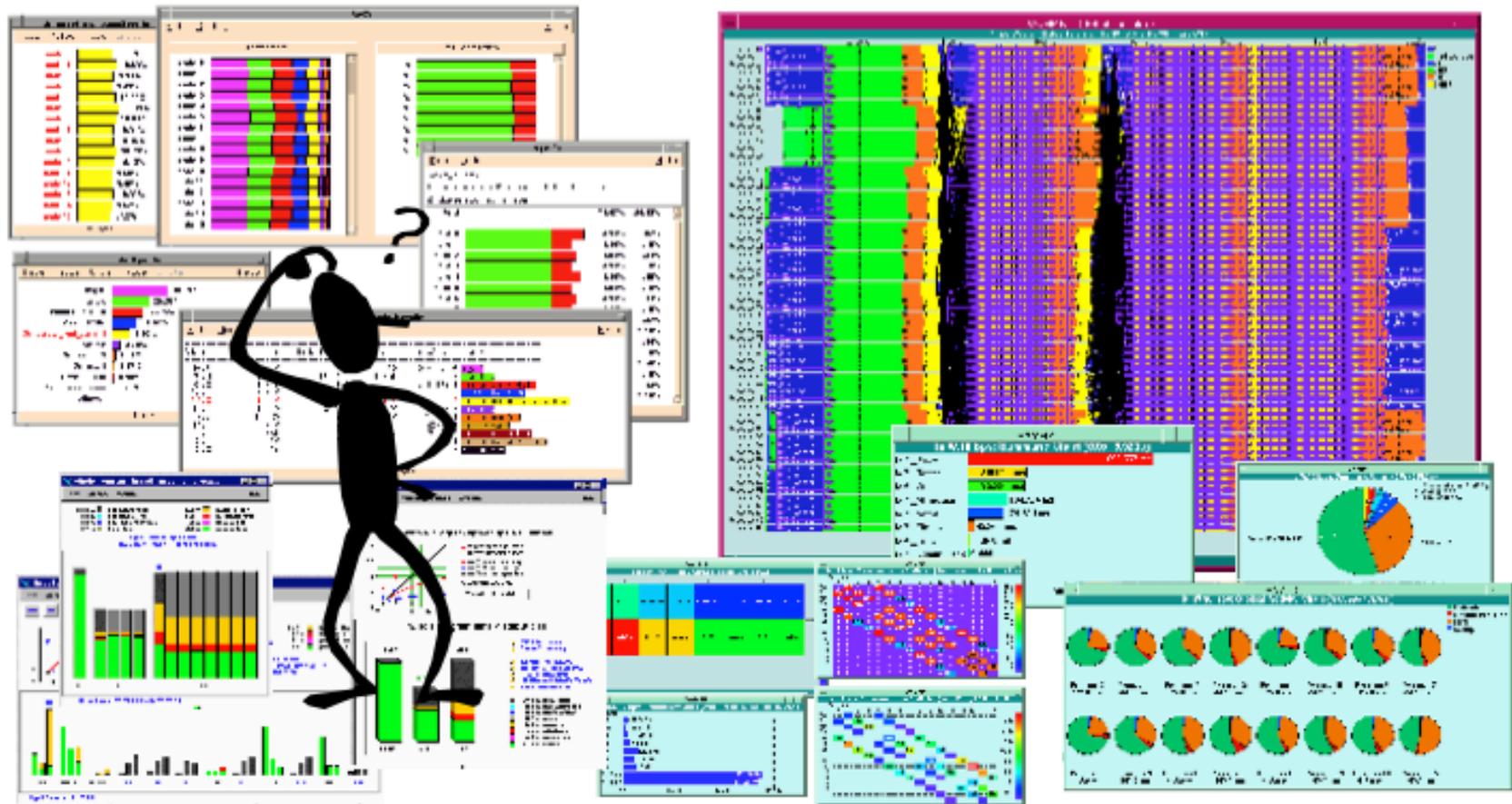
Understanding Bottlenecks Is Essential



- Different tools measure different performance metrics
 - ❖ **Profiling tools:** Aggregating statistics at run time, e.g., run time on each function, total size of message sent or received. Profiling data volumes are small.
 - ❖ **Gprof, mpiP, Intel Profiler, VTune Amplifier, IPM, TAU**
 - ❖ **Tracing tools:** Collecting event history (*when* the events take place in each process along a timeline). Tracing data volumes are large.
 - ❖ **VampirTrace, TAU**
 - ❖ **Memory Profiling tools:** Collecting heap memory usage per processor, and detecting memory errors and leaks.
 - ❖ **TotalView/MemScape, Valgrind/Memcheck, TAU**



Now, Which Tool?





Tuning and Analysis Utilities (TAU)



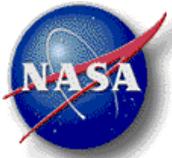
- Performance problem solving framework for HPC
 - ❖ Portable, scalable, flexible. And open source
 - ❖ Target all parallel programming/execution paradigms
 - ❖ Fortran, C/C++. CUDA and OpenCL
 - ❖ Multi-threading, MPI, MPI/OpenMP hybrid
 - ❖ Even including MIC. Native support coming up
- Integrated performance toolkit includes
 - ❖ Instrumentation, Measurement, Analysis, Visualization
 - ❖ Widely-ported performance profiling / tracing system



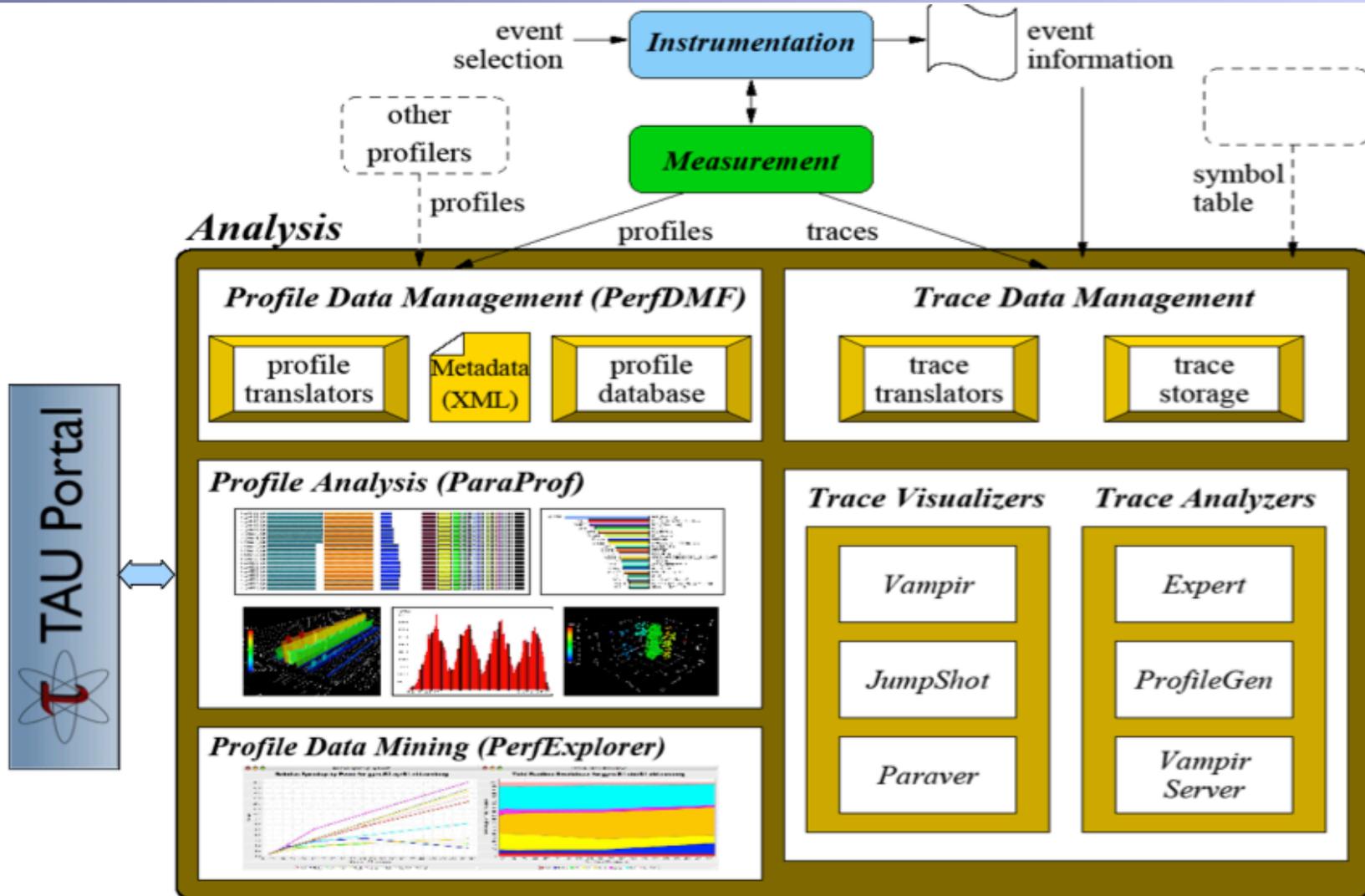
TAU: Usage Scenarios



- How much time is spent on each routine and outer *loops*? Within loops, what is the time contribution of each line statement?
- What is the *peak heap memory* usage of the code? When and where is memory allocated/deallocated? Any *memory leaks*?
- How does the code *scale* with different core counts? Efficiency and run time breakdown of performance?
- How much time used performing *I/O* in the code? What is the peak read and write *bandwidth* of individual calls, and total volume?
- How many instructions are executed in some code regions? *Floating point*, L1 and L2 *cache misses*, hits, branches taken?



Big Picture, First





Big Picture, Explained



Instrumentation: Adds probes to perform measurements

- Source code instrumentation using pre-processors and compiler scripts
- External library wrapping (MPI, I/O, Memory, CUDA, OpenCL, pthread)

Measurement: Profiling or Tracing using wallclock time or HW counters (PAPI unavailable on Discover though)

- Interval events measure exclusive and inclusive durations
- Throttling and run time control of low-level events that execute frequently

Analysis: Visualization of profiles and traces

- 3D visualization of profile data in paraprof and perfexplorer
- Trace conversion/display in external tools (Vampir, Jumpshot, ParaVer)



Take Home Messages



- TAU satisfies many of our code performance tuning needs, and a whole lot more...
- It has a lot of components, and requires steep learning curve to master the tool. But...
- No worries! There are many ways to do the same things here, **so just learning a couple of tricks could be sufficient for you**



Take Home Messages (*Cont'd*)



- This brown-bag will focus on:
 - ❖ Getting you started using TAU on Discover
 - ❖ Showing you where to find help while you are trying it
- Useful references:
 - ❖ <http://tau.uoregon.edu/tau.ppt>
 - ❖ <http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/10-tau-gpu-tutorial-part1.pdf>
 - ❖ <http://www.vi-hps.org/datapool/vihpstw8/TAU.pdf>
 - ❖ Under our Primer:
<http://www.nccs.nasa.gov/primer/computing.html#tau>



Ok, Let's get started...



We will explain the concepts, steps, and details involved in performance evaluation with TAU using two real case usage scenarios:

- Demo 1: Performance tuning for an OpenMP application (*My openmp code does not run faster with multiple threads, what should I do?*)
- Demo 2: Performance tuning for an MPI application (*How does the load balance of my MPI code look like? What is the heap memory usage for each routine? Any memory leaks?*)



Demo 1: Performance Tuning for an OpenMP application



1. Instrumentation:

- **Compiler based instrumentation** easily generates routine level performance data
- **Automatic source instrumentation** uses PDT for more detailed instrumentation at the fine-grained loop, I/O tracking, memory allocation, etc
- Program Database Toolkit (PDT) is a framework for analyzing source code in multiple languages
- **You will have to set a couple of environment variables and substitute the name of your compiler with a TAU shell script**
 - ❖ Use **tau_f90.sh**, **tau_cxx.sh**, or **tau_cc.sh** to replace ifort/mpif90, icpc/mpicxx, or icc/mpicc respectively



Demo 1 – Hands-on



```
$ setenv PATH ${PATH}:/discover/nobackup/cpan2/lib/tau-2.21.2/x86_64/bin
```

Or place it in your shell startup files, e.g. .cshrc

```
$ setenv TAU_MAKEFILE /discover/nobackup/cpan2/lib/tau-2.21.2/x86_64/lib/  
Makefile.tau-icpc-pdt-openmp-opari
```

```
$ setenv TAU_OPTIONS "-optVerbose -optKeepFiles"
```

Edit the Makefile, e.g.,

```
FC = tau_f90.sh # to replace ifort or mpif90
```

```
$ make
```

TAU uses different TAU_MAKEFILE for different configuration measurements, e.g., to configure TAU using PDT and OpenMP

```
$ ./configure -openmp -c++=icpc -fortran=intel -cc=icc \
```

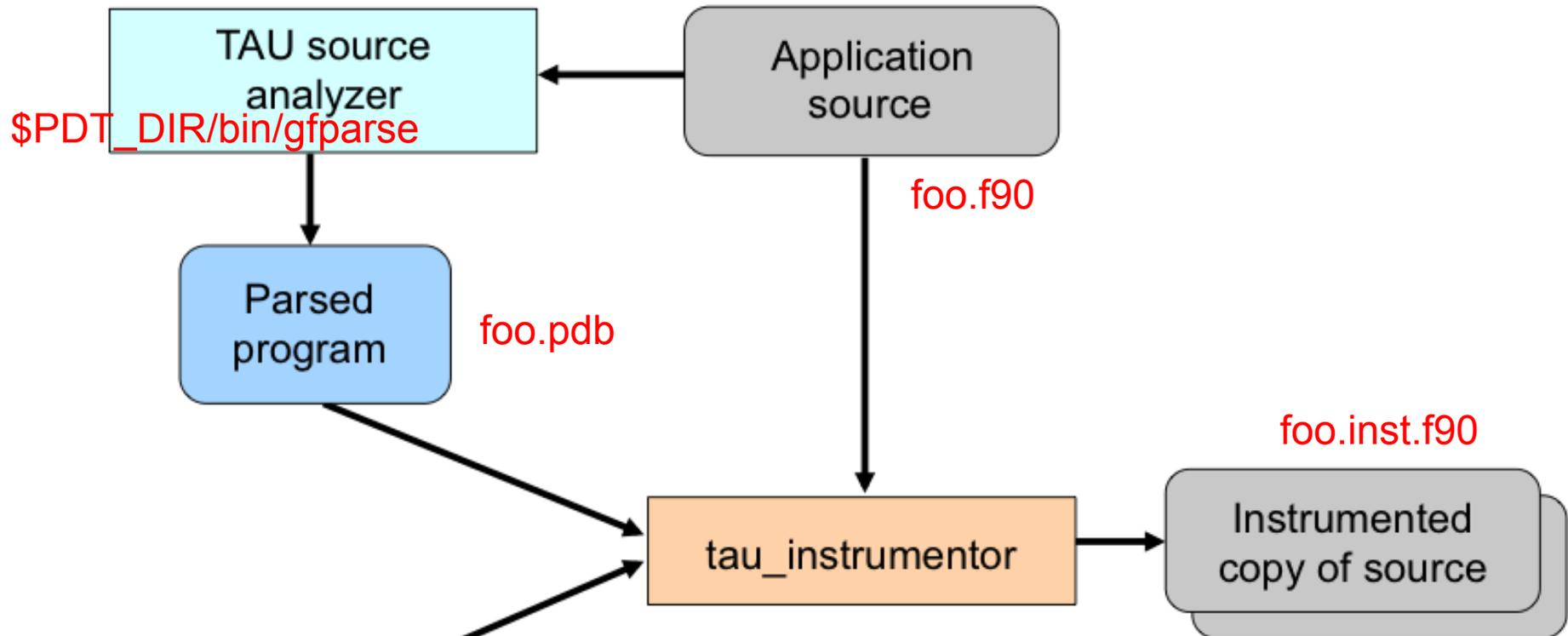
```
-pdt=/discover/nobackup/cpan2/lib/pdtoolkit-3.17 \
```

```
-opari -opari_region -opari_construct
```

```
$ make install
```



Source instrumentation using tau_xx.sh and PDT



Optional user-defined specification file, e.g.
`BEGIN_INSTRUMENT_SECTION`
`Loops file="foo.f90" routine="#"`
`END_INSTRUMENT_SECTION`



Compile-time options TAU_OPTIONS



% tau_compiler.sh

- optVerbose Turn on verbose debugging messages
- optComplnst Use compiler based instrumentation
- optNoComplnst 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*)
- 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*) ...



Demo 1 – Hands-on



2. Measurement

Run the job via PBS using the executable compiled with tau_instrumentor:

```
#PBS -l select=1:ncpus=12
#PBS ...
...
setenv OMP_NUM_THREADS 8
setenv OMP_STACKSIZE 2G
setenv OMP_AFFINITY compact
setenv I_MPI_PIN_DOMAIN auto

setenv TAU_CALLPATH 1
setenv TAU_CALLPATH_DEPTH 100

cd ..
./<executable>
```

3. Analysis

Multiple profile data will be generated after the job completes.

```
$ ls profile*
profile.0.0.0 profile.0.0.1
... profile.0.0.8
$ paraprof --pack \
openmp_example_baseline.ppk
$ paraprof \
openmp_example_baseline.ppk
```

You can look at the bundled profile data examples using “paraprof”:

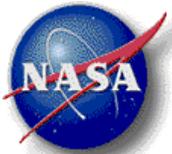
[/discover/nobackup/cpan2/Brownbag/openmp_example_baseline.ppk](#)
and [openmp_example_4t_final.ppk](#)



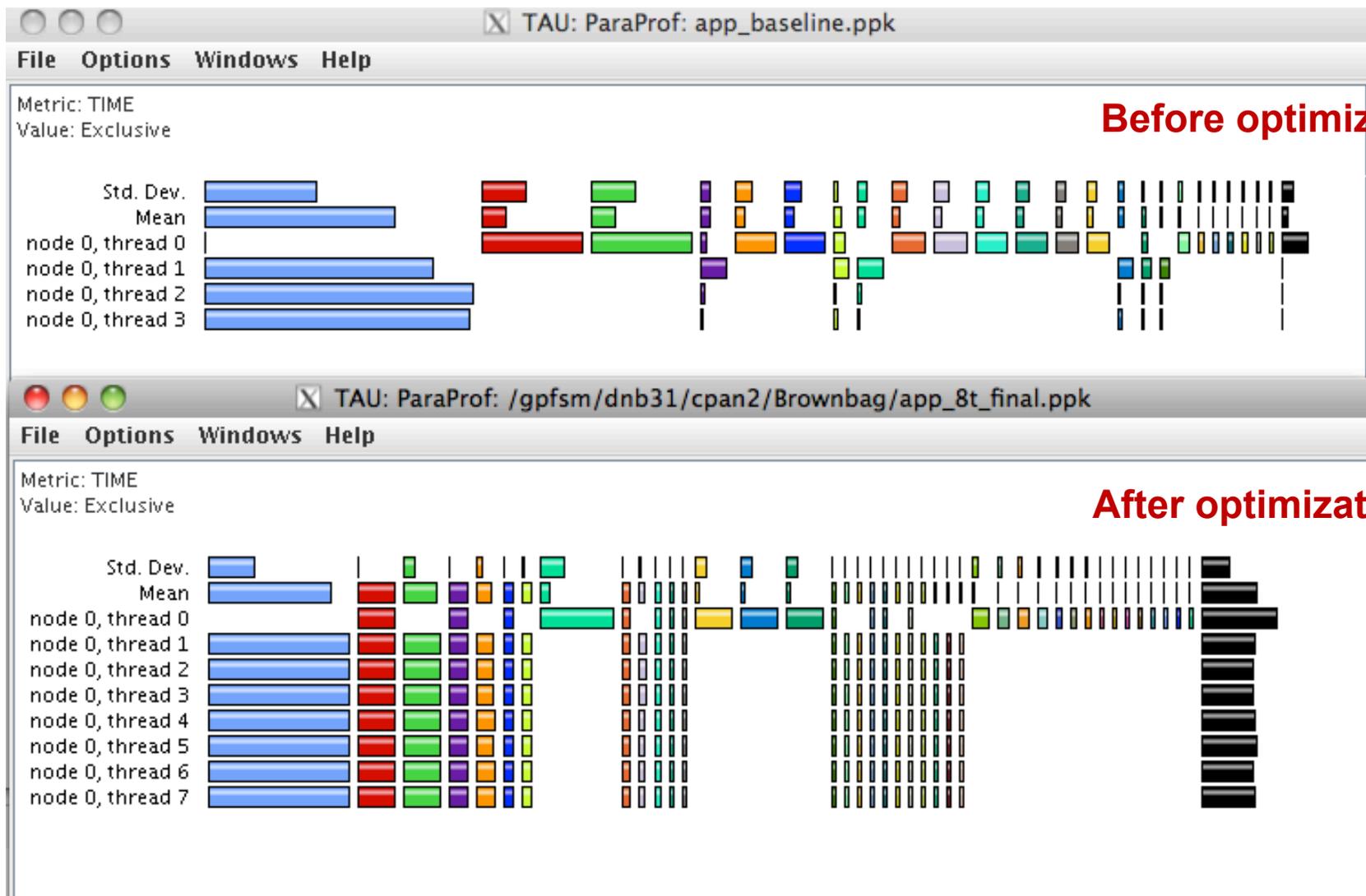
Environment Variables in TAU



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_HEAP or TAU_TRACK_HEADROOM	0	Setting to 1 turns on tracking heap memory/headroom at routine entry & exit using context events (e.g., Heap at Entry: main=>foo=>bar)
TAU_CALLPATH_DEPTH	2	Specifies depth of callpath.
TAU_SYNCHRONIZE_CLOCKS	1	Synchronize clocks across nodes to correct timestamps in traces
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:linux timers:PAPI_FP_OPS:PAPI_NATIVE_<event>)



Demo 1: Paraprof



Before optimization

After optimization



Demo 2: Performance measurement and memory leak detection for an MPI application



1. Instrumentation:

```
$ setenv PATH ${PATH}:/discover/nobackup/cpan2/lib/tau-2.21.2/x86_64/bin
```

Or place it in your shell startup files, e.g. .cshrc

```
$ setenv /discover/nobackup/cpan2/lib/tau-2.21.2/x86_64/lib/Makefile.tau-icpc-  
mpi-pdt
```

```
$ setenv TAU_OPTIONS "-optVerbose -optKeepFiles -optDetectMemoryLeaks  
-optTauSelectFile=/discover/nobackup/cpan2/Brownbag/select.tau"
```

```
$ gmake clean
```

```
$ gmake install FOPT=-g FC=tau_f90.sh |& tee make.log
```

To configure TAU using PDT and MPI:

```
$ ./configure -mpi -mpiinc=/usr/local/intel/mpi/3.2.2.006/include64 \  
-mpilib=/usr/local/intel/mpi/3.2.2.006/lib64 -c++=icpc -fortran=intel -cc=icc \  
-useropt="-L/usr/local/intel/Compiler/11.0/083/lib/intel64 -lirc" \  
-pdt=/discover/nobackup/cpan2/lib/pdtoolkit-3.17  
$ make install
```



Selective Instrumentation File



- Specify an EXCLUDE/INCLUDE list of routines/files
- User instrumentation commands are placed in INSTRUMENT section
- ? and * used as wildcard characters for file name, # for routine name
- Outer-loop level instrumentation
- Arbitrary code insertion

An example select.tau file:

```
BEGIN_FILE_EXCLUDE_LIST
m_fpe*.F90
catch_types*.F90
MAPL_Base*.F90
END_FILE_EXCLUDE_LIST

BEGIN_INSTRUMENT_SECTION
memory file="*.F90" routine="#"
loops file="*.F90" routine="#"
io file="foo.F90" routine="matrix#"
END_INSTRUMENT_SECTION
```



Demo 2 – Hands-on



2. Measurement

```
#PBS -l
select=12:ncpus=12:mpiprocs=6
#PBS ...
...

setenv TAU_TRACK_HEAP 1
setenv TAU_CALLPATH 1
setenv TAU_CALLPATH_DEPTH 100

cd $WORKDIR
mpirun -perhost 6 -np 72 ./<exe>
```

3. Analysis

```
$ ls profile*
profile.0.0.0 profile.1.0.0
... profile.72.0.0
$ paraprof --pack mpi_example.ppk
$ paraprof mpi_example.ppk
```

You can look at the bundled profile data examples using “paraprof”:
[/discover/nobackup/cpan2/Brownbag/mpi_example.ppk](#)

