Parallel Performance Evaluation Using TAU

Workshop at LLNL,

Laboratory Training Center 2, T-1889

Livermore, CA
9am – 5pm, Nov 3-4, 2010

Sameer Shende

sameer@paratools.com

http://www.paratools.com/llnl10

Outline

Outline

- Introduction to performance evaluation tools: TAU, PAPI, Scalasca, and Vampir
- Performance strategies
  - MPI, Loop level optimizations, memory optimizations
- Hands-on:
  - TAU instrumentation at routine, loop level, PAPI hardware performance counter data collection, derived metrics, analyzing performance using TAU's paraprof profile browser, Scalasca bottleneck detection tools
- Hands-on:
  - PerfExplorer, perfdmf configuration using Derby, memory leak detection, trace visualization, workshop examples including the NAS Parallel Benchmarks 3.1

Workshop 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 improving and understanding code performance
  - How to collect and analyze data from hardware performance counters using PAPI
  - How to instrument your programs with TAU
    - Automatic instrumentation at the routine level and outer loop level
    - Manual instrumentation at the loop/statement level
  - Measurement options provided by TAU
  - Environment variables used for choosing metrics, generating performance data
  - How to use the TAU's profile browser, ParaProf
  - How to use TAU's database for storing and retrieving performance data
  - General familiarity with TAU's use for Fortran, Python, C++,C, MPI for mixed language programming
  - How to generate trace data in different formats
  - How to use Scalasca for detecting performance bottlenecks
  - How to analyze trace data using Vampir, and Jumpshot

More Information

- PAPI References:
  - PAPI documentation page available from the PAPI website:
    http://icl.cs.utk.edu/papi/
- TAU References:
  - TAU Users Guide and papers available from the TAU website:
    http://tau.uoregon.edu/
- VAMPIR References
  - VAMPIR-NG website
    http://www.vampir-ng.de/
- Scalasca/KOJAK References
  - Scalasca documentation page
    http://www.scalasca.org/
- Eclipse PTP References
  - Documentation available from the Eclipse PTP website:
    http://www.eclipse.org/ptp/

TAU: A Quick Reference

Part I: TAU: A Tutorial

TAU Performance System

- http://tau.uoregon.edu/
- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Integration in complex software, systems, applications

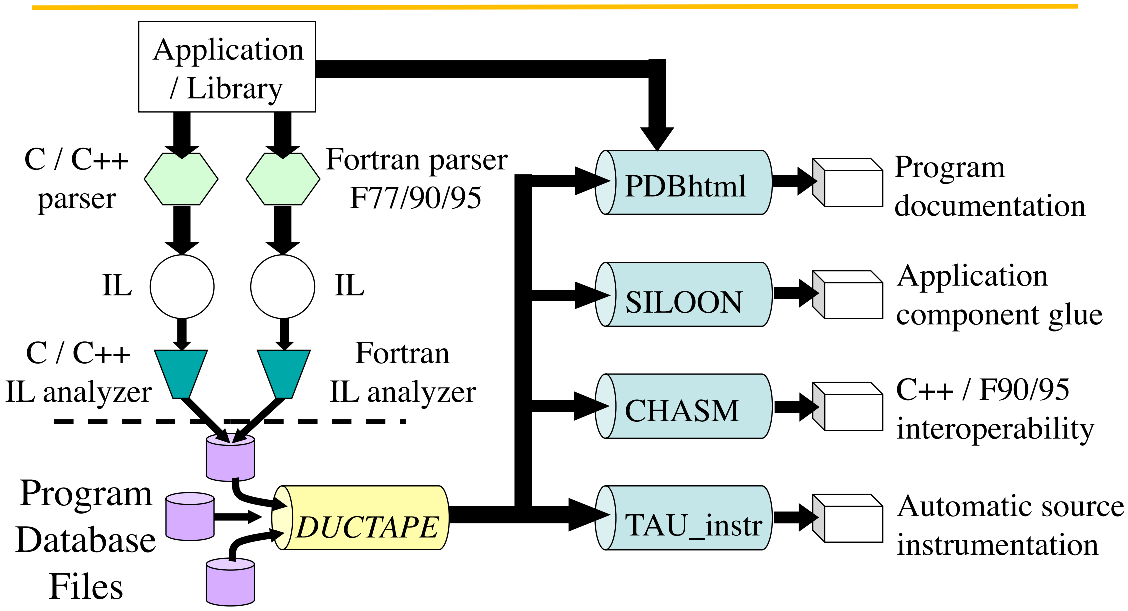
What is TAU?

- TAU is a performance evaluation tool
- It supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you *when* the events take place in each process along a timeline
- TAU uses a package called PDT for automatic instrumentation of the source code
- Profiling and tracing can measure time as well as hardware performance counters from your CPU
- TAU can automatically instrument your source code (routines, loops, I/O, memory, phases, etc.)
- TAU runs on all HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - paraprof is TAU's 3D profile browser
- To use TAU's automatic source instrumentation, you need to set a couple of environment variables and substitute the name of your compiler with a TAU shell script

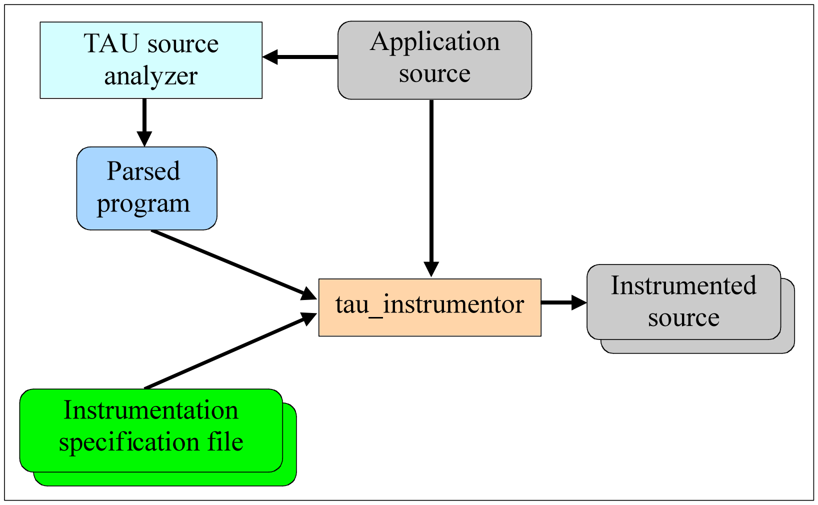
TAU Instrumentation Approach

- Based on direct performance observation
  - Direct instrumentation of program (system) code (probes)
  - Instrumentation invokes performance measurement
  - Event measurement: performance data, meta-data, context
- Support for standard program events
  - Routines, classes and templates
  - Statement-level blocks and loops
  - Begin/End events (Interval events)
- Support for user-defined events
  - Begin/End events specified by user
  - Atomic events (e.g., size of memory allocated/freed)
  - Flexible selection of event statistics
- Provides static events and dynamic events

Program Database Toolkit (PDT)

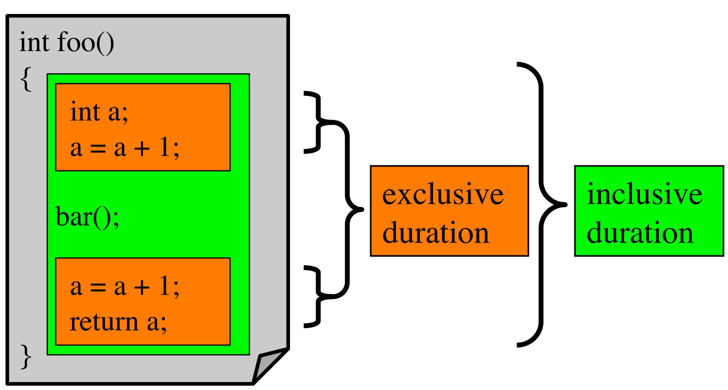


Automatic Source-Level Instrumentation in TAU using Program Database Toolkit (PDT)



Inclusive and Exclusive Profiles

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



Interval Events, Atomic Events in TAU

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.187        1,105           1          44    1105659 int main(int, char **) C
 93.2        1,030        1,030           1           0    1030654 MPI_Init()
  5.9        0.879           65          40         320       1637 void func(int, int) C
  4.6           51           51          40           0       1277 MPI_Barrier()
  1.2           13           13         120           0        111 MPI_Recv()
  0.8            9            9           1           0       9328 MPI_Finalize()
  0.0        0.137        0.137         120           0          1 MPI_Send()
  0.0        0.086        0.086          40           0          2 MPI_Bcast()
  0.0        0.002        0.002           1           0          2 MPI_Comm_size()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.138E+04      44.39   3.09E+04  1.234E+04  Heap Memory Used (KB) : Entry
       365  5.138E+04       2064  3.115E+04   1.21E+04  Heap Memory Used (KB) : Exit
        40         40         40         40          0  Message size for broadcast
---------------------------------------------------------------------------------------
```

Interval event e.g., routines (start/stop)

Atomic events (trigger with value)

% setenv TAU_CALLPATH_DEPTH 0
% setenv TAU_TRACK_HEAP 1

Atomic Events, Context Events

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.253        1,106           1          44    1106701 int main(int, char **) C
 93.2        1,031        1,031           1           0    1031311 MPI_Init()
  6.0            1           66          40         320       1650 void func(int, int) C
  5.7           63           63          40           0       1588 MPI_Barrier()
  0.8            9            9           1           0       9119 MPI_Finalize()
  0.1            1            1         120           0         10 MPI_Recv()
  0.0        0.141        0.141         120           0          1 MPI_Send()
  0.0        0.085        0.085          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
        40         40         40         40          0  Message size for broadcast
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
        40  5.139E+04       3097  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Bcast()
         1       2067       2067       2067          0  Heap Memory Used (KB) : Entry : MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04  0.0006905  Heap Memory Used (KB) : Entry : MPI_Finalize()
         1      57.56      57.56      57.56          0  Heap Memory Used (KB) : Entry : MPI_Init()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Recv()
       120  5.139E+04  1.129E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Send()
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
        40  5.036E+04       2068  3.011E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C
```

Atomic event

Context event = atomic event + executing context

% setenv TAU_CALLPATH_DEPTH 1
% setenv TAU_TRACK_HEAP 1

Context Events (Default)

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.357        1,114           1          44    1114040 int main(int, char **) C
 92.6        1,031        1,031           1           0    1031066 MPI_Init()
  6.7           72           74          40         320       1865 void func(int, int) C
  0.7            8            8           1           0       8002 MPI_Finalize()
  0.1            1            1         120           0         12 MPI_Recv()
  0.1        0.608        0.608          40           0         15 MPI_Barrier()
  0.0        0.136        0.136         120           0          1 MPI_Send()
  0.0        0.095        0.095          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
         1       2068       2068       2068          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Finalize()
         1      57.58      57.58      57.58          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Init()
        40  5.036E+04       2069  3.011E+04  1.228E+04  Heap Memory Used (KB) : Entry : int main(int, char **) C => void func(int, int) C
        40  5.139E+04       3098  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Bcast()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Recv()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Send()
       365  5.139E+04       2065  3.116E+04   1.21E+04  Heap Memory Used (KB) : Exit
                                                                                                3,7
```

% setenv TAU_CALLPATH_DEPTH 2
% setenv TAU_TRACK_HEAP 1

Context event = atomic event + executing context

Steps of Performance Evaluation

- Collect basic routine-level timing profile to determine where most time is being spent
- Collect routine-level hardware counter data to determine types of performance problems
- Collect callpath profiles to determine sequence of events causing performance problems
- Conduct finer-grained profiling and/or tracing to pinpoint performance bottlenecks
  - Loop-level profiling with hardware counters
  - Tracing of communication operations

Using TAU: A brief Introduction

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

  % source /usr/global/tools/tau/training/tau.bashrc
  % export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
  % export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh -help)

  And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:

  % mpif90 foo.f90

  changes to

  % tau_f90.sh foo.f90
- Execute application and analyze performance data:

  % pprof (for text based profile display)
  % paraprof (for GUI)

TAU Measurement Configuration

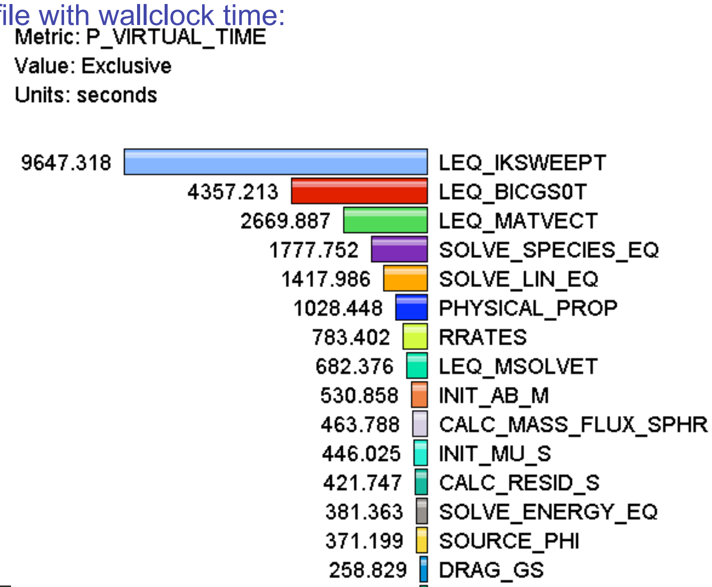
```
% cd /usr/global/tools/tau/training/tau_latest/x86_64/lib; ls
   Makefile.*
Makefile.tau-pdt
Makefile.tau-mpi-pdt
Makefile.tau-pthread-pdt
Makefile.tau-papi-mpi-pdt
Makefile.tau-papi-pthread-pdt
Makefile.tau-mpi-papi-pdt
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-mpi-pdt-vampirtrace-trace
```

- For an MPI+F90 application, you may want to start with:

Makefile.tau-mpi-pdt
- Supports MPI instrumentation & PDT for automatic source instrumentation
- `% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt`
- `% tau_f90.sh matrix.f90 -o matrix`

Usage Scenarios: Routine Level Profile

- Goal: What routines account for the most time? How much?
- Flat profile with wallclock time:

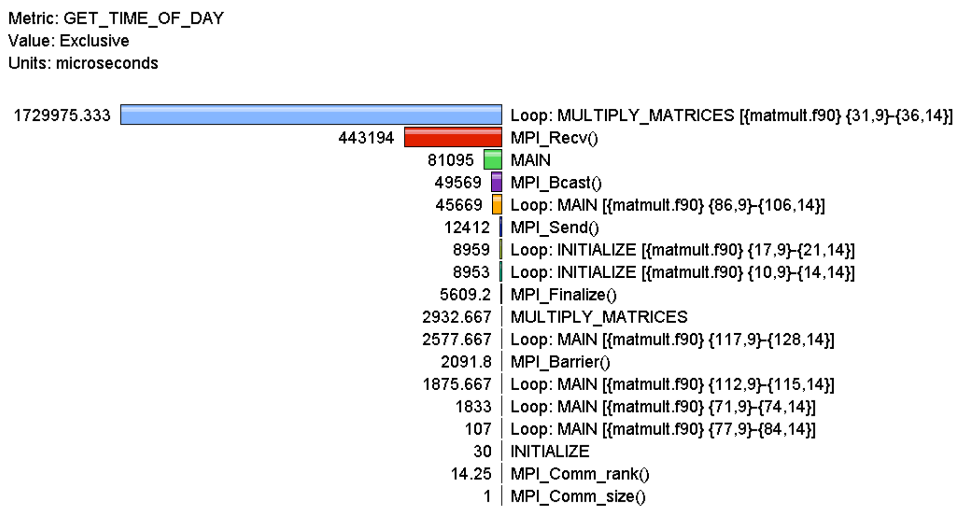


Solution: Generating a flat profile with MPI

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/
x86_64/lib/Makefile.tau-mpi-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:
$PATH
OR
% source /usr/global/tools/tau/training/tau.bashrc
% tau_f90.sh matmult.f90 -o matmult
(Or edit Makefile and change F90=tau_f90.sh)

% mpirun -np 4 ./matmult
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```

Usage Scenarios: Loop Level Instrumentation

- Goal: What loops account for the most time? How much?
- Flat profile with wallclock time with loop instrumentation:

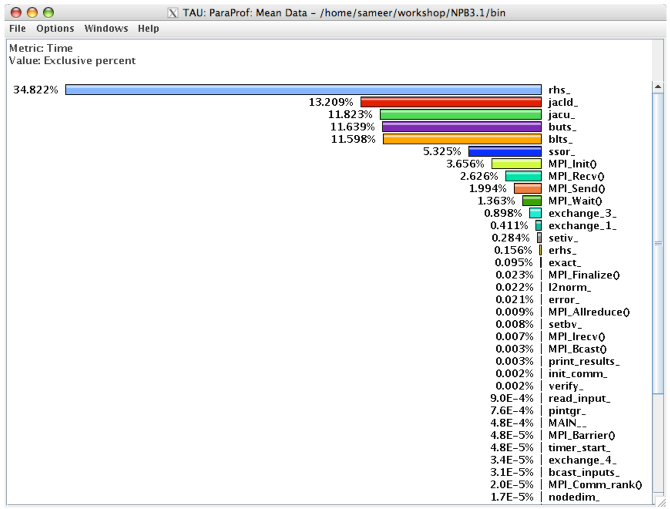


Solution: Generating a loop level profile

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/
x86_64/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:
$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```

Usage Scenarios: Compiler-based Instrumentation

- Goal: Easily generate routine level performance data using the compiler instead of PDT for parsing the source code

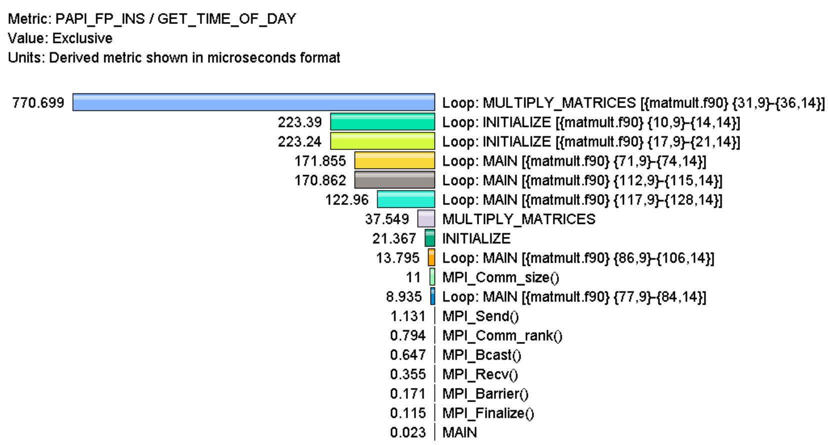


Use Compiler-Based Instrumentation

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                 /lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optCompInst -optVerbose'
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:
$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)


% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```

Usage Scenarios: Calculate mflops in Loops

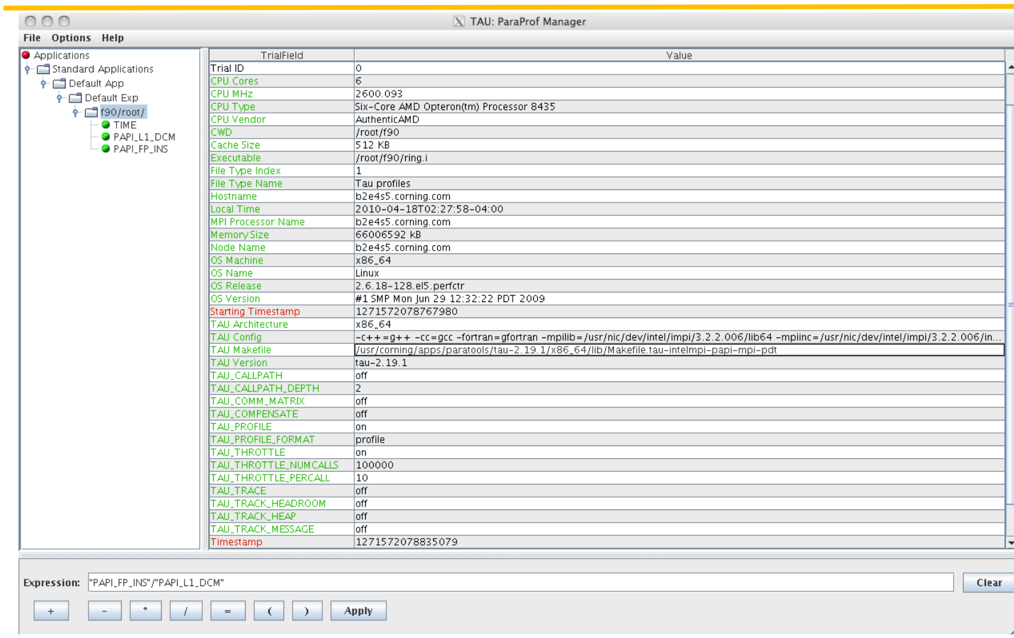
- Goal: What MFlops am I getting in all loops?
- Flat profile with PAPI_FP_INS/OPS and time with loop instrumentation:



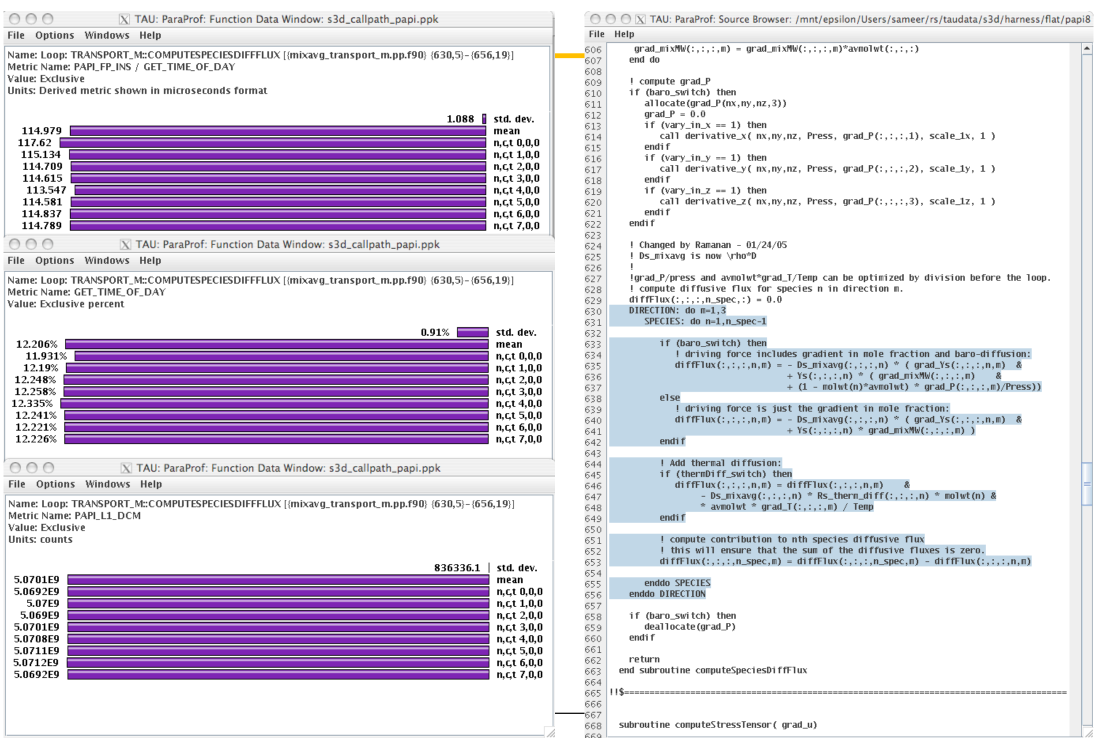
Generate a PAPI profile with 2 or more counters

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                                /lib/Makefile.tau-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
% mpirun -np 4  ./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> "PAPI_FP_INS", click "/", "TIME", click "Apply"
     choose.
```

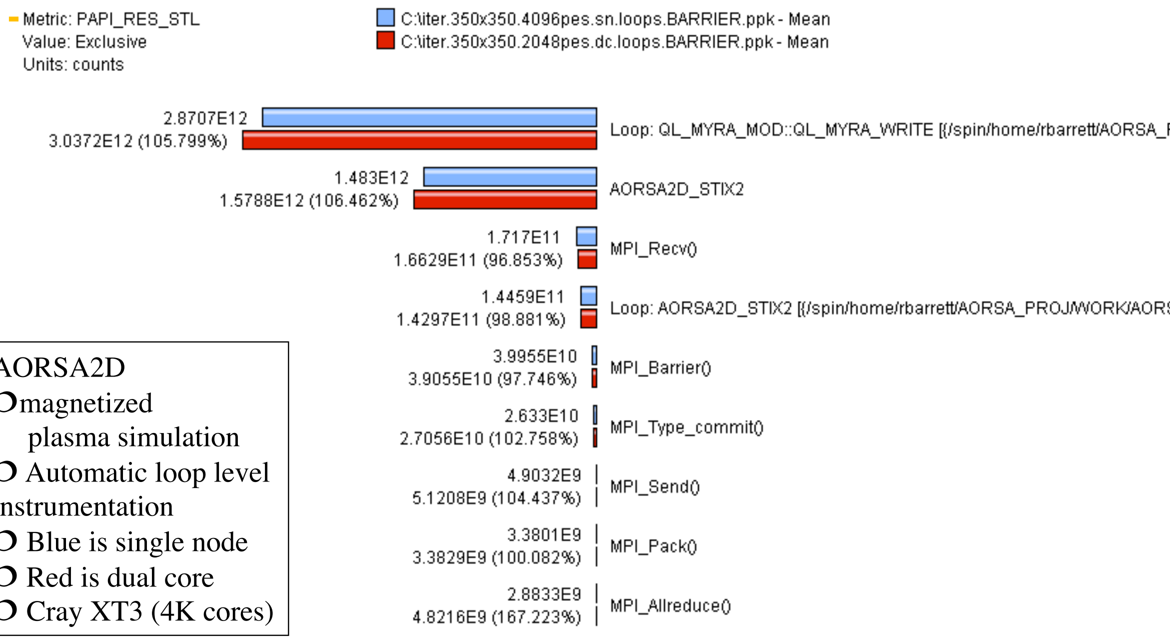
Derived Metrics in ParaProf



ParaProf's Source Browser: Loop Level Instrumentation



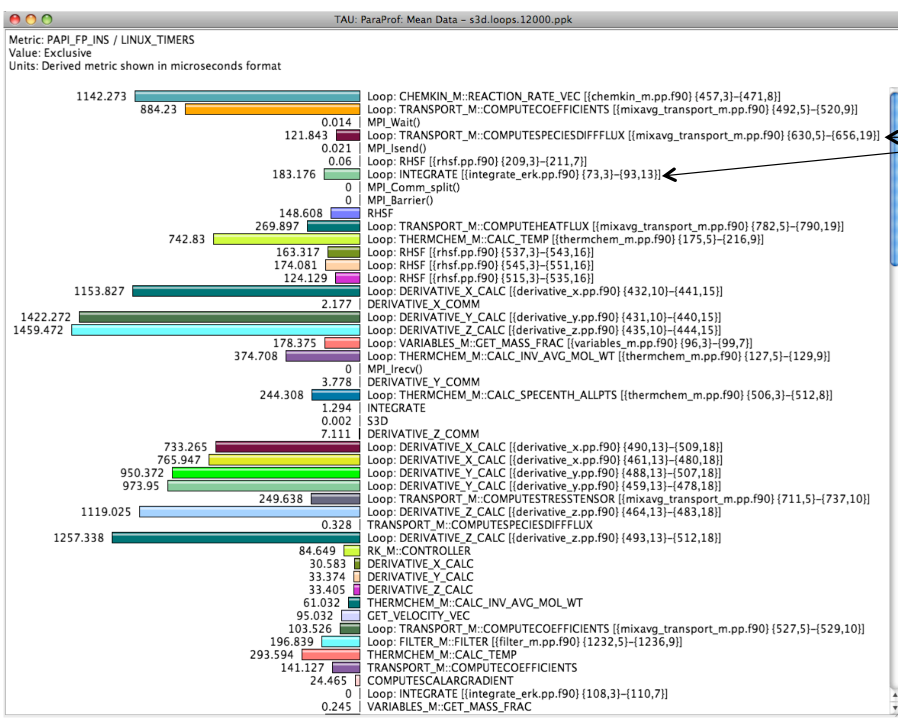
Comparing Effects of Multi-Core Processors



AORSA2D
- magnetized plasma simulation
- Automatic loop level instrumentation
- Blue is single node
- Red is dual core
- Cray XT3 (4K cores)

ParaTools

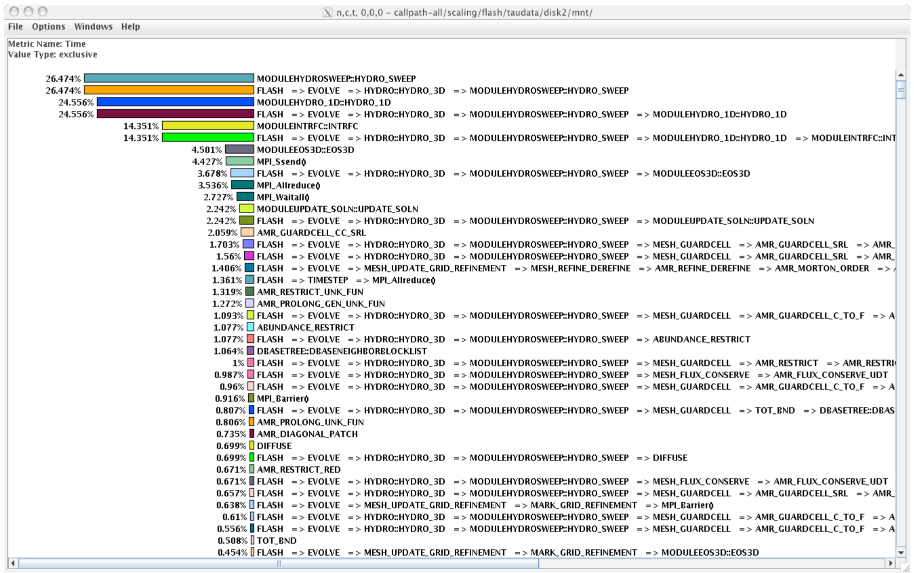
ParaProf: Mflops Sorted by Exclusive Time



low mflops?

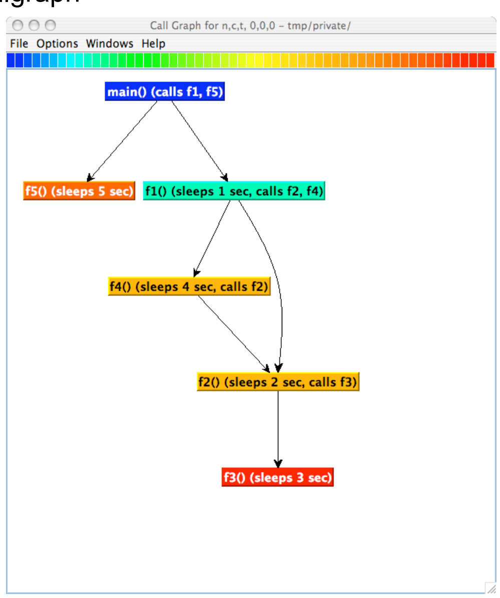
Usage Scenarios: Generating Callpath Profile

- Callpath profile for a given callpath depth:



Callpath Profile

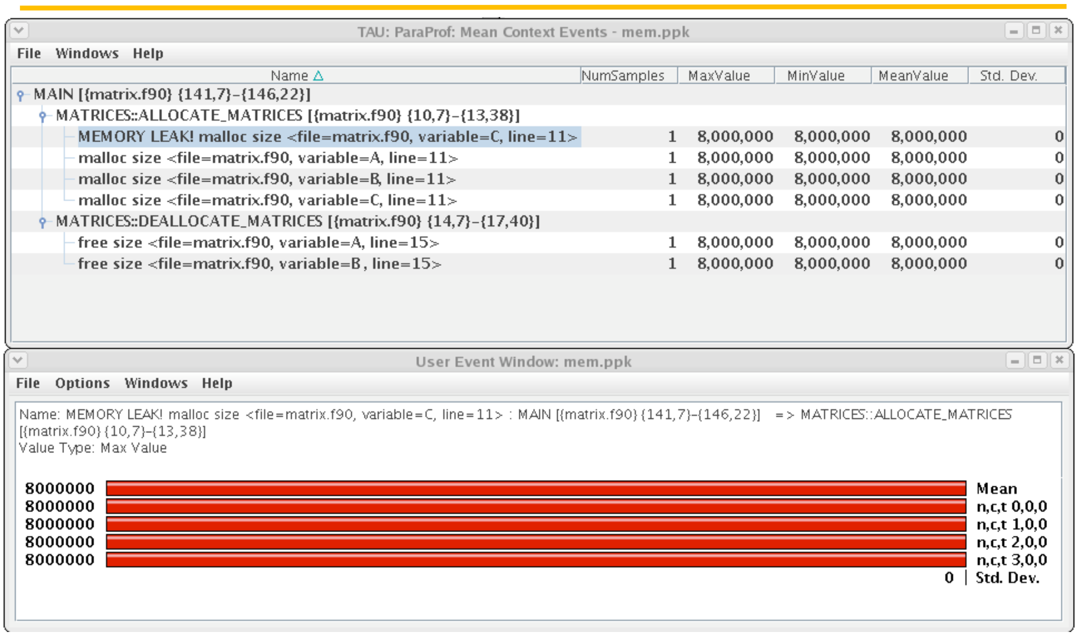
- Generates program callgraph



Generate a Callpath Profile

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                              /lib/Makefile.tau-mpi-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100

% mpirun -np 4  ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```


Usage Scenario: Detect Memory Leaks

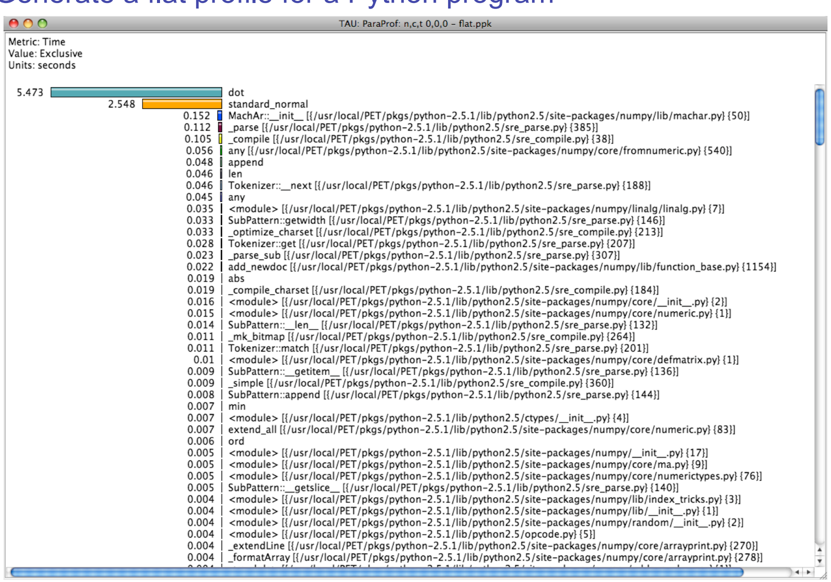


Detect Memory Leaks

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                        /lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optDetectMemoryLeaks -optVerbose'
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_CALLPATH_DEPTH=100

% mpirun -np 4  ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Context Event Window -> Select thread -> select...
   expand tree)
(Windows -> Thread -> User Event Bar Chart -> right click LEAK
-> Show User Event Bar Chart)
```


Usage Scenarios: Instrument a Python program

- Goal: Generate a flat profile for a Python program



Usage Scenarios: Instrument a Python program

Original code:

```
% cat foo.py
#!/usr/bin/env python
import numpy
ra=numpy.random
la=numpy.linalg

size=2000
a=ra.standard_normal((size,size))
b=ra.standard_normal((size,size))
c=la.linalg.dot(a,b)
print c
```

Create a wrapper:

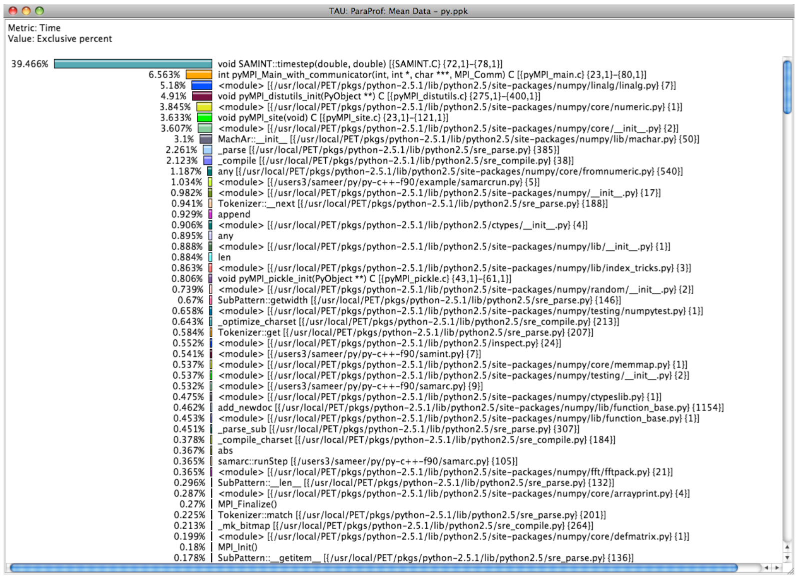
```
% cat wrapper.py
#!/usr/bin/env python

# setenv PYTHONPATH $PET_HOME/pkgs/tau-2.17.3/ppc64/lib/bindings-gnu-python-pdt

import tau

def OurMain():
    import foo

tau.run('OurMain()')
```


Generate a Python Profile

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                /lib/Makefile.tau-python-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:
$PATH
% cat wrapper.py
  import tau
  def OurMain():
      import foo
  tau.run('OurMain()')
Uninstrumented:
% ./foo.py
Instrumented:
% export PYTHONPATH= <taudir>/x86_64/<lib>/bindings-python-pdt
(same options string as TAU_MAKEFILE)
% export LD_LIBRARY_PATH=<taudir>/x86_64/lib/bindings-python-pdt:
$LD_LIBRARY_PATH
% ./wrapper.py

Wrapper invokes foo and generates performance data
% pprof/paraprof
```

Usage Scenarios: Mixed Python+F90+C+pyMPI

- Goal: Generate multi-level instrumentation for Python+MPI+C+F90+C++ ...

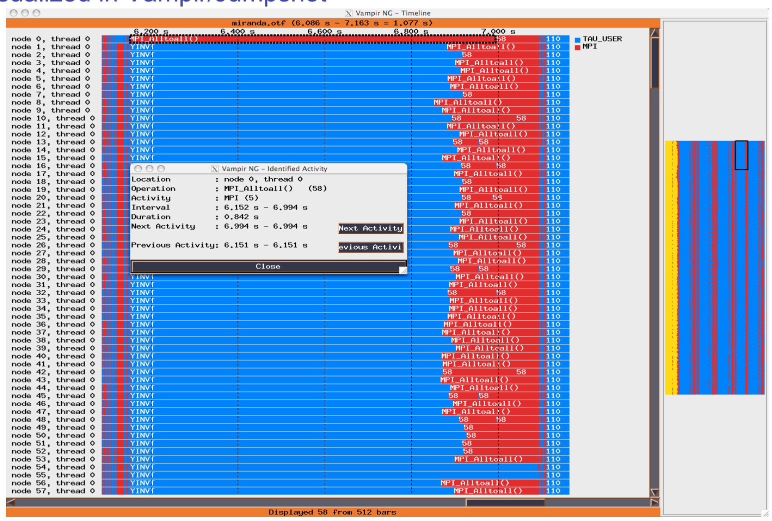


Generate a Multi-Language Profile w/ Python

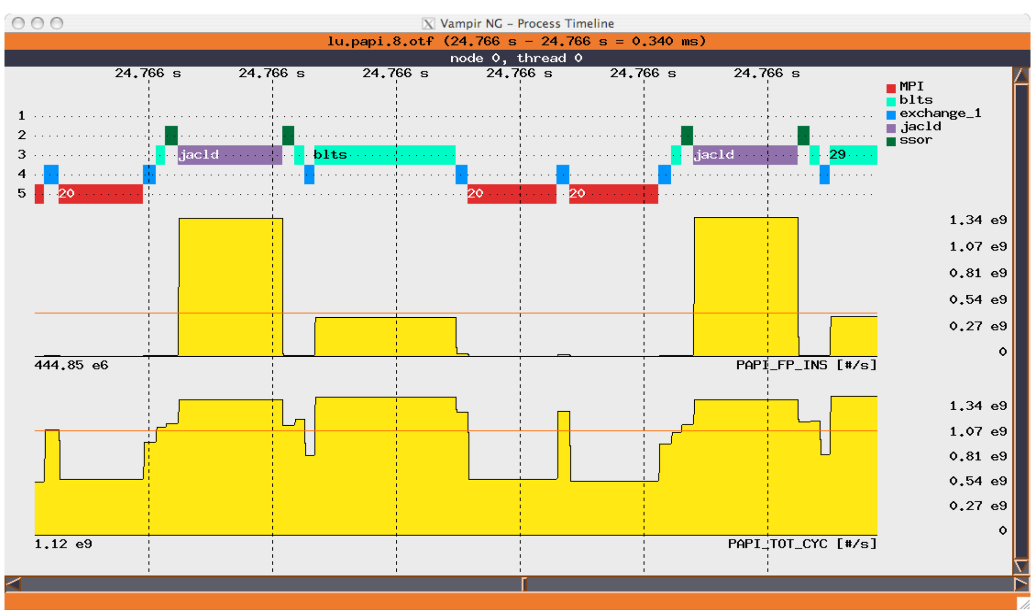
```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
               /lib/Makefile.tau-python-mpi-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% export TAU_OPTIONS='-optShared -optVerbose…'
(Python needs shared object based TAU library)
% make F90=tau_f90.sh CXX=tau_cxx.sh CC=tau_cc.sh  (build libs, pyMPI w/TAU)
% cat wrapper.py
  import tau
  def OurMain():
      import App
  tau.run('OurMain()')
Uninstrumented:
% mpirun –np 4 pyMPI ./App.py
Instrumented:
% export PYTHONPATH= <taudir>/x86_64/<lib>/bindings-python-mpi-pdt
(same options string as TAU_MAKEFILE)
% export LD_LIBRARY_PATH=<taudir>/x86_64/lib/bindings-python-mpi-pdt:
$LD_LIBRARY_PATH
% mpirun –np 4 <pkgs>/pyMPI-2.5b0-TAU/bin/pyMPI
./wrapper.py                (Instrumented pyMPI with wrapper.py)
```

Usage Scenarios: Generating a Trace File

- Goal: Identify the temporal aspect of performance. What happens in my code at a given time? When?
- Event trace visualized in Vampir/Jumpshot



VNG Process Timeline with PAPI Counters



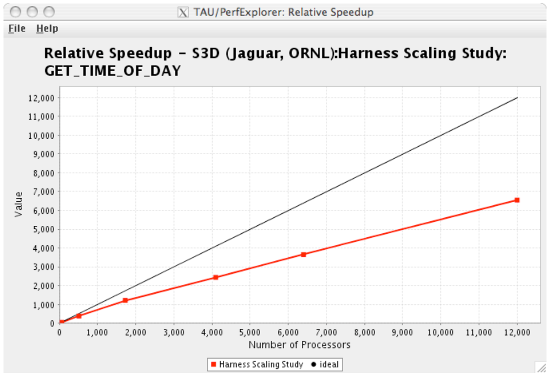
Vampir Counter Timeline Showing I/O BW

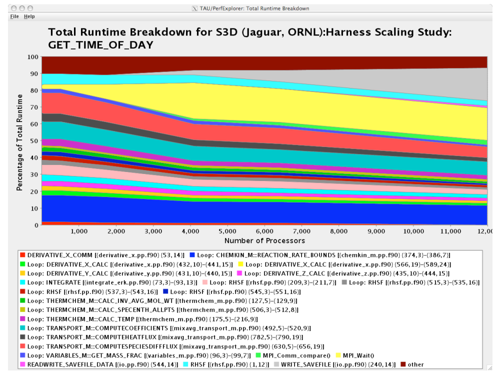
Generate a Trace File

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/
tau_latest/x86_64/lib/Makefile.tau-mpi-pdt

% export TAU_TRACE=1
% export PATH=/usr/global/tools/tau/training/tau_latest/
x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 4  ./a.out
% tau_treemerge.pl
(merges binary traces to create tau.trc and tau.edf files)
JUMPSHOT:
% tau2slog2 tau.trc tau.edf -o app.slog2
% jumpshot app.slog2
   OR
VAMPIR:
% tau2otf tau.trc tau.edf app.otf -n 4 -z
(4 streams, compressed output trace)
% vampir app.otf
```

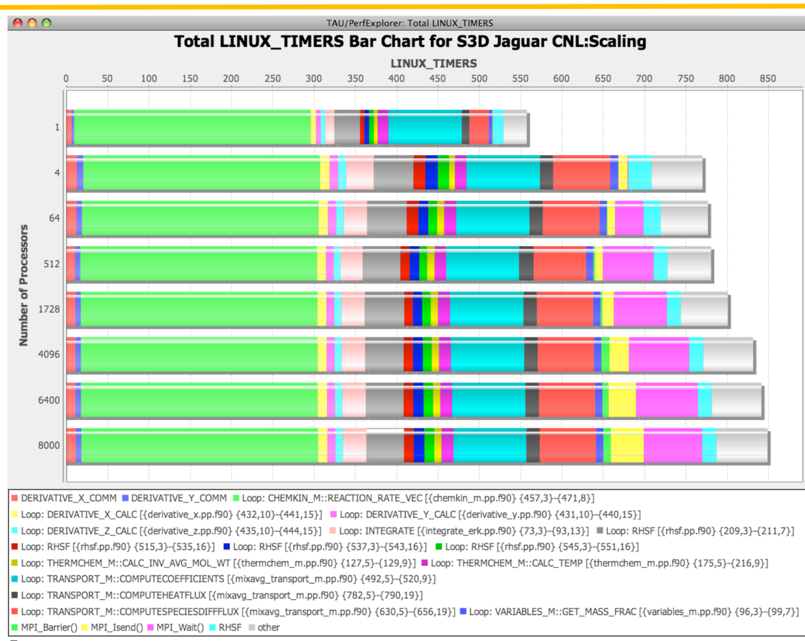
Usage Scenarios: Evaluate Scalability

- Goal: How does my application scale? What bottlenecks occur at what core counts?
- Load profiles in PerfDMF database and examine with PerfExplorer

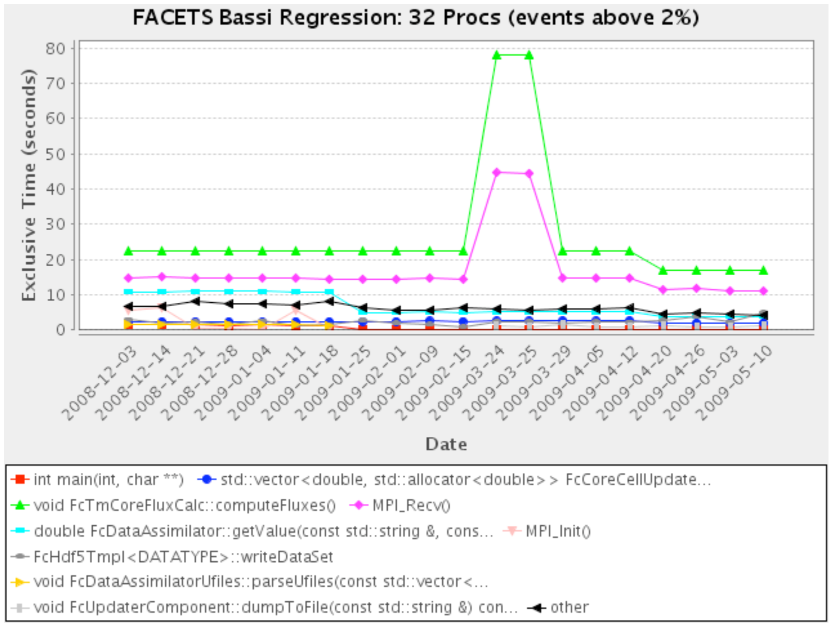




Usage Scenarios: Evaluate Scalability



Performance Regression Testing

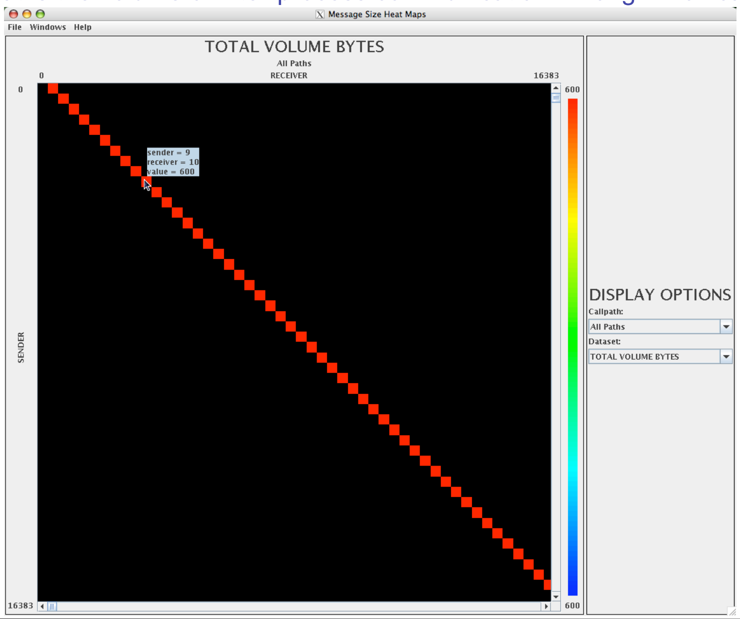


Evaluate Scalability using PerfExplorer Charts

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                              /lib/Makefile.tau-mpi-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 1 ./a.out
% paraprof --pack 1p.ppk
% mpirun -np 2 ./a.out …
% paraprof --pack 2p.ppk … and so on.
On your client:
% perfdmf_configure --create-default
(Chooses derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK) OR use
   perfdmf_loadtrial
Then,
% perfexplorer
(Select experiment, Menu: Charts -> Speedup)
```

Communication Matrix Display

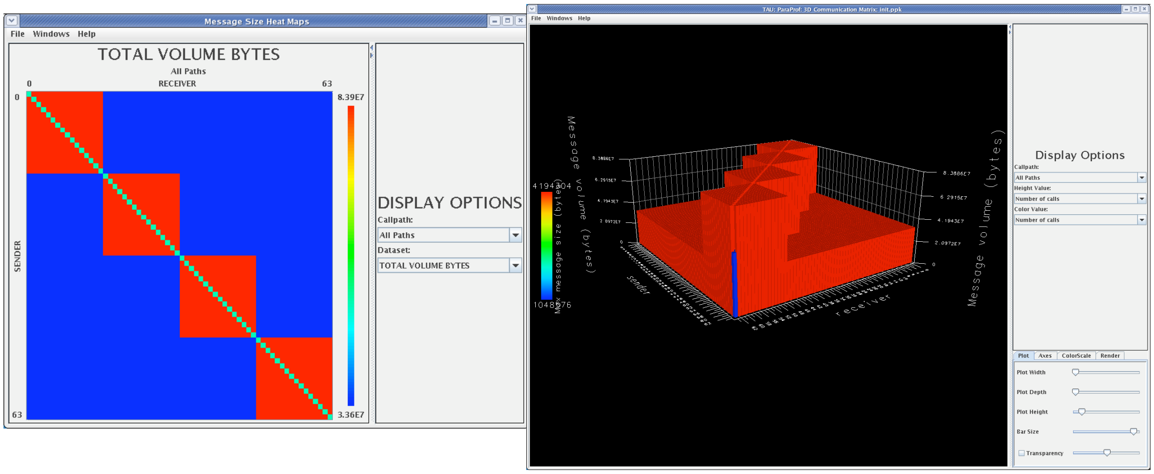
- Goal: What is the volume of inter-process communication? Along which calling path?



Evaluate Scalability using PerfExplorer Charts

```
% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64
                              /lib/Makefile.tau-mpi-pdt
% export PATH=/usr/global/tools/tau/training/tau_latest/x86_64/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_COMM_MATRIX=1

% mpirun -np 4 ./a.out (setting the environment variables)

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

ParaProf: Communication Matrix Display



Instrumentation Issues

- Dynamic Instrumentation using DyninstAPI [U. Wisconsin, Madison, and U. Maryland]
- Pre-execution instrumentation
- Shell script spawned the task on the node and instrumented it
- As the number of processors increased, more time was wasted:
  - transferring un-instrumented executables to the compute nodes,
  - Instrumenting the application binary
- Solution: Binary re-writing!

Re-writing Binaries

- 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
- Simplify the usage of TAU:
  - To instrument:
    - % tau_run a.out –o a.inst
  - To perform measurements, execute the application:
    - % mpirun –np 4 ./a.inst
  - To analyze the data:
    - % paraprof

ParaTools

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 &
```

Library interposition/wrapping: tau_exec, tau_wrap

- TAU provides 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
- Track GPGPU operations
  - % mpirun –np 256 tau_exec –cuda ./a.out

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_MEMORY_LEAKS	0	Setting to 1 turns on leak detection
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. 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_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 separted list generates other metrics. (e.g., TIME:linuxtimers:PAPI_FP_OPS:PAPI_NATIVE_<event>)

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()						

TAU Instrumentation Mechanisms

- Source code
  - Manual (TAU API, TAU component API)
  - Automatic (robust)
    - C, C++, F77/90/95 (Program Database Toolkit (PDT))
    - OpenMP (directive rewriting (Opari2), OpenMP 3.0 spec)
    - Library header wrapping
- Object code
  - Pre-instrumented libraries (e.g., MPI using PMPI)
  - Statically- and dynamically-linked (with LD_PRELOAD)
- Executable code
  - Binary and dynamic instrumentation (DyninstAPI)
  - Virtual machine instrumentation (e.g., Java using JVMTI)
- TAU compiler scripts to automate instrumentation process

ParaTools

Library wrapping: tau_wrap –r <library.so>

- 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 instrumentation files
  - Library loads the original library using the dlopen() call
- 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

ParaTools

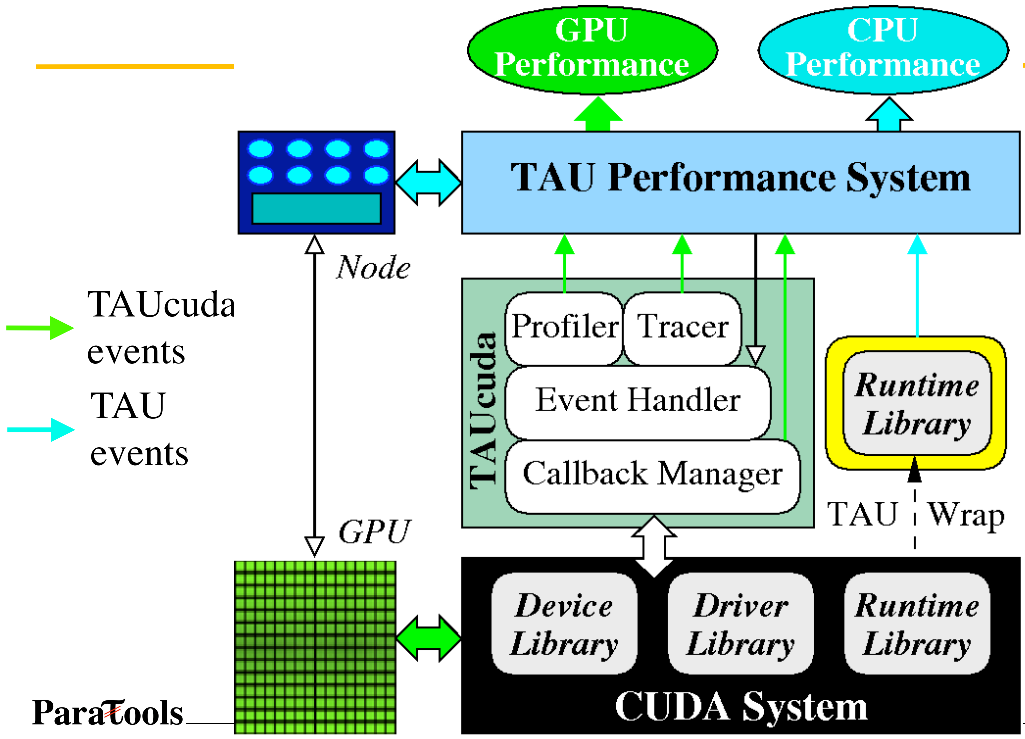
HDF5 Library Wrapping

```
[sameer@zorak]$ tau_wrap hdf5.h.pdb hdf5.h -o hdf5.inst.c -f select.tau -g hdf5 -r
libhdf5.so; cd wrapper; make

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)
• tau_exec loads libhdf5_wrap.so shared library using LD_PRELOAD
• 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 .TAU Application
 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()
```

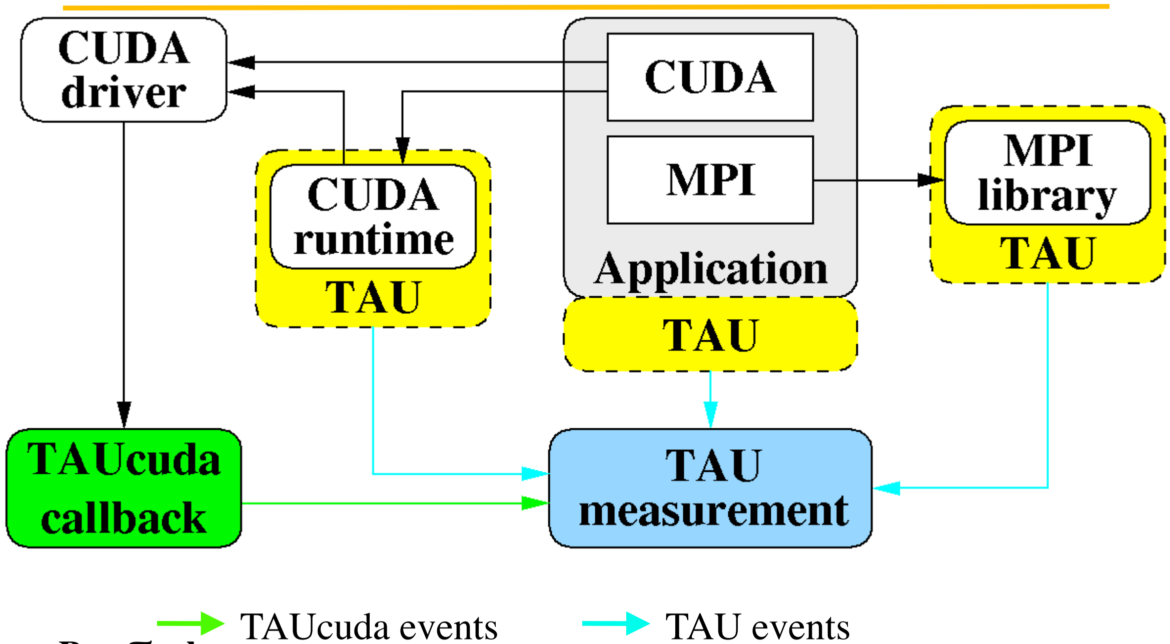

Profiling GPGPU Executions

- GPGPU compilers (e.g., CAPS hmpp and PGI) can now automatically generate GPGPU code using manual annotation of loop-level constructs and routines (hmpp)
- The loops (and routines for HMPP) are transferred automatically to the GPGPU
- TAU intercepts the runtime library routines and examines the arguments
- Shows events as seen from the host
- Profiles and traces GPGPU execution

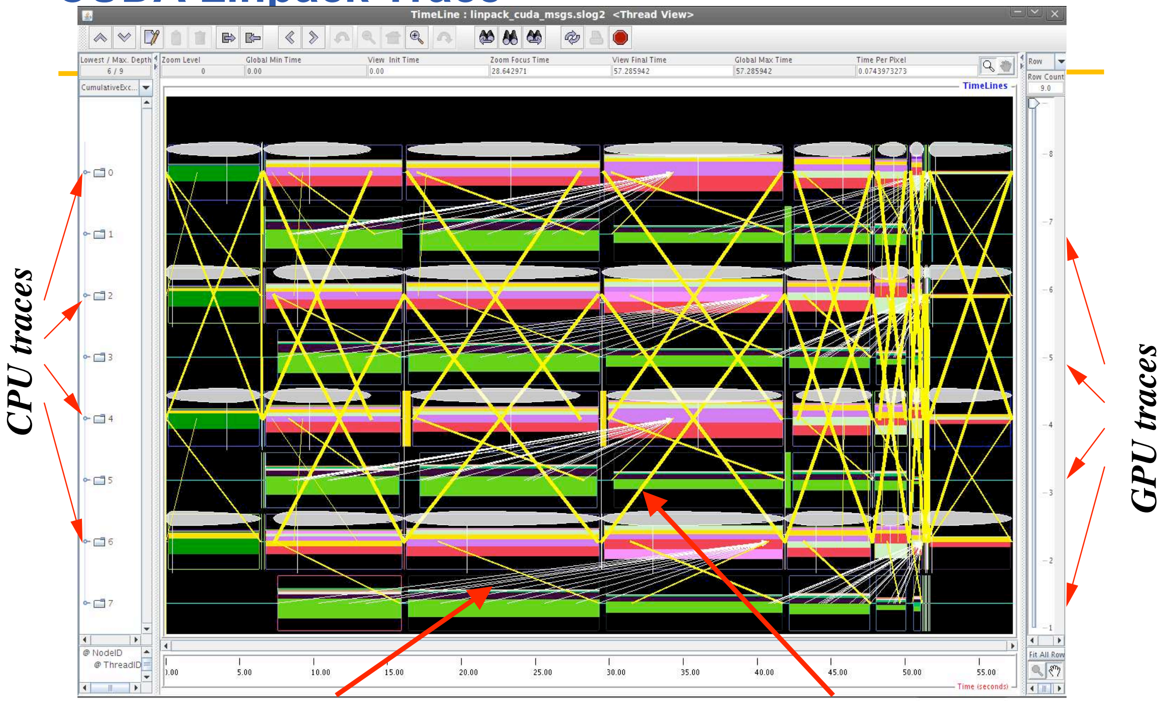
TAUcuda Architecture



TAUcuda Instrumentation



CUDA Linpack Trace



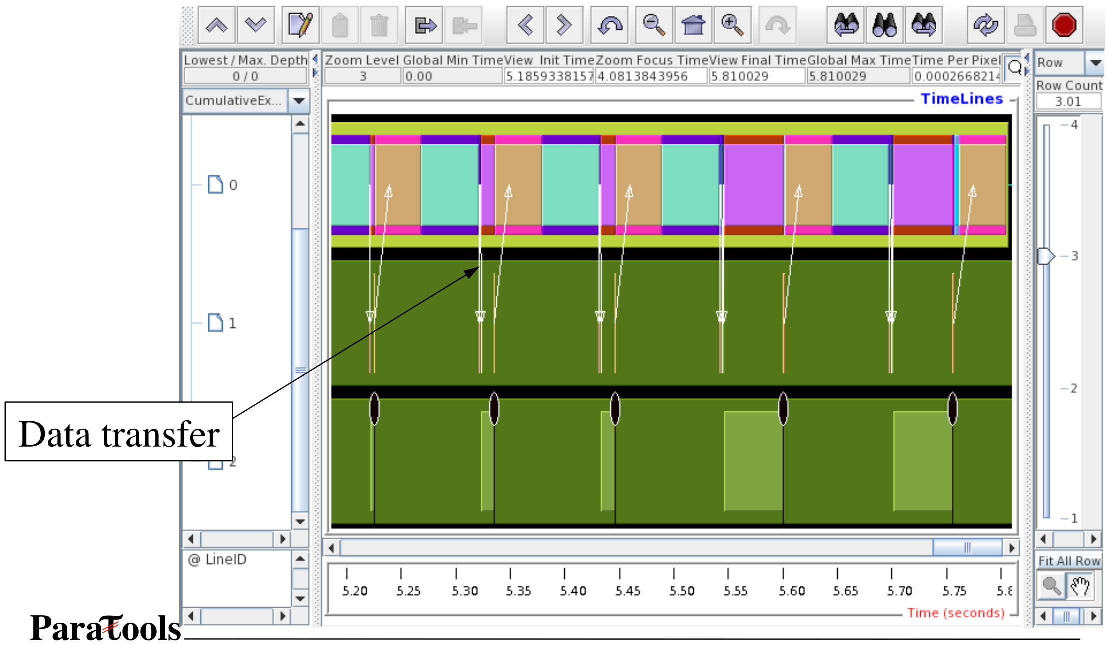
CUDA memory transfer (white) MPI communication (yellow)

SHOC Stencil2D (512 iterations, 4 CPUxGPU)

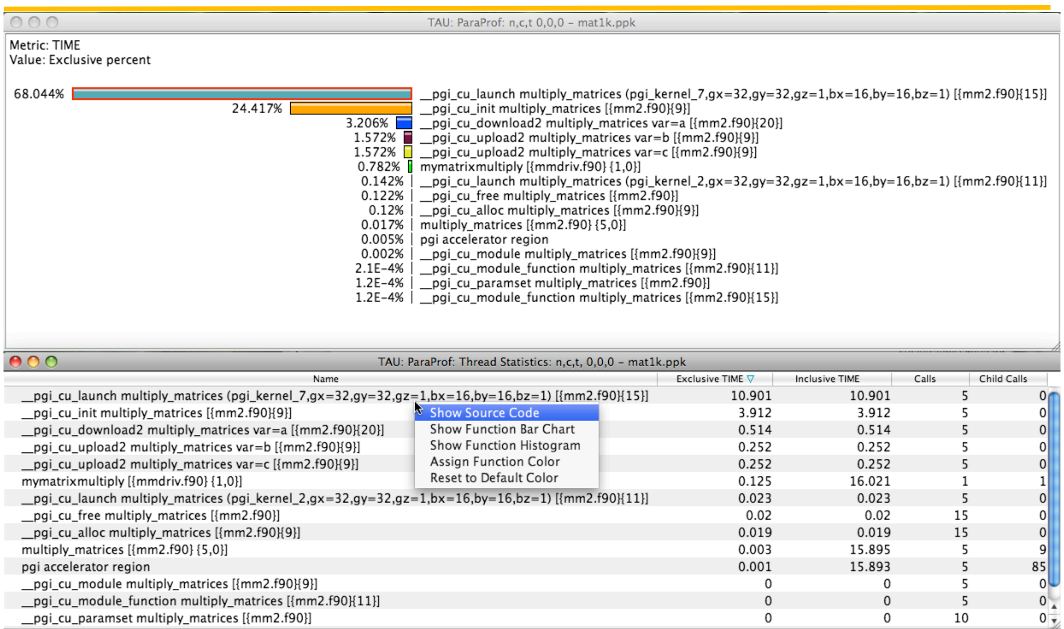
- Scalable HeterOgenerous Computing benchmark suite

CUDA memory transfer (white)

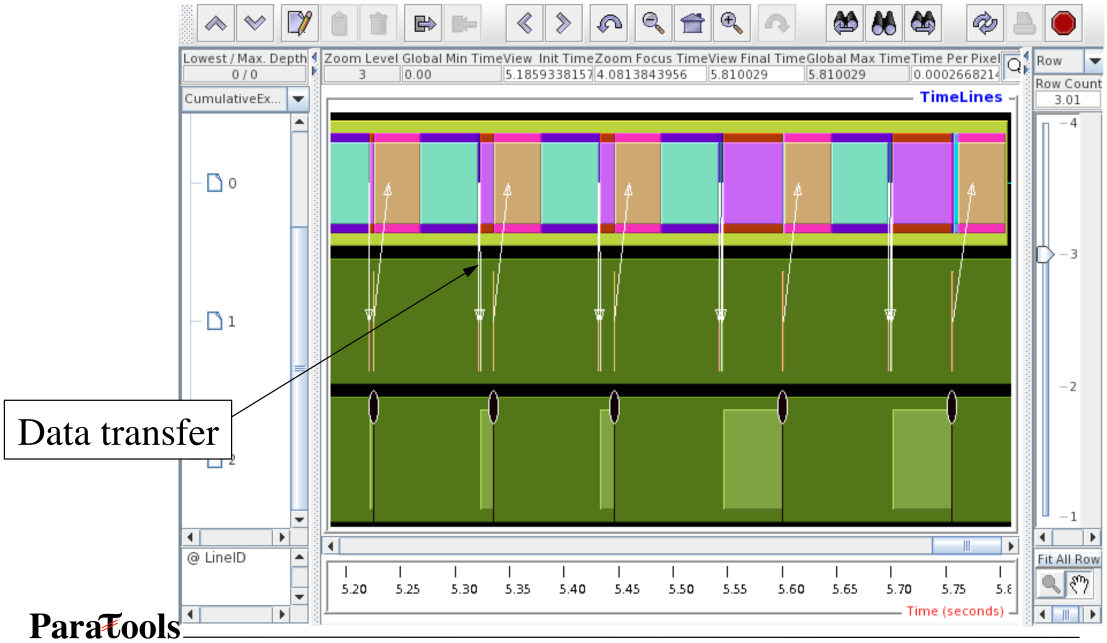
Scaling NAMD with CUDA (Jumpshot with TAU)



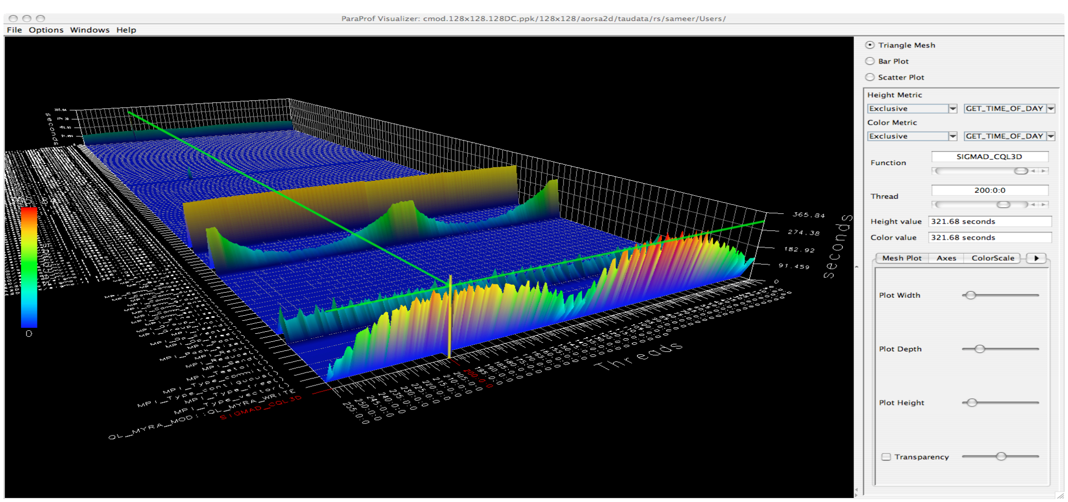
Measuring Performance of PGI GPGPU Accelerated Code



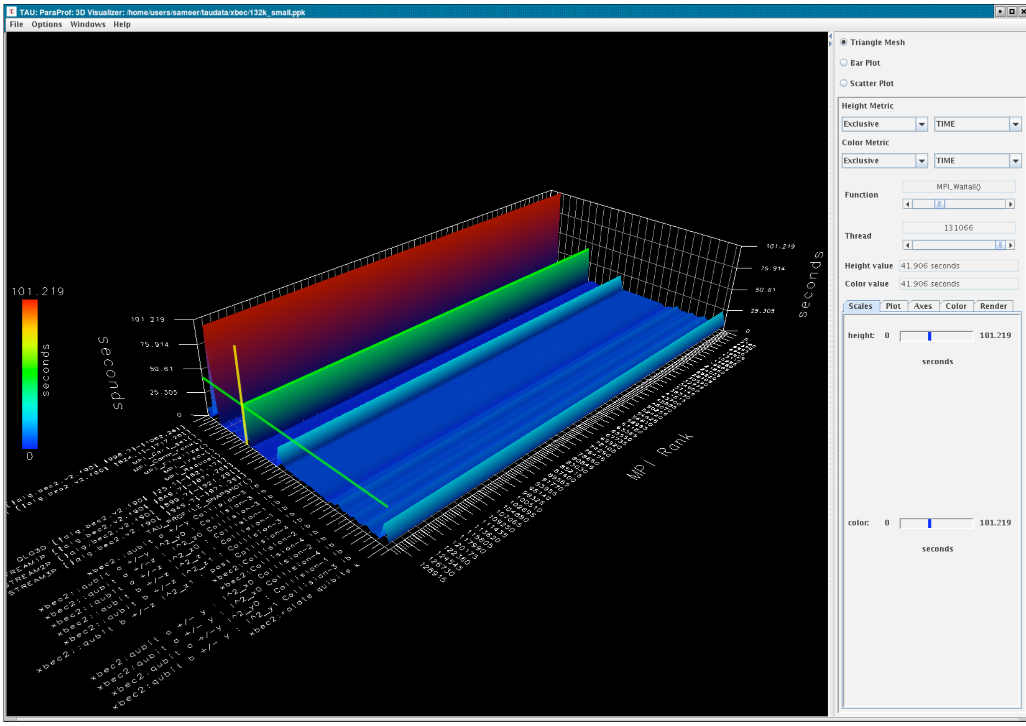
Scaling NAMD with CUDA (Jumpshot with TAU)



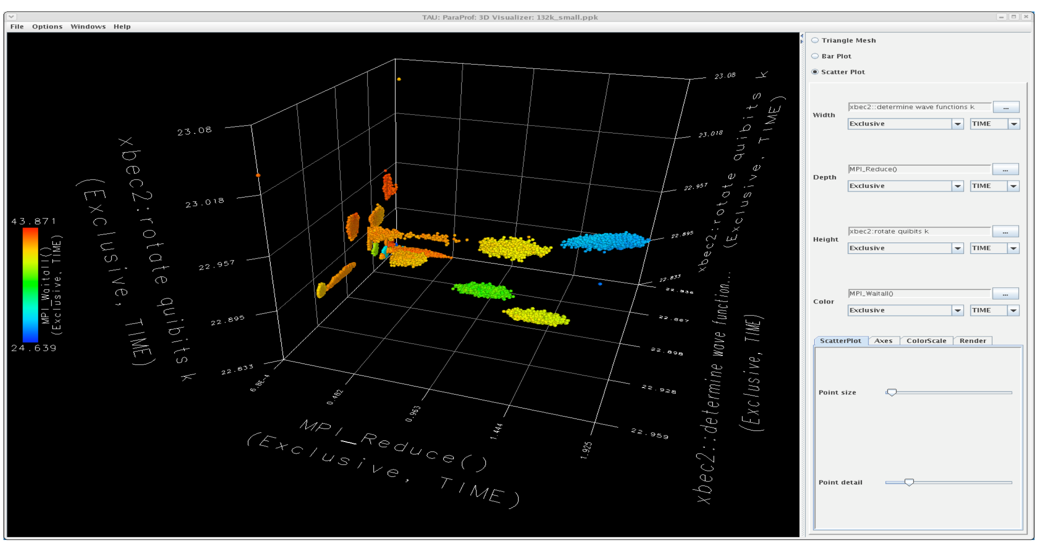
Parallel Profile Visualization: ParaProf



Scalable Visualization: ParaProf (128k cores)



Scatter Plot: ParaProf (128k cores)



Labs

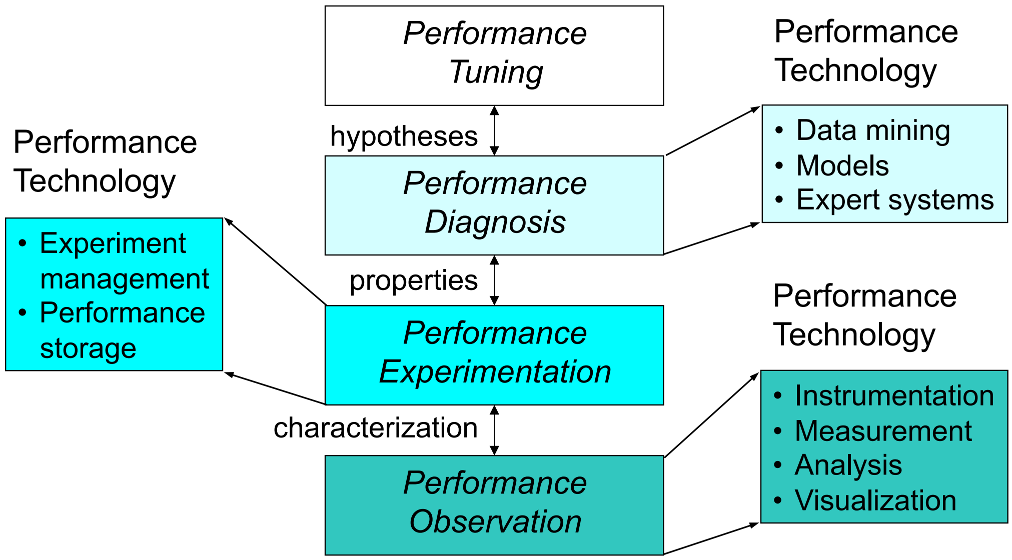
- Add one of
  source /usr/global/tools/tau/training/tau.bashrc
  or
  source /usr/global/tools/tau/training/tau.cshrc
  to the end of your **.login** file (for bash or csh/tcsh users respectively)
  These files contain DSRC specific location information.

- wget http://www.paratools.com/llnl10/workshop.tar.gz
  or
  cp /usr/global/tools/tau/training/workshop.tar.gz .
  and follow the README file.

Part II: Introduction to Performance Engineering

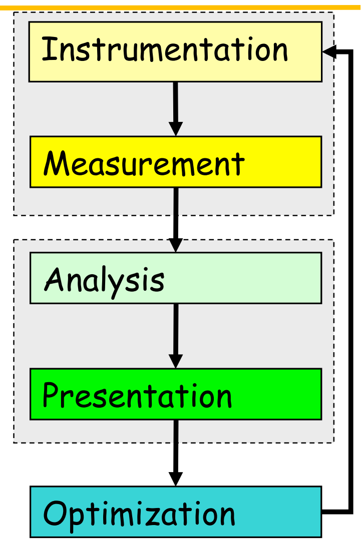
Performance Engineering

- Optimization process
- Effective use of performance technology



Performance Optimization Cycle

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Tune performance



Parallel Performance Properties

- Parallel code performance is influenced by both sequential and parallel factors?
- Sequential factors
  - Computation and memory use
  - Input / output
- Parallel factors
  - Thread / process interactions
  - Communication and synchronization

Performance Observation

- Understanding performance requires observation of performance properties
- Performance tools and methodologies are primarily distinguished by what observations are made and how
  - What aspects of performance factors are seen
  - What performance data is obtained
- Tools and methods cover broad range

Metrics and Measurement

- Observability depends on measurement
- A metric represents a type of measured data
  - Count, time, hardware counters
- A measurement records performance data
  - Associates with program execution aspects
- Derived metrics are computed
  - Rates (e.g., flops)
- Metrics / measurements decided by need

Execution Time

- Wall-clock time
  - Based on realtime clock
- Virtual process time
  - Time when process is executing
    - serial time and system time
  - Does not include time when process is stalled
- Parallel execution time
  - Runs whenever any parallel part is executing
  - Global time basis

Direct Performance Observation

- Execution *actions* exposed as *events*
  - In general, actions reflect some execution state
    - presence at a code location or change in data
    - occurrence in parallelism context (thread of execution)
  - Events encode actions for observation
- Observation is *direct*
  - Direct instrumentation of program code (probes)
  - Instrumentation invokes performance measurement
  - Event measurement = performance data + context
- Performance experiment
  - Actual events + performance measurements

Indirect Performance Observation

- Program code instrumentation is not used
- Performance is observed indirectly
  - Execution is interrupted
    - can be triggered by different events
  - Execution state is queried (sampled)
    - different performance data measured
  - *Event-based sampling* (ESB)
- Performance attribution is inferred
  - Determined by execution context (state)
  - Observation resolution determined by interrupt period
  - Performance data associated with context for period

Direct Observation: Events

- Event types
  - Interval events (begin/end events)
    - measures performance between begin and end
    - metrics monotonically increase
  - Atomic events
    - used to capture performance data state
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops
- User-defined events
  - Specified by the user
- Abstract mapping events

Direct Observation: Instrumentation

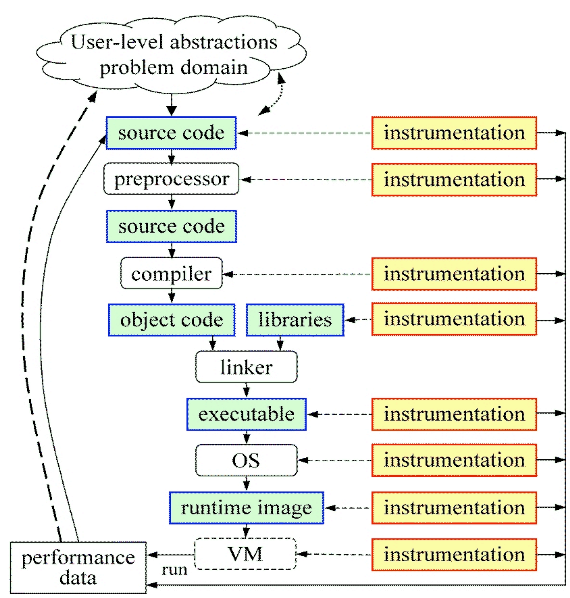
- Events defined by instrumentation access
- Instrumentation levels
  - Source code
  - Object code
  - Runtime system
  - Library code
  - Executable code
  - Operating system
- Different levels provide different information
- Different tools needed for each level
- Levels can have different granularity

Direct Observation: Techniques

- Static instrumentation
  - Program instrumented prior to execution
- Dynamic instrumentation
  - Program instrumented at runtime
- Manual and automatic mechanisms
- Tool required for automatic support
  - Source time: preprocessor, translator, compiler
  - Link time: wrapper library, preload
  - Execution time: binary rewrite, dynamic
- Advantages / disadvantages

Direct Observation: Mapping

- Associate performance data with high-level semantic abstractions
- Abstract events at user-level provide semantic context



Indirect Observation: Events/Triggers

- Events are actions external to program code
  - Timer countdown, HW counter overflow, …
  - Consequence of program execution
  - Event frequency determined by:
    - Type, setup, number enabled (exposed)
- Triggers used to invoke measurement tool
  - Traps when events occur (interrupt)
  - Associated with events
  - May add differentiation to events

Indirect Observation: Context

- When events trigger, execution context determined at time of trap (interrupt)
  - Access to PC from interrupt frame
  - Access to information about process/thread
  - Possible access to call stack
    - requires call stack unwinder
- Assumption is that the context was the same during the preceding period
  - Between successive triggers
  - Statistical approximation valid for long running programs

Direct / Indirect Comparison

- Direct performance observation
  - ☺ Measures performance data exactly
  - ☺ Links performance data with application events
  - 😐 Requires instrumentation of code
  - ☹ Measurement overhead can cause execution intrusion and possibly performance perturbation
- Indirect performance observation
  - ☺ Argued to have less overhead and intrusion
  - ☺ Can observe finer granularity
  - ☺ No code modification required (may need symbols)
  - ☹ Inexact measurement and attribution

Measurement Techniques

- When is measurement triggered?
  - External agent (indirect, asynchronous)
    - interrupts, hardware counter overflow, …
  - Internal agent (direct, synchronous)
    - through code modification
- How are measurements made?
  - Profiling
    - summarizes performance data during execution
    - per process / thread and organized with respect to context
  - Tracing
    - trace record with performance data and timestamp
    - per process / thread

Measured Performance

- Counts
- Durations
- Communication costs
- Synchronization costs
- Memory use
- Hardware counts
- System calls

Critical issues

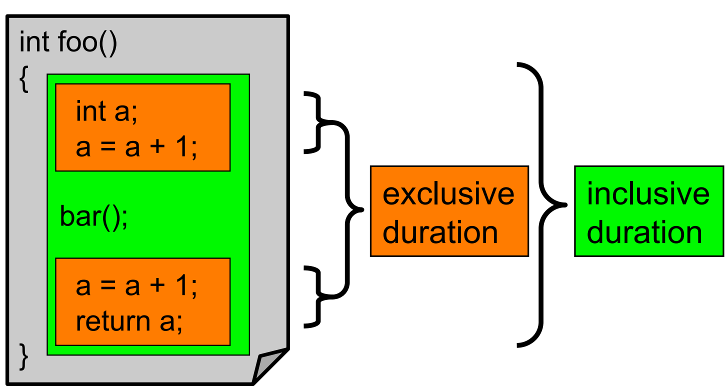
- Accuracy
  - Timing and counting accuracy depends on resolution
  - Any performance measurement generates overhead
    - Execution on performance measurement code
  - Measurement overhead can lead to intrusion
  - Intrusion can cause perturbation
    - alters program behavior
- Granularity
  - How many measurements are made
  - How much overhead per measurement
- Tradeoff (general wisdom)
  - Accuracy is inversely correlated with granularity

Profiling

- Recording of aggregated information
  - Counts, time, …
- … about program and system entities
  - Functions, loops, basic blocks, …
  - Processes, threads
- Methods
  - Event-based sampling (indirect, statistical)
  - Direct measurement (deterministic)

Inclusive and Exclusive Profiles

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



Terminology – Example

- For routine "int main( )":
- Exclusive time
  - 100-20-50-20=10 secs
- Inclusive time
  - 100 secs
- Calls
  - 1 call
- Subrs (no. of child routines called)
  - 3
- Inclusive time/call
  - 100secs

```
int main( )
{ /* takes 100 secs */

  f1(); /* takes 20 secs */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */

  /* other work */
}

/*
Time can be replaced by  counts
from PAPI e.g., PAPI_FP_INS. */
```


Flat and Callpath Profiles

- Static call graph
  - Shows all parent-child calling relationships in a program
- Dynamic call graph
  - Reflects actual execution time calling relationships
- Flat profile
  - Performance metrics for when event is active
  - Exclusive and inclusive
- Callpath profile
  - Performance metrics for calling path (event chain)
  - Differentiate performance with respect to program execution state
  - Exclusive and inclusive

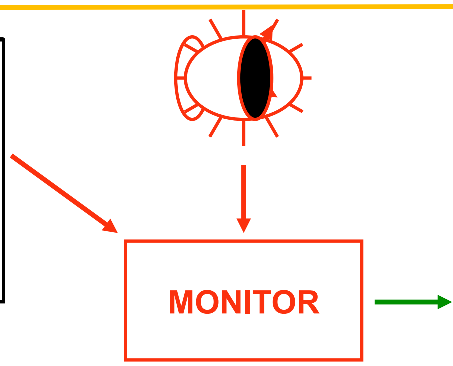
Tracing Measurement

Process A:

```
void master {
  trace(ENTER, 1);
  ...
  trace(SEND, B);
  send(B, tag, buf);
  ...
  trace(EXIT, 1);
}
```

Process B:

```
void worker {
  trace(ENTER, 2);
  ...
  recv(A, tag, buf);
  trace(RECV, A);
  ...
  trace(EXIT, 2);
}
```



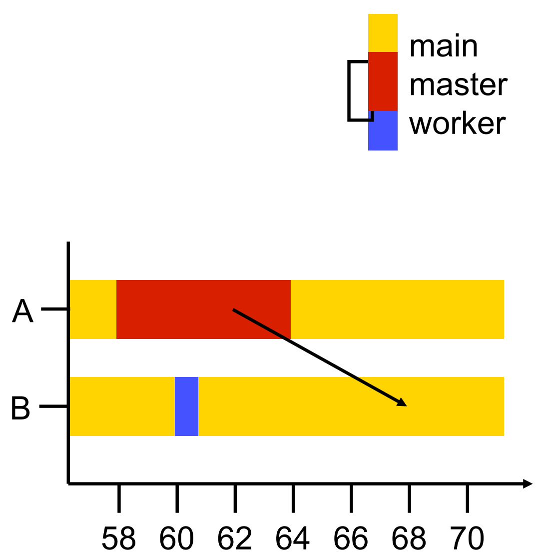
1	master
2	worker
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Tracing Analysis and Visualization

1	master
2	worker
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



Trace Formats

- Different tools produce different formats
  - Differ by event types supported
  - Differ by ASCII and binary representations
    - Vampir Trace Format (VTF)
    - KOJAK (EPILOG)
    - Jumpshot (SLOG-2)
    - Paraver
- Open Trace Format (OTF)
  - Supports interoperation between tracing tools

Profiling / Tracing Comparison

- Profiling
  - ☺ Finite, bounded performance data size
  - ☺ Applicable to both direct and indirect methods
  - 😐 Loses time dimension (not entirely)
  - ☹ Lacks ability to fully describe process interaction
- Tracing
  - ☺ Temporal and spatial dimension to performance data
  - ☺ Capture parallel dynamics and process interaction
  - 😐 Some inconsistencies with indirect methods
  - ☹ Unbounded performance data size (large)
  - ☹ Complex event buffering and clock synchronization

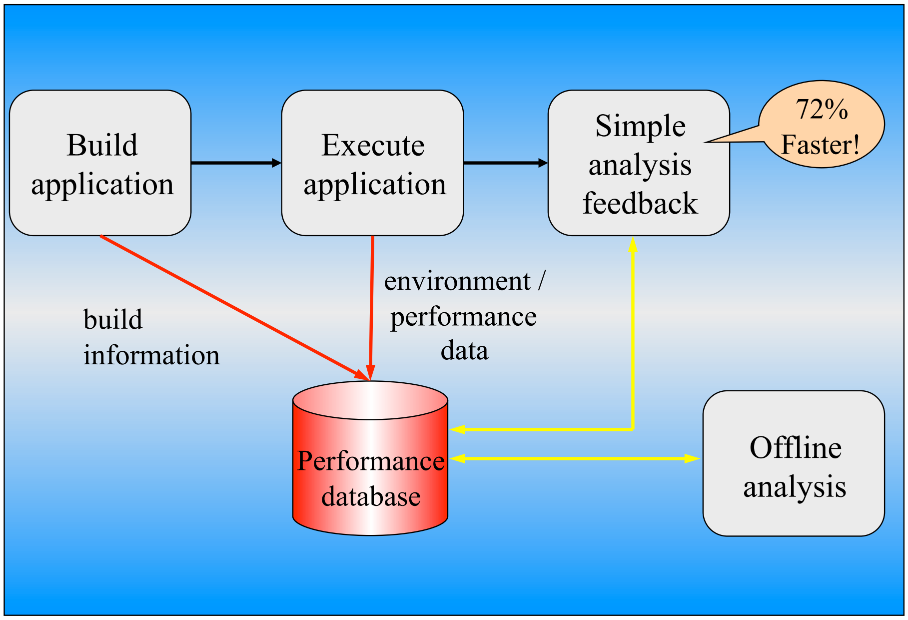
Performance Problem Solving Goals

- Answer questions at multiple levels of interest
  - High-level performance data spanning dimensions
    - machine, applications, code revisions, data sets
    - examine broad performance trends
  - Data from low-level measurements
    - use to predict application performance
- Discover general correlations
  - performance and features of external environment
  - Identify primary performance factors
- Benchmarking analysis for application prediction
- Workload analysis for machine assessment

Performance Analysis Questions

- How does performance vary with different compilers?
- Is poor performance correlated with certain OS features?
- Has a recent change caused unanticipated performance?
- How does performance vary with MPI variants?
- Why is one application version faster than another?
- What is the reason for the observed scaling behavior?
- Did two runs exhibit similar performance?
- How are performance data related to application events?
- Which machines will run my code the fastest and why?
- Which benchmarks predict my code performance best?

Automatic Performance Analysis



Performance Data Management

- Performance diagnosis and optimization involves multiple performance experiments
- Support for common performance data management tasks augments tool use
  - Performance experiment data and metadata storage
  - Performance database and query
- What type of performance data should be stored?
  - Parallel profiles or parallel traces
  - Storage size will dictate
  - Experiment metadata helps in meta analysis tasks
- Serves tool integration objectives

Metadata Collection

- Integration of metadata with each parallel profile
  – Separate information from performance data
- Three ways to incorporate metadata
  – Measured hardware/system information
    – CPU speed, memory in GB, MPI node IDs, …
  – Application instrumentation (application-specific)
    – Application parameters, input data, domain decomposition
    – Capture arbitrary name/value pair and save with experiment
  – Data management tools can read additional metadata
    – Compiler flags, submission scripts, input files, …
    – Before or after execution
- Enhances analysis capabilities

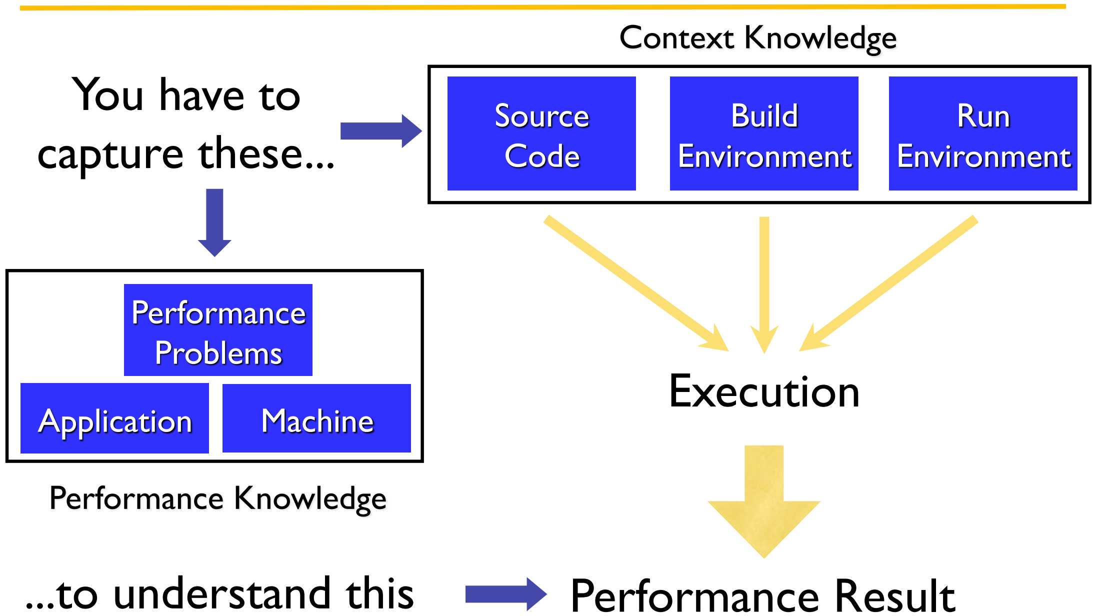
Performance Data Mining

- Conduct parallel performance analysis in a systematic, collaborative and reusable manner
  – Manage performance complexity and automate process
  – Discover performance relationship and properties
  – Multi-experiment performance analysis
- Data mining applied to parallel performance data
  – Comparative, clustering, correlation, characterization, …
  – Large-scale performance data reduction
- Implement extensible analysis framework
  – Abtraction / automation of data mining operations
  – Interface to existing analysis and data mining tools

How to explain performance?

- Should not just redescribed performance results
- Should explain performance phenomena
  - What are the causes for performance observed?
  - What are the factors and how do they interrelate?
  - Performance analytics, forensics, and decision support
- Add *knowledge* to do more intelligent things
  - Automated analysis needs good informed feedback
  - Performance model generation requires interpretation
- Performance knowledge discovery framework
  - Integrating meta-information
  - Knowledge-based performance problem solving

Metadata and Knowledge Role



Performance Optimization Process

- Performance characterization
  - Identify major performance contributors
  - Identify sources of performance inefficiency
  - Utilize timing and hardware measures
- Performance diagnosis (Performance Debugging)
  - Look for conditions of performance problems
  - Determine if conditions are met and their severity
  - What and where are the performance bottlenecks
- Performance tuning
  - Focus on dominant performance contributors
  - Eliminate main performance bottlenecks

Part III:
PAPI

Shirley Moore, David Cronk, Heike Jagode, and Dan Terpstra
Innovative Computing Lab
University of Tennessee, Knoxville

Outline

- Introduction
- PAPI Utilities
- An Example
- Some PAPI counters
- PAPI and Multi-core
- Component PAPI – The New Wave

Hardware Counters

Hardware performance counters available on most modern microprocessors can provide insight into:

1. Whole program timing
2. Cache behaviors
3. Branch behaviors
4. Memory and resource access patterns
5. Pipeline stalls
6. Floating point efficiency
7. Instructions per cycle

Hardware counter information can be obtained with:

1. Subroutine or basic block resolution
2. Process or thread attribution

What's PAPI?

- Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.
- Countable events are defined in two ways:
  - Platform-neutral *preset* events
  - Platform-dependent native events
- Presets can be derived from multiple *native events*
- All events are referenced by name and collected in EventSets for sampling
- Events can be multiplexed if counters are limited
- Statistical sampling implemented by:
  - Hardware overflow if supported by the platform
  - Software overflow with timer driven sampling

Where's PAPI

- PAPI runs on most modern processors and operating systems of interest to HPC:
  - IBM POWER / AIX / Linux
  - Blue Gene / L / P…
  - Intel Pentium, Core2, Core i7, Atom / Linux
  - Intel Itanium / Linux
  - AMD Athlon, Opteron / Linux
  - Cray XT(n) / CLE
  - Altix, Sparc, Niagara …

PAPI Utilities: *papi_cost*

```
$ utils/papi_cost -h
This is the PAPI cost program.
It computes min / max / mean / std. deviation for PAPI start/stop pairs;
   for PAPI reads, and for PAPI_accums.
   Usage:

    cost [options] [parameters]
    cost TESTS_QUIET

Options:

  -b BINS       set the number of bins for the graphical
                 distribution of costs. Default: 100
  -d            show a graphical distribution of costs
  -h            print this help message
  -s            show number of iterations above the first
                 10 std deviations
  -t THRESHOLD  set the threshold for the number of
                 iterations. Default: 100,000
```


PAPI Utilities: *papi_cost*

```
$ utils/papi_cost
Cost of execution for PAPI start/stop and PAPI read.
This test takes a while. Please be patient...
Performing start/stop test...

Total cost for PAPI_start/stop(2 counters) over 1000000 iterations
min cycles   : 63
max cycles   : 17991
mean cycles  : 69.000000
std deviation: 34.035263
 Performing start/stop test...

Performing read test...

Total cost for PAPI_read(2 counters) over 1000000 iterations
min cycles   : 288
max cycles   : 102429
mean cycles  : 301.000000
std deviation: 144.694053
 cost.c                                  PASSED
```

PAPI Utilities: *papi_cost*

```
Cost distribution profile

      63:**************************** 999969 counts ****************************
     153:
     243:
     [...]
    1863:
    1953:********************
    2043:
    2133:********************
    2223:
    2313:
    2403:********************
    2493:********************
    2583:****************************************
    2673:****************************************
    2763:****************************************************************************
    2853:****************************************
    2943:
    3033:********************
    3123:****************************************
    3213:****************************************
    3303:
    3393:
    3483:
    3573:
    3663:********************
```


PAPI Utilities: *papi_avail*

```
$ utils/papi_avail -h
Usage: utils/papi_avail [options]
Options:

General command options:
    -a, --avail   Display only available preset events
    -d, --detail  Display detailed information about all preset events
    -e EVENTNAME  Display detail information about specified preset or native event
    -h, --help    Print this help message

Event filtering options:
        --br          Display branch related PAPI preset events
        --cache       Display cache related PAPI preset events
        --cnd         Display conditional PAPI preset events
        --fp          Display Floating Point related PAPI preset events
        --ins         Display instruction related PAPI preset events
        --idl         Display Stalled or Idle PAPI preset events
        --l1          Display level 1 cache related PAPI preset events
        --l2          Display level 2 cache related PAPI preset events
        --l3          Display level 3 cache related PAPI preset events
        --mem         Display memory related PAPI preset events
        --msc         Display miscellaneous PAPI preset events
        --tlb         Display Translation Lookaside Buffer PAPI preset events

This program provides information about PAPI preset and native events.
PAPI preset event filters can be combined in a logical OR.
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 4.0.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel Core i7 (21)
CPU Revision             : 5.000000
CPUID Info               : Family: 6  Model: 26  Stepping: 5
CPU Megahertz            : 2926.000000
CPU Clock Megahertz      : 2926
Hdw Threads per core     : 1
Cores per Socket         : 4
NUMA Nodes               : 2
CPU's per Node           : 4
Total CPU's              : 8
Number Hardware Counters : 7
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    [MORE…]
```


PAPI Utilities: *papi_avail*

```
     [CONTINUED…]

--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No    No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses

[…]

PAPI_VEC_SP  0x80000069  Yes   No   Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   No   Double precision vector/SIMD instructions
-------------------------------------------------------------------------
Of 107 possible events, 34 are available, of which 9 are derived.

avail.c                                  PASSED
```

PAPI Utilities: *papi_avail*

```
$ utils/papi_avail -e PAPI_FP_OPS
[…]
---------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Event name:                   PAPI_FP_OPS
Event Code:                   0x80000066
Number of Native Events:      2
Short Description:           |FP operations|
Long Description:            |Floating point operations|
Developer's Notes:           ||
Derived Type:                |DERIVED_ADD|
Postfix Processing String:   ||
 Native Code[0]: 0x4000801b |FP_COMP_OPS_EXE:SSE_SINGLE_PRECISION|
 Number of Register Values: 2
 Register[ 0]: 0x0000000f |Event Selector|
 Register[ 1]: 0x00004010 |Event Code|
 Native Event Description: |Floating point computational micro-ops, masks:SSE* FP single precision Uops|

 Native Code[1]: 0x4000081b |FP_COMP_OPS_EXE:SSE_DOUBLE_PRECISION|
 Number of Register Values: 2
 Register[ 0]: 0x0000000f |Event Selector|
 Register[ 1]: 0x00008010 |Event Code|
 Native Event Description: |Floating point computational micro-ops, masks:SSE* FP double precision Uops|
-------------------------------------------------------------------------
```


PAPI Utilities: *papi_native_avail*

```
UNIX> utils/papi_native_avail
Available native events and hardware information.
--------------------------------------------------------------------------------
[…]
Event Code   Symbol  | Long Description |
--------------------------------------------------------------------------------
0x40000010   BR_INST_EXEC  | Branch instructions executed                      |
  40000410   :ANY  | Branch instructions executed                                |
  40000810   :COND  | Conditional branch instructions executed                   |
  40001010   :DIRECT  | Unconditional branches executed                          |
  40002010   :DIRECT_NEAR_CALL  | Unconditional call branches executed           |
  40004010   :INDIRECT_NEAR_CALL  | Indirect call branches executed              |
  40008010   :INDIRECT_NON_CALL  | Indirect non call branches executed           |
  40010010   :NEAR_CALLS  | Call branches executed                               |
  40020010   :NON_CALLS  | All non call branches executed                        |
  40040010   :RETURN_NEAR  | Indirect return branches executed                   |
  40080010   :TAKEN  | Taken branches executed                                   |
--------------------------------------------------------------------------------
0x40000011   BR_INST_RETIRED  | Retired branch instructions                    |
  40000411   :ALL_BRANCHES  | Retired branch instructions (Precise Event)        |
  40000811   :CONDITIONAL  | Retired conditional branch instructions (Precise  |
           | Event)                                                            |
  40001011   :NEAR_CALL  | Retired near call instructions (Precise Event)        |
--------------------------------------------------------------------------------
[…]
```

PAPI Utilities: *papi_native_avail*

```
UNIX> utils/papi_native_avail -e DATA_CACHE_REFILLS
Available native events and hardware information.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Event name:                   DATA_CACHE_REFILLS
Event Code:                   0x4000000b
Number of Register Values:    2
Description:                 |Data Cache Refills from L2 or System|
 Register[ 0]:    0x0000000f  |Event Selector|
 Register[ 1]:    0x00000042  |Event Code|

Unit Masks:
 Mask Info:                  |:SYSTEM|Refill from System|
  Register[ 0]:  0x0000000f  |Event Selector|
  Register[ 1]:  0x00000142  |Event Code|
 Mask Info:                  |:L2_SHARED|Shared-state line from L2|
  Register[ 0]:  0x0000000f  |Event Selector|
  Register[ 1]:  0x00000242  |Event Code|
 Mask Info:                  |:L2_EXCLUSIVE|Exclusive-state line from L2|
  Register[ 0]:  0x0000000f  |Event Selector|
  Register[ 1]:  0x00000442  |Event Code|
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser PRESET PAPI_FP_OPS
Event Chooser: Available events which can be added with given events.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------
    Name        Code     Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No    Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No    Level 1 instruction cache misses
PAPI_L2_ICM  0x80000003  No    Level 2 instruction cache misses
[…]
PAPI_L1_DCA  0x80000040  No    Level 1 data cache accesses
PAPI_L2_DCR  0x80000044  No    Level 2 data cache reads
PAPI_L2_DCW  0x80000047  No    Level 2 data cache writes
PAPI_L1_ICA  0x8000004c  No    Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  No    Level 2 instruction cache accesses
PAPI_L2_TCA  0x80000059  No    Level 2 total cache accesses
PAPI_L2_TCW  0x8000005f  No    Level 2 total cache writes
PAPI_FML_INS 0x80000061  No    Floating point multiply instructions
PAPI_FDV_INS 0x80000063  No    Floating point divide instructions
-------------------------------------------------------------------------
Total events reported: 34
event_chooser.c                          PASSED
```

PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser PRESET PAPI_FP_OPS PAPI_L1_DCM
Event Chooser: Available events which can be added with given events.
--------------------------------------------------------------------------------
[…]
--------------------------------------------------------------------------------

    Name        Code    Deriv Description (Note)
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_TOT_CYC 0x8000003b  No   Total cycles
-------------------------------------------------------------------------
Total events reported: 2
event_chooser.c                          PASSED
```


PAPI Utilities: *papi_event_chooser*

```
$ utils/papi_event_chooser NATIVE RESOURCE_STALLS:LD_ST X87_OPS_RETIRED
    INSTRUCTIONS_RETIRED
[…]
--------------------------------------------------------------------------------
UNHALTED_CORE_CYCLES    0x40000000
 |count core clock cycles whenever the clock signal on the specific core is running (not
    halted). Alias to event CPU_CLK_UNHALTED:CORE_P|
 |Register Value[0]: 0x20003       Event Selector|
 |Register Value[1]: 0x3c          Event Code|
-------------------------------------------------------------------------
UNHALTED_REFERENCE_CYCLES       0x40000002
 |Unhalted reference cycles. Alias to event CPU_CLK_UNHALTED:REF|
 |Register Value[0]: 0x40000       Event Selector|
 |Register Value[1]: 0x13c         Event Code|
-------------------------------------------------------------------------
CPU_CLK_UNHALTED        0x40000028
 |Core cycles when core is not halted|
 |Register Value[0]: 0x60000       Event Selector|
 |Register Value[1]: 0x3c          Event Code|
    0x40001028  :CORE_P  |Core cycles when core is not halted|
    0x40008028  :NO_OTHER  |Bus cycles when core is active and the other is halted|
-------------------------------------------------------------------------
Total events reported: 3
event_chooser.c                          PASSED
```

PAPI Utilities: *papi_command_line*

```
$ papi_command_line PAPI_FP_OPS
Successfully added: PAPI_FP_OPS

PAPI_FP_OPS :   100000000

----------------------------------
Verification: None.
 This utility lets you add events from the command line interface to see if they work.
command_line.c                              PASSED


$ papi_command_line PAPI_FP_OPS PAPI_L1_DCA
Successfully added: PAPI_FP_OPS
Successfully added: PAPI_L1_DCA

PAPI_FP_OPS :   100000000
PAPI_L1_DCA :   120034404

----------------------------------
Verification: None.
 This utility lets you add events from the command line interface to see if they work.
command_line.c                              PASSED
```

The Code

```
#define ROWS 1000              // Number of rows in each matrix
#define COLUMNS 1000           // Number of columns in each matrix
```

```
void classic_matmul()
{
   // Multiply the two matrices
   int i, j, k;
   for (i = 0; i < ROWS; i++) {
      for (j = 0; j < COLUMNS; j++) {
         float sum = 0.0;
         for (k = 0; k < COLUMNS; k++) {
            sum +=
               matrix_a[i][k] * matrix_b[k][j];
         }
         matrix_c[i][j] = sum;
      }
   }
}
```

```
void interchanged_matmul()
{
   // Multiply the two matrices
   int i, j, k;
   for (i = 0; i < ROWS; i++) {
      for (k = 0; k < COLUMNS; k++) {
         for (j = 0; j < COLUMNS; j++) {
            matrix_c[i][j] +=
               matrix_a[i][k] * matrix_b[k][j];
         }
      }
   }
}

// Note that the nesting of the innermost loops
// has been changed. The index variables j and k
// change the most frequently and the access
// pattern through the operand matrices is
// sequential using a small stride (one.) This
// change improves access to memory data through
// the data cache. Data translation lookaside
// buffer (DTLB) behavior is also improved.
```

IPC – instructions per cycle

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_IPC Test (PAPI_ipc)		
Real time	13.6093 sec	2.9796 sec
Processor time	13.5359 sec	2.9556 sec
IPC	0.3697	1.6936
Instructions	9007035063	9009011383
High Level IPC Test (PAPI_{start,stop}_counters)		
Real time	13.6106 sec	2.9762 sec
IPC	0.3697	1.6939
PAPI_TOT_CYC	24362605525	5318626915
PAPI_TOT_INS	9007034503	9009011245
Low Level IPC Test (PAPI low level calls)		
Real time	13.6113 sec	2.9772 sec
IPC	0.3697	1.6933
PAPI_TOT_CYC	24362750167	5320395138
PAPI_TOT_INS	9007034381	9009011130

- All three PAPI methods consistent
- Roughly 460% improvement in reordered code

Data Cache Access

Data Cache Misses can be considered in 3 categories:

- **Compulsory: Occurs on first reference to a data item.**
  - Prefetching
- **Capacity: Occurs when the working set exceeds the cache capacity.**
  - Spatial locality
  - Smaller working set (blocking/tiling algorithms)
- **Conflict: Occurs when a data item is referenced after the cache line containing the item was evicted earlier.**
  - Temporal locality
  - Data layout; memory access patterns

L1 Data Cache Access

Measurement	Classic mat_mul	Reordered mat_mul
DATA_CACHE_ACCESSES	2002807841	3008528961
DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED	205968263	60716301
DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED	61970925	1950282
PAPI_L1_DCA	2002808034	3008528895
PAPI_L1_DCM	268010587	62680818
Data Cache Request Rate	0.2224 req/inst	0.3339 req/inst
Data Cache Miss Rate	0.0298 miss/inst	0.0070 miss/inst
Data Cache Miss Ratio	0.1338 miss/req	0.0208 miss/req

- Two techniques
  - First uses native events
  - Second uses PAPI presets only
- ~50% more requests from reordered code
- 1/4 as many misses per instruction
- 1/6 as many misses per request

Branching

Measurement	Classic mat_mul	Reordered mat_mul
PAPI_BR_INS	1001028240	1001006987
PAPI_BR_MSP	1028256	1006984
PAPI_BR_TKN	1000027233	1000005980
Branch Rate	0.1111 br/inst	0.1111 br/inst
Branch Miss Rate	0.0001 miss/inst	0.0001 miss/inst
Branch Miss Ratio	0.0010 miss/br	0.0010 miss/br
Branch Taken Rate	0.1110 tkn/inst	0.1110 tkn/inst
Branch Taken Ratio	0.9990 tkn/br	0.9990 tkn/br
Instr / Branch	8.9978 inst/br	8.9999 inst/br

- Uses all PAPI Presets!
- Branch behavior nearly identical in both codes
- Roughly 1 branch every 9 instructions
- 1 miss per 1000 branches (remember ROWS?)
- Branching and branch misses can be reduced with loop unrolling, loop fusion and function in-lining.

Performance Measurement Categories

- Efficiency
  - Instructions per cycle (IPC)
  - Memory bandwidth
- Caches
  - Data cache misses and miss ratio
  - Instruction cache misses and miss ratio
- Lower level cache misses and miss ratio
- Translation lookaside buffers (TLB)
  - Data TLB misses and miss ratio
  - Instruction TLB misses and miss ratio
- Control transfers
  - Branch mispredictions
  - Near return mispredictions
- Special cases
  - Unaligned data access
  - Floating point operations
  - Floating point exceptions

The Multicore Dilemma

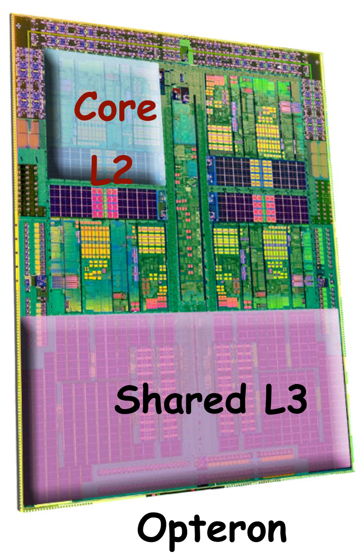
- Multicore is the (near term) future of Petascale computing
- Minimizing Resource contention is key
  - Memory bandwidth
  - Cache sharing & collisions
  - Bus and other resource contention
- Current tools don't support first-person counting of shared events
- Current architectures don't encourage first-person counting of shared events

Current "State of the Art"

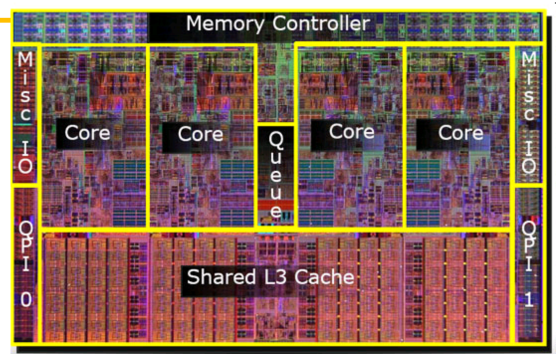
- Counter support for shared resources is broken
  - Every vendor has a different approach
  - Often 3rd person, not 1st person
  - Counts often polluted by other cores
  - No exclusive reservation of shared counter resources
  - No migration of events with tasks
- PAPI research is underway to address this

Multicore counter support

- Intel Core2 Duos:
  - SELF/ANY
  - L2 shared cache, bus, snoop
  - 39 events/~140 are core qualified
- AMD Opteron Shanghai
  - 4 L3 shared cache events:
    - READ_REQUEST_TO_L3_CACHE
    - L3_CACHE_MISSES
    - L3_FILLS_CAUSED_BY_L2_EVICTIONS
    - L3_EVICTIONS
  - First 3 are qualified per core:
    - CORE0, CORE1, CORE2, CORE3
    - Only 1 core can (safely) count these events at a time



Multicore counter support (cont.)



- Intel i7 (Nehalem)
  - 7 counters per core
    - 3 fixed, 4 programmable
  - 8 counters shared on-chip for "Uncore" events
    - Require global, not process level access
    - Not currently supported by PAPI
  - 117 native events available to PAPI users
  - 28 PAPI PRESET events

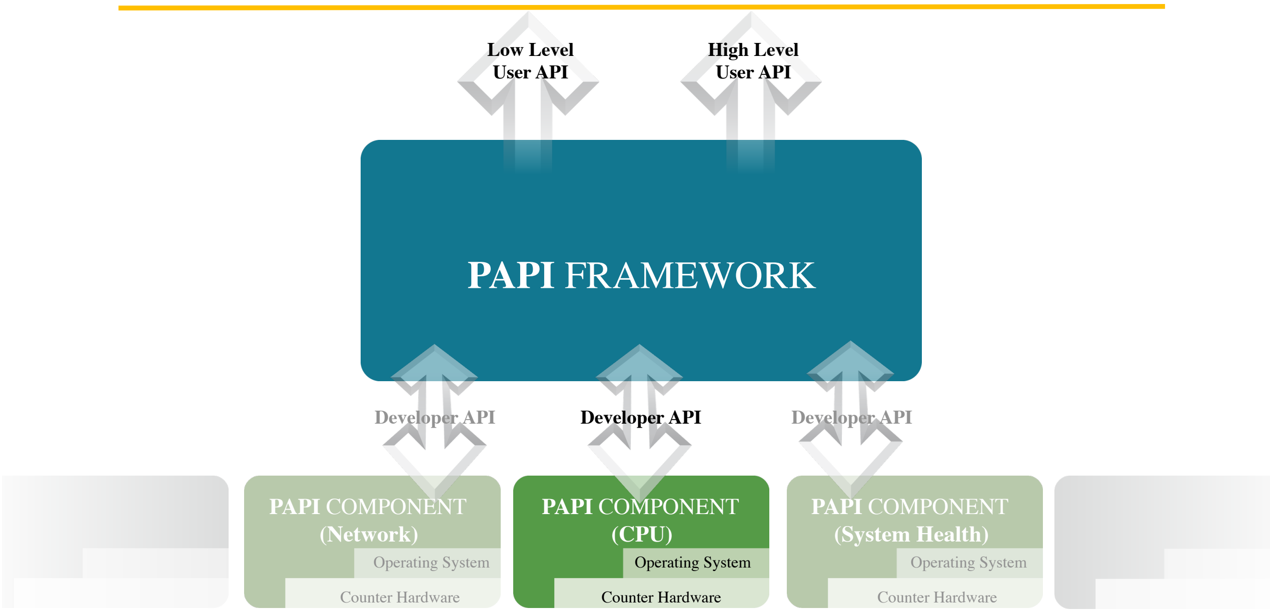
Extending PAPI beyond the CPU

- PAPI has historically targeted on on-processor performance counters
- Several categories of off-processor counters exist
  - network interfaces: Myrinet, Infiniband, GigE
  - memory interfaces: Cray SeaStar, Gemini
  - thermal and power interfaces: ACPI, lm-sensors
  - accelerators?
- CHALLENGE:
  - Extend the PAPI interface to address multiple counter domains
  - Preserve the PAPI calling semantics, ease of use, and platform independence for existing applications

Component PAPI Goals

- Support simultaneous access to on- and off-processor counters
- Isolate hardware dependent code in separable ‘component’ modules
- Extend platform independent code to support multiple simultaneous components
- Add or modify API calls to support access to any of several components
- Modify build environment for easy selection and configuration of multiple available components

Component **PAPI**



File System Components: Lustre

- Measures data collected in: `/proc/…/stats`
  and: `/proc/…/read_ahead_stats`

```
Hits631592284
misses                          9467662
readpage not consecutive        931757
miss inside window              81301
failed grab_cache_page          5621647
failed lock match               2135855
read but discarded              2089608
zero size window                6136494
read-ahead to EOF               160554
hit max r-a issue               25610
```

- Snippet of available native events for Lustre:

```
0x44000002  fastfs_llread       | bytes read on this lustre client
0x44000003  fastfs_llwrite      | bytes written on this lustre client
0x44000004  fastfs_wrong_readahead         | bytes read but discarded due to readahead
0x44000005  work_llread         | bytes read on this lustre client
0x44000006  work_llwrite        | bytes written on this lustre client
0x44000007  work_wrong_readahead | bytes read but discarded due to readahead
```

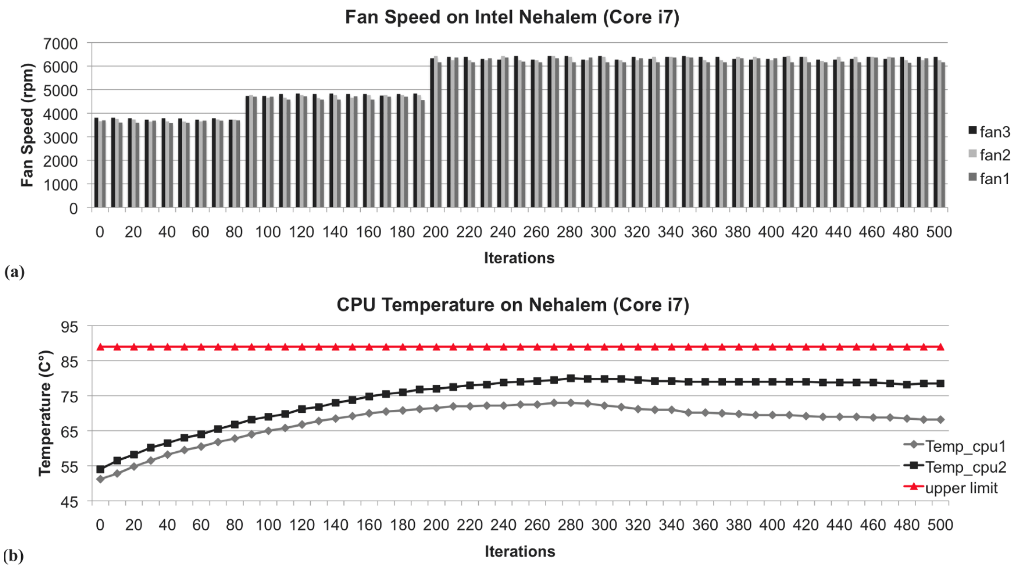

System Health Components: lm-sensors

- Access computer health monitoring sensors,
  exposed by lm_sensors library
- user is able to closely monitor the system's hardware health
  - observe feedback between performance and environmental conditions
- Available features and monitored events depend on hardware setup
- Snippet of available native events for lm-sensors:

```
…
0x4c000000   LM_SENSORS.max1617-i2c-0-18.temp1.temp1_input
0x4c000001   LM_SENSORS.max1617-i2c-0-18.temp1.temp1_max
0x4c000002   LM_SENSORS.max1617-i2c-0-18.temp1.temp1_min
…
0x4c000049   LM_SENSORS.w83793-i2c-0-2f.fan1.fan1_input
0x4c00004a   LM_SENSORS.w83793-i2c-0-2f.fan1.fan1_min
0x4c00004b   LM_SENSORS.w83793-i2c-0-2f.fan1.fan1_alarm
…
```

lm-sensors Component Example

- libsensors version 3.1.1



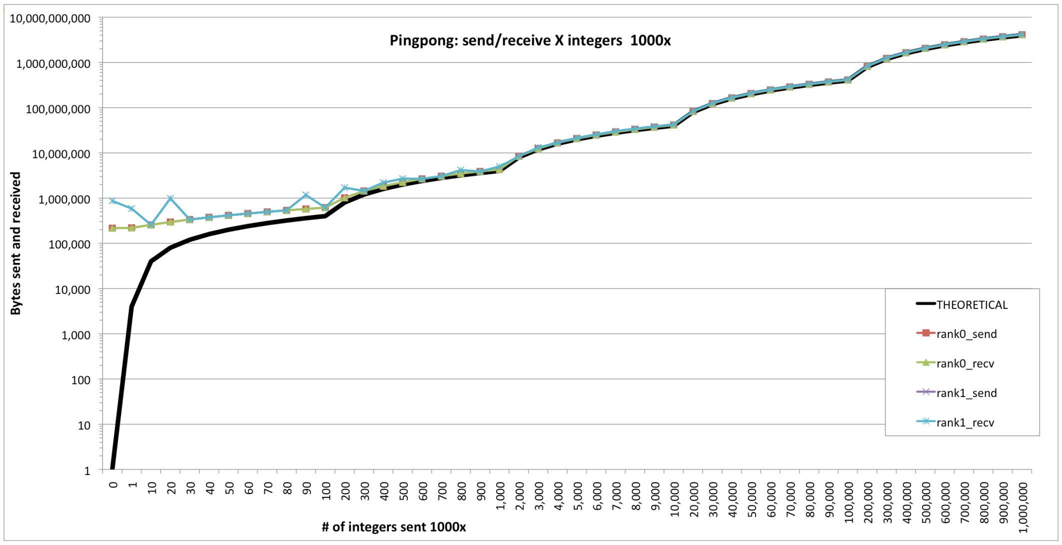
(a) (b)

Network Components: InfiniBand

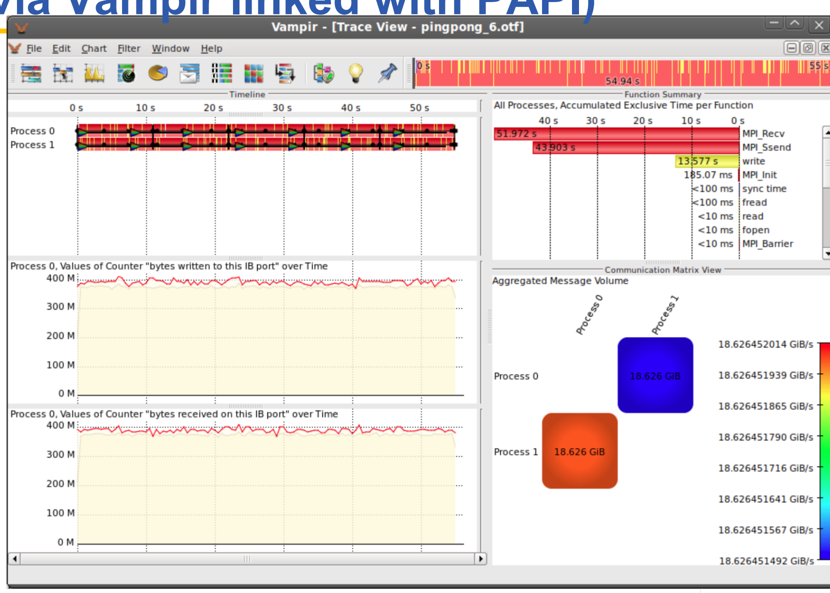
- Measures everything that is provided by the libibmad:
- Errors, Bytes, Packets, local IDs (LID), global IDs (GID), etc.
- ibmad library provides low-layer IB functions for use by the IB diagnostic and management programs, including MAD, SA, SMP, and other basic IB functions
- Snippet of available native events on a machine with 2 IB devices, mthca0 and mthca1:

```
…
0x44000000   mthca0_1_recv        | bytes received on this IB port
0x44000001   mthca0_1_send        | bytes written to this IB port
0x44000002   mthca1_1_recv        | bytes received on this IB port
0x44000003   mthca1_1_send        | bytes written to this IB port
…
```

InfiniBand Component Results

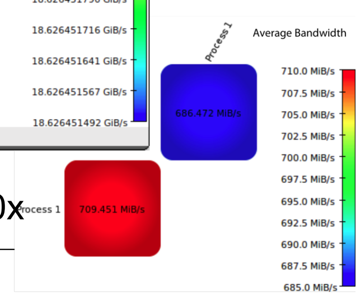


InfiniBand events measured over time (via Vampir linked with PAPI)



IB Counter resolution in Vampir:

1 sec



- Run Pingpong 5x:
  send 1,000,000 integers 1000x
  (theor: ~19GB)

For more information

- PAPI Website: http://icl.cs.utk.edu/papi/
  - Software
  - Release notes
  - Documentation
  - Links to tools that use PAPI
  - Mailing/discussion lists

TAU Performance System®

Part IV: TAU Internals

Performance Tools FAQ/Concerns

- Does it automatically instrument my code? At the routine level? At the outer-loop level?
- Can it show me where time is spent in my code? PAPI Flops? L1 data cache misses? Can I measure more than one quantity in a trial?
- Does the tool support profiling (runtime summarization) as well as tracing (time-line based displays)? What about profile snapshots? Callpath (parent-child) profiles? Can I use it to easily benchmark codes?
- Can I observe the performance data at runtime as the application executes?
- Can it show me memory utilization? Memory leaks? Mallocs/frees? When and where?
- What about I/O? Can I observe bandwidth of reads/writes? Volume of I/O? What about Kernel events? User space+Kernel?
- What is the typical overhead? Can I reduce it to < 5%? < 1%? Can it compensate and remove timer overhead from performance data? Can it throttle away instrumentation in lightweight routines at runtime to reduce overhead?
- I already have profile data from <XYZ> tool. Can it import my legacy data?
- I prefer <XYZ> performance tool for visualization. Can it hook up with this tool? Are there converters?

Performance Tools FAQ/Concerns (contd.)

- Can I use it for multi-core CPUs? Compare the performance of application running on a single vs. multi-core processor? Can I observe multi-core data snoops, invalidates?
- Can I share the performance data with my colleagues in a secure manner (web/database)? Can it automatically track progress of my application over time (~ 6 mos)? Can I use it for scalability studies? Over multiple platforms?
- Are the GUI client tools available under Linux? MS Windows? Apple?
- Does it run on all Cray, IBM, SGI, HP … platforms? CNL? Catamount?
- Does it support MPI? MPI2? Threads? Hybrid MPI+Pthreads/MPI+OpenMP?
- Does it support Fortran? C++, C? Java? Python? Python+MPI+F90+C++…?
- Does it support Intel/PGI/PathScale/IBM/Cray/Sun compilers?
- Are tools available in command-line form & GUI? IDE GUI? Web-based? 3D?
- Is it already installed and supported on my HPC system? What about systems at NERSC? ANL? LLNL? LANL? NASA? DoD? NSF sites?...
- Is there support (phone/e-mail) available for the tool? Professional support? For instrumentation? Analysis?
- Will it work on the new <XYZ> HPC platform scheduled for release six months from now?
- Is it free? BSD license? …

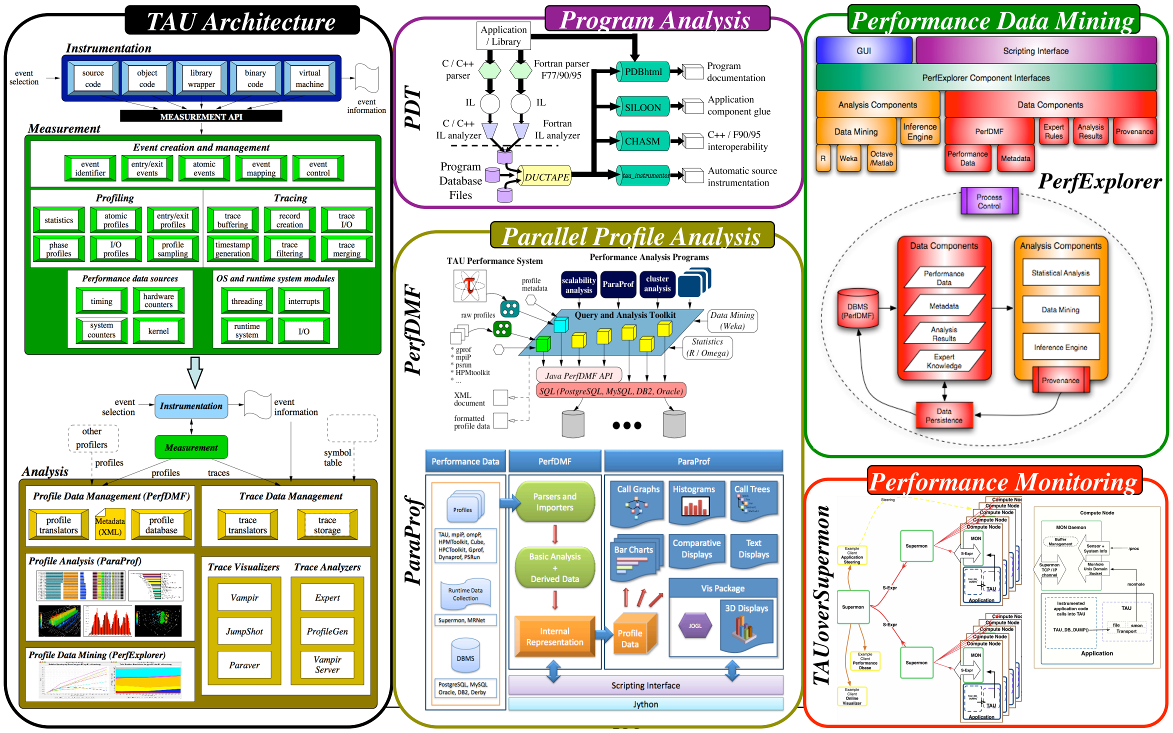
TAU Performance System® Project

- ***T****uning and* ***A****nalysis* ***U****tilities* (15+ year project effort)
- **Performance system framework for HPC systems**
  - Integrated, scalable, and flexible
  - Target parallel programming paradigms
- **Integrated toolkit for performance problem solving**
  - Instrumentation, measurement, analysis, and visualization
  - Portable performance profiling and tracing facility
  - Performance data management and data mining
- ***Partners***
  - LLNL, ANL, LANL
  - Research Centre Jülich, TU Dresden

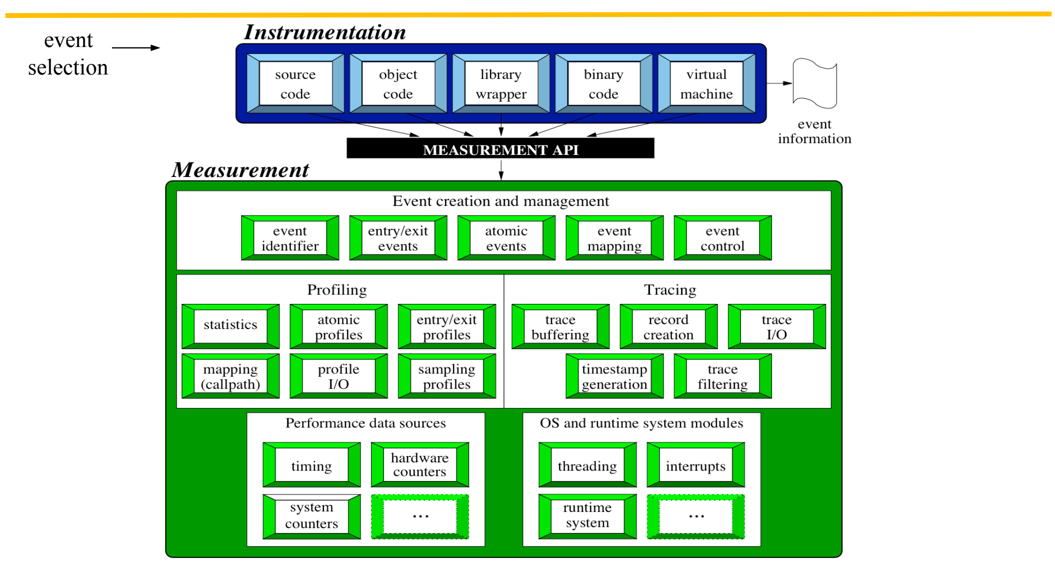
TAU Parallel Performance System Goals

- **Portable (open source) parallel performance system**
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Multi-level, multi-language performance instrumentation
- **Flexible and configurable performance measurement**
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid, object oriented (generic), component-based
- Support for performance mapping
- Integration of leading performance technology
- **Scalable (very large) parallel performance analysis**

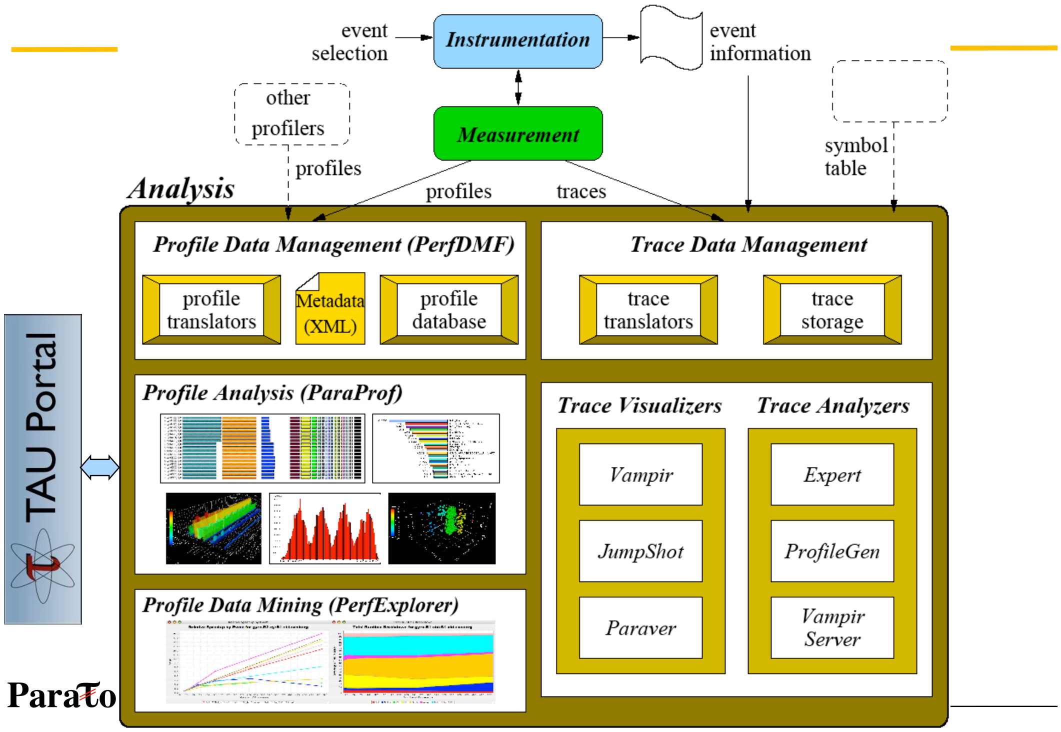
TAU Performance System Components



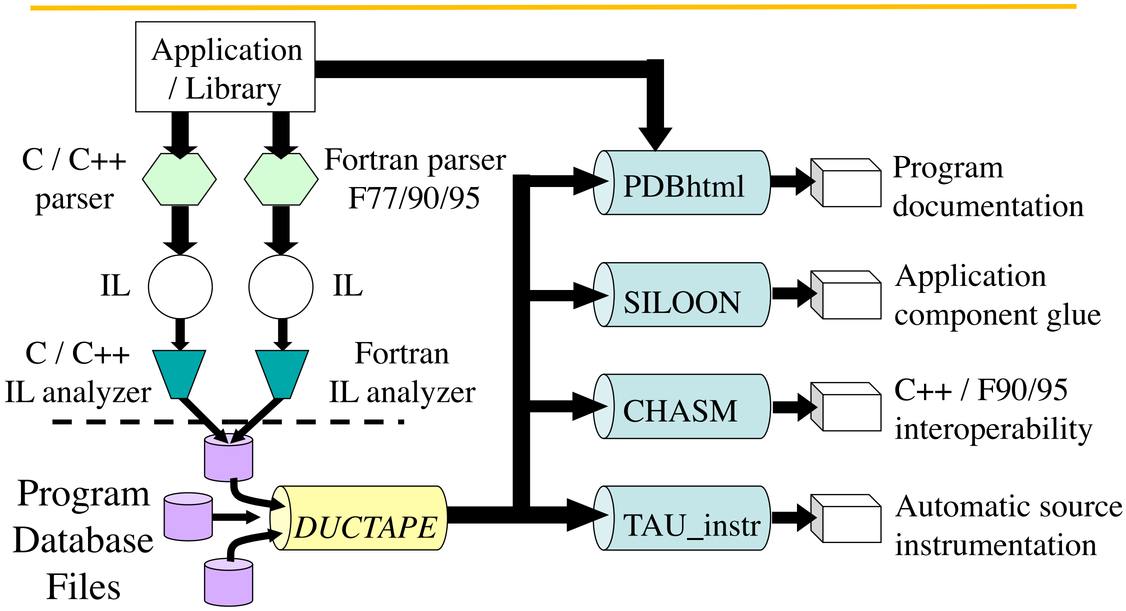
TAU Performance System Architecture



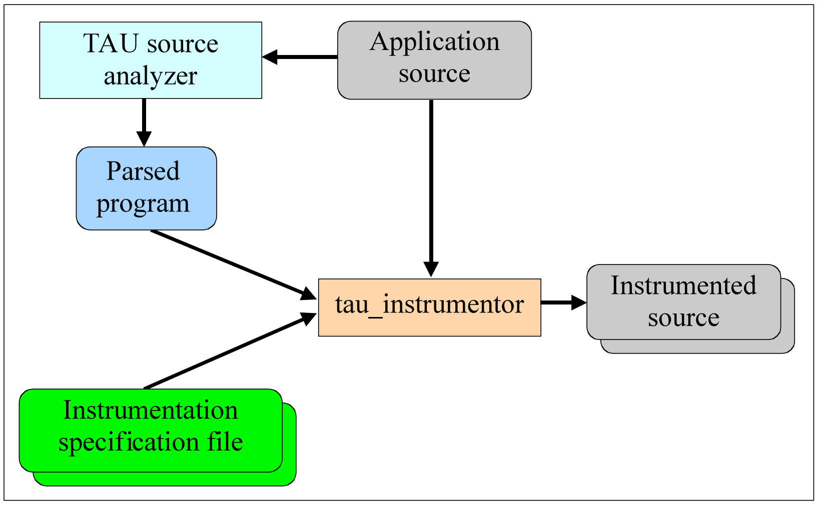
TAU Performance System Architecture



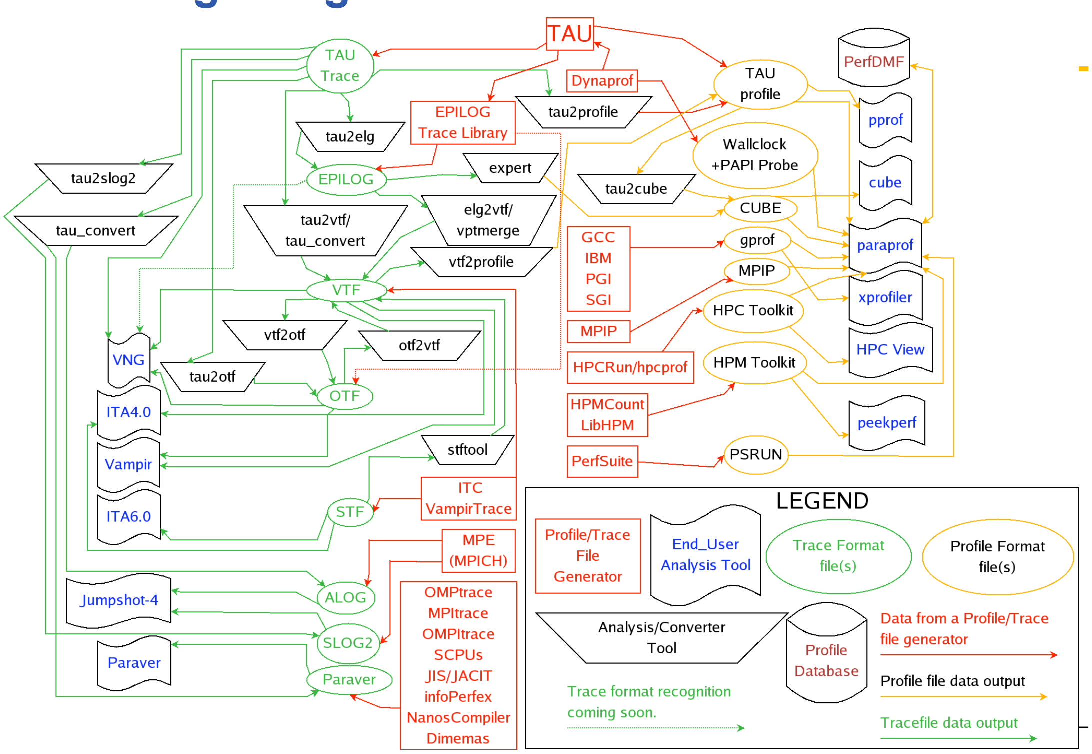
Program Database Toolkit (PDT)



Automatic Source-Level Instrumentation in TAU



Building Bridges to Other Tools



Installing TAU on Linux x86_64

- Install PAPI and PDT
  - PAPI:
    - ./configure –prefix=/usr/global/tools/tau/training/papi-4.0.0;
    - make ; make install
  - PDT:
    - ./configure –prefix=/usr/global/tools/tau/training/pdtoolkit-3.15 –PGI
    - make; make install
- Install TAU:
  - ./installtau –pdt=/usr/global/tools/tau/training/pdtoolkit-3.15 –papi=/usr/global/tools/tau/training/papi-4.0.0
    -c++=pgCC -cc=pgcc -fortran=pgi –mpi
    - Configures multiple typically requested versions for you in x86_64/lib/Makefile.tau-* configurations
  - tau_validate –html –build x86_64 &> results.html
  - mozilla results.html

Validating an Install

results.html
file:///Users/sameer/tmp/results.html

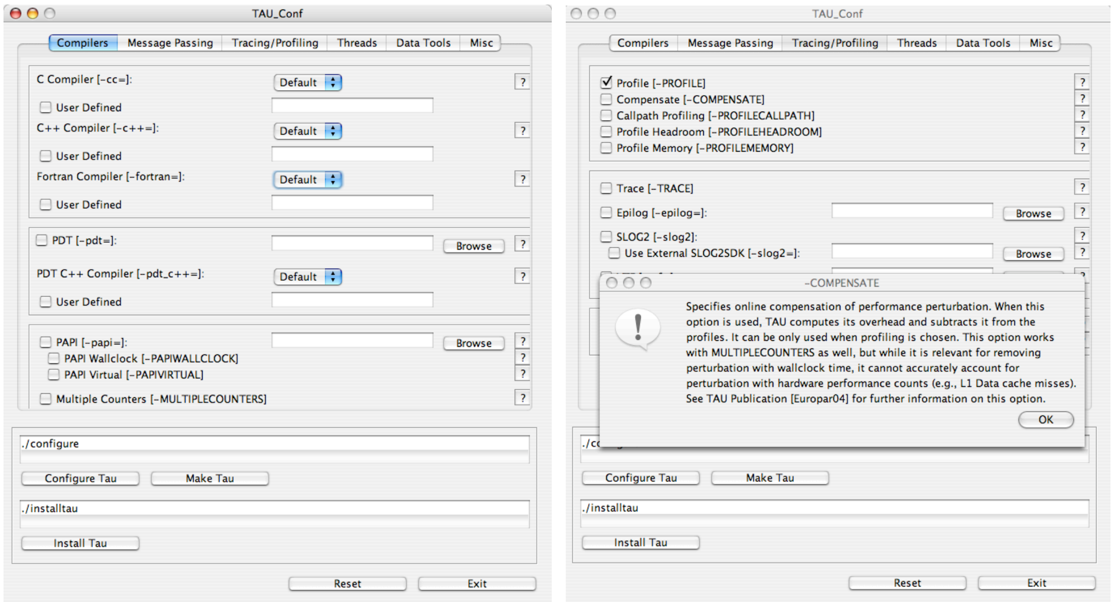
Skipping, not configured with PDT

na/na : PDT-MPI (GFortran) : Makefile.tau-trace

Stub Makefile	C		Fortran (flink)		Fortran (cpplink)		Fortran (clink)		MPI (C)		MPI (Fortran)		CompInst (C)		CompInst (C++)		CompInst (F90)		PDT (C)		PDT (C++)		PDT (Fortran)		PDT (GFortran)		PDT-MPI (C)		PDT-MPI (C++)		PDT-M (Fort
	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build	run	build
Makefile.tau-intelmpi-mpi-pdt	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass
Makefile.tau-trace	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Makefile.tau-intelmpi-papi-mpi-pdt-epilog-scalasca-trace	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pass	pass	pass	pass	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Makefile.tau-intelmpi-papi-mpi-pdt	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass
Makefile.tau-intelmpi-param-papi-mpi-pdt	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass
Makefile.tau-pdt	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	N/A
Makefile.tau-pthread-pdt	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	N/A
Makefile.tau	pass	pass	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	pass	pass	pass	pass	pass	pass	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Makefile.tau-intelmpi-papi-mpi-pdt-vampirtrace-trace	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass
Makefile.tau-intelmpi-mpi-pthread-pdt	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass	pass

No Errors!

TAU_SETUP: A GUI for Installing TAU



Upgrading TAU v2.19.2 configurations to 2.20

- Upgrade TAU
  - Previous installation in /usr/global/tools/tau/training/tau-2.19.2
    - `cd tau-2.20`
    - `./upgradetau  /usr/global/tools/tau/training/tau-2.19.2`
      - Builds all previous configurations in the current dir
      - You may also upgrade with a new package say PDT 3.16
    - `./upgradetau /usr/global/tools/tau/training/tau-2.19.2 -pdt=/usr/global/tools/tau/training/pdtoolkit-3.16`

- Validate your new installation (in bash)

    - `./tau_validate -html -build x86_64 &> results.html`
    - ``mozilla `pwd`/results.html``

Using TAU

- Install TAU
  % ./configure [options]; make clean install
- Replace the names of your compiler with tau_f90.sh, tau_cxx.sh and tau_cc.sh in your makefiles
- Set environment variables
  - Choose the measurement option and compile your code:
    - export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
    - export TAU_OPTIONS='-optVerbose -optKeepFiles -optPreProcess'
  - At runtime, if more than one metric is measured
    - export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_NATIVE_<native_event_name>
      - Use papi_native_avail, papi_avail, and papi_event_chooser to select these preset and native event names
- Build the application, run it, analyze performance data

Using TAU: A brief Introduction

- To instrument source code:
  % export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
  And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:
  % mpif90 foo.f90
  changes to
  % tau_f90.sh foo.f90
- Execute application and then run:
  % pprof (for text based profile display)
  % paraprof (for GUI)
- LABS:
  % source /usr/global/tools/tau/training/tau.bashrc
  % cp /usr/global/tools/tau/training/workshop.tar.gz .
  and follow instructions in README file

TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events ("user-defined timers")
  - Atomic events (e.g., size of memory allocated/freed)
  - Selection of event statistics
- Support definition of "semantic" entities for mapping
- Support for event groups
- Instrumentation optimization (eliminate instrumentation in lightweight routines)

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual (TAU API, TAU Component API)
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
      - OpenMP (directive rewriting (*Opari*), *POMP spec*)
  - Object code
    - pre-instrumented libraries (e.g., MPI using *PMPI*)
    - statically-linked and dynamically-linked
  - Executable code
    - dynamic instrumentation (pre-execution) (*DynInstAPI*)
    - virtual machine instrumentation (e.g., Java using *JVMPI*)
    - Python interpreter based instrumentation at runtime
  - Proxy Components

TAU Measurement Approach

- Portable and scalable parallel profiling solution
  - Multiple profiling types and options
  - Event selection and control (enabling/disabling, throttling)
  - Online profile access and sampling
  - Online performance profile overhead compensation
- Portable and scalable parallel tracing solution
  - Trace translation to Open Trace Format (OTF)
  - Trace streams and hierarchical trace merging
- Robust timing and hardware performance support
- Multiple counters (hardware, user-defined, system)
- Performance measurement for CCA component software

Using TAU

- Configuration
- Instrumentation
  - Manual
  - MPI – Wrapper interposition library
  - PDT- Source rewriting for C,C++, F77/90/95
  - Compiler-based instrumentation for C, C++, F90
  - OpenMP – Directive rewriting
  - Component based instrumentation – Proxy components
  - Binary Instrumentation
    - DyninstAPI – Runtime Instrumentation/Rewriting binary
    - Java – Runtime instrumentation
    - Python – Runtime instrumentation
- Measurement
- Performance Analysis

TAU Measurement System Configuration

- configure [OPTIONS]

{-c++=<CC>, -cc=<cc>}	Specify C++ and C compilers
-pdt=<dir>	Specify location of PDT
-opari=<dir>	Specify location of Opari OpenMP tool
-papi=<dir>	Specify location of PAPI
-vampirtrace=<dir>	Specify location of VampirTrace
-mpi[inc/lib]=<dir>	Specify MPI library instrumentation
-dyninst=<dir>	Specify location of DynInst Package
-shmem[inc/lib]=<dir>	Specify PSHMEM library instrumentation
-python[inc/lib]=<dir>	Specify Python instrumentation
-tag=<name>	Specify a unique configuration name
-epilog=<dir>	Specify location of EPILOG
-slog2	Build SLOG2/Jumpshot tracing package
-otf=<dir>	Specify location of OTF trace package
-arch=<architecture>	Specify architecture explicitly (bgl, xt3,x86_64,x86_64linux…)
{-pthread, -sproc}	Use pthread or SGI sproc threads
-openmp	Use OpenMP threads
-jdk=<dir>	Specify Java instrumentation (JDK)
-fortran=[vendor]	Specify Fortran compiler

TAU Measurement System Configuration

- configure [OPTIONS]

-TRACE	Generate binary TAU traces
-PROFILE (default)	Generate profiles (summary)
-PROFILECALLPATH	Generate call path profiles
-PROFILEPHASE	Generate phase based profiles
-PROFILEMEMORY	Track heap memory for each routine
-PROFILEHEADROOM	Track memory headroom to grow
-MULTIPLECOUNTERS	Use hardware counters + time
-COMPENSATE	Compensate timer overhead
-CPUTIME	Use usertime+system time
-PAPIWALLCLOCK	Use PAPI's wallclock time
-PAPIVIRTUAL	Use PAPI's process virtual time
-SGITIMERS	Use fast IRIX timers
-LINUXTIMERS	Use fast x86 Linux timers

TAU Measurement Configuration – Examples

- ./configure –pdt=/usr/global/tools/tau/training/pdtoolkit-3.16 -mpi
  Configure using PDT and MPI
- ./configure -papi=/usr/local/tools/papi
  -pdt=<dir> -mpi ; make clean install
  - Use PAPI counters (one or more) with C/C++/F90 automatic instrumentation. Also instrument the MPI library.
- Typically configure multiple measurement libraries using installtau
- Past configurations are stored in TAU's .all_configs file and .installflags
- Each configuration creates a unique <arch>/lib/Makefile.tau<options> stub makefile. It corresponds to the configuration options used. e.g.,
  - /usr/global/tools/tau/training/x86_64/lib/Makefile.tau-mpi-pdt
  - /usr/global/tools/tau/training/x86_64/lib/Makefile.tau-mpi-papi-pdt

TAU Measurement Configuration – Examples

```
% cd /usr/global/tools/tau/training/tau_latest/x86_64/lib; ls
   Makefile.*
Makefile.tau-pdt
Makefile.tau-mpi-pdt
Makefile.tau-pthread-pdt
Makefile.tau-papi-mpi-pdt
Makefile.tau-papi-pthread-pdt
Makefile.tau-mpi-papi-pdt
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-mpi-pdt-vampirtrace-trace
```

- For an MPI+F90 application, you may want to start with:

Makefile.tau-mpi-pdt
  - Supports MPI instrumentation & PDT for automatic source instrumentation
  - `% export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt`
  - `% tau_f90.sh matrix.f90 -o matrix`

Compile-Time Environment Variables

- Optional parameters for TAU_OPTIONS: [tau_compiler.sh –help]

-optVerbose	Turn on verbose debugging messages
-optCompInst	Use compiler based instrumentation
-optNoCompInst	Do not revert to compiler instrumentation if source instrumentation fails.
-optDetectMemoryLeaks	Turn on debugging memory allocations/ de-allocations to track leaks
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess Fortran sources before instrumentation
-optTauSelectFile=""	Specify selective instrumentation file for tau_instrumentor
-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)
-optPdtF95Reset=""	Reset options for Fortran parser in PDT (f95parse/gfparse)
-optPdtCOpts=""	Options for C parser in PDT (cparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtCxxOpts=""	Options for C++ parser in PDT (cxxparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

...

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_MEMORY_LEAKS	0	Setting to 1 turns on leak detection
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. 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_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 separted list generates other metrics. (e.g., TIME:linuxtimers:PAPI_FP_OPS:PAPI_NATIVE_<event>)

Configuration Parameters in Stub Makefiles

- Each TAU stub Makefile resides in <tau>/<arch>/lib directory
- Variables:

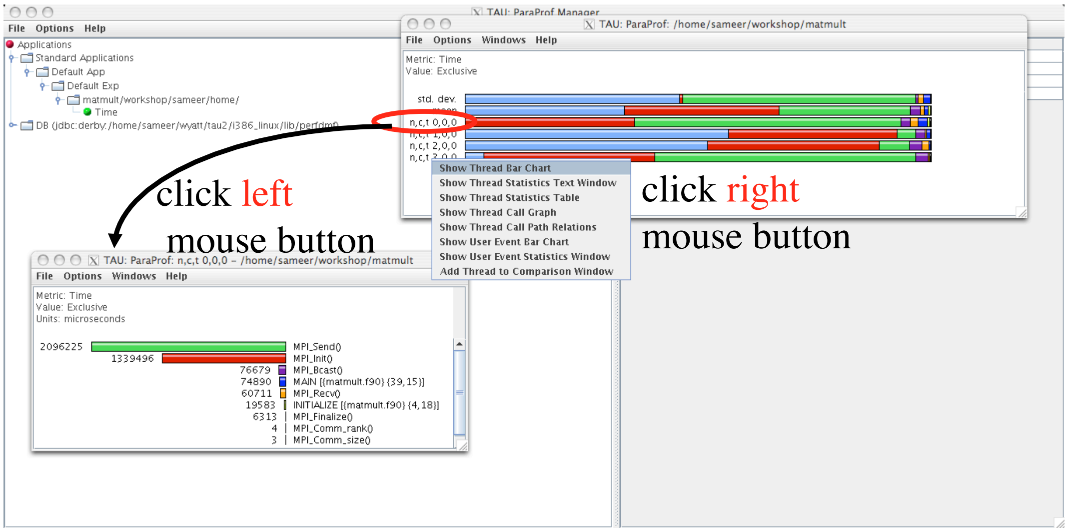
– TAU_CXX	Specify the C++ compiler used by TAU
– TAU_CC, TAU_F90	Specify the C, F90 compilers
– TAU_DEFS	Defines used by TAU. Add to CFLAGS
– TAU_LDFLAGS	Linker options. Add to LDFLAGS
– TAU_INCLUDE	Header files include path. Add to CFLAGS
– TAU_LIBS	Statically linked TAU library. Add to LIBS
– TAU_SHLIBS	Dynamically linked TAU library
– TAU_MPI_LIBS	TAU's MPI wrapper library for C/C++
– TAU_MPI_FLIBS	TAU's MPI wrapper library for F90
– TAU_FORTRANLIBS	Must be linked in with C++ linker for F90
– TAU_CXXLIBS	Must be linked in with F90 linker
– TAU_INCLUDE_MEMORY	Use TAU's malloc/free wrapper lib
– TAU_DISABLE	TAU's dummy F90 stub library
– TAU_COMPILER	Instrument using tau_compiler.sh script

- Each stub makefile encapsulates the parameters that TAU was configured with
- It represents a specific instance of the TAU libraries. TAU scripts use stub makefiles to identify what performance measurements are to be performed.
- Each configuration produces a shared library called libTAUsh-<options>.so

Using TAU

- Install TAU
  % configure [options]; make clean install
- Typically modify application makefile and choose TAU configuration
  – Select TAU's stub makefile, change name of compiler in Makefile
  % export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
  % export TAU_OPTIONS='-optVerbose -optKeepFiles ...'
  – F90 = tau_f90.sh CXX = tau_cxx.sh CC = tau_cc.sh
- Set environment variables
  – Directory where profiles/traces are to be stored/counter selection
- Execute application
  % jsub run.job
- Analyze performance data
  – paraprof, vampir, pprof, paraver …

ParaProf Main Window




```
% paraprof matmult.ppk
```


TAU's MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
  - Provides name shifted interface
    - MPI_Send = PMPI_Send
    - Weak bindings
- Interpose TAU's MPI wrapper library between MPI and TAU
  - -lmpi replaced by –lTauMpi –lpmpi –lmpi
- No change to the source code!
  - Just re-link the application to generate performance data
  - export TAU_MAKEFILE=<dir>/<arch>/lib/Makefile.tau-mpi -[options]
  - Use tau_cxx.sh, tau_f90.sh and tau_cc.sh as compilers

Runtime MPI Shared Library Instrumentation

- We can now interpose the MPI wrapper library for applications that have already been compiled
  - No re-compilation or re-linking necessary!
- Uses LD_PRELOAD for Linux
- On AIX, TAU uses MPI_EUILIB / MPI_EUILIBPATH
- Simply compile TAU with MPI support and prefix your MPI program with tauex

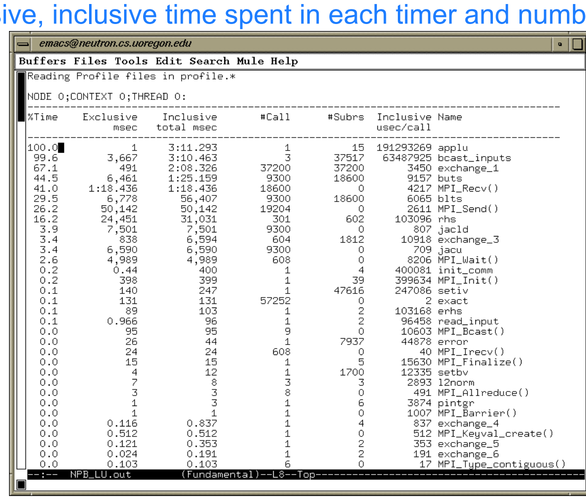
```
% mpirun -np 4 tauex a.out
```

- Requires shared library MPI - does not work on XT3
- Approach will work with other shared libraries

-PROFILE Configuration Option

- Generates flat profiles (one for each MPI process)
  - It is the default option.
- Uses wallclock time (gettimeofday() sys call)
- Calculates exclusive, inclusive time spent in each timer and number of calls

% pprof


emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1	3:11.293	1	15	191293269	applu
99.6	3,667	3:10.463	3	37517	63487925	bcast_inputs
67.1	491	2:08.326	37200	37200	3450	exchange_1
44.5	6,461	1:25.159	9300	18600	9157	buts
41.0	1:18.436	1:18.436	18600	0	4217	MPI_Recv()
29.5	6,778	56,407	9300	18600	6065	blts
26.2	50,142	50,142	19204	0	2611	MPI_Send()
16.2	24,451	31,031	301	602	103096	rhs
3.9	7,501	7,501	9300	0	807	jacld
3.4	838	6,594	604	1812	10918	exchange_3
3.4	6,590	6,590	9300	0	709	jacu
2.6	4,989	4,989	608	0	8206	MPI_Wait()
0.2	0.44	400	1	4	400081	init_comm
0.2	398	399	1	39	399634	MPI_Init()
0.1	140	247	1	47616	247086	setiv
0.1	131	131	57252	0	2	exact
0.1	89	103	1	2	103168	erhs
0.1	0.966	96	1	2	96458	read_input
0.0	95	95	9	0	10603	MPI_Bcast()
0.0	26	44	1	7937	44878	error
0.0	24	24	608	0	40	MPI_Irecv()
0.0	15	15	1	5	15630	MPI_Finalize()
0.0	4	12	1	1700	12335	setbv
0.0	7	8	3	3	2893	l2norm
0.0	3	3	8	0	491	MPI_Allreduce()
0.0	1	3	1	6	3874	pintgr
0.0	1	1	1	0	1007	MPI_Barrier()
0.0	0.116	0.837	1	4	837	exchange_4
0.0	0.512	0.512	1	0	512	MPI_Keyval_create()
0.0	0.121	0.353	1	2	353	exchange_5
0.0	0.024	0.191	1	2	191	exchange_6
0.0	0.103	0.103	6	0	17	MPI_Type_contiguous()

--:-- NPB_LU.out   (Fundamental)--L8--Top--

-PAPI Configuration Option

- Instead of one metric, profile or trace with more than one metric
  - % export TAU_METRICS=TIME:PAPI_L2_DCM:PAPI_FP_OPS...
- When used with –TRACE option, the first counter **must** be TIME
  - % export TAU_METRICS=TIME:...
  - Provides a globally synchronized real time clock for tracing
- -papi appears in the name of the stub Makefile
- Often used with –papi=<dir> to measure hardware performance counters and time
- papi_native_avail and papi_avail are two useful tools

Papi_avail

```
cfe1.sameer 66> ./papi_avail | more
Available events and hardware information.
-------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : Itanium 2 (1)
CPU Revision             : 5.000000
CPU Megahertz            : 1500.000000
CPU's in this Node       : 28
Nodes in this System     : 1
Total CPU's              : 28
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Name            Code            Avail   Deriv   Description (Note)
PAPI_L1_DCM     0x80000000      Yes     No      Level 1 data cache misses
PAPI_L1_ICM     0x80000001      Yes     No      Level 1 instruction cache misses
PAPI_L2_DCM     0x80000002      Yes     Yes     Level 2 data cache misses
PAPI_L2_ICM     0x80000003      Yes     No      Level 2 instruction cache misses
PAPI_L3_DCM     0x80000004      Yes     Yes     Level 3 data cache misses
PAPI_L3_ICM     0x80000005      Yes     No      Level 3 instruction cache misses
PAPI_L1_TCM     0x80000006      Yes     Yes     Level 1 cache misses
PAPI_L2_TCM     0x80000007      Yes     No      Level 2 cache misses

...
```

Papi_native_avail

```
cfe1.sameer 67> ./papi_native_avail | more
Available native events and hardware information.
-------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : Itanium 2 (1)
CPU Revision             : 5.000000
CPU Megahertz            : 1500.000000
CPU's in this Node       : 28
Nodes in this System     : 1
Total CPU's              : 28
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Symbol                           Event Code   Long Description
 Register Name[n]
 Register Value[n]
ALAT_CAPACITY_MISS_ALL           0x40000000   ALAT Entry Replaced -- both integer and floating point i
nstructions
ALAT_CAPACITY_MISS_FP            0x40000001   ALAT Entry Replaced -- only floating point instructions
ALAT_CAPACITY_MISS_INT           0x40000002   ALAT Entry Replaced -- only integer instructions
…
```

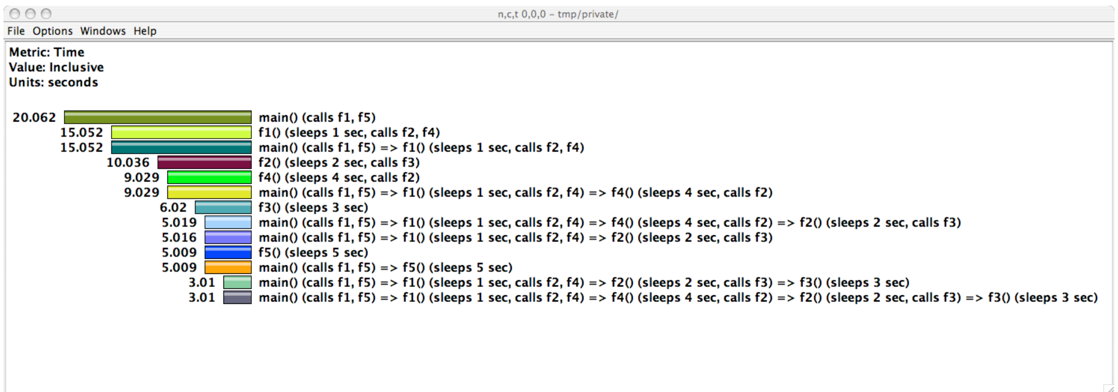
Papi_event_chooser on IA-64

```
cfe1.sameer 68> ./papi_event_chooser
Usage: eventChooser NATIVE|PRESET evt1 evet2 ...

cfe1.sameer 72> ./papi_event_chooser PRESET PAPI_FP_OPS PAPI_L1_DCM PAPI_TOT_CYC
Test case eventChooser: Available events which can be added with given events.
-------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : Itanium 2 (1)
CPU Revision             : 5.000000
CPU Megahertz            : 1500.000000
CPU's in this Node       : 28
Nodes in this System     : 1
Total CPU's              : 28
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------------------------------------
Name            Derived Description (Mgr. Note)
PAPI_L1_ICM     No      Level 1 instruction cache misses ()
PAPI_L2_ICM     No      Level 2 instruction cache misses ()
PAPI_L3_ICM     No      Level 3 instruction cache misses

…
```

PAPI_L1_ICM may be counted with these counters on IA64

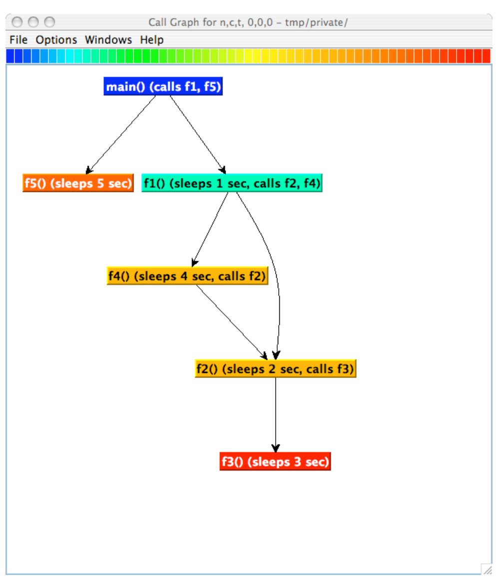
-PROFILECALLPATH Configuration Option

- Generates profiles that show the calling order (edges & nodes in callgraph)
  - A=>B=>C shows the time spent in C when it was called by B and B was called by A
  - Control the depth of callpath using TAU_CALLPATH_DEPTH env. Variable
  - -callpath in the name of the stub Makefile name
  - In TAU 2.18.2+, any executable can generate callpath profiles using
  - `% export TAU_CALLPATH=1`



-PROFILECALLPATH Configuration Option

- Generates program callgraph



Profile Measurement – Three Flavors

- Flat profiles
  - Time (or counts) spent in each routine (nodes in callgraph).
  - Exclusive/inclusive time, no. of calls, child calls
  - E.g,: MPI_Send, foo, …
- Callpath Profiles
  - Flat profiles, **plus**
  - Sequence of actions that led to poor performance
  - Time spent along a calling path (edges in callgraph)
  - E.g., "main=> f1 => f2 => MPI_Send" shows the time spent in MPI_Send when called by f2, when f2 is called by f1, when it is called by main. Depth of this callpath = 4 (TAU_CALLPATH_DEPTH environment variable)
- Phase based profiles
  - Flat profiles, **plus**
  - Flat profiles under a phase (nested phases are allowed)
  - Default "main" phase has all phases and routines invoked outside phases
  - Supports static or dynamic (per-iteration) phases
  - E.g., "IO => MPI_Send" is time spent in MPI_Send in IO phase

-DEPTHLIMIT Configuration Option

- Allows users to enable instrumentation at runtime based on the depth of a calling routine on a callstack.
  - Disables instrumentation in all routines a certain depth away from the root in a callgraph
- TAU_DEPTH_LIMIT environment variable specifies depth

  % export TAU_DEPTH_LIMIT=1
  enables instrumentation in only "main"
  % export TAU_DEPTH_LIMIT=2
  enables instrumentation in main and routines that are directly called by main

- Stub makefile has -depthlimit in its name:

  export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-depthlimit-pdt

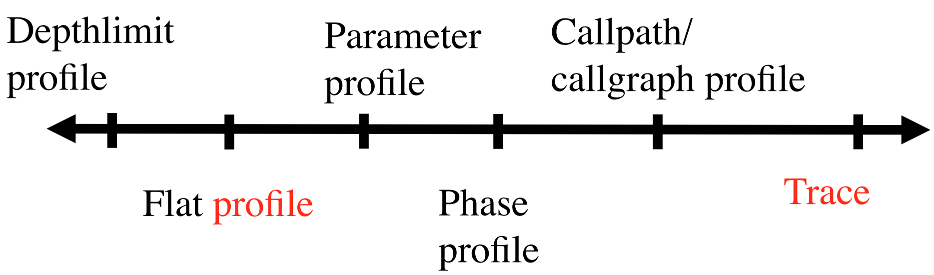
-COMPENSATE Configuration Option

- Specifies online compensation of performance perturbation
- TAU computes its timer overhead and subtracts it from the profiles
- Works well with time or instructions based metrics
- Does not work with level 1/2 data cache misses
- export TAU_COMPENSATE=1 (in TAU v2.18.2+)

-TRACE Configuration Option

- Generates event-trace logs, rather than summary profiles
- Traces show when and where an event occurred in terms of location and the process that executed it
- Traces from multiple processes are merged:
  % tau_treemerge.pl
  - generates tau.trc and tau.edf as merged trace and event definition file
- TAU traces can be converted to Vampir's OTF/VTF3, Jumpshot SLOG2, Paraver trace formats:
  % tau2otf tau.trc tau.edf app.otf
  % tau2vtf tau.trc tau.edf app.vpt.gz
  % tau2slog2 tau.trc tau.edf -o app.slog2
  % tau_convert -paraver tau.trc tau.edf app.prv
- Stub Makefile has -trace in its name
  % export TAU_MAKEFILE=<taudir>/<arch>/lib/
  Makefile.tau-mpi-pdt-trace-pdt

Performance Evaluation Alternatives



Each alternative has:
- one metric/counter
- multiple counters

Volume of performance data →

-PROFILEPARAM Configuration Option

- Idea: partition performance data for individual functions based on runtime parameters
- Enable by configuring with –PROFILEPARAM
- TAU call: TAU_PROFILE_PARAM1L (value, “name”)
- Simple example:

```
void foo(long input) {
   TAU_PROFILE("foo", "", TAU_DEFAULT);
   TAU_PROFILE_PARAM1L(input, "input");
   ... }
```

Workload Characterization

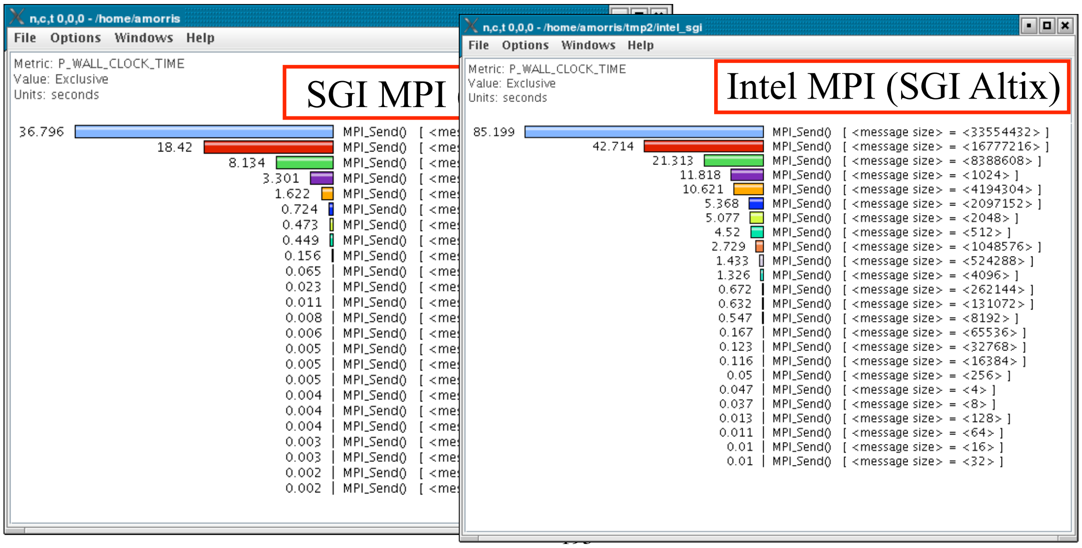
- 5 seconds spent in function "foo" becomes
  - 2 seconds for "foo [<input> = <25>]"
  - 1 seconds for "foo [<input> = <5>]"
  - …
- Currently used in MPI wrapper library
  - Allows for partitioning of time spent in MPI routines based on parameters (message size, message tag, destination node)
  - Can be extrapolated to infer specifics about the MPI subsystem and system as a whole

Workload Characterization

```
#include <stdio.h>
#include <mpi.h>
int buffer[8*1024*1024];

int main(int argc, char **argv) {
  int rank, size, i, j;
  MPI_Init(&argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  for (i=0;i<1000;i++)
    for (j=1;j<=8*1024*1024;j*=2) {
      if (rank == 0) {
        MPI_Send(buffer,j,MPI_INT,1,42,MPI_COMM_WORLD);
      } else {
        MPI_Status status;
        MPI_Recv(buffer,j,MPI_INT,0,42,MPI_COMM_WORLD,&status);
      }
    }
  MPI_Finalize();
}
```

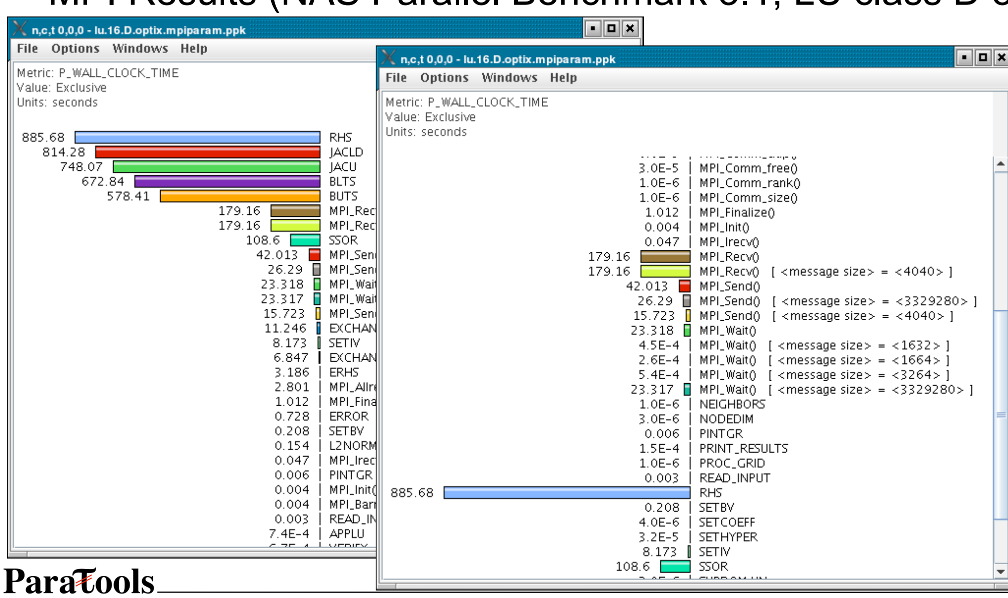
Workload Characterization

```
% icc mpi.c -lmpi
% mpirun -np 2 tauex a.out
```



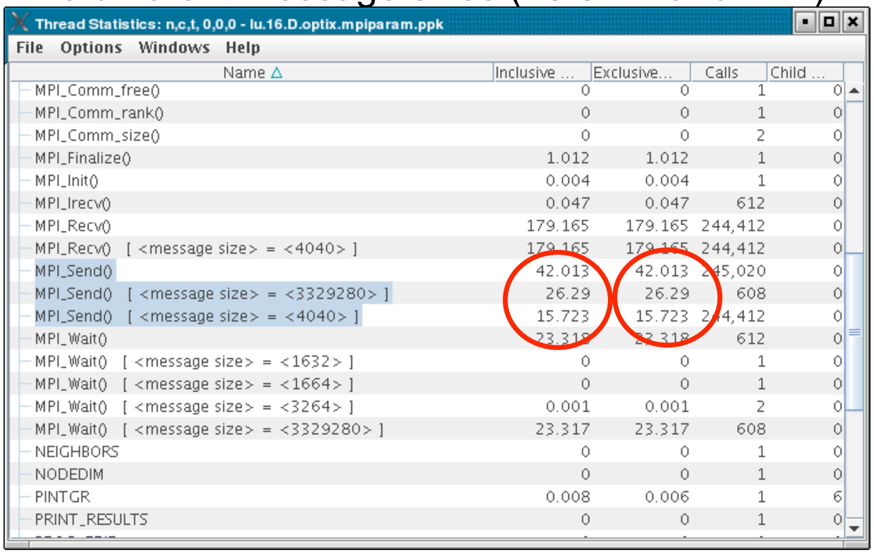
Workload Characterization

- MPI Results (NAS Parallel Benchmark 3.1, LU class D on



Workload Characterization

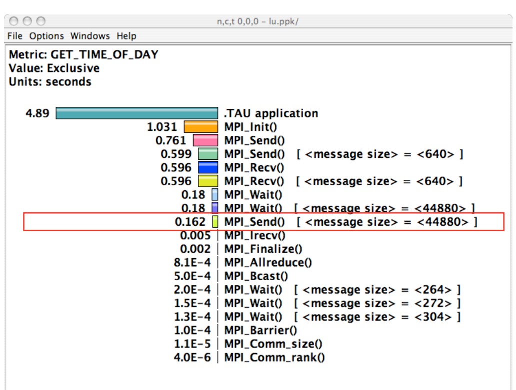
- Two different message sizes (~3.3MB and ~4K)

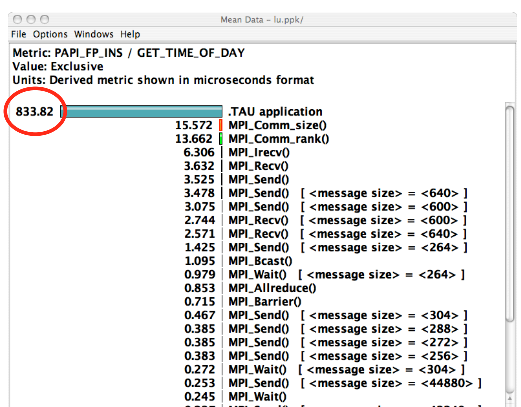


Thread Statistics: n,c,t, 0,0,0 - lu.16.D.optix.mpiparam.ppk

Name	Inclusive ...	Exclusive...	Calls	Child ...
MPI_Comm_free()	0	0	1	0
MPI_Comm_rank()	0	0	1	0
MPI_Comm_size()	0	0	2	0
MPI_Finalize()	1.012	1.012	1	0
MPI_Init()	0.004	0.004	1	0
MPI_Irecv()	0.047	0.047	612	0
MPI_Recv()	179.165	179.165	244,412	0
MPI_Recv() [<message size> = <4040>]	179.165	179.165	244,412	0
MPI_Send()	42.013	42.013	245,020	0
MPI_Send() [<message size> = <3329280>]	26.29	26.29	608	0
MPI_Send() [<message size> = <4040>]	15.723	15.723	244,412	0
MPI_Wait()	23.318	23.318	612	0
MPI_Wait() [<message size> = <1632>]	0	0	1	0
MPI_Wait() [<message size> = <1664>]	0	0	1	0
MPI_Wait() [<message size> = <3264>]	0.001	0.001	2	0
MPI_Wait() [<message size> = <3329280>]	23.317	23.317	608	0
NEIGHBORS	0	0	1	0
NODEDIM	0	0	1	0
PINTGR	0.008	0.006	1	6
PRINT_RESULTS	0	0	1	0

Job Tracking: ParaProf profile browser





LU spent 0.162 seconds sending messages of size 44880

It got 833.82 Mflops!

Memory Profiling in TAU

- Configuration option –PROFILEMEMORY
  - Records global heap memory utilization for each function
  - Takes one sample at beginning of each function and associates the sample with function name
- Configuration option -PROFILEHEADROOM
  - Records headroom (amount of free memory to grow) for each function
  - Takes one sample at beginning of each function and associates it with the callstack [TAU_CALLPATH_DEPTH env variable]
  - Useful for debugging memory usage on IBM BG/L.
- Independent of instrumentation/measurement options selected
- No need to insert macros/calls in the source code
- User defined atomic events appear in profiles/traces

Memory Profiling in TAU (Atomic events)

Sorted By: number of userEvents

NumSamples	Max	Min	Mean	Std. Dev	Name
252032	2022.7	1181.2	1534.3	410.04	MODULEHYDRO_1D::HYDRO_1D - Heap Memory (KB)
252032	2022.8	1181.7	1534.3	410.04	MODULEINTRFC::INTRFC - Heap Memory (KB)
104559	2023.2	331.13	1526.6	409.54	MODULEEOS3D::EOS3D - Heap Memory (KB)
63008	2022.7	1182	1534.3	410.01	MODULEUPDATE_SOLN::UPDATE_SOLN - Heap Memory (KB)
55545	2023.3	333.07	1514.2	408.31	DBASETREE::DBASENEIGHBORBLOCKLIST - Heap Memory (KB)
51374	2023	1179.4	1497.7	402.53	AMR_PROLONG_GEN_UNK_FUN - Heap Memory (KB)
42120	2022.7	1187.5	1533.5	409.83	ABUNDANCE_RESTRICT - Heap Memory (KB)
41958	2023	346.12	1514.9	408.39	AMR_RESTRICT_UNK_FUN - Heap Memory (KB)
31832	2022.8	1187.4	1534.1	409.91	AMR_RESTRICT_RED - Heap Memory (KB)
31504	2022.7	1181.8	1534.3	410.04	DIFFUSE - Heap Memory (KB)
26042	2023	1179.2	1501.9	403.61	AMR_PROLONG_UNK_FUN - Heap Memory (KB)

Flash2 code profile (-PROFILEMEMORY) on IBM BlueGene/L [MPI rank 0]

Memory Profiling in TAU

- Instrumentation based observation of global heap memory (not per function)
  - call TAU_TRACK_MEMORY()
  - call TAU_TRACK_MEMORY_HEADROOM()
    - Triggers one sample every 10 secs
  - call TAU_TRACK_MEMORY_HERE()
  - call TAU_TRACK_MEMORY_HEADROOM_HERE()
    - Triggers sample at a specific location in source code
  - call TAU_SET_INTERRUPT_INTERVAL(seconds)
    - To set inter-interrupt interval for sampling
  - call TAU_DISABLE_TRACKING_MEMORY()
  - call TAU_DISABLE_TRACKING_MEMORY_HEADROOM()
    - To turn off recording memory utilization
  - call TAU_ENABLE_TRACKING_MEMORY()
  - call TAU_ENABLE_TRACKING_MEMORY_HEADROOM()
    - To re-enable tracking memory utilization

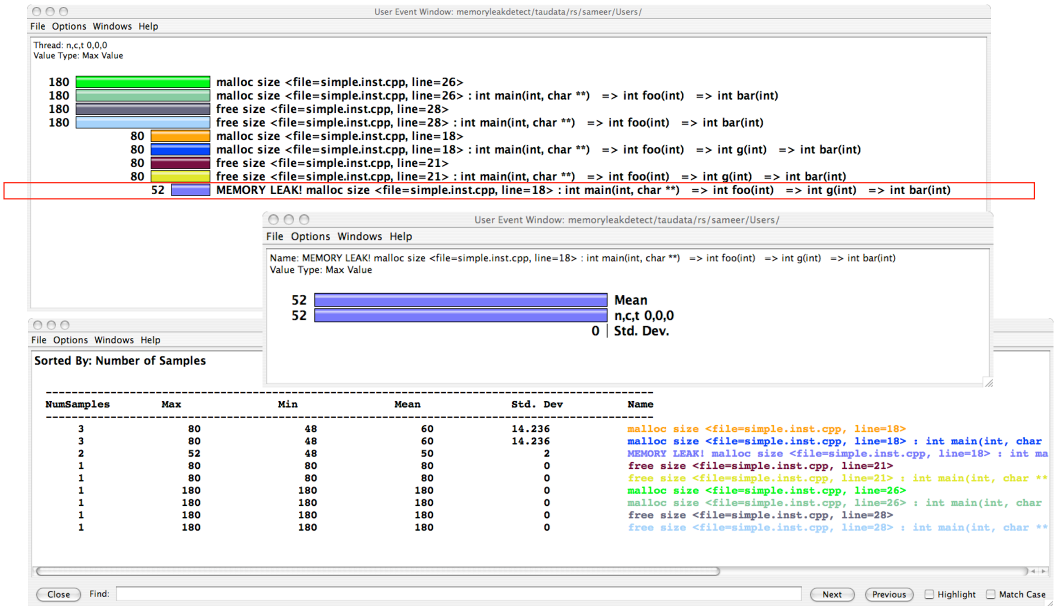
Detecting Memory Leaks in C/C++

- TAU wrapper library for malloc/realloc/free
- During instrumentation, specify
  -optDetectMemoryLeaks option to TAU_COMPILER

```
% export TAU_OPTIONS='-optVerbose -optDetectMemoryLeaks'
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt...
% tau_cxx.sh foo.cpp ...
```

- Tracks each memory allocation/de-allocation in parsed files
- Correlates each memory event with the executing callstack
- At the end of execution, TAU detects memory leaks
- TAU reports leaks based on allocations and the executing callstack
- Set **TAU_CALLPATH_DEPTH** environment variable to limit callpath data
  - default is 2
- Future work
  - Support for C++ new/delete planned
  - Support for Fortran 90/95 allocate/deallocate planned

Memory Leak Detection



Detecting Memory Leaks in Fortran

```
subroutine foo(x)
integer:: x
integer, allocatable :: A(:), B(:), C(:)

print *, "inside foo"
allocate(A(x), B(x), C(x))
deallocate(A, C)
print *, "exiting foo"

end subroutine foo

program main
call foo(5)
end program main
```

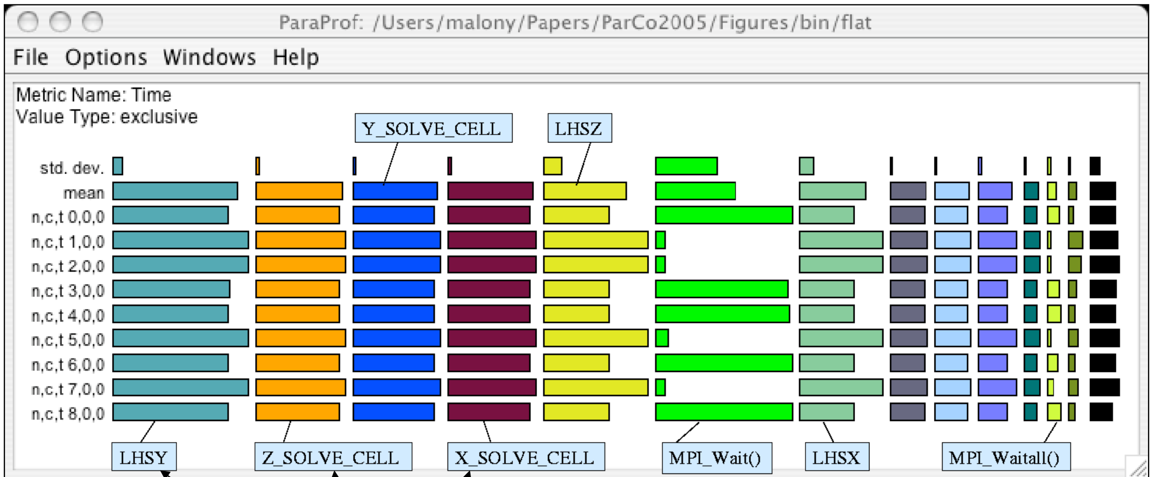
Detecting Memory Leaks in Fortran

```
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples  MaxValue  MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
  1   5   5   5   0  MEMORY LEAK! malloc size <file=simple.f, variable=B, line=6> : MAIN => FOO
  1   5   5   5   0  free size <file=simple.f, variable=A, line=7>
  1   5   5   5   0  free size <file=simple.f, variable=A, line=7> : MAIN => FOO
  1   5   5   5   0  free size <file=simple.f, variable=C, line=7>
  1   5   5   5   0  free size <file=simple.f, variable=C, line=7> : MAIN => FOO
  1   5   5   5   0  malloc size <file=simple.f, variable=A, line=6>
  1   5   5   5   0  malloc size <file=simple.f, variable=A, line=6> : MAIN => FOO
  1   5   5   5   0  malloc size <file=simple.f, variable=B, line=6>
  1   5   5   5   0  malloc size <file=simple.f, variable=B, line=6> : MAIN => FOO
  1   5   5   5   0  malloc size <file=simple.f, variable=C, line=6>
  1   5   5   5   0  malloc size <file=simple.f, variable=C, line=6> : MAIN => FOO
---------------------------------------------------------------------------------------
```


tau_exec

- Uninstrumented execution
  - % mpirun –np 256 ./a.out
- Track MPI Performance
  - % mpirun –np 256 tau_exec ./a.out
- Track I/O Performance (MPI enabled by default)
  - % mpirun –np 256 tau_exec –io ./a.out
- Track Memory
  - % setenv TAU_TRACK_MEMORY_LEAKS 1
  - % mpirun –np 256 tau_exec –memory ./a.out
- Track I/O and Memory
  - % mpirun –np 256 tau_exec –io –memory ./a.out

Phase Profiling (NAS BT, Flat Profile)



Application routine names reflect phase semantics

How is MPI_Wait() distributed relative to solver direction?

NAS BT – Phase Profile (Main and X, Y, Z)

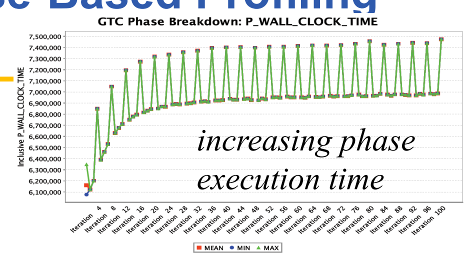
Main phase shows nested phases and immediate events

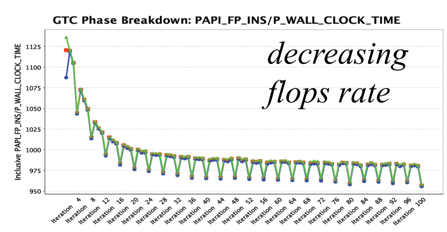
TAU Timers and Phases

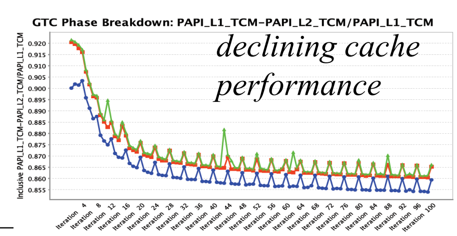
- Static timer
  - Shows time spent in all invocations of a routine (foo)
  - E.g., “foo()” 100 secs, 100 calls
- Dynamic timer
  - Shows time spent in each invocation of a routine
  - E.g., “foo() 3” 4.5 secs, “foo 10” 2 secs (invocations 3 and 10 respectively)
- Static phase
  - Shows time spent in all routines called (directly/indirectly) by a given routine (foo)
  - E.g., “foo() => MPI_Send()” 100 secs, 10 calls shows that a total of 100 secs were spent in MPI_Send() when it was called by foo.
- Dynamic phase
  - Shows time spent in all routines called by a given invocation of a routine.
  - E.g., “foo() 4 => MPI_Send()” 12 secs, shows that 12 secs were spent in MPI_Send when it was called by the 4th invocation of foo.

Performance Dynamics: Phase-Based Profiling

- Profile phases capture performance with respect to application-defined ‘phases’ of execution
  - Separate full profile produce for each phase
- GTC particle-in-cell simulation of fusion turbulence
- Phases assigned to iterations
- Data change affects cache







TAU's MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
  - Provides name shifted interface
    - MPI_Send = PMPI_Send
    - Weak bindings
- Interpose TAU's MPI wrapper library between MPI and TAU
  - -lmpi replaced by –lTauMpi –lpmpi –lmpi
- No change to the source code! Just re-link the application to generate performance data
  - `export TAU_MAKEFILE=<dir>/<arch>/lib/Makefile.tau-mpi-[options]`
  - Use tau_cxx.sh, tau_f90.sh and tau_cc.sh as compilers

Using TAU

- Install TAU
  - Configuration
  - Measurement library creation
- Instrument application
  - Manual or automatic source instrumentation
  - Instrumented library (e.g., MPI – wrapper interposition library)
  - Binary instrumentation
- Create performance experiments
  - Integrate with application build environment
  - Set experiment variables
- Execute application
- Analyze performance

Integration with Application Build Environment

- Try to minimize impact on user's application build procedures
- Handle process of parsing, instrumentation, compilation, linking
- Dealing with Makefiles
  - Minimal change to application Makefile
  - Avoid changing compilation rules in application Makefile
  - No explicit inclusion of rules for process stages
- Some applications do not use Makefiles
  - Facilitate integration in whatever procedures used
- Two techniques:
  - TAU shell scripts (tau_<compiler>.sh)
    - Invokes all PDT parser, TAU instrumenter, and compiler
  - TAU_COMPILER

Using Program Database Toolkit (PDT)

1. **Parse the Program to create foo.pdb:**

```
% cxxparse foo.cpp -I/usr/local/mydir -DMYFLAGS …
```

or

```
% cparse foo.c -I/usr/local/mydir -DMYFLAGS …
```

or

```
% f95parse foo.f90 -I/usr/local/mydir …
% f95parse *.f -omerged.pdb -I/usr/local/mydir -R free
```

2. **Instrument the program:**

```
% tau_instrumentor foo.pdb   foo.f90 -o foo.inst.f90
    -f select.tau
```

3. **Compile the instrumented program:**

```
% ifort foo.inst.f90 -c -I/usr/local/mpi/include -o foo.o
```

Tau_[cxx,cc,f90].sh – Improves Integration in Makefiles

```
# set TAU_MAKEFILE and TAU_OPTIONS env vars
CC = tau_cc.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
    $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.c.o:
    $(CC) $(CFLAGS) -c $<
.f90.o:
    $(F90) $(FFLAGS) -c $<
```


Automatic Instrumentation

- We now provide compiler wrapper scripts
  - Simply replace `mpif90` with `tau_f90.sh`
  - Automatically instruments Fortran source code, links with TAU MPI Wrapper libraries.
- Use `tau_cc.sh` and `tau_cxx.sh` for C/C++

Before

```
CXX = mpicc
F90 = mpif90
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
    $(LIBS)
.cpp.o:
    $(CC) $(CFLAGS) -c $<
```

After

```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
    $(LIBS)
.cpp.o:
    $(CC) $(CFLAGS) -c $<
```

TAU_COMPILER Commandline Options

- See `<taudir>/<arch>/bin/tau_compiler.sh –help`
- Compilation:

```
% mpxlf90 -c foo.f90
```

Changes to

```
% f95parse foo.f90 $(OPT1)
% tau_instrumentor foo.pdb foo.f90 –o foo.inst.f90 $(OPT2)
% ftn –c foo.f90 $(OPT3)
```

- Linking:

```
% ftn foo.o bar.o –o app
```

Changes to

```
% ftn foo.o bar.o –o app $(OPT4)
```

- Where options OPT[1-4] default values may be overridden by the user:

```
F90 = tau_f90.sh
```


Compile-Time Environment Variables

- Optional parameters for TAU_OPTIONS: [tau_compiler.sh –help]

-optVerbose	Turn on verbose debugging messages
-optCompInst	Use compiler based instrumentation
-optNoCompInst	Do not revert to compiler instrumentation if source instrumentation fails.
-optDetectMemoryLeaks	Turn on debugging memory allocations/ de-allocations to track leaks
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess Fortran sources before instrumentation
-optTauSelectFile=""	Specify selective instrumentation file for tau_instrumentor
-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)
-optPdtF95Reset=""	Reset options for Fortran parser in PDT (f95parse/gfparse)
-optPdtCOpts=""	Options for C parser in PDT (cparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtCxxOpts=""	Options for C++ parser in PDT (cxxparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

...

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_MEMORY_LEAKS	0	Setting to 1 turns on leak detection
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. 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_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 separted list generates other metrics. (e.g., TIME:linuxtimers:PAPI_FP_OPS:PAPI_NATIVE_<event>)

Compiling Fortran Codes with TAU

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:
  % export TAU_OPTIONS=‘-optPdtF95Opts=“-R free” -optVerbose ’

- To use the compiler based instrumentation instead of PDT (source-based):
  % export TAU_OPTIONS=‘-optCompInst -optVerbose’

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):
  % export TAU_OPTIONS=‘-optPreProcess -optVerbose -optDetectMemoryLeaks’

- To use an instrumentation specification file:
  % export TAU_OPTIONS=‘-optTauSelectFile=mycmd.tau -optVerbose -optPreProcess’
  % cat mycmd.tau
  BEGIN_INSTRUMENT_SECTION
  memory file=“foo.f90” routine=“#”
  # instruments all allocate/deallocate statements in all routines in foo.f90
  loops file=“*” routine=“#”
  io file=“abc.f90” routine=“FOO”
  END_INSTRUMENT_SECTION

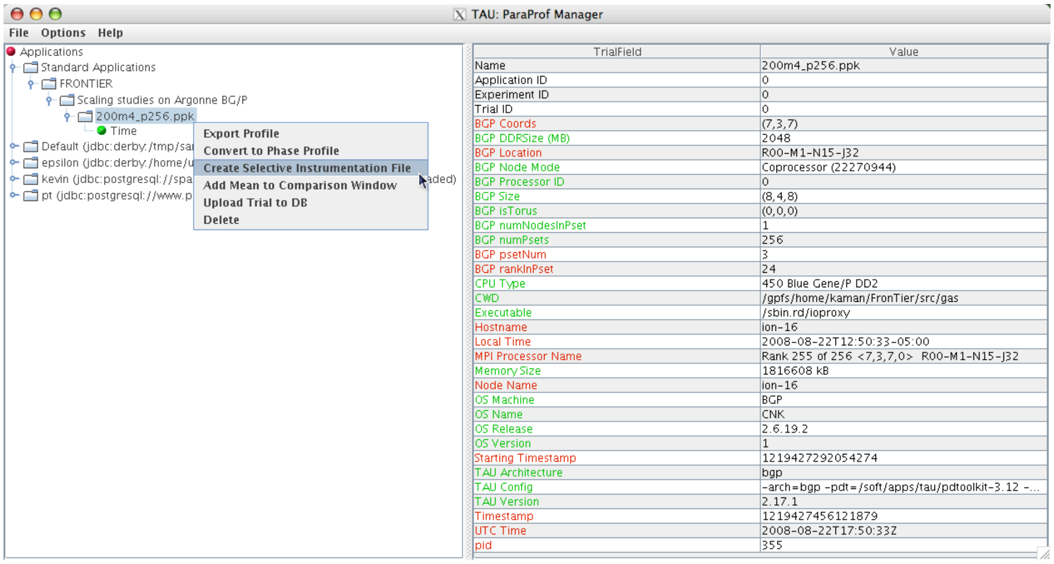
Overriding Default Options:TAU_COMPILER

```
% cat Makefile
F90 = tau_f90.sh
OBJS = f1.o f2.o f3.o …
LIBS = -Lappdir -lapplib1 -lapplib2 …

app: $(OBJS)
     $(F90) $(OBJS) -o app $(LIBS)
.f90.o:
     $(F90) -c $<
% export TAU_OPTIONS='-optVerbose
       -optTauSelectFile=select.tau -optKeepFiles'
% export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-mpi-pdt
```

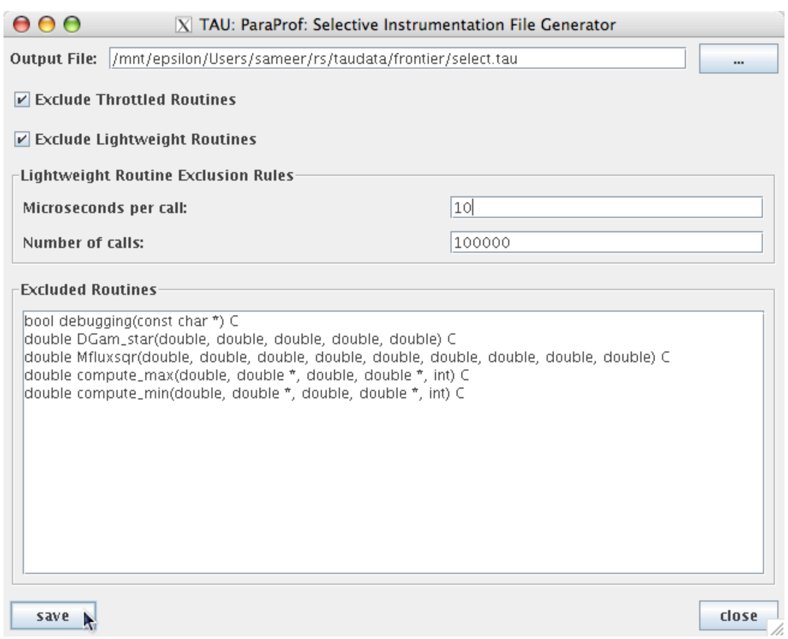

Optimization of Program Instrumentation

- Need to eliminate instrumentation in frequently executing lightweight routines
- Throttling of events at runtime (default in tau-2.17.2+):
  `% export TAU_THROTTLE=1`
  Turns off instrumentation in routines that execute over 100000 times (TAU_THROTTLE_NUMCALLS) and take less than 10 microseconds of inclusive time per call (TAU_THROTTLE_PERCALL). Use TAU_THROTTLE=0 to disable.
- Selective instrumentation file to filter events
  `% tau_instrumentor [options] -f <file>` ***OR***
  `% export TAU_OPTIONS='-optTauSelectFile=tau.txt'`
- Compensation of local instrumentation overhead
  `% configure -COMPENSATE`
  `or`
  `% export TAU_COMPENSATE=1   (in tau-2.19.2+)`

ParaProf: Creating Selective Instrumentation File



Choosing Rules for Excluding Routines



Selective Instrumentation File

- Specify a list of routines to exclude or include (case sensitive)
- # is a wildcard in a routine name. It cannot appear in the first column.

```
BEGIN_EXCLUDE_LIST
Foo
Bar
D#EMM
END_EXCLUDE_LIST
```

- Specify a list of routines to include for instrumentation

```
BEGIN_INCLUDE_LIST
int main(int, char **)
F1
F3
END_INCLUDE_LIST
```

- Specify either an include list or an exclude list!

Selective Instrumentation File

- Optionally specify a list of files to exclude or include (case sensitive)
- * and ? may be used as wildcard characters in a file name

```
BEGIN_FILE_EXCLUDE_LIST
f*.f90
Foo?.cpp
END_FILE_EXCLUDE_LIST
```

- Specify a list of routines to include for instrumentation

```
BEGIN_FILE_INCLUDE_LIST
main.cpp
foo.f90
END_FILE_INCLUDE_LIST
```

Selective Instrumentation File

- User instrumentation commands are placed in INSTRUMENT section
- ? and * used as wildcard characters for file name, # for routine name
- \ as escape character for quotes
- Routine entry/exit, arbitrary code insertion
- Outer-loop level instrumentation

```
BEGIN_INSTRUMENT_SECTION
loops file=“foo.f90” routine=“matrix#”
memory file=“foo.f90” routine=“#”
io routine=“matrix#”
[static/dynamic] phase routine=“MULTIPLY”
dynamic [phase/timer] name=“foo” file=“foo.cpp” line=22 to line=35
file=“foo.f90” line = 123 code = "  print *, \" Inside foo\""
exit routine = “int foo()” code = "cout <<\"exiting foo\"<<endl;"
END_INSTRUMENT_SECTION
```


Instrumentation Specification

```
% tau_instrumentor
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
For selective instrumentation, use –f option
% tau_instrumentor foo.pdb foo.cpp –o foo.inst.cpp –f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST
# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```

Instrumentation of OpenMP Constructs

- **O**penMP **P**ragma **A**nd **R**egion **I**nstrumentor [UTK, FZJ]
- Source-to-Source translator to insert POMP calls around OpenMP constructs and API functions
- Done: Supports
  - Fortran77 and Fortran90, OpenMP 2.0
  - C and C++, OpenMP 1.0
  - POMP Extensions
  - EPILOG and TAU POMP implementations
  - Preserves source code information (`#line` ***line file***)
- tau_ompcheck
  - Balances OpenMP constructs (DO/END DO) and detects errors
  - Invoked by tau_compiler.sh prior to invoking Opari
- KOJAK Project website http://icl.cs.utk.edu/kojak

OpenMP API Instrumentation

- Transform
  - `omp_#_lock()` → `pomp_#_lock()`
  - `omp_#_nest_lock()` → `pomp_#_nest_lock()`

  `[# = init|destroy|set|unset|test]`

- POMP version
  - Calls omp version internally
  - Can do extra stuff before and after call

Example: !$OMP PARALLEL DO Instrumentation

```
call pomp_parallel_fork(d)
!$OMP PARALLEL other-clauses...
      call pomp_parallel_begin(d)
      call pomp_do_enter(d)
      !$OMP DO schedule-clauses, ordered-clauses,
               lastprivate-clauses
            do loop
      !$OMP END DO NOWAIT
      call pomp_barrier_enter(d)
      !$OMP BARRIER
      call pomp_barrier_exit(d)
      call pomp_do_exit(d)
      call pomp_parallel_end(d)
!$OMP END PARALLEL DO
call pomp_parallel_join(d)
```


Opari Instrumentation: Example

```
pomp_for_enter(&omp_rd_2);
#line 252 "stommel.c"
#pragma omp for schedule(static) reduction(+: diff) private(j)
  firstprivate (a1,a2,a3,a4,a5) nowait
for( i=i1;i<=i2;i++) {
  for(j=j1;j<=j2;j++){
    new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1]
      + a4*psi[i][j-1] - a5*the_for[i][j];
    diff=diff+fabs(new_psi[i][j]-psi[i][j]);
  }
}
pomp_barrier_enter(&omp_rd_2);
#pragma omp barrier
pomp_barrier_exit(&omp_rd_2);
pomp_for_exit(&omp_rd_2);
```

Using Opari with TAU

Configure TAU with Opari (used here with MPI and PDT)

```
% configure -opari -arch=x86_64 -mpi  -pdt=/usr/contrib/TAU/pdtoolkit-3.15
% make clean; make install
% export TAU_MAKEFILE=/tau/<arch>/lib/Makefile.tau-…opari-…
% tau_cxx.sh -c foo.cpp
% tau_cxx.sh -c bar.f90
% tau_cxx.sh *.o -o app
```


Dynamic Instrumentation

- TAU uses DyninstAPI for runtime code patching
- Developed by U. Wisconsin and U. Maryland
- http://www.dyninst.org
- ***tau_run*** (mutator) loads measurement library
- Instruments mutatee
- MPI issues:
  - one mutator per executable image [TAU, DynaProf]
  - one mutator for several executables [Paradyn, DPCL]

Using DyninstAPI with TAU

Step I: Install DyninstAPI[Download from http://www.dyninst.org]

```
% cd dyninstAPI-6/core; make
```

Set DyninstAPI environment variables (including LD_LIBRARY_PATH)

Step II: Configure TAU with Dyninst

```
% configure -dyninst=/usr/local/dyninstAPI-6
% make clean; make install
```

Builds <taudir>/<arch>/bin/tau_run

```
% tau_run [<-o outfile>] [-Xrun<libname>][-f <select_inst_file>] [-v] <infile>
% tau_run -o a.inst.out a.out
```

Rewrites a.out

```
% tau_run klargest
```

Instruments klargest with TAU calls and executes it

```
% tau_run -XrunTAUsh-papi a.out
```

Loads libTAUsh-papi.so instead of libTAU.so for measurements

Virtual Machine Performance Instrumentation

- Integrate performance system with VM
  - Captures robust performance data (e.g., thread events)
  - Maintain features of environment
    - portability, concurrency, extensibility, interoperation
  - Allow use in optimization methods
- JVM Profiling Interface (JVMPI)
  - Generation of JVM events and hooks into JVM
  - Profiler agent (TAU) loaded as shared object
    - registers events of interest and address of callback routine
  - Access to information on dynamically loaded classes
  - No need to modify Java source, bytecode, or JVM

Using TAU with Java Applications

Step I: Sun JDK 1.4+ [download from www.javasoft.com]

Step II: Configure TAU with JDK (v 1.2 or better)

```
% configure -jdk=/usr/java2 -TRACE -PROFILE
% make clean; make install
```

Builds <taudir>/<arch>/lib/libTAU.so

For Java (without instrumentation):

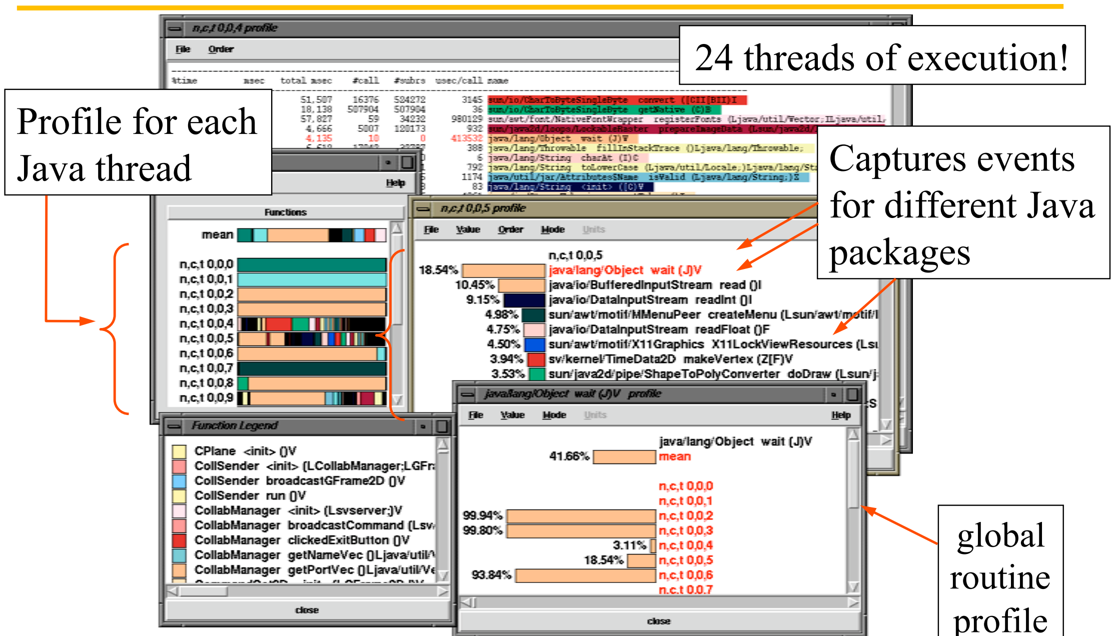
```
% java application
```

With instrumentation:

```
% java -XrunTAU application
% java -XrunTAU:exclude=sun/io,java application
```

Excludes sun/io/* and java/* classes

TAU Profiling of Java Application (SciVis)



Using TAU with Python Applications

Step I: Configure TAU with Python

```
% configure -pythoninc=/usr/include/python2.5/include
% make clean; make install
```

Builds <taudir>/<arch>/lib/<bindings>/pytau.py and tau.py packages for manual and automatic instrumentation respectively

```
% export PYTHONPATH= $PYTHONPATH:<taudir>/<arch>/lib/[<dir>]
```

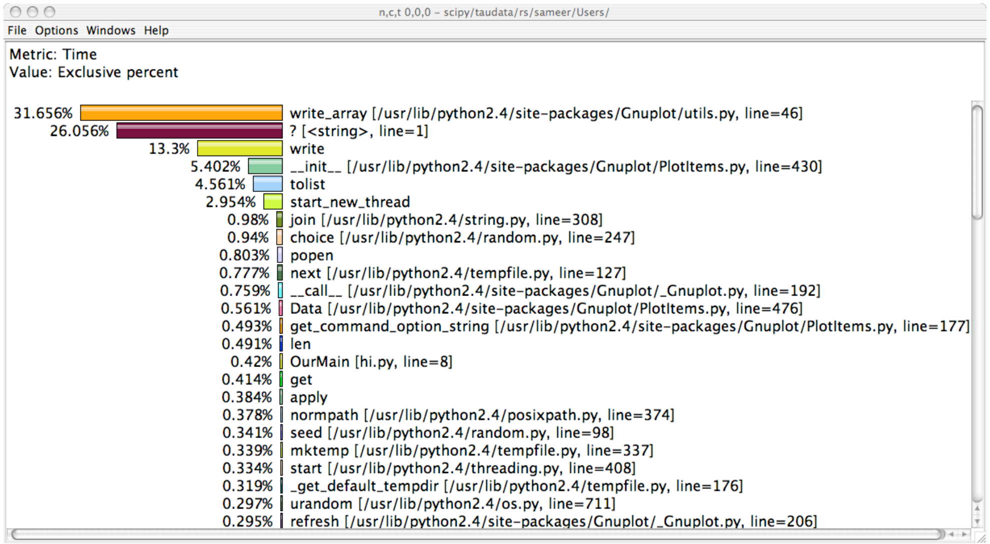

Python Automatic Instrumentation Example

```
#!/usr/bin/env/python

import tau
from time import sleep

def f2():
    print " In f2: Sleeping for 2 seconds "
    sleep(2)
def f1():
    print " In f1: Sleeping for 3 seconds "
    sleep(3)

def OurMain():
    f1()
tau.run('OurMain()')
```

Running:

```
% export PYTHONPATH= <tau>/<arch>/lib/bindings-python
% ./auto.py
Instruments OurMain, f1, f2, print…
```

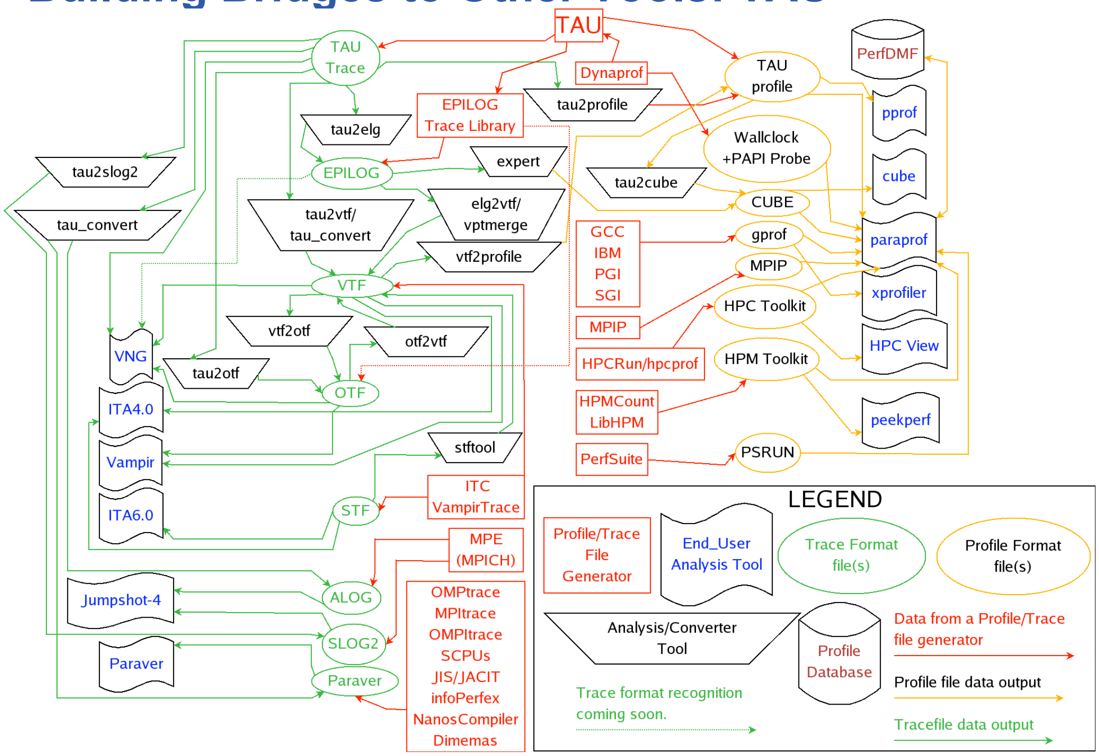
Python Instrumentation: SciPy



Performance Analysis

- paraprof profile browser (GUI)
- pprof (text based profile browser)
- TAU traces can be exported to many different tools
  - Vampir/VNG [T.U. Dresden] (formerly Intel (R) Trace Analyzer)
  - EXPERT [FZJ]
  - Jumpshot (bundled with TAU) [Argonne National Lab] ...

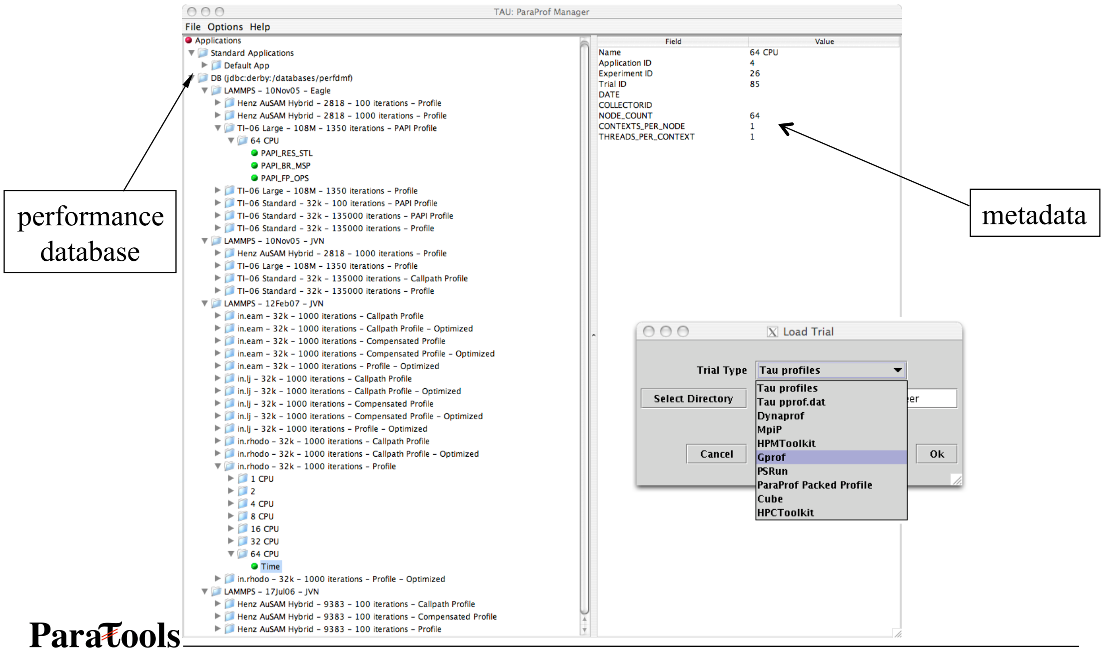
Building Bridges to Other Tools: TAU



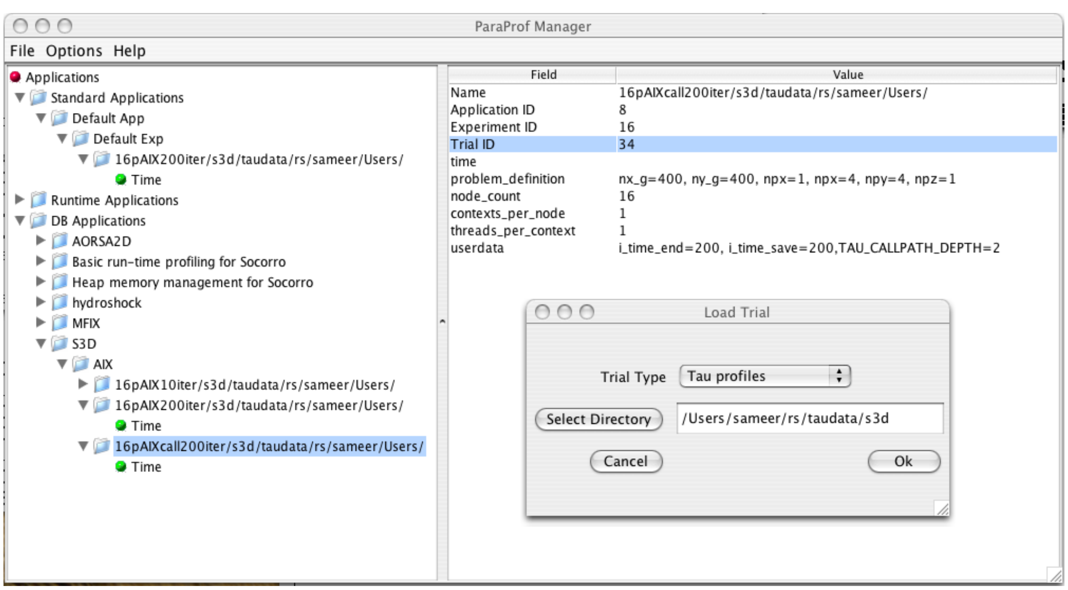
TAU Performance System Interfaces

- PDT [U. Oregon, LANL, FZJ] for instrumentation of C++, C99, F95 source code
- PAPI [UTK] for accessing hardware performance counters data
- DyninstAPI [U. Maryland, U. Wisconsin] for runtime instrumentation
- KOJAK [FZJ, UTK]
  - Epilog trace generation library
  - CUBE callgraph visualizer
  - Opari OpenMP directive rewriting tool
- Vampir/VNG Trace Analyzer [TU Dresden]
- VTF3/OTF trace generation library [TU Dresden] (available from TAU website)
- Paraver trace visualizer [CEPBA]
- Jumpshot-4 trace visualizer [MPICH, ANL]
- JVMPI from JDK for Java program instrumentation [Sun]
- Paraprof profile browser/PerfDMF database supports:
  - TAU format
  - Gprof [GNU]
  - HPM Toolkit [IBM]
  - MpiP [ORNL, LLNL]
  - Dynaprof [UTK]
  - PSRun [NCSA]

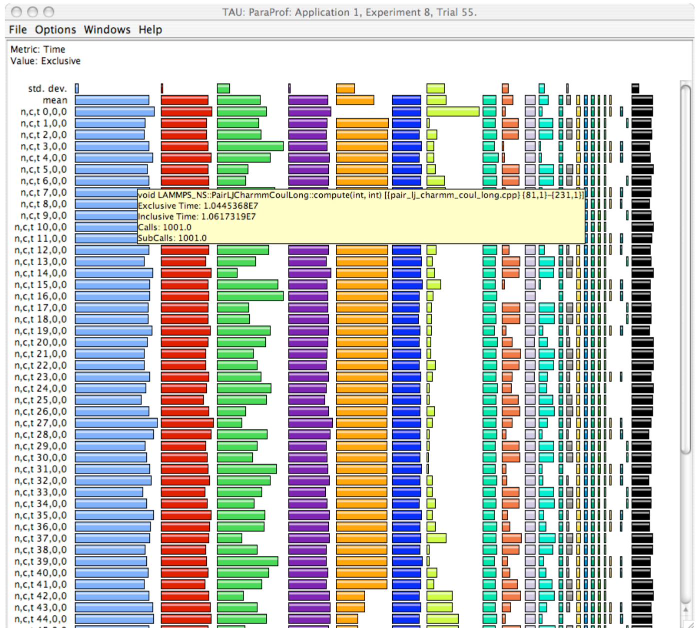
ParaProf – Manager Window



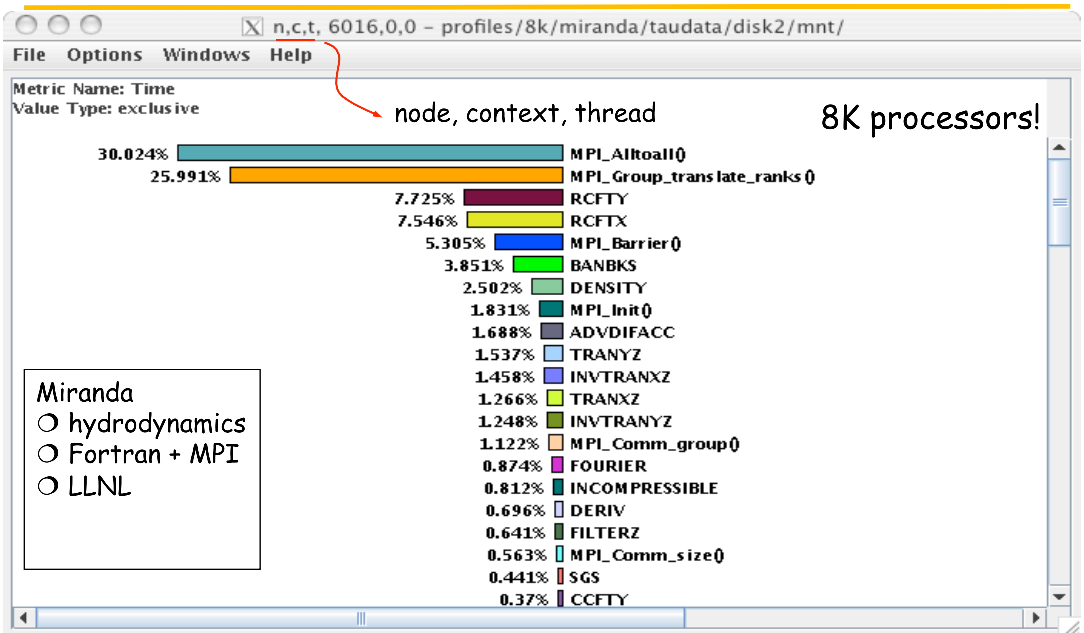
Performance Database: Storage of MetaData



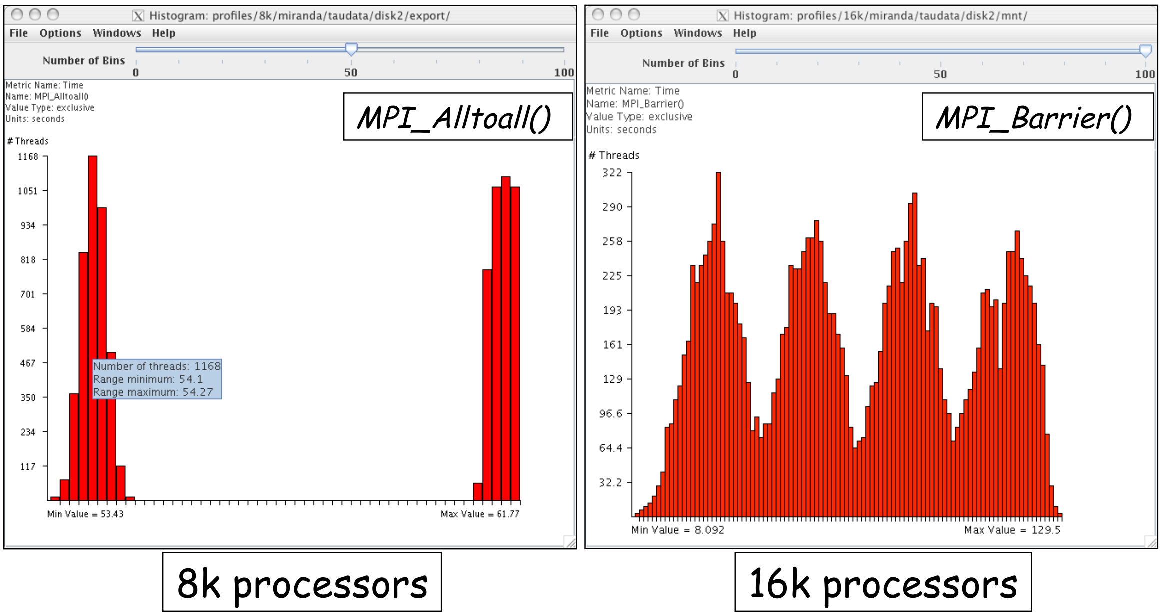
ParaProf Main Window (Lammps)



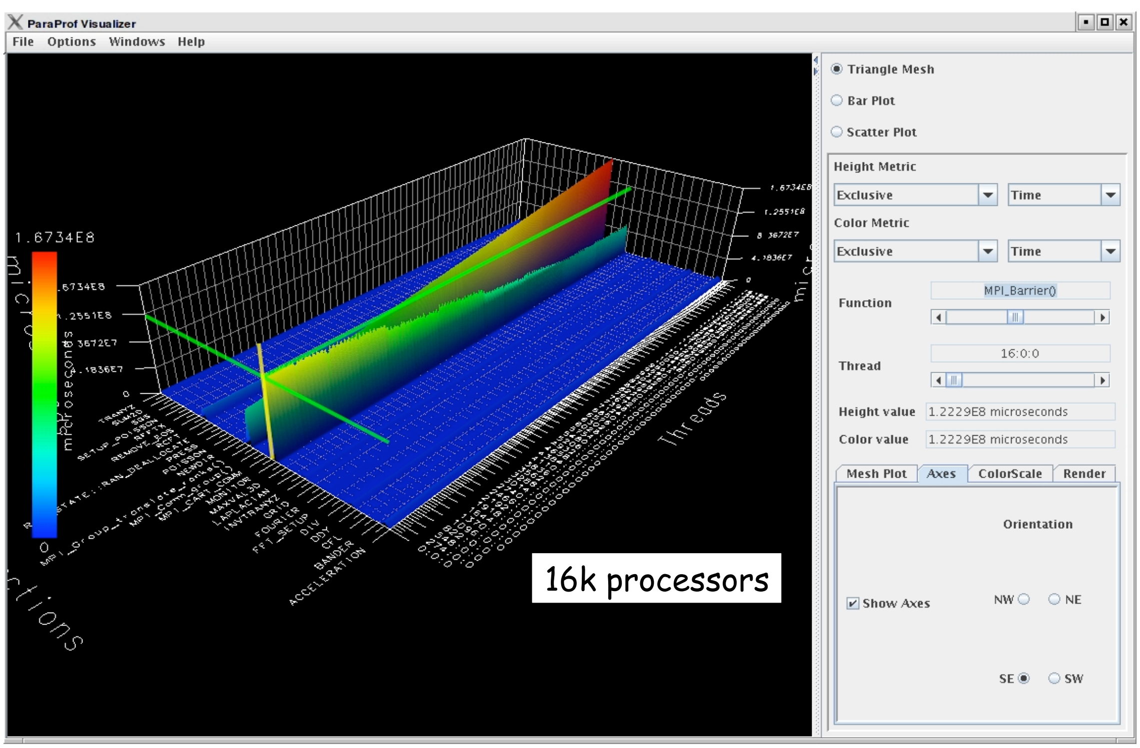
ParaProf – Flat Profile (Miranda)



ParaProf – Histogram View (Miranda)

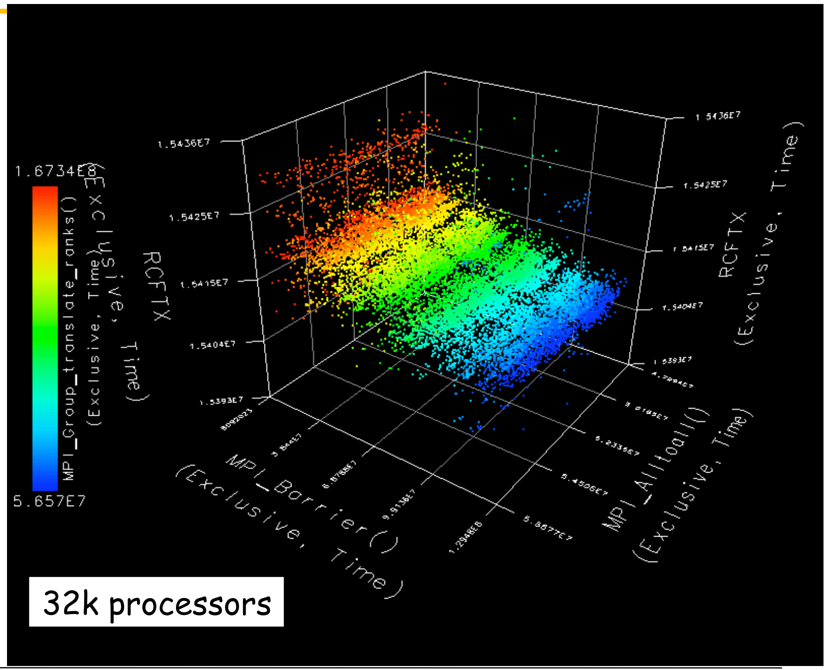


ParaProf – 3D Full Profile (Miranda)

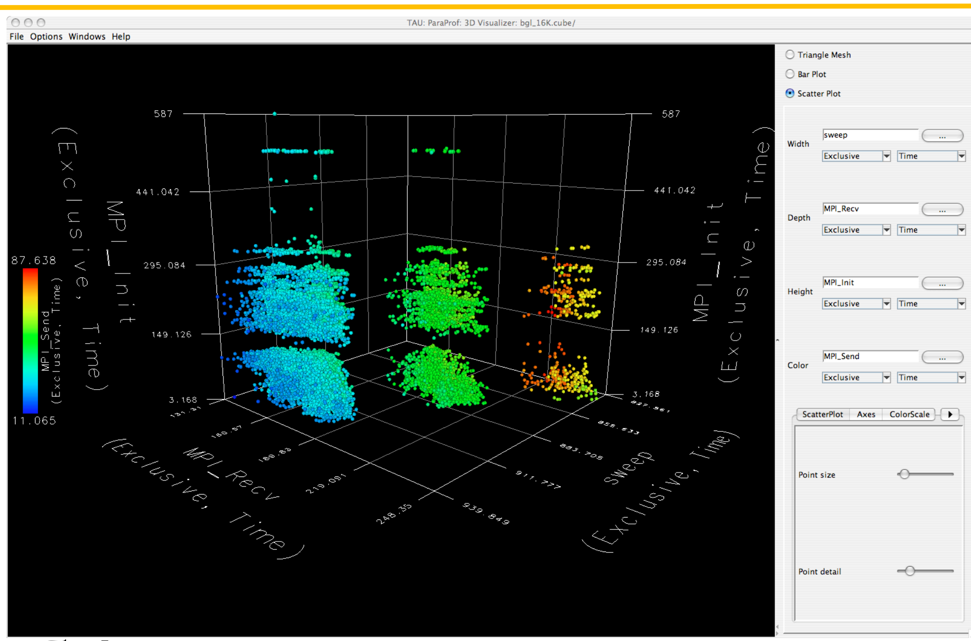


ParaProf – 3D Scatterplot (Miranda)

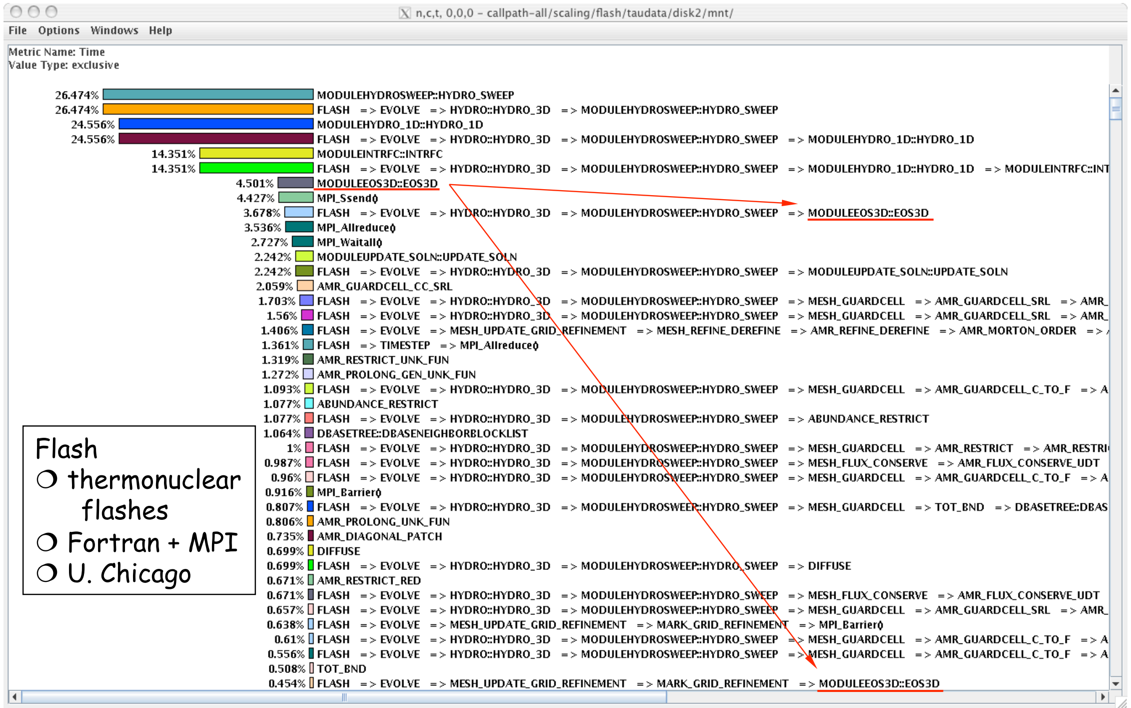
- Each point is a "thread" of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library
  - JOGL



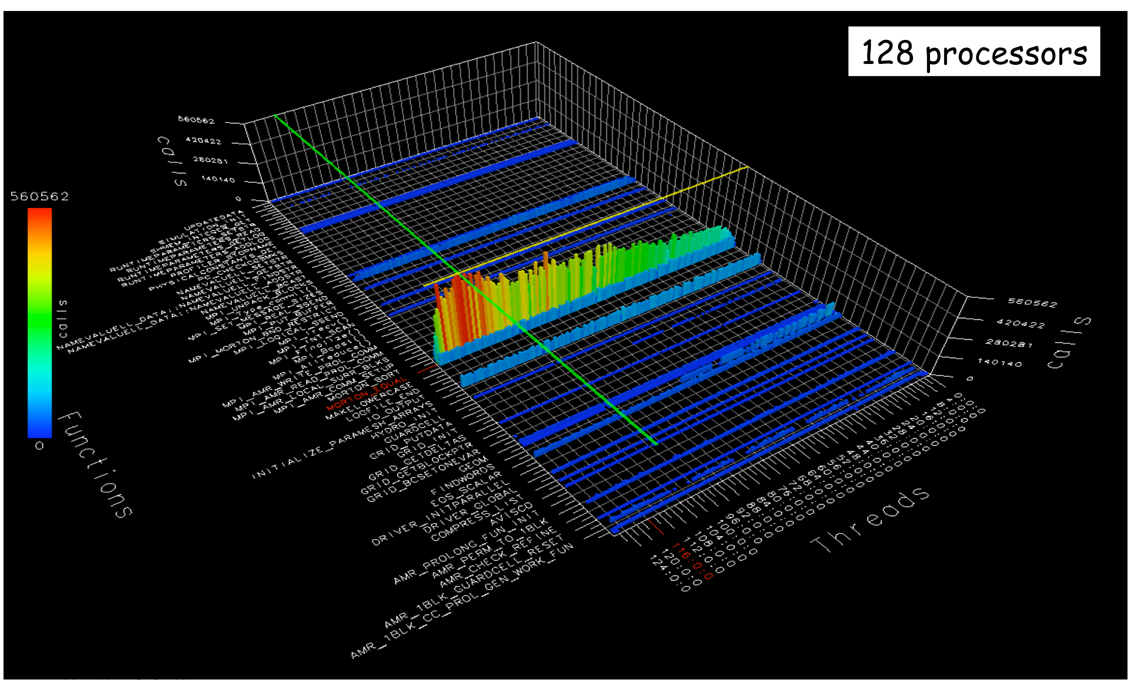
ParaProf – 3D Scatterplot (SWEEP3D CUBE)



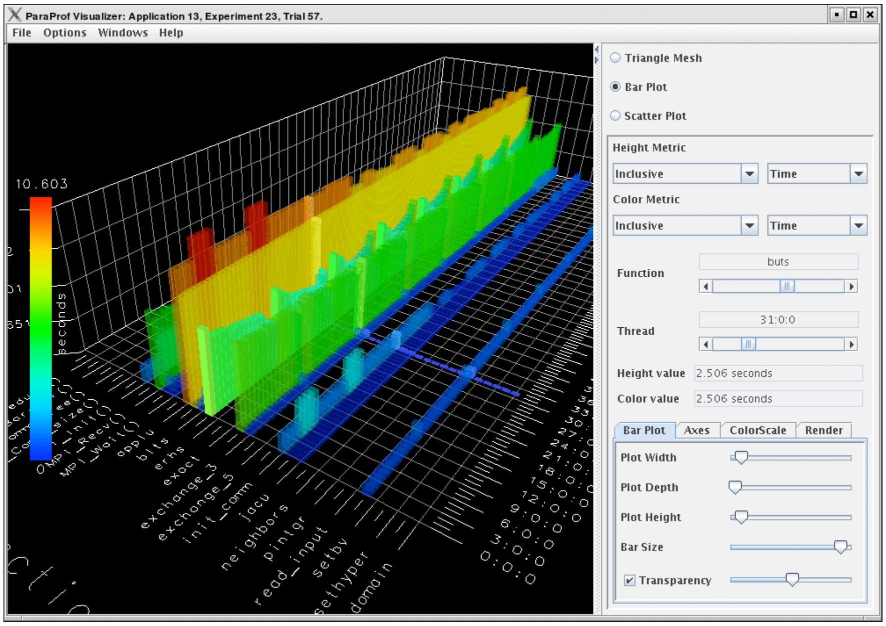
ParaProf – Callpath Profile (Flash)



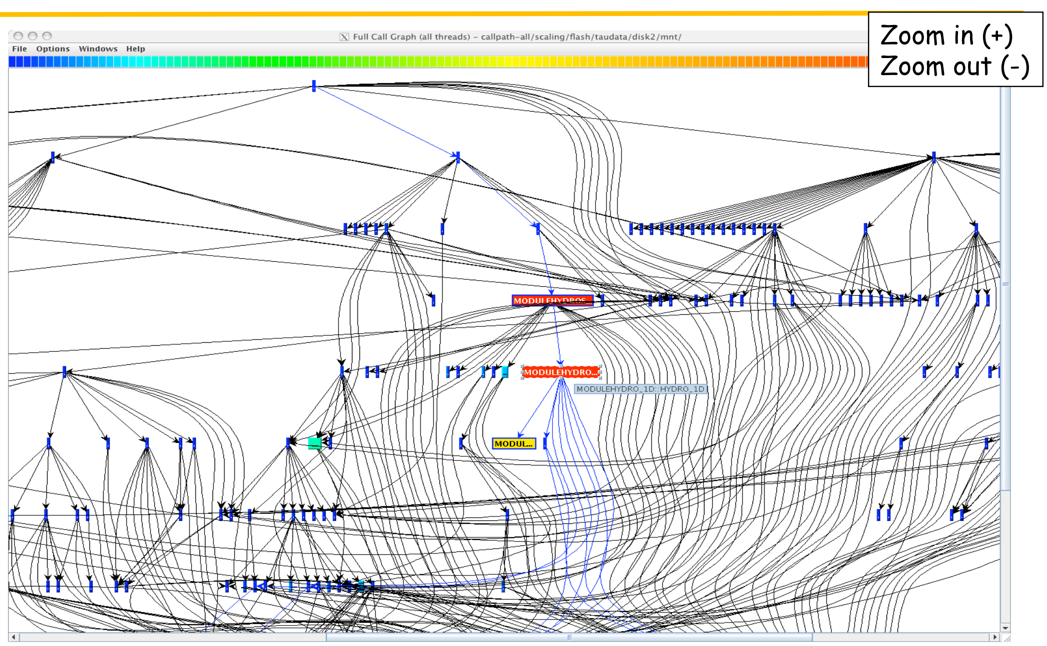
ParaProf – 3D Full Profile Bar Plot (Flash)



ParaProf Bar Plot (Zoom in/out +/-)



ParaProf – Callgraph Zoomed (Flash)



ParaProf - Thread Statistics Table (GSI)

Thread Statistics: n,c,t, 0,0,0 – comp.ppk/

File Options Windows Help

Name	Inclusive Time	Exclusive Time	Calls	Child Calls
GSI	5,223.564	0.098	1	30
SPECMOD::INIT_SPEC_VARS	0.26	0.26	1	0
MPI_Init()	0.056	0.054	1	1
GSISUB	5,223.094	0.012	1	13
RADINFO::RADINFO_READ	0.103	0.101	1	1,196
PCPINFO::PCPINFO_READ	0.042	0.042	1	0
GLBSOI	5,212.171	0.024	1	12
MPI_Finalize()	1.004	1.004	1	0
OBS_PARA	3.635	0.181	1	56
JFUNC::CREATE_JFUNC	0.142	0.142	1	0
GUESS_GRIDS::CREATE_GES_BIAS_GRIDS	0.059	0.059	1	0
READ_GUESS	1,406.412	0.023	1	8
READ_OBS	3,770.188	0.016	1	6
MPI_Allreduce()	3,725.802	3,725.802	3	0
READ_BUFRTOVS	44.369	0.254	1	871,535
SATTHIN::MAKEGVALS	0	0	1	0
W3FS21	0	0	1	1
BINARY_FILE_UTILITY::OPEN_BINARY_FILE	0.025	0.012	1	3
INITIALIZE::INITIALIZE_RTM	0.099	0.001	1	2
GUESS_GRIDS::CREATE_SFC_GRIDS	0	0	1	0
M_FVANAGRID::ALLGETLIST_	30.582	0	1	10
ERROR_HANDLER::DISPLAY_MESSAGE	0	0	1	0
JFUNC::SET_POINTER	0	0	1	0
OZINFO::OZINFO_READ	0.016	0.016	1	0
DETER_SUBDOMAIN	0.008	0.008	1	0
GRIDMOD::CREATE_MAPPING	0.005	0.005	1	0
INIT_COMMVARS	0.004	0.004	1	0
M_FVANAGRID::ALLGETLIST_	10.711	0	1	1
GRIDMOD::CREATE_GRID_VARS	0	0	1	0

ParaProf - Callpath Thread Relations Window

Call Path Data n,c,t, 0,0,0 – comp.ppk/

File Options Windows Help

Metric Name: Time
Sorted By: Exclusive
Units: seconds

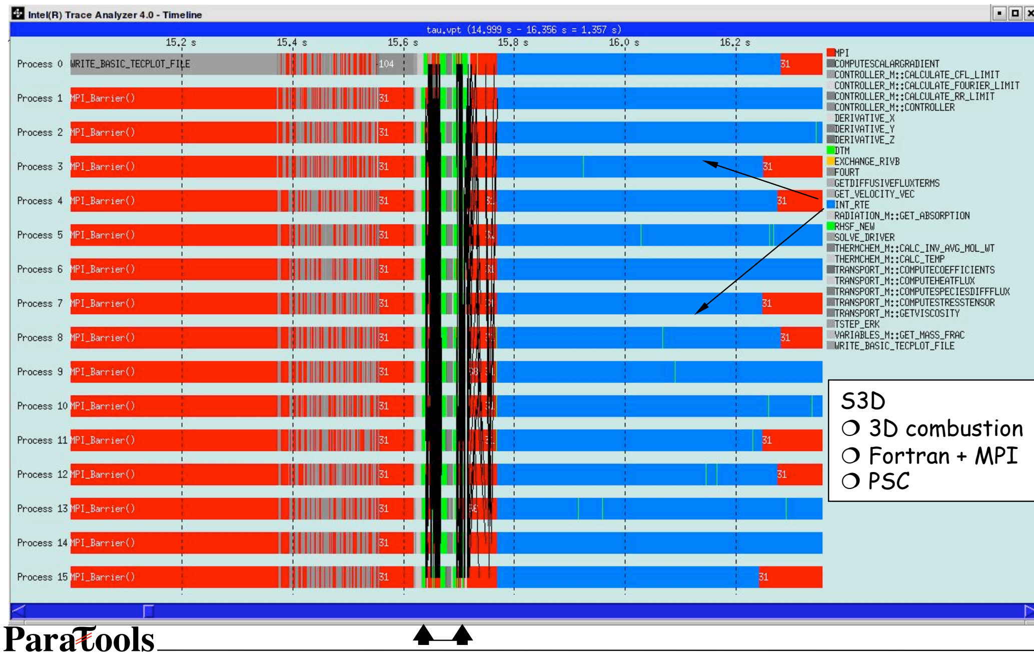
	Exclusive	Inclusive	Calls/Tot.Calls	Name[id]
	0.023	0.023	3/430	COMPUTE_DERIVED[55]
	2.02	2.02	104/430	DPRODXMOD::DPRODX[66]
	0.33	0.33	104/430	INTALLMOD::INTALL[1708]
	0.003	0.003	1/430	M_FVANAGRID::ALLGETLIST_[1773]
	1.639	1.639	1/430	OBS_PARA[1802]
	3725.802	3725.802	3/430	READ_OBS[1860]
	214.294	214.294	6/430	SETUPRHSALL[1900]
	20.069	20.069	208/430	STPCALCMOD::STPCALC[1942]
-->	3964.18	3964.18	430	MPI_Allreduce()[1762]
	2.6E-4	30.582	1/15	GLBSOI[93]
	0.007	0.036	1/15	GSI[107]
	2.7E-4	10.711	1/15	GSISUB[1690]
	31.273	1347.703	3/15	M_FVANAGRID::ALLGETLIST_[1773]
	0.412	0.412	1/15	PREWGT[1831]
	70.198	1406.389	4/15	READ_GUESS[1857]
	0.952	0.952	3/15	SATTHIN::GETSFC_GLOBAL[1882]
	86.937	95.933	1/15	WRITE_ALL[2004]
-->	196.61	1575.595	15	M_FVANAGRID::ALLGETLIST_[1773]
	6.2E-5	6.2E-5	1/1	BALMOD::CREATE_BALANCE_VARS[7]
	4.6E-5	4.6E-5	1/1	BALMOD::DESTROY_BALANCE_VARS[8]
	3.494	3.494	1/1	BALMOD::PREBAL[9]
	0.017	0.017	1/1	BERROR::CREATE_BERROR_VARS[11]
	2.0E-4	2.0E-4	1/1	BERROR::DESTROY_BERROR_VARS[12]
	8.6E-5	8.6E-5	1/1	BERROR::SET_PREDICTORS_VAR[16]
	5.7E-5	5.7E-5	1/1	COMPACT_DIFFS::CREATE_CDIFF_COEFS[34]
	4.9E-5	4.9E-5	1/1	COMPACT_DIFFS::DESTROY_CDIFF_COEFS[35]
	0.015	0.042	1/1	COMPACT_DIFFS::INISPH[41]
	0.052	8.196	3/3	COMPUTE_DERIVED[55]
	1.4E-4	3.1E-4	3/3	GETLIST_::MOVDATE_[89]
	4.2E-5	4.2E-5	1/1	GRIDMOD::DESTROY_GRID_VARS[98]
	8.2E-5	8.2E-5	1/1	GRIDMOD::DESTROY_MAPPING[99]
	0.169	0.169	3/3	GUESS_GRIDS::CREATE_ATM_GRIDS[1692]
	3.3E-4	3.3E-4	3/3	GUESS_GRIDS::DESTROY_ATM_GRIDS[1695]
	9.1E-5	9.1E-5	1/1	GUESS_GRIDS::DESTROY_GES_BIAS_GRIDS[1696]
	2.2E-4	2.2E-4	1/1	GUESS_GRIDS::DESTROY_SFC_GRIDS[1697]
	6.6E-5	6.4E-4	1/1	INITIALIZE::DESTROY_RTM[1705]
	5.8E-5	5.8E-5	1/1	JFUNC::DESTROY_JFUNC[1739]
	0.003	0.003	1/430	MPI_Allreduce()[1762]
	0.017	0.017	68/116	MPI_Bcast()[1764]
	0.004	0.004	297/409	MPI_Comm_rank()[1765]

Parent

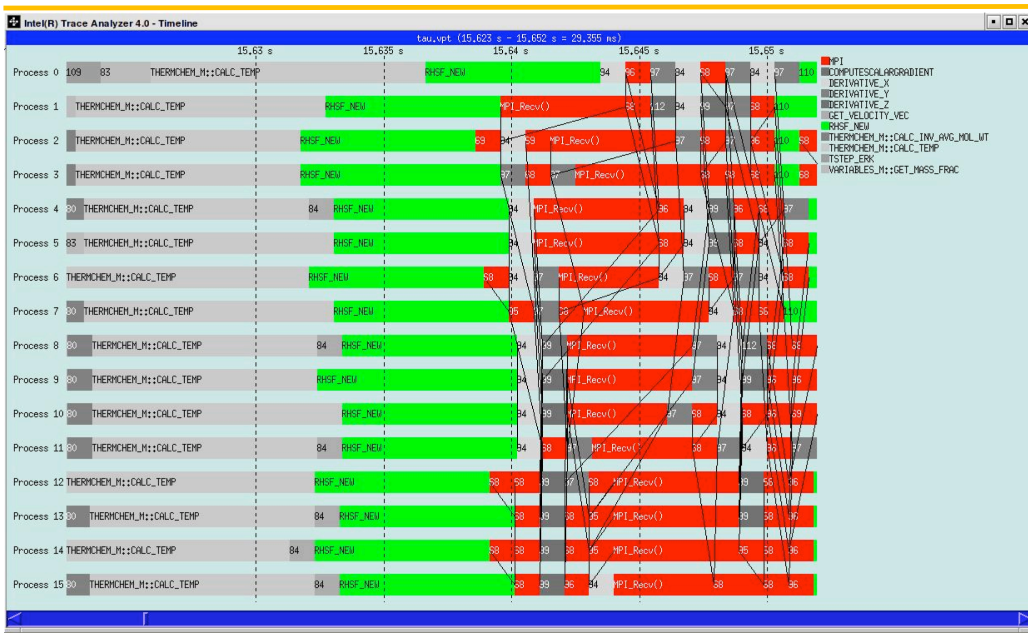
Routine

Children

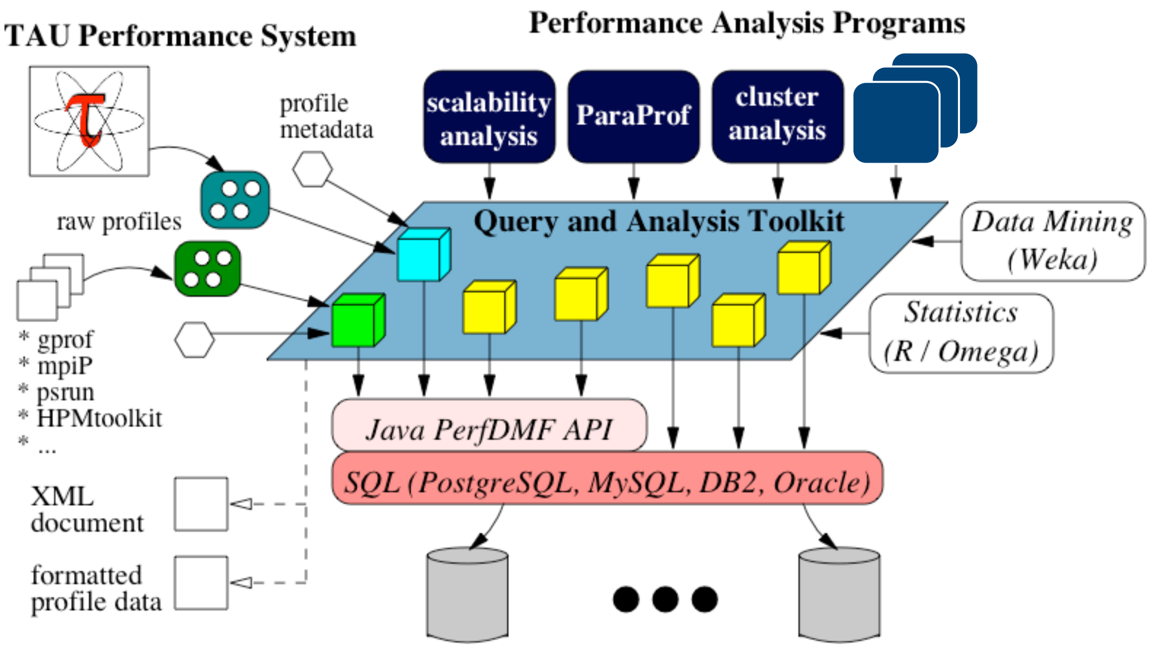
Vampir – Trace Analysis (TAU-to-VTF3) (S3D)



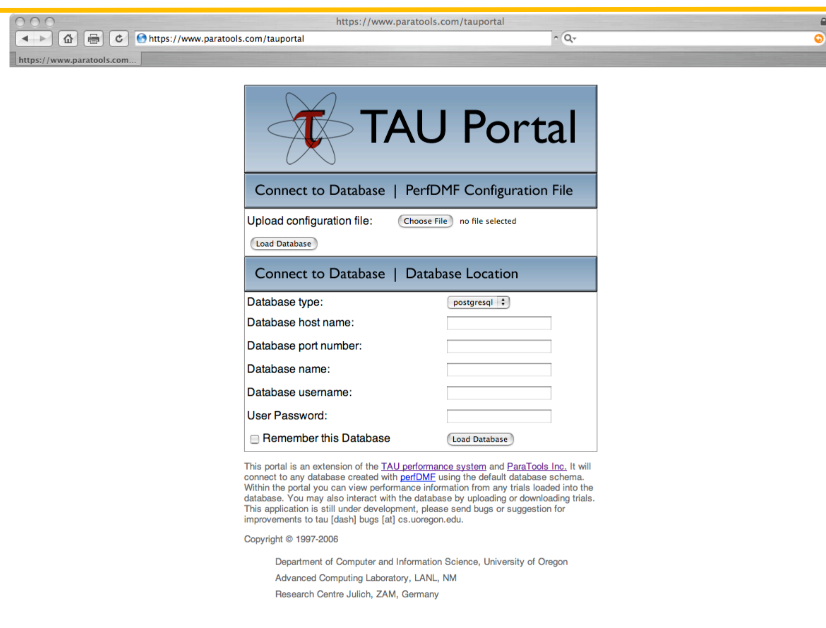
Vampir – Trace Zoomed (S3D)



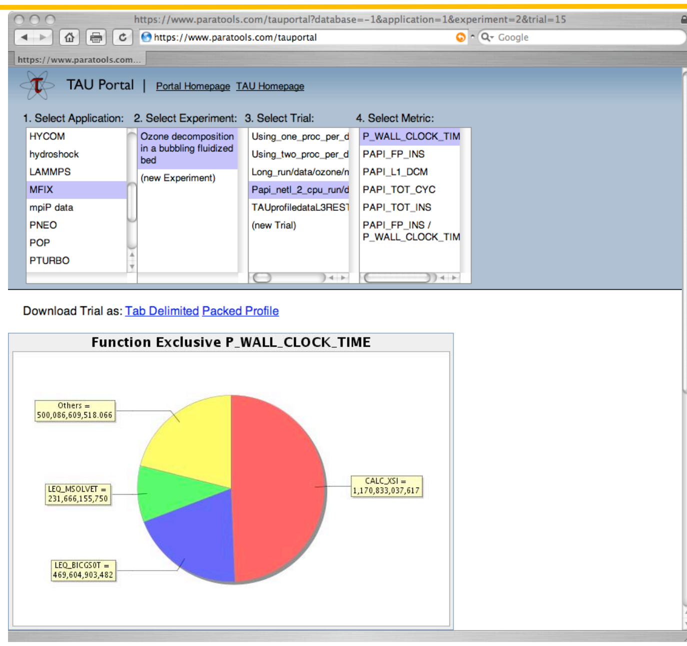
PerfDMF: Performance Data Mgmt. Framework



TAU Portal - www.paratools.com/tauportal



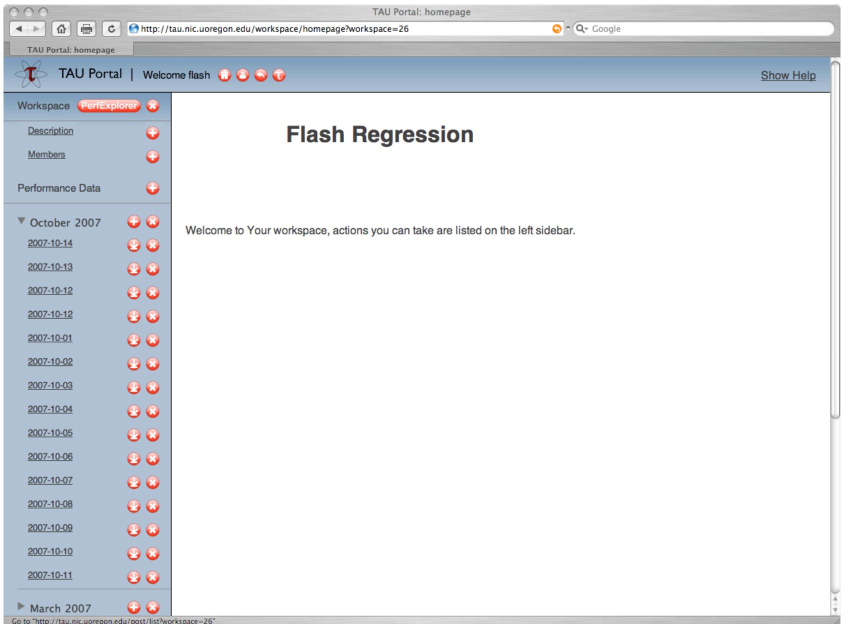
TAU Portal



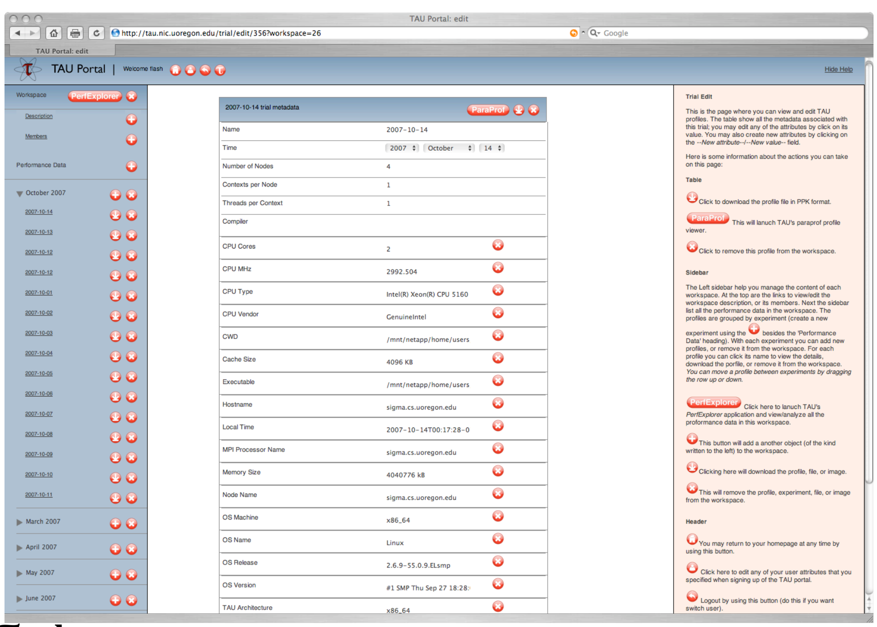
TAU Portal

- Web-based access to TAU
- Support collaborative performance study
  - Secure performance data sharing
  - Does not require TAU installation
  - Launch TAU performance tools with Java WebStart
    - ParaProf, PerfExplorer
- FLASH regression testing
  - Nightly regression testcases
  - Uploaded to the database automatically
  - Interactive review of performance through TAU portal
  - Multi-experiment analysis

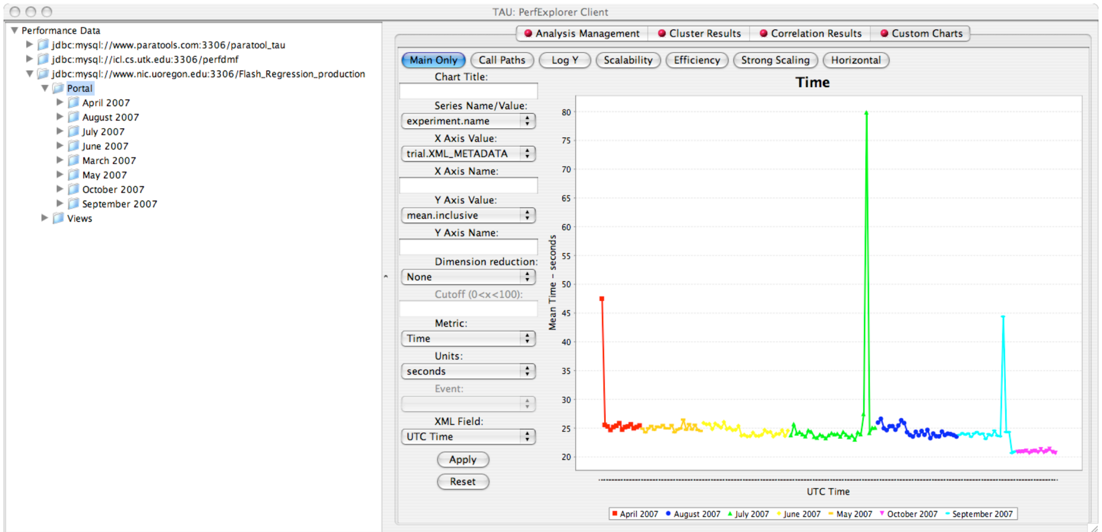
Portal: Nightly Performance Regression Testing



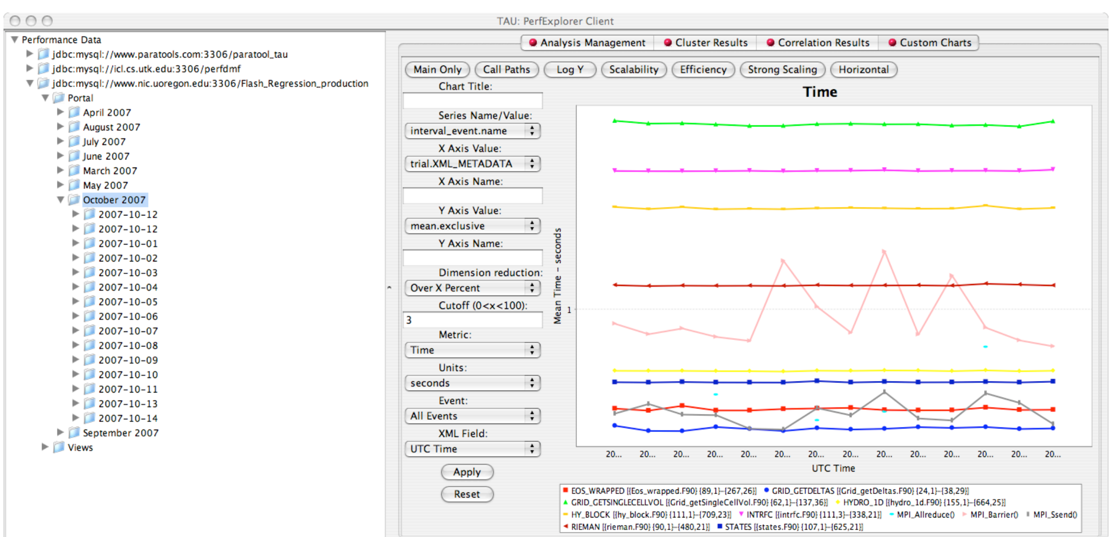
TAU Portal: Launch ParaProf/PerfExplorer



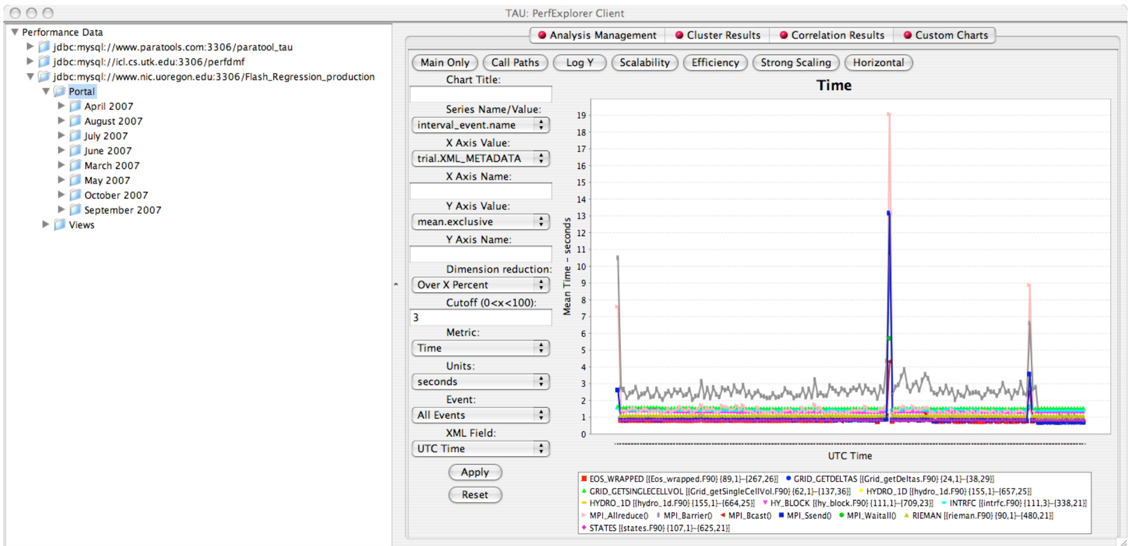
PerfExplorer: Regression Testing



PerfExplorer: Limiting Events (> 3%), Oct 2007



PerfExplorer: Exclusive Time for Events (2007)



Using Performance Database (PerfDMF)

- Configure PerfDMF (Done by each user)
  % perfdmf_configure --create-default
  - Choose derby, PostgreSQL, MySQL, Oracle or DB2
  - Hostname
  - Username
  - Password
  - Say yes to downloading required drivers (we are not allowed to distribute these)
  - Stores parameters in your ~/.ParaProf/perfdmf.cfg file
- Configure PerfExplorer (Done by each user)
  % perfexplorer_configure
- Execute PerfExplorer
  % perfexplorer

PerfDMF and the TAU Portal

- Development of the TAU portal
  - Common repository for collaborative data sharing
  - Profile uploading, downloading, user management
  - Paraprof, PerfExplorer can be launched from the portal using Java Web Start (no TAU installation required)
- Portal URL
  http://tau.nic.uoregon.edu

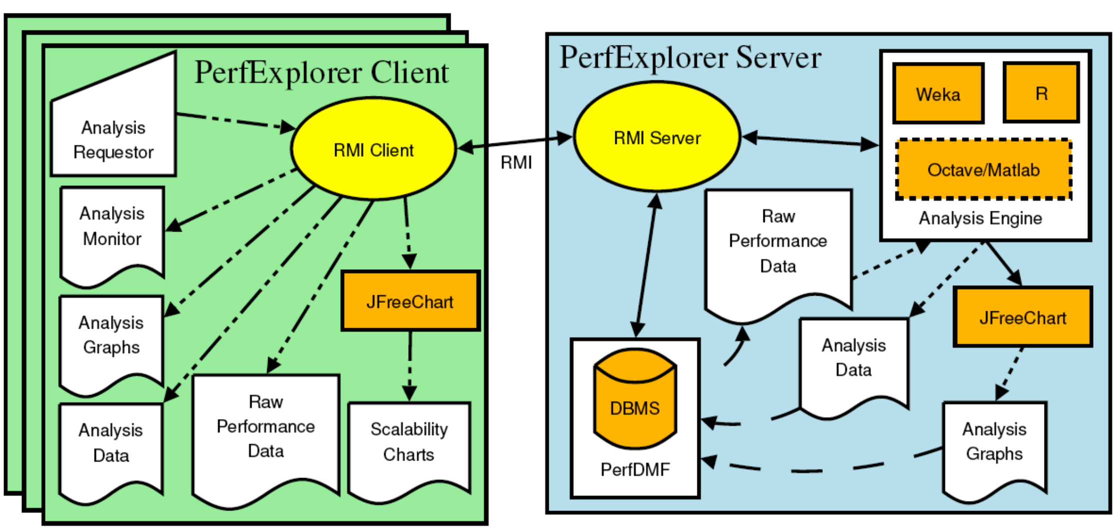
Performance Data Mining (Objectives)

- Conduct parallel performance analysis process
  - In a systematic, collaborative and reusable manner
  - Manage performance complexity
  - Discover performance relationship and properties
  - Automate process
- Multi-experiment performance analysis
- Large-scale performance data reduction
  - Summarize characteristics of large processor runs
- Implement extensible analysis framework
  - Abstraction / automation of data mining operations
  - Interface to existing analysis and data mining tools

Performance Data Mining (PerfExplorer)

- Performance knowledge discovery framework
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, …
  - Use the existing TAU infrastructure
    - TAU performance profiles, PerfDMF
  - Client-server based system architecture
- Technology integration
  - Java API and toolkit for portability
  - PerfDMF
  - R-project/Omegahat, Octave/Matlab statistical analysis
  - WEKA data mining package
  - JFreeChart for visualization, vector output (EPS, SVG)

Performance Data Mining (PerfExplorer)



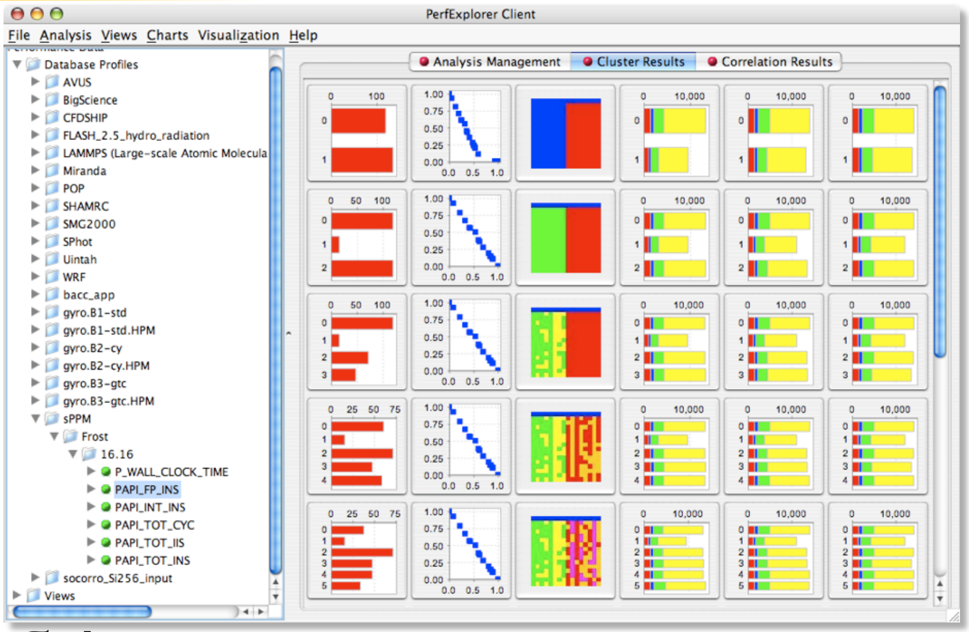
PerfExplorer - Analysis Methods

- Data summaries, distributions, scatter plots
- Clustering
  - *k*-means
  - Hierarchical
- Correlation analysis
- Dimension reduction
  - PCA
  - Random linear projection
  - Thresholds
- Comparative analysis
- Data management views

PerfExplorer - Cluster Analysis

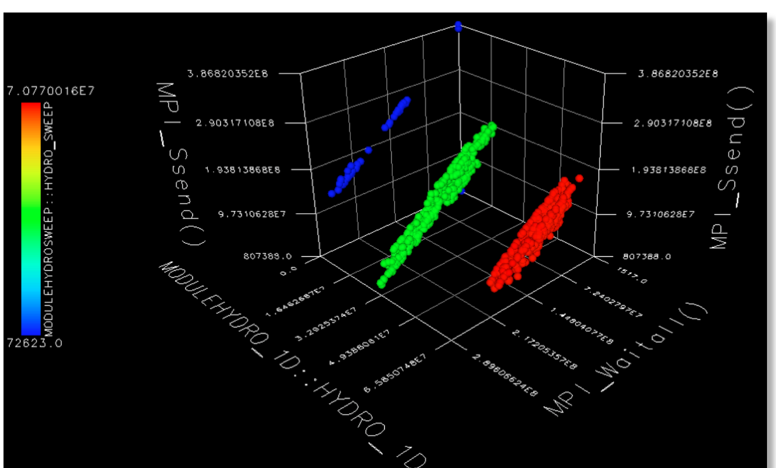
- Performance data represented as vectors - each dimension is the cumulative time for an event
- *k*-means: *k* random centers are selected and instances are grouped with the "closest" (Euclidean) center
- New centers are calculated and the process repeated until stabilization or max iterations
- Dimension reduction necessary for meaningful results
- Virtual topology, summaries constructed

PerfExplorer - Cluster Analysis (sPPM)



PerfExplorer - Cluster Analysis

- Four significant events automatically selected (from 16K processors)
- Clusters and correlations are visible

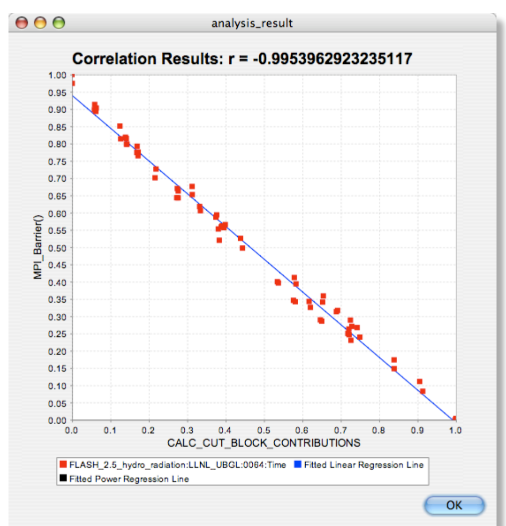


PerfExplorer - Correlation Analysis (Flash)

- Describes strength and direction of a linear relationship between two variables (events) in the data

PerfExplorer - Correlation Analysis (Flash)

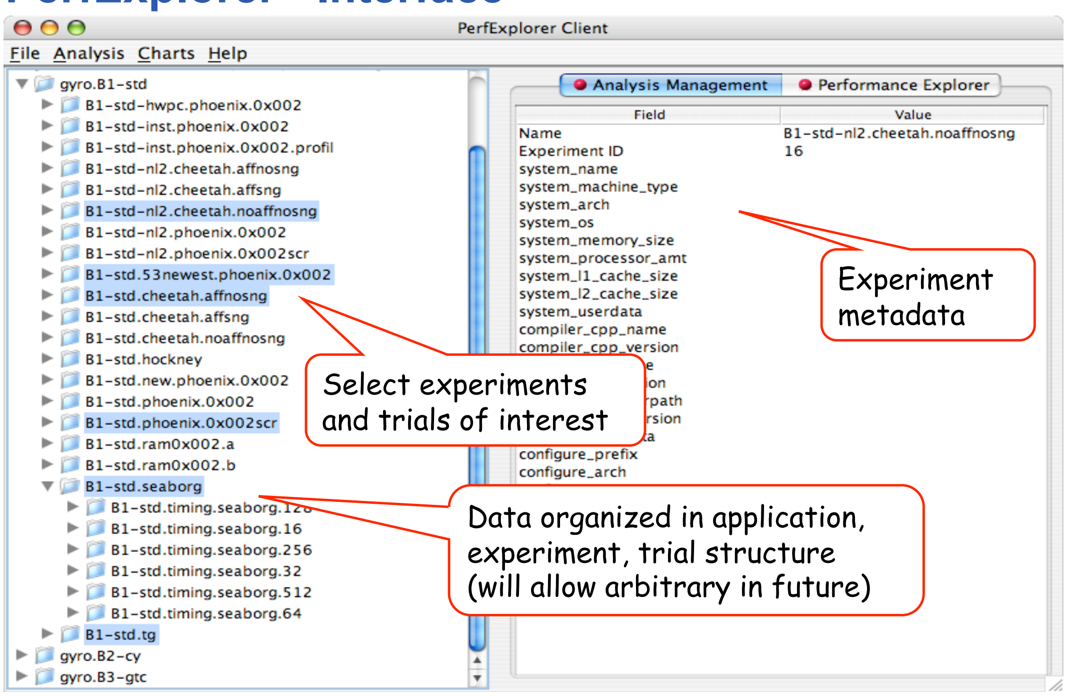
- -0.995 indicates strong, negative relationship
- As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases



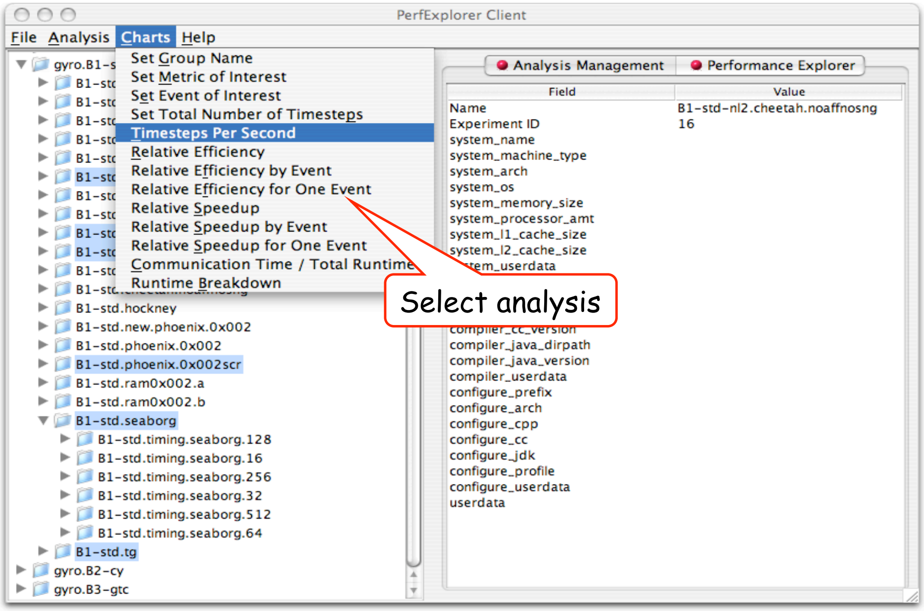
PerfExplorer - Comparative Analysis

- Relative speedup, efficiency
  - total runtime, by event, one event, by phase
- Breakdown of total runtime
- Group fraction of total runtime
- Correlating events to total runtime
- Timesteps per second
- Performance Evaluation Research Center (PERC)
  - PERC tools study (led by ORNL, Pat Worley)
  - In-depth performance analysis of select applications
  - Evaluation performance analysis requirements
  - Test tool functionality and ease of use

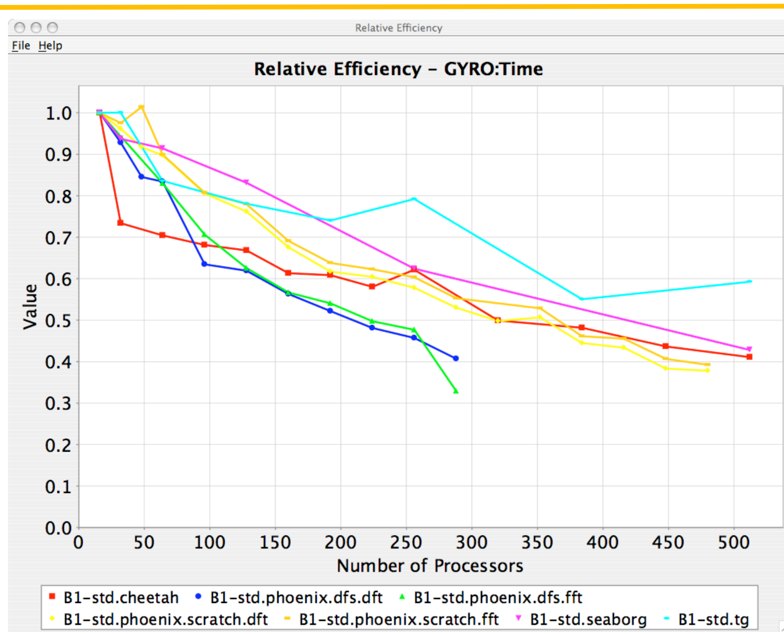
PerfExplorer - Interface



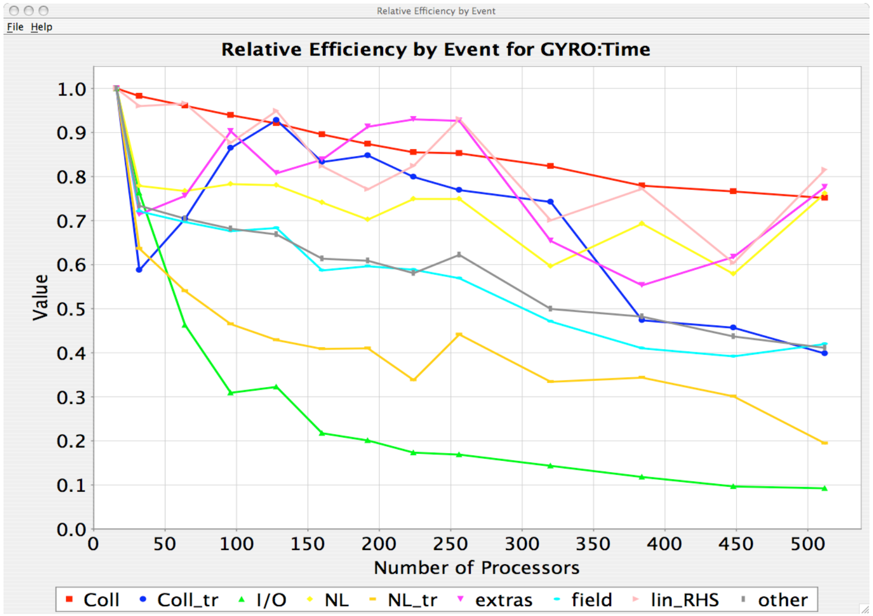
PerfExplorer - Interface



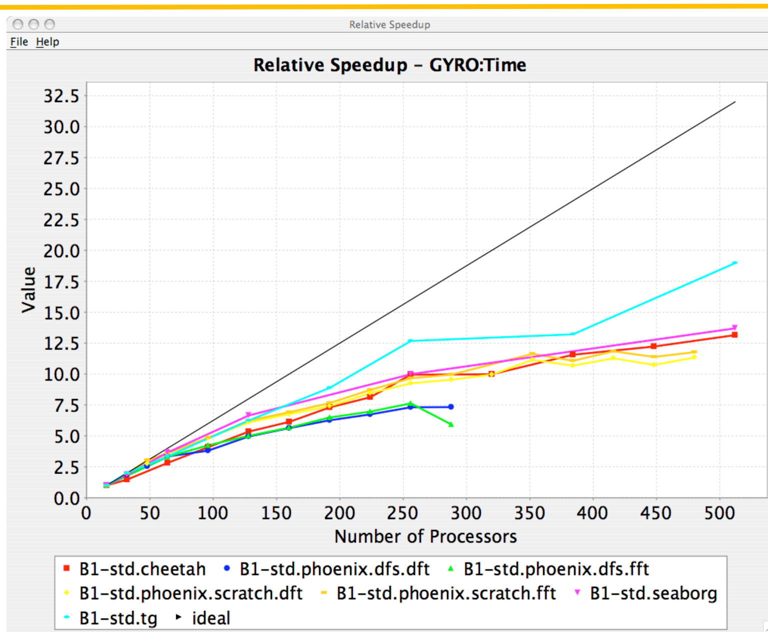
PerfExplorer - Relative Efficiency Plots



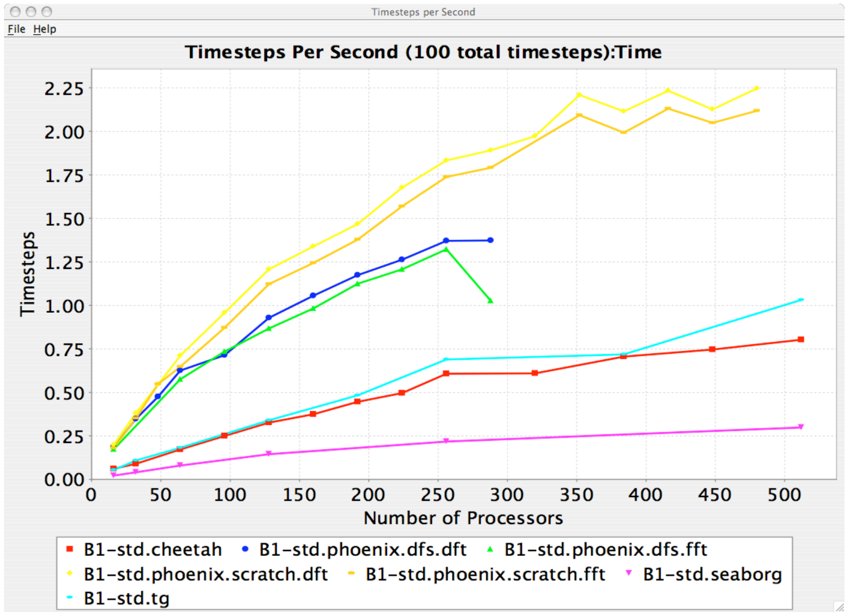
PerfExplorer - Relative Efficiency by Routine



PerfExplorer - Relative Speedup

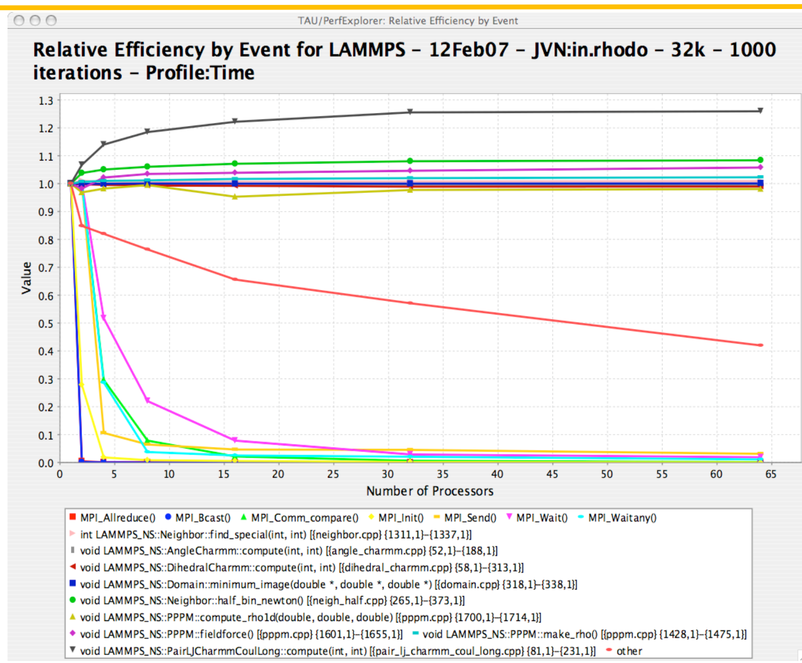


PerfExplorer - Timesteps Per Second



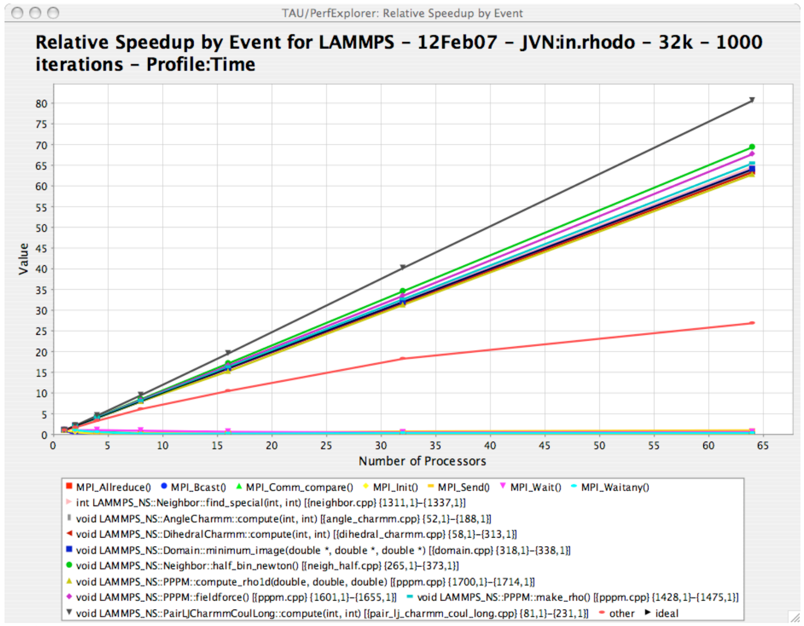
ParaTools

PerfExplorer - Relative Efficiency

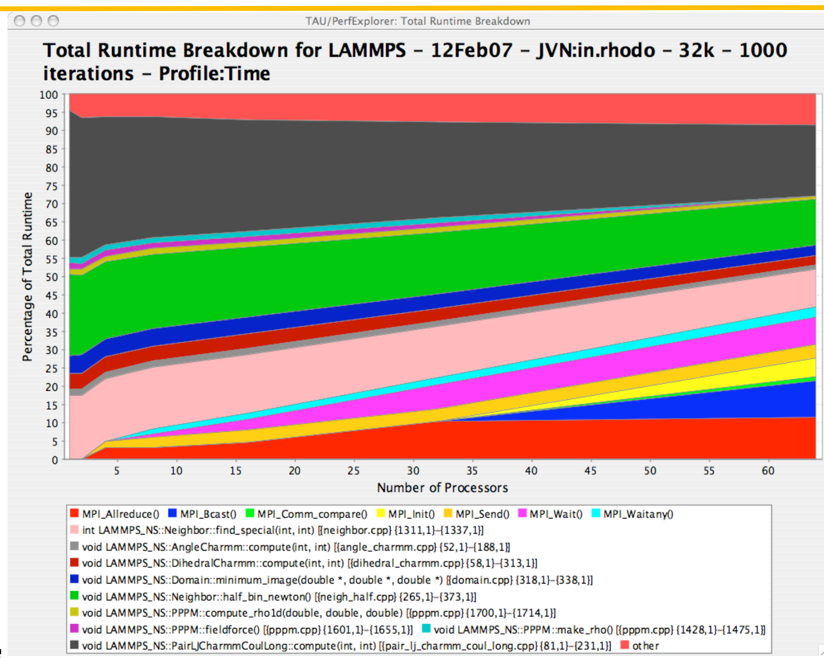


ParaTools

PerfExplorer - Relative Speedup by Event

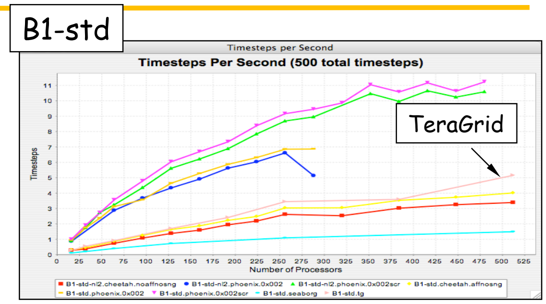


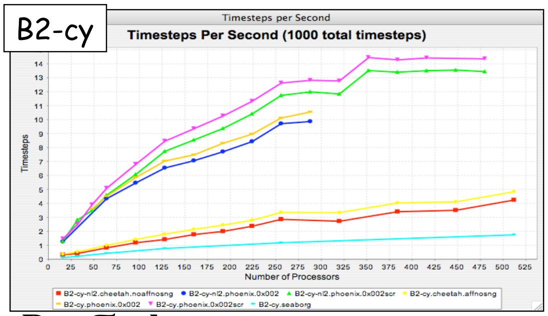
PerfExplorer - Runtime Breakdown

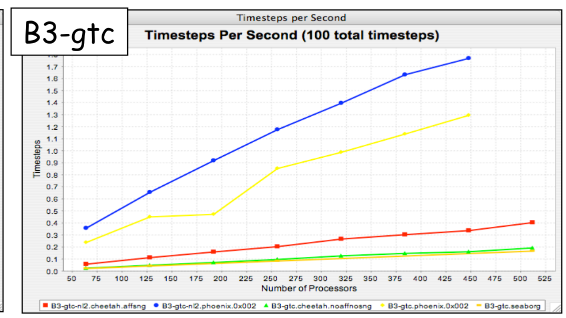


PerfExplorer - Timesteps per Second for GYRO

- Cray X1 is the fastest to solution
  - In all 3 tests
- FFT (nl2) improves time
  - B3-gtc only
- TeraGrid faster than p690
  - For B1-std?
- All plots generated automatically

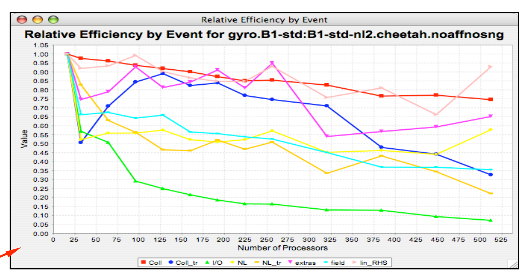


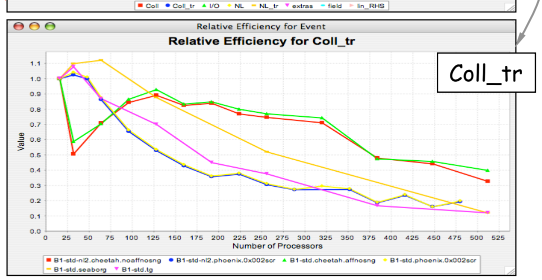


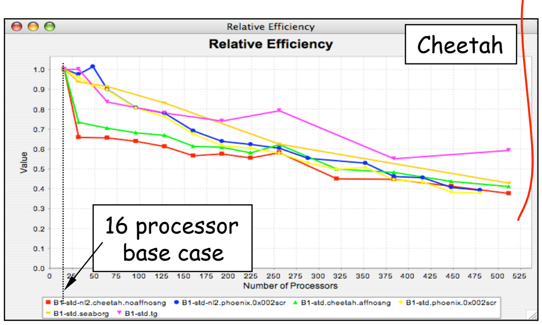


PerfExplorer - Relative Efficiency (B1-std)

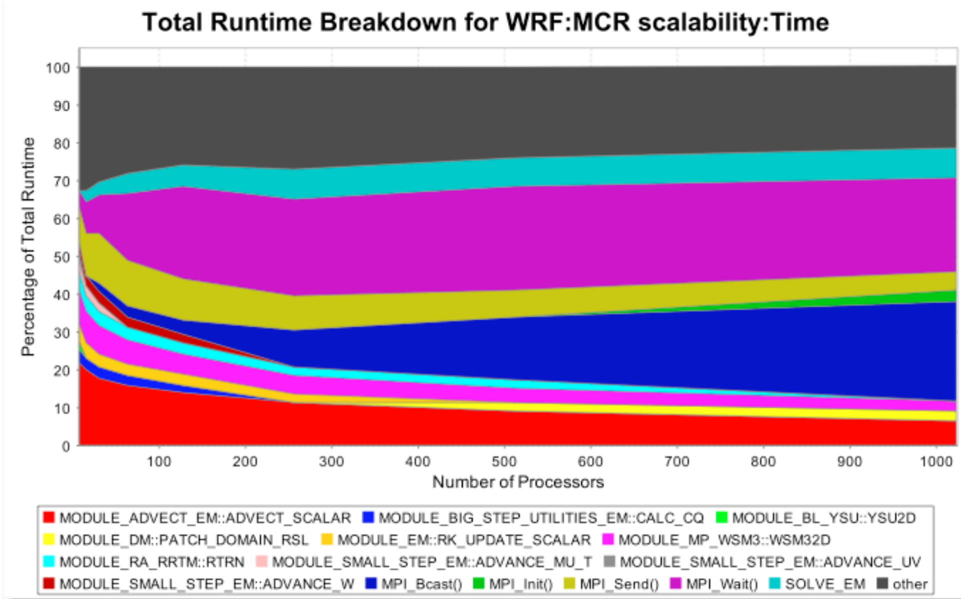
- By experiment (B1-std)
  - Total runtime (Cheetah (red))
- By event for one experiment
  - Coll_tr (blue) is significant
- By experiment for one event
  - Shows how Coll_tr behaves for all experiments



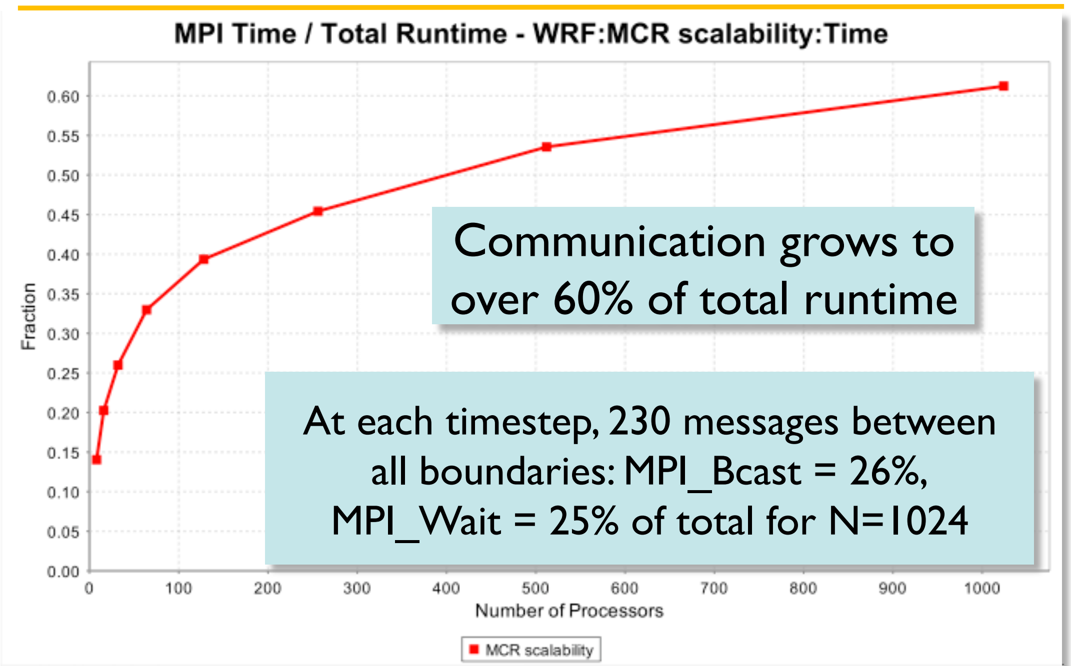




PerfExplorer - Runtime Breakdown



Group % of Total

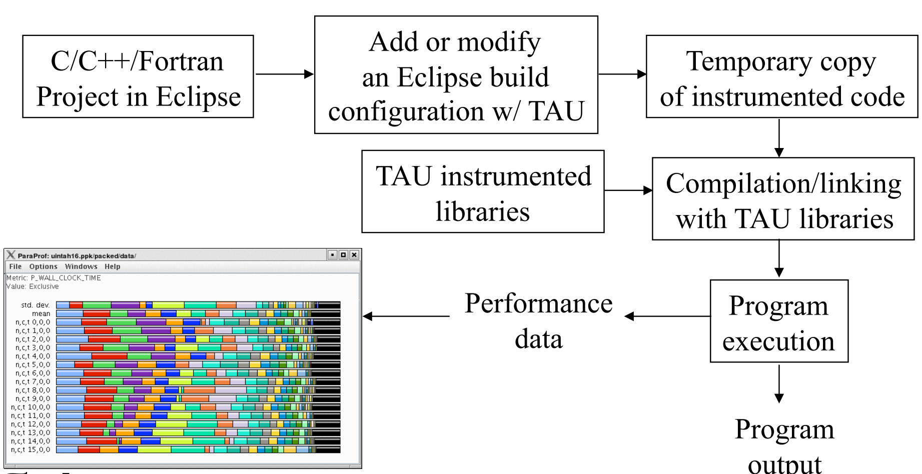


TAU Integration with IDEs

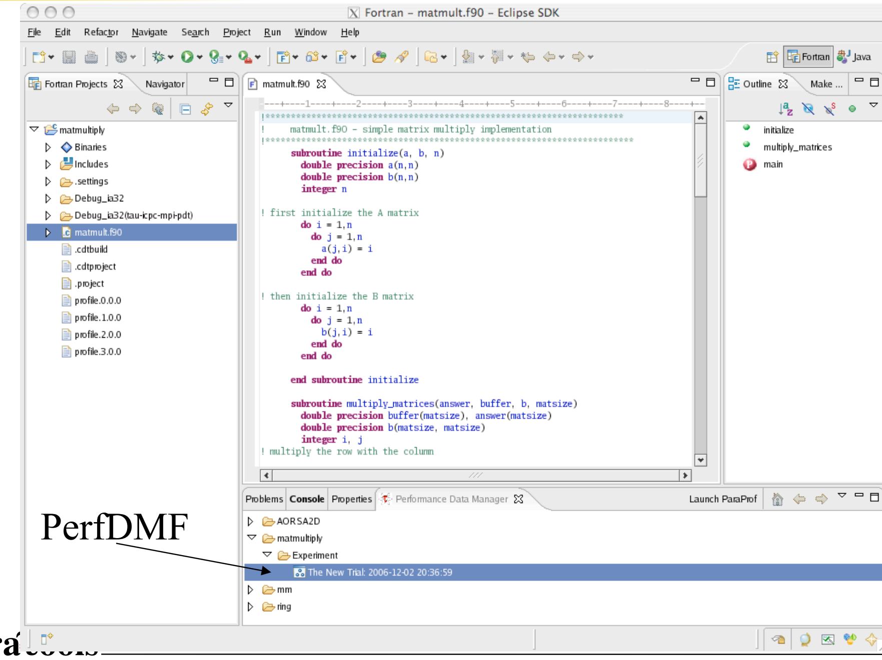
- High performance software development environments
  - Tools may be complicated to use
  - Interfaces and mechanisms differ between platforms / OS
- Integrated development environments
  - Consistent development environment
  - Numerous enhancements to development process
  - Standard in industrial software development
- Integrated performance analysis
  - Tools limited to single platform or programming language
  - Rarely compatible with 3rd party analysis tools
  - Little or no support for parallel projects

TAU and Eclipse

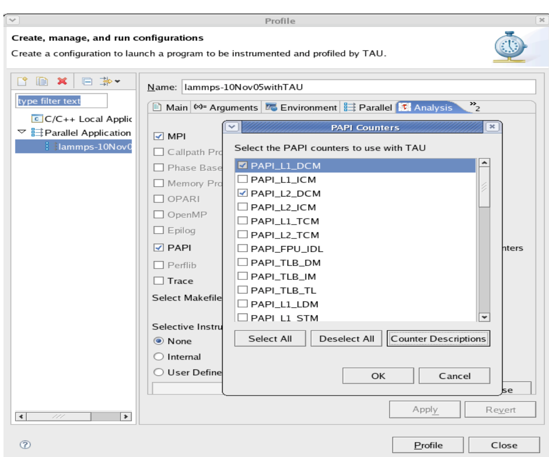
- Provide an interface for configuring TAU's automatic instrumentation within Eclipse's build system
- Manage runtime configuration settings and environment variables for execution of TAU instrumented programs

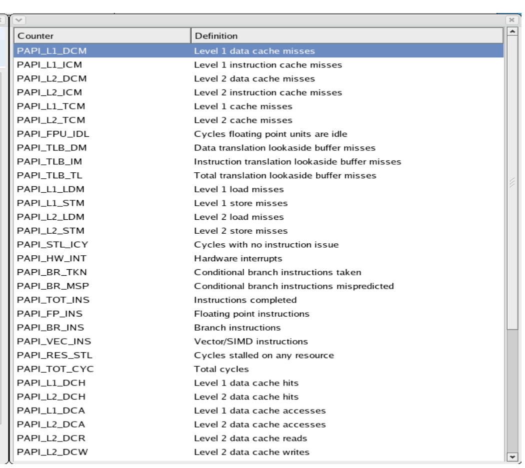


TAU and Eclipse



Choosing PAPI Counters with TAU in Eclipse






```
% /usr/global/tools/tau/training/eclipse/eclipse
```

Part III: MPI and Compiler Tips and Tricks!

General notes

- Measurement experiment typically limited to parallel region between MPI_Init and MPI_Finalize
  - experiment preparation before/during MPI_Init
  - experiment finalization during/after MPI_Finalize
- Tools typically rely on normal application termination
  - if execution is killed or exits without successful MPI_Finalize measurement experiment report not produced or incomplete
  - additional finalization time should be included in job runtime
- Measurement will necessarily dilate parallel execution
  - small amounts of dilation are unavoidable
  - dilation is uneven and most severe for small routines
    - often such routines are in-lined by compiler optimizer
    - might need to adjust instrumentation/measurement

I/O notes

- Application file I/O performance often highly variable
  - depends on load on shared filesystem/network resources
    - and application/system configuration at time of measurement
  - tuning requires very careful extensive benchmarking
    - worst care performance very different from typical case
  - current tools don't deal well with this
- Optimal I/O is no I/O!
  - preferable to eliminate non-essential I/O during measurement
  - configure tools to avoid intermediate measurement I/O (e.g., trace buffer flushes) where appropriate
  - configure measurement or analysis to exclude I/O phases
    - typically part of one-off application initialization/finalization cost which would be amortized in long production execution

Scaling notes

- Typically want to analyze execution at production scale
  - performance may vary during course of execution
  - performance at small scales may differ considerably
  - best understood in context of performance scaling study
- Large-scale performance measurement & analysis often a “grand challenge” in its own right
  - quantity of measurements & analyses are proportional to number of processes (for constant application complexity)
  - event traces grow linearly with duration of measurement
- Advisable to start small and increase scale in stages
  - profiling/summarization requirements invariant with time
  - trace targeted short sections (e.g., only a few iterations)

Communication & synchronization

- Expect communication costs to grow with num. processes
  - Examine number of messages & bytes transferred
    - redundant data transfers become increasingly costly
    - however, read+broadcast often better than parallel read
  - Synchronizing operations become more serious
    - blocking/waiting time is wasted time
    - collectives may synchronize all ranks unnecessarily
- Origins of imbalance need careful examination
  - often earlier than it ultimately manifests

Computation

- Optimized libraries are customized to platform
  - often provide a drop-in replacement for user functions
- Optimizing compilers can often do a good job
  - at least when given a reasonable chance
- Tuning experts study assembly code produced by compiler
  - and work with the compiler to do better
- Most tuning can be done at very small scale
  - with only a single compute node (or less!)

MPI Performance Tuning Tips

- Aggregate many small messages into a single large message to reduce latency
- Aggregate data in collective operations
  - More efficient to do one 2-element allreduce than two 1-element operations
- Overlap computation and communication using non-blocking MPI calls (persistent)
- Evaluate load balancing and scaling using performance tools
- Use a different algorithm that can be parallelized better
- Tradeoffs: slower communication for faster computation
- For iterative solvers, check convergence after a set of steps instead of at each iteration
- Loop unrolling:
  - Loop: I=1, N
    call Compute(I)
    exchange data for I
  - Loop: I=1, N, 2
    call Compute(I)
    call Compute(I+1)
    exchange data for I and I+1

Part IV: VAMPIRTRACE & VAMPIR INTRODUCTION AND OVERVIEW

Andreas Knüpfer
Technische Universität Dresden

Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Introduction

Why bother with performance analysis?

- Well, why are you here after all?
- Efficient usage of expensive and limited resources
- Scalability to achieve next bigger simulation

Profiling and Tracing

- Have an optimization phase!
- Use tools!
- Avoid do-it-yourself-with-printf solutions, really!!!

Introduction: Profiling & Tracing

Program Instrumentation

- Detect run-time events (points of interest)
- Pass information to run-time measurement library

Profile Recording

- Collect aggregated information (Time, Counts, …)
- About program and system entities
  - functions, loops, basic blocks per process/thread

Trace Recording

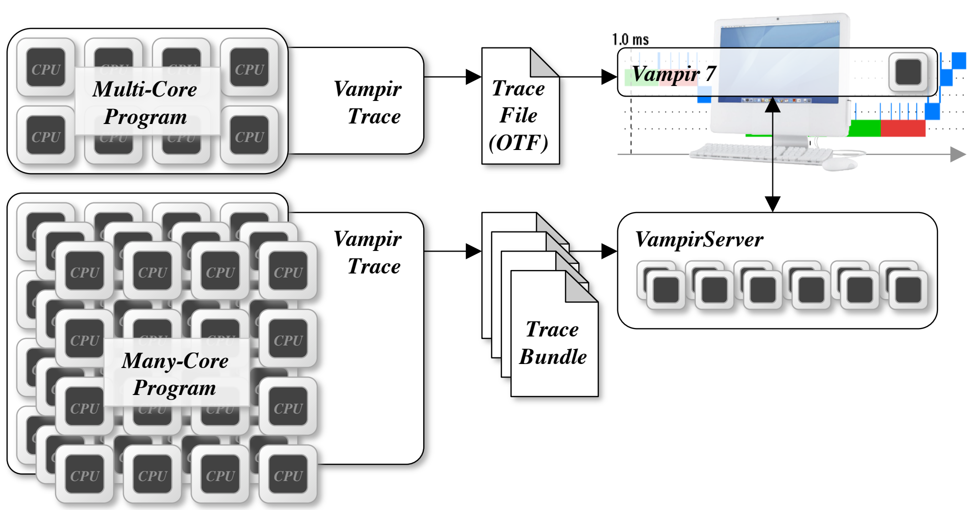
- Individual event records
- Precise time stamp, process/thread ID
- Event specific information

Event Trace Visualization

Trace Visualization

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior visually
- Provide statistics and performance metrics
  - global timeline for parallel processes/threads
  - process timeline plus performance counters
  - statistic summary display
  - communication statistics, more …
- Interactive browsing, zooming, selecting
  - adapt statistics to zoom level (time interval)
  - also for very large and highly parallel traces

Vampir Toolset Architecture



Overview

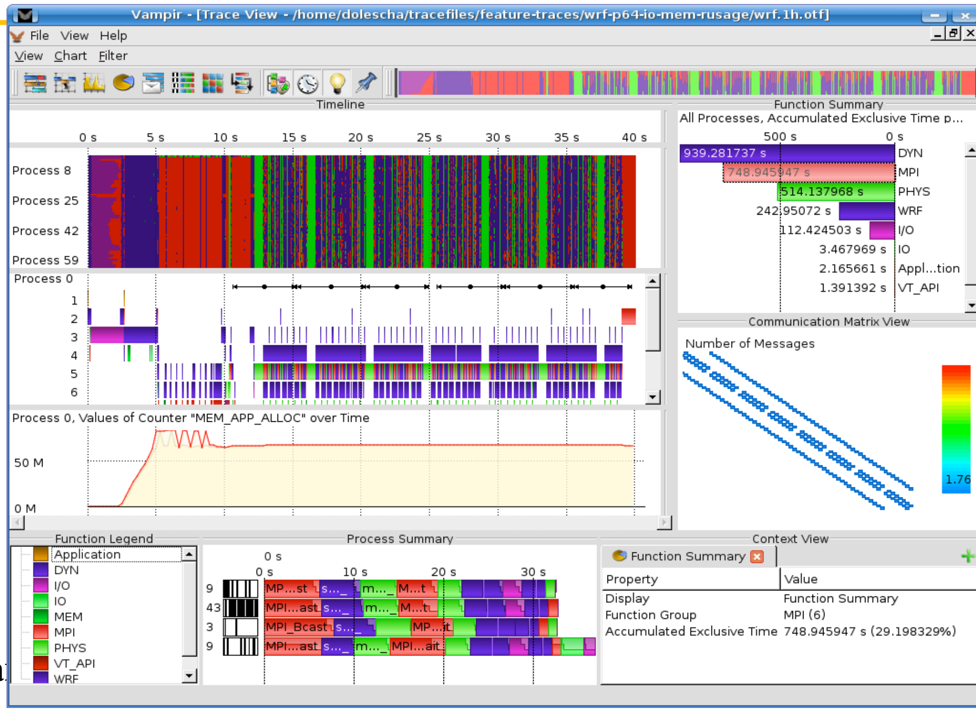
- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Vampir Displays

The main displays of Vampir:

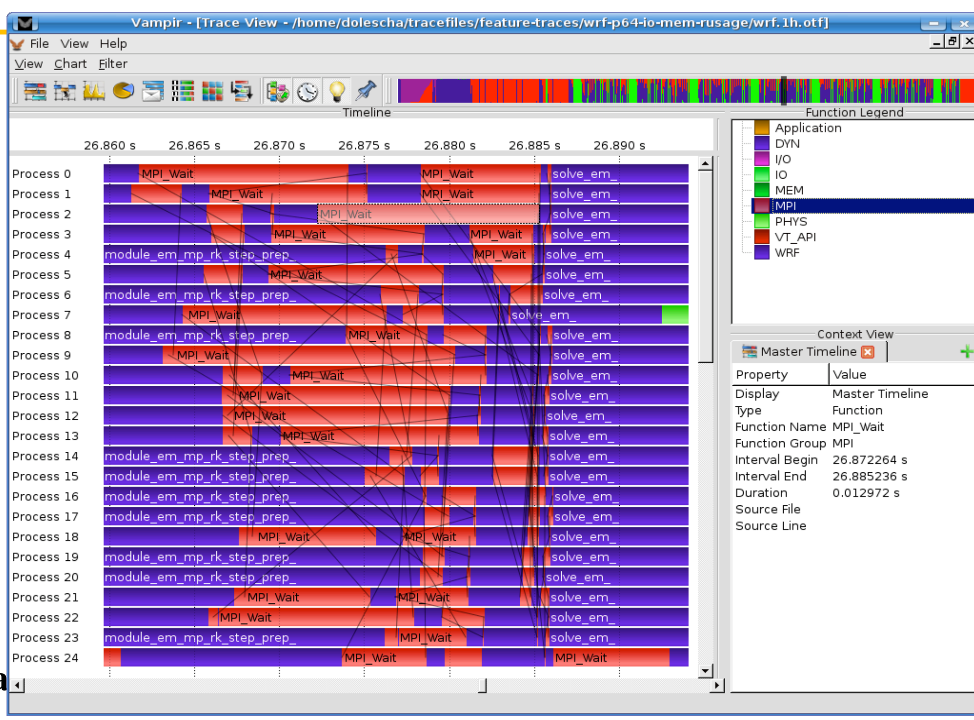
- Master Timeline
- Process and Counter Timeline
- Function Summary
- Message Summary
- Process Summary
- Communication Matrix
- Call Tree

Vampir 7 Display Overview



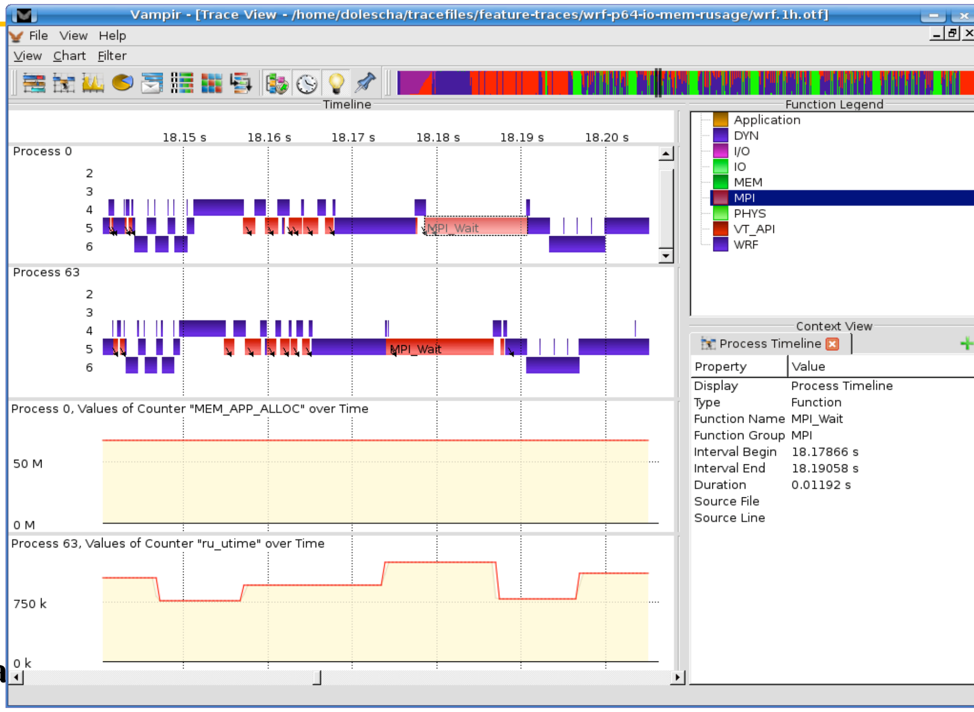
Pa

Master Timeline

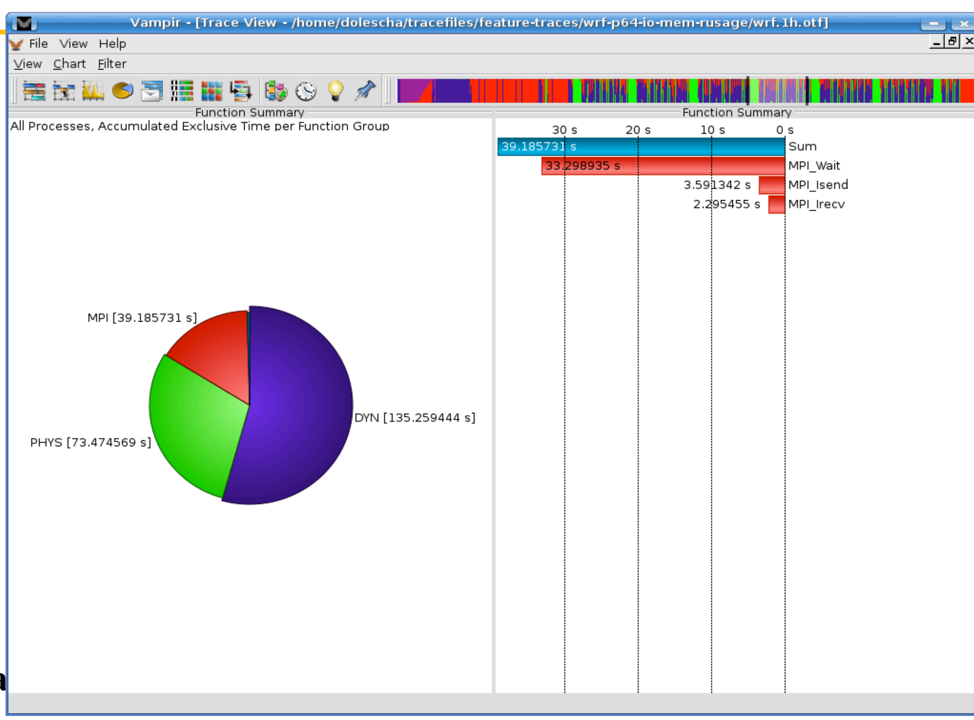


Pa

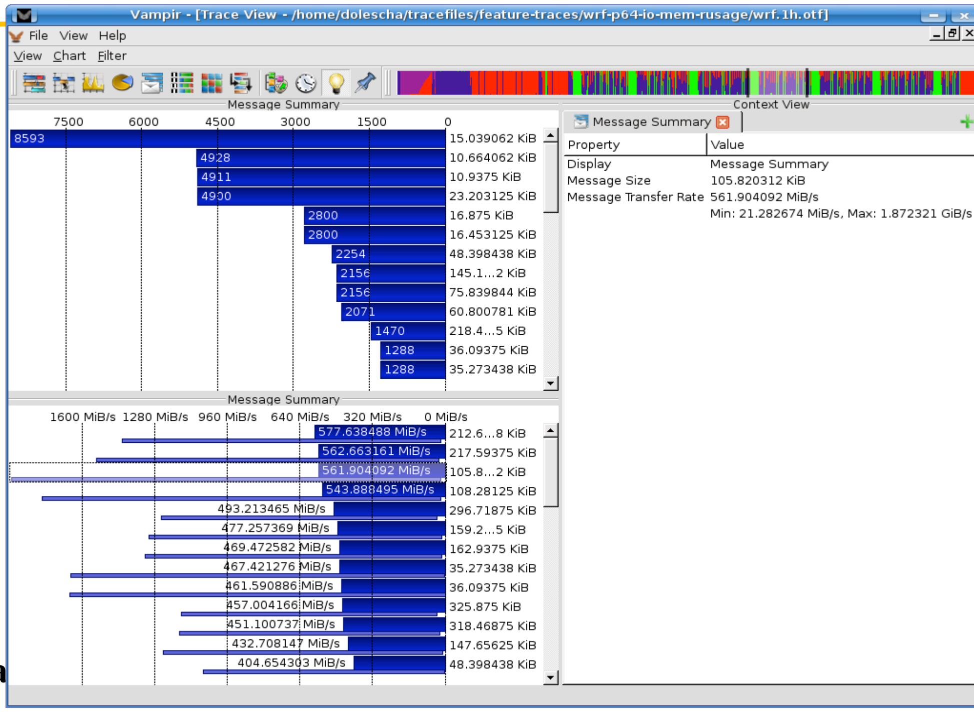
Process and Counter Timelines



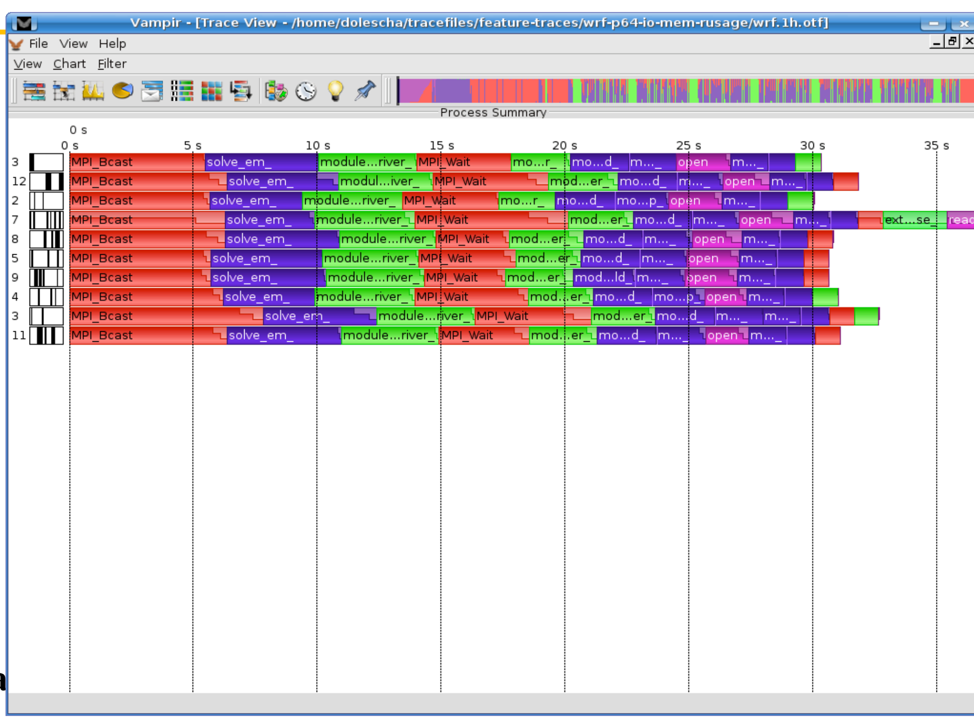
Function Summary



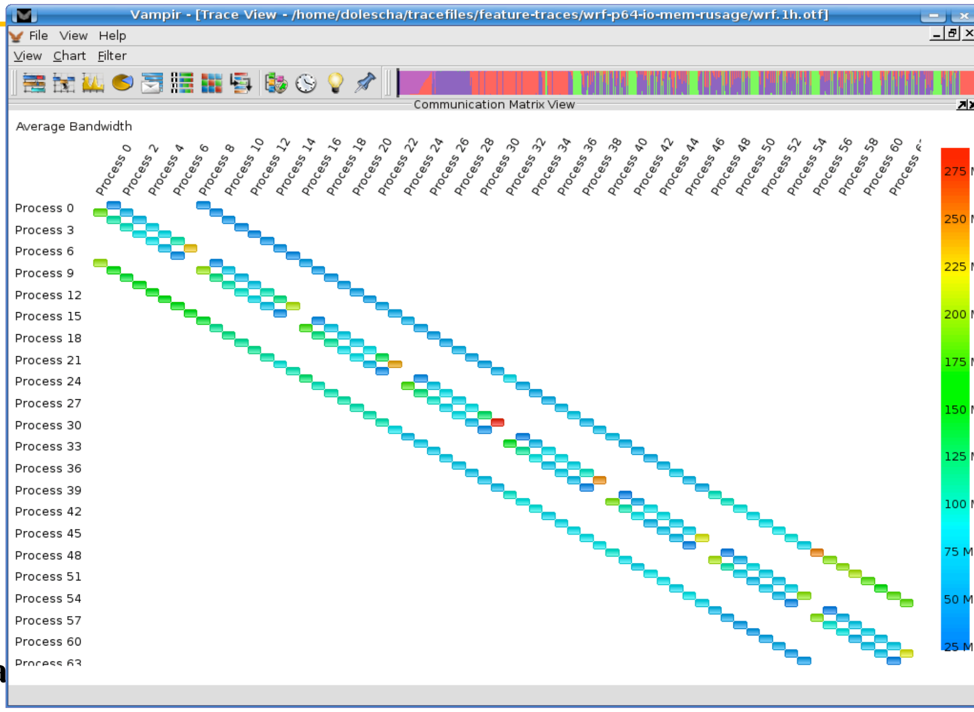
Message Summary



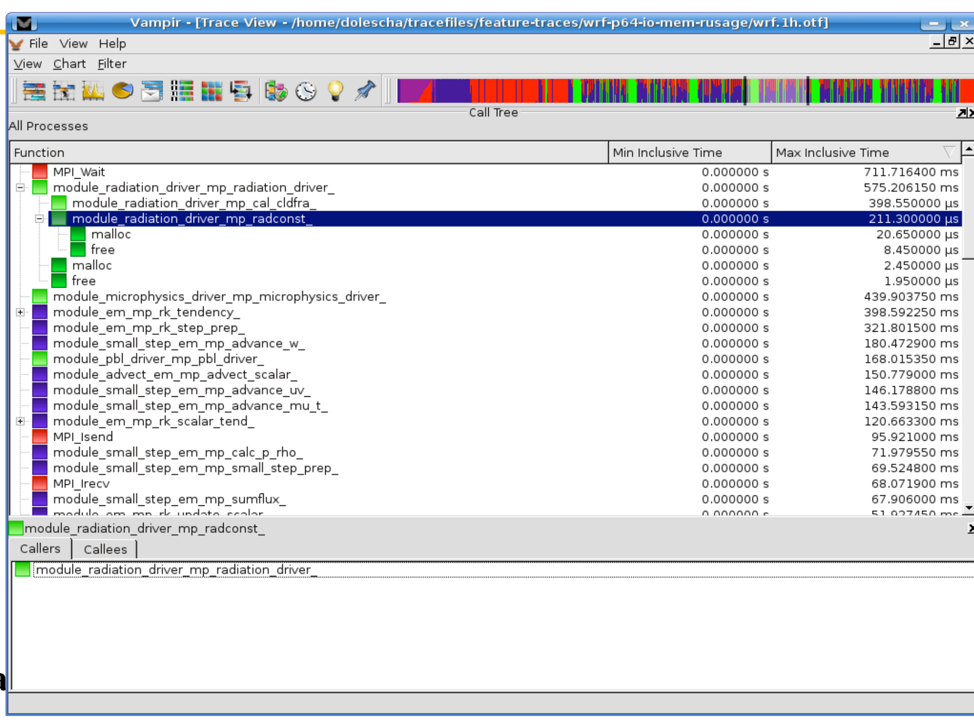
Process Summary



Communication Matrix



Call Tree



Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Profiling and Tracing

- Tracing Advantages
  - preserve temporal and spatial relationships
  - allow reconstruction of dynamic behavior on any required abstraction level
  - profiles can be calculated from trace
- Tracing Disadvantages
  - traces can become very large
  - may cause perturbation
  - instrumentation and tracing is complicated (event buffering, clock synchronization, …)

Common Event Types

- Enter/leave of function/routine/region
  – time, process/thread, function ID
- Send/receive of P2P message (MPI)
  – time, sender, receiver, length, tag, comm.
- Collective communication (MPI)
  – time, process, root, communicator, # bytes
- Hardware performance counter values
  – time, process, counter ID, value

Instrumentation

- Instrumentation:
  Process of modifying target programs
  to detect and report events

  – call instrumentation functions
  – provided by trace library
  – call for every run-time event of interest

  – various ways

Instrumentation & Measurement

What do you need to do for it?

- Instrumentation (automatic with compiler wrappers)

`CC=icc`	`CC=vtcc`
`CXX=icpc`	`CXX=vtcxx`
`F90=ifc`	`F90=vtf90`
`MPICC=mpicc`	`MPICC=vtcc`

- Re-compile & re-link
- Trace Run (run with appropriate test data set)
- More details later

Instrumentation & Measurement

What does VampirTrace do?

- During Instrumentation:
  - via compiler wrappers
  - by underlying compiler with specific options
  - MPI instrumentation with replacement lib
  - OpenMP instrumentation with Opari
  - also binary instrumentation with Dyninst
  - partial manual instrumentation

Instrumentation & Measurement

What does VampirTrace do?

- During trace run:
  - event data collection
  - precise time measurement
  - parallel timer synchronization
  - collecting parallel process/thread traces
  - collecting performance counters (from PAPI, memory usage, POSIX I/O calls and fork/ system/exec calls, and more ...
  - filtering and grouping of function calls

The Tools

- VampirTrace
  - convenient instrumentation and measurement
  - hides away complicated details
  - provides many options and switches for experts
  - part of Open MPI 1.3 package
- Vampir & VampirServer
  - interactive trace visualization, browsing, zooming
  - scalable to large trace data sizes (100 GB)
  - scalable to high parallelism (16 000 processes)

The Tools

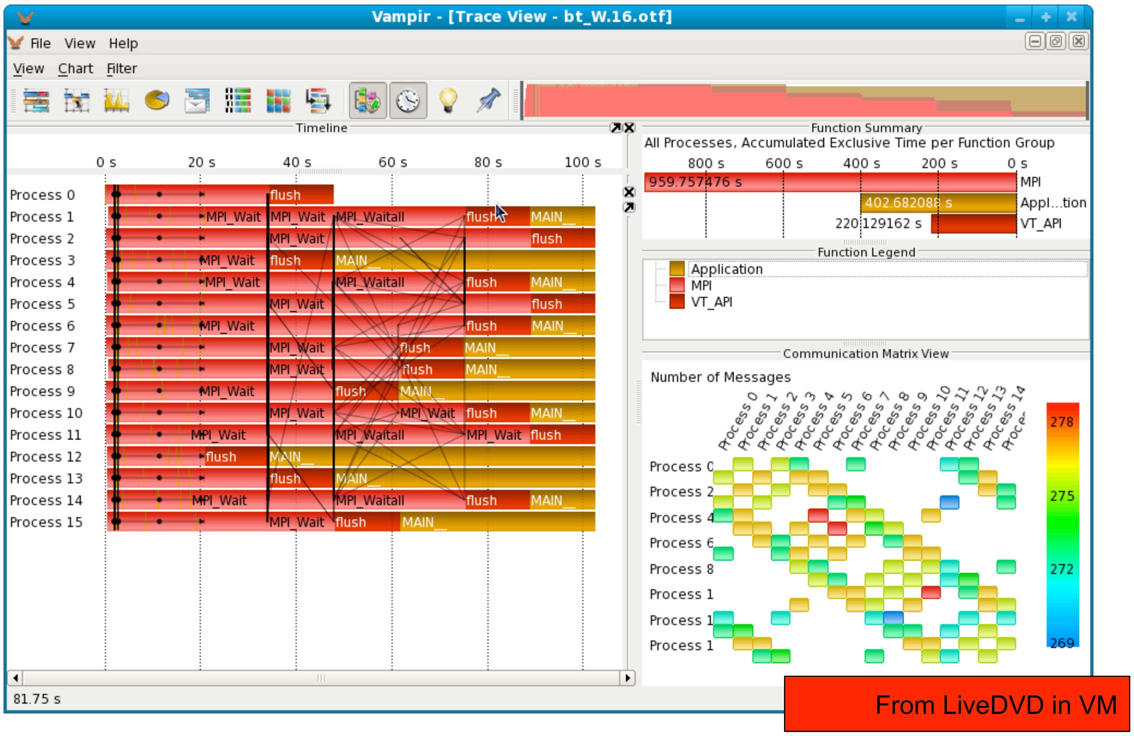
The Open Trace Format (OTF)

- Open source trace file format
- Available at http://www.tu-dresden.de/zih/otf/
- Includes libotf for reading/parsing/writing
- Two-level API:
  – High level interface for analysis tools
  – Low level interface for trace libraries
- Actively developed at TU Dresden in cooperation with the University of Oregon and the Lawrence Livermore National Laboratory (LLNL)

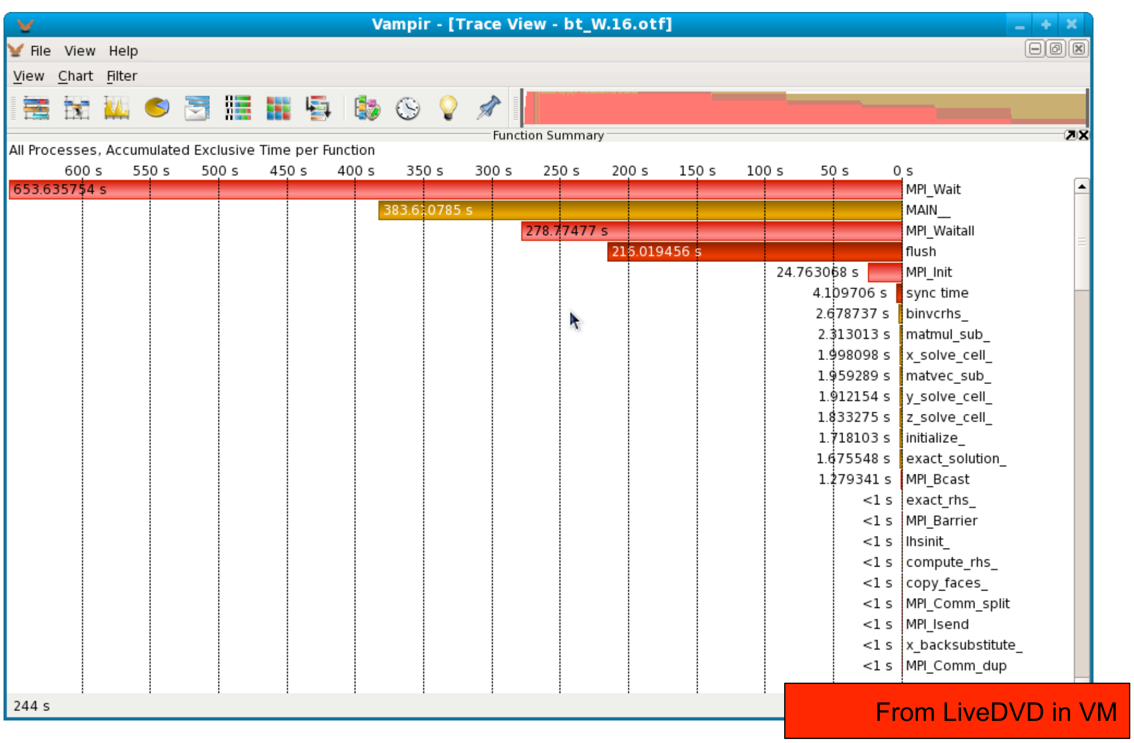
Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

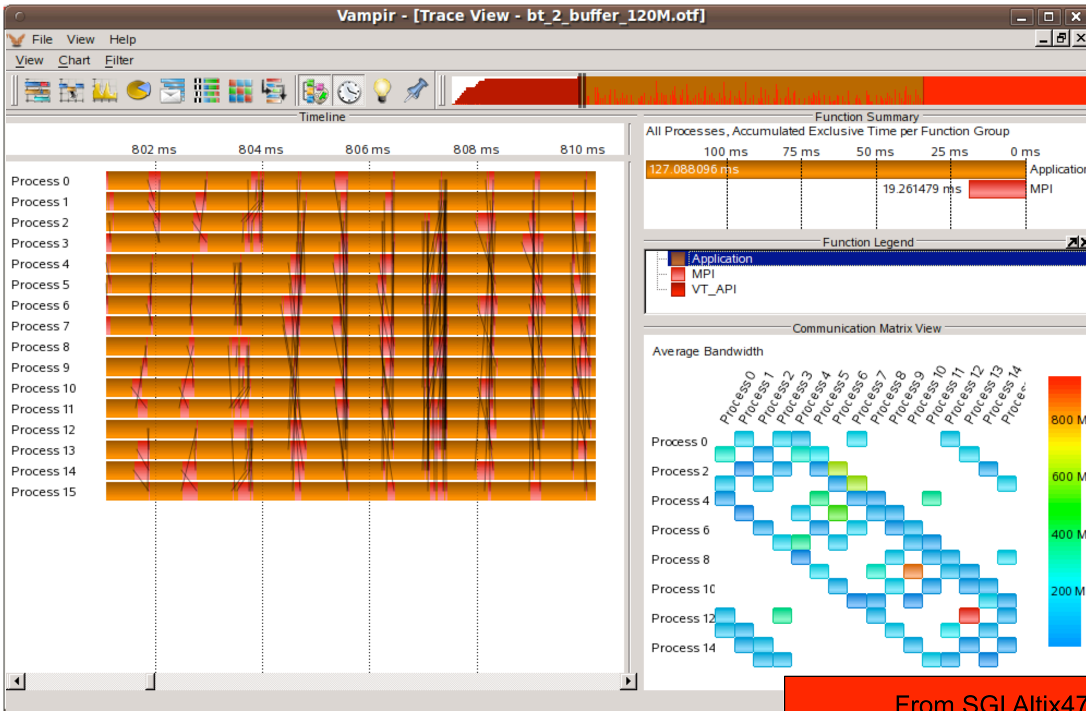
Hands-on: NPB 3.3 BT-MPI



Hands-on: NPB 3.3 BT-MPI

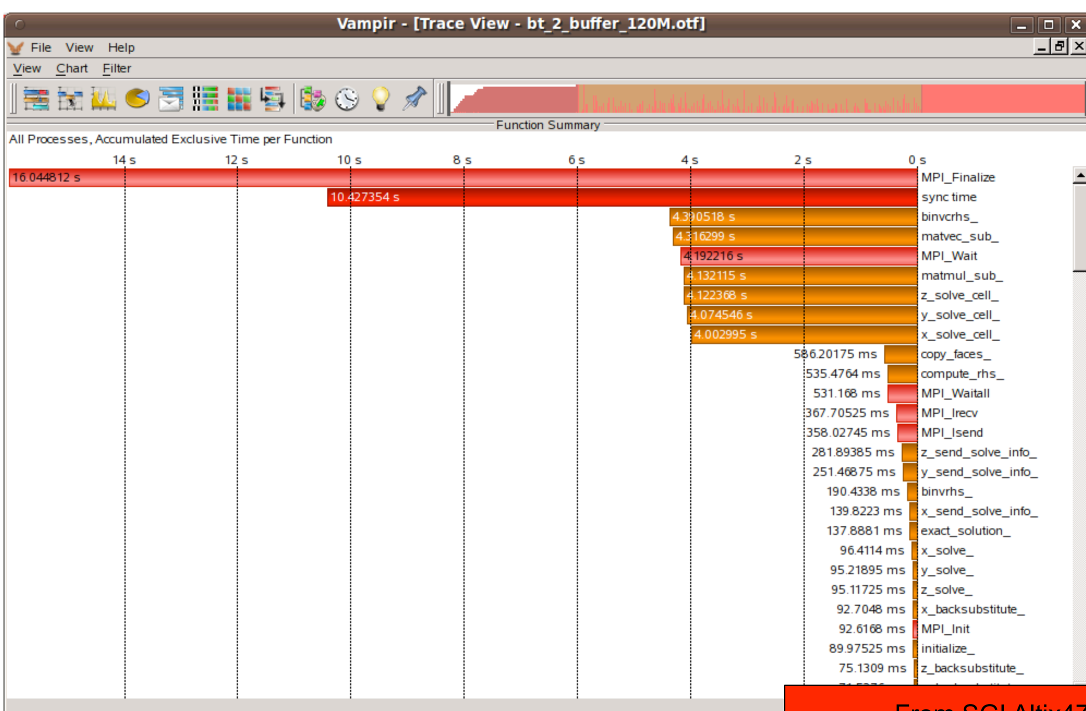


Hands-on: NPB 3.3 BT-MPI



From SGI Altix4700

Hands-on: NPB 3.3 BT-MPI



From SGI Altix4700

Function Filtering

- Filtering is one of the ways to reduce trace size
- Environment variable VT_FILTER_SPEC

```
% export VT_FILTER_SPEC=/home/user/filter.spec
```

- Filter file example:

```
my_*;test_* -- 1000
debug_* -- 0
calculate -- -1
* -- 1000000
```

- See also the vtfilter tool
  – To generate a customized filter file
  – To reduce the size of existing trace files

Function Grouping

- Grouping of related functions
  – Assign different colors
  – Highlight different activities
- Environment variable VT_GROUPS_SPEC

```
% export VT_GROUPS_SPEC = /home/user/groups.spec
```

- Group file example

```
CALC=calculate
MISC=my*;test
UNKNOWN=*
```

Hands-on: NPB 3.3 BT-MPI

- Generate filter specification file

```
% vtfilter -gen -fo filter.txt -r 10 -stats  \
  -p bt_2_buffer_120M.otf
% export VT_FILTER_SPEC=filter.txt
```

Use bt_1_initial if less than 2GB main memory.

-

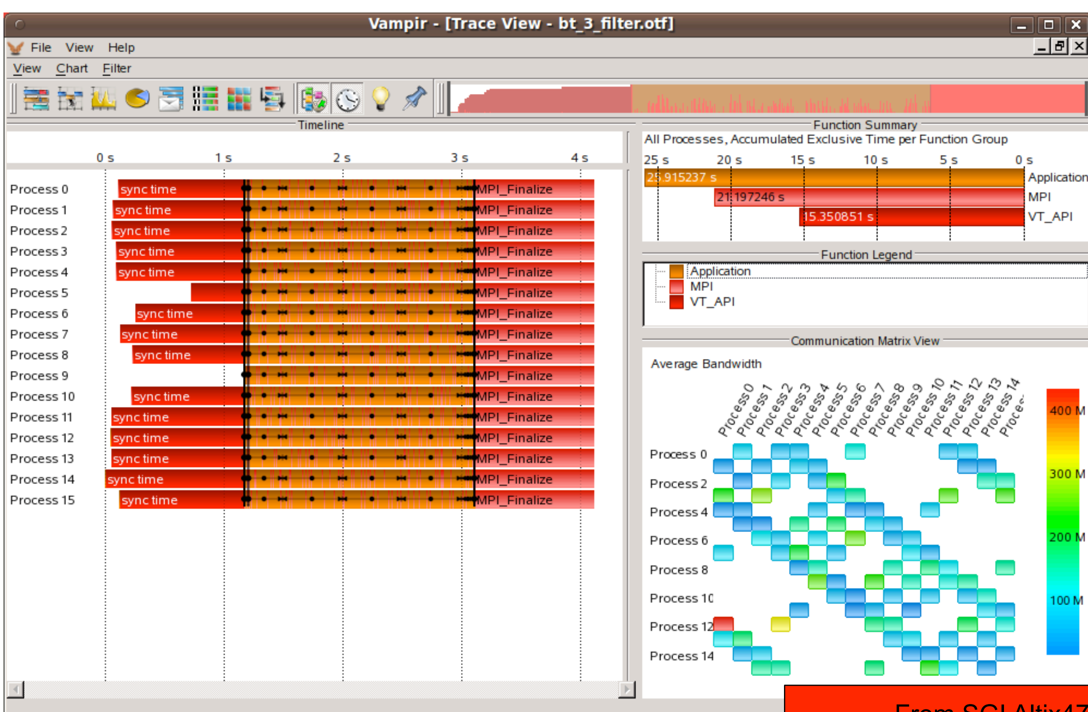
```
% export VT_FILE_PREFIX=bt_3_filter
```

Remove the old trace first on LiveDVD !

```
% mpiexec -np 16 bt_W.16
```

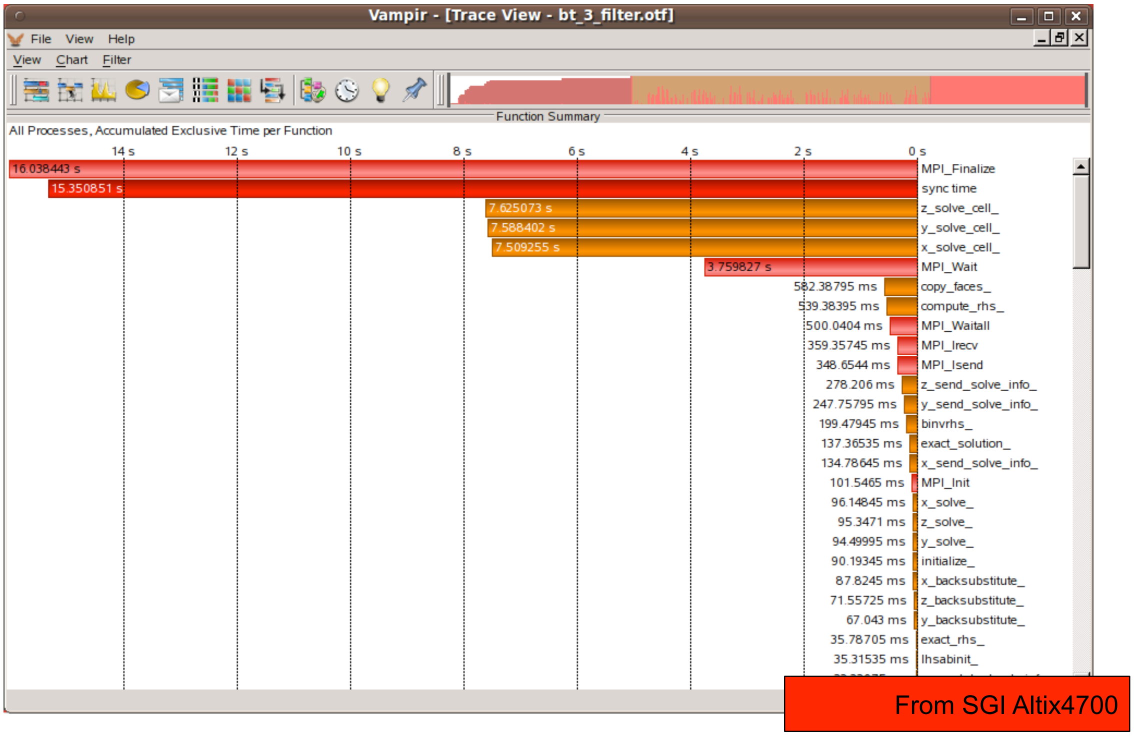
- Launch as MPI application

Hands-on: NPB 3.3 BT-MPI



From SGI Altix4700

Hands-on: NPB 3.3 BT-MPI



PAPI

- PAPI counters can be included in traces
  - If VampirTrace was build with PAPI support
  - If PAPI is available on the platform
- VT_METRICS specifies a list of PAPI counters

```
% export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```

- see also the PAPI commands
  papi_avail, papi_command_line,
  papi_event_chooser

Mem Allocation and I/O Counters

- Memory allocation counters with GNU glibc
- Intercept “malloc”, “free”, etc.
- Environment variable VT_MEMTRACE

```
% export VT_MEMTRACE = yes
```

- I/O counters
- Intercept standard I/O calls “open”, “read”, etc.

```
% export VT_IOTRACE = yes
```

- Environment variable VT_IOTRACE

Hands-on: NPB 3.3 BT-MPI

- Record PAPI hardware counters

```
% papi_avail
% papi_event_chooser PRESET PAPI_FP_OPS
% export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```

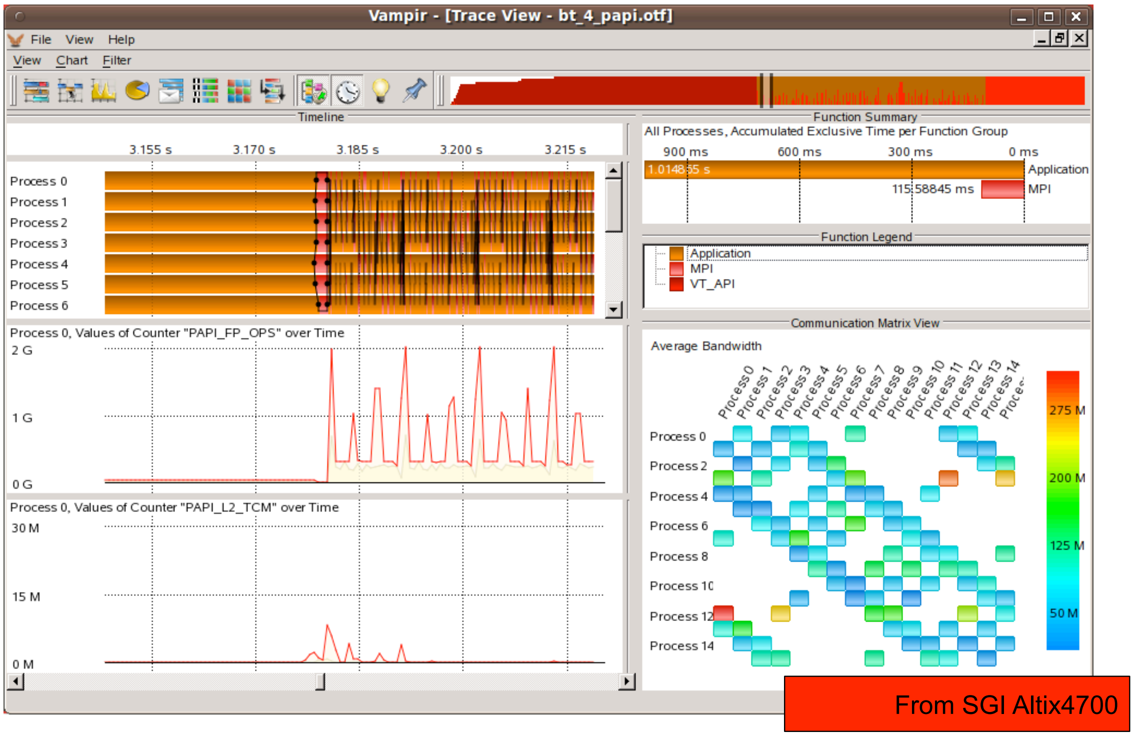
- Set a new file prefix

```
% export VT_FILE_PREFIX=bt_4_papi
```

Launch as MPI application

```
% mpiexec -np 16 bt_W.16
```

Hands-on: NPB 3.3 BT-MPI



VampirTrace Run-Time Options (1)

VT_PFORM_GDIR	Directory for final trace files
VT_PFORM_LDIR	Directory for intermediate files
VT_FILE_PREFIX	Trace file name
VT_BUFFER_SIZE	Internal trace buffer size
VT_MAX_FLUSHES	Max number of buffer flushes
VT_MPITRACE	Enable MPI tracing
VT_MPICHECK	Enable MPI checking

VampirTrace Run-Time Options (2)

VT_FILTER_SPEC	Name of filter definition file
VT_GROUPS_SPEC	Name of group definition file
VT_METRICS	Counter selection
VT_MEMTRACE	Enable mem allocation tracing
VT_IOTRACE	Enable I/O tracing

Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Finding Bottlenecks

- Vampir trace visualization
  - several displays with many options
  - identify essential parts of an application (initialization, main iteration, I/O, finalization)
  - identify important components of the code (serial computation, MPI P2P, collective MPI, OpenMP)
  - make a hypothesis about performance problems
  - consider application's internal workings if known
  - select the appropriate displays
  - use statistic displays in conjunction with timelines

Finding Bottlenecks

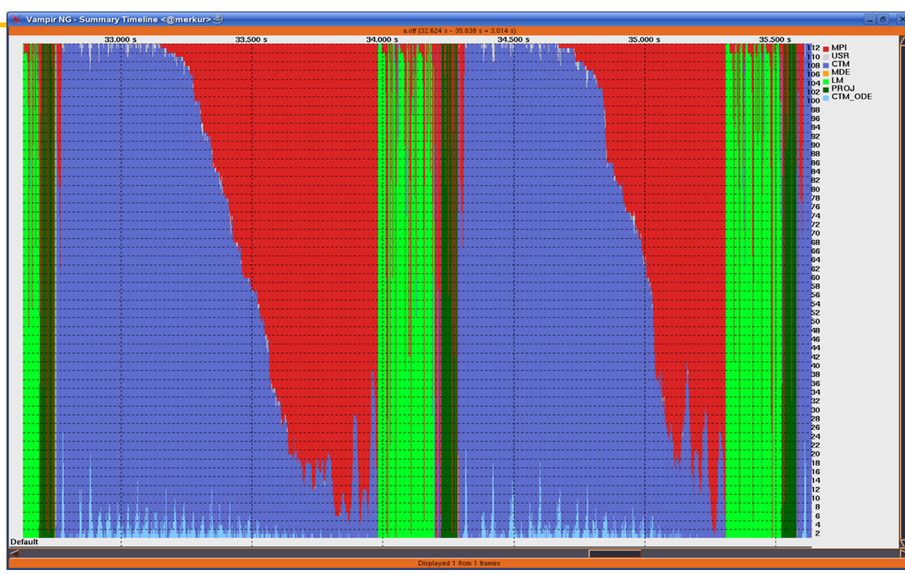
- Communication
- Computation
- Memory, I/O, etc
- Tracing itself

Bottlenecks in Communication

- communication as such (dominating over computation)
- late sender, late receiver
- point-to-point messages instead of collective communication
- unmatched messages
- overcharge of MPI's buffers
- bursts of large messages (bandwidth)
- frequent short messages (latency)
- unnecessary synchronization (barrier)

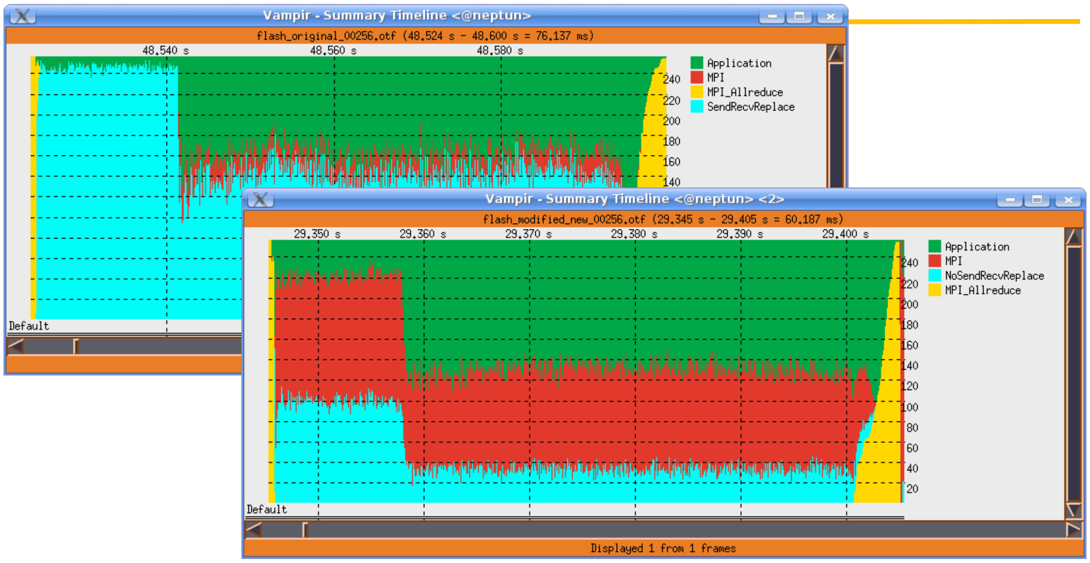
The above usually result in a high MPI time share

Bottlenecks in Communication



Prevalent communication

Bottlenecks in Communication

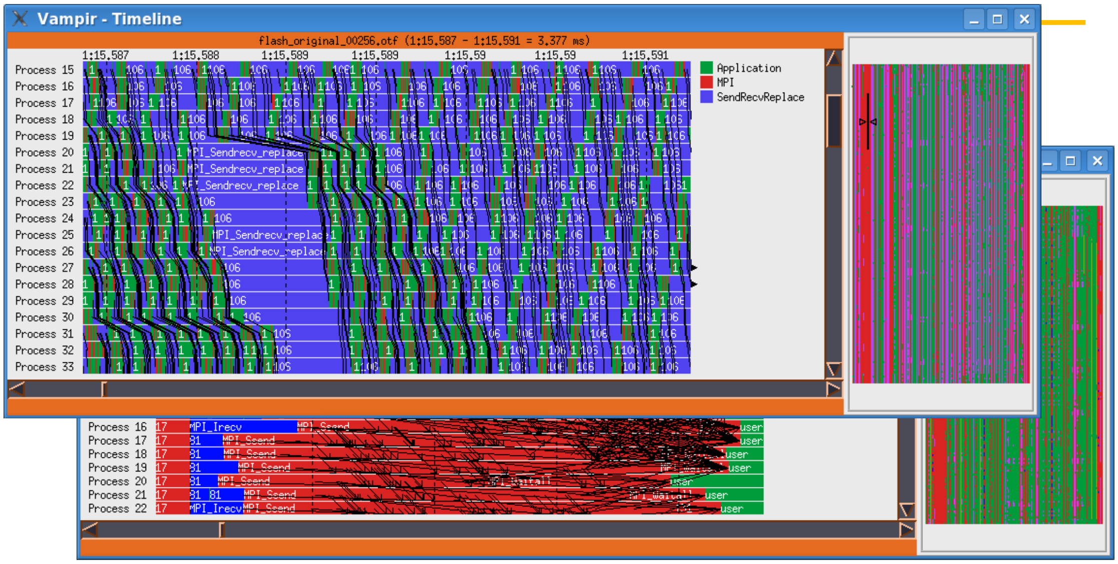


Prevalent communication: MPI_Allreduce

Bottlenecks in Communication

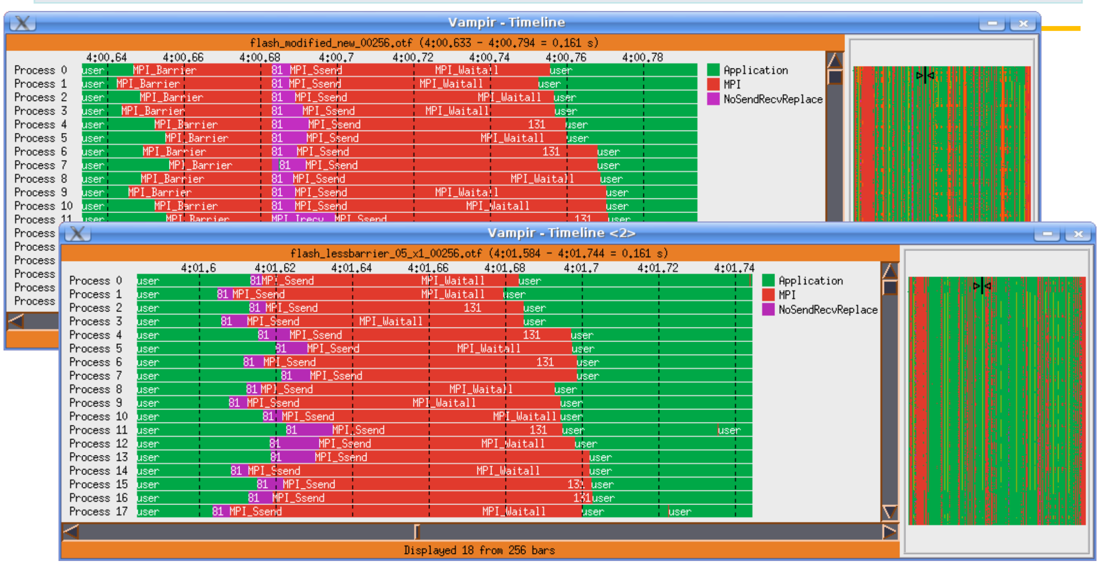
Prevalent communication: Timeline view

Bottlenecks in Communication



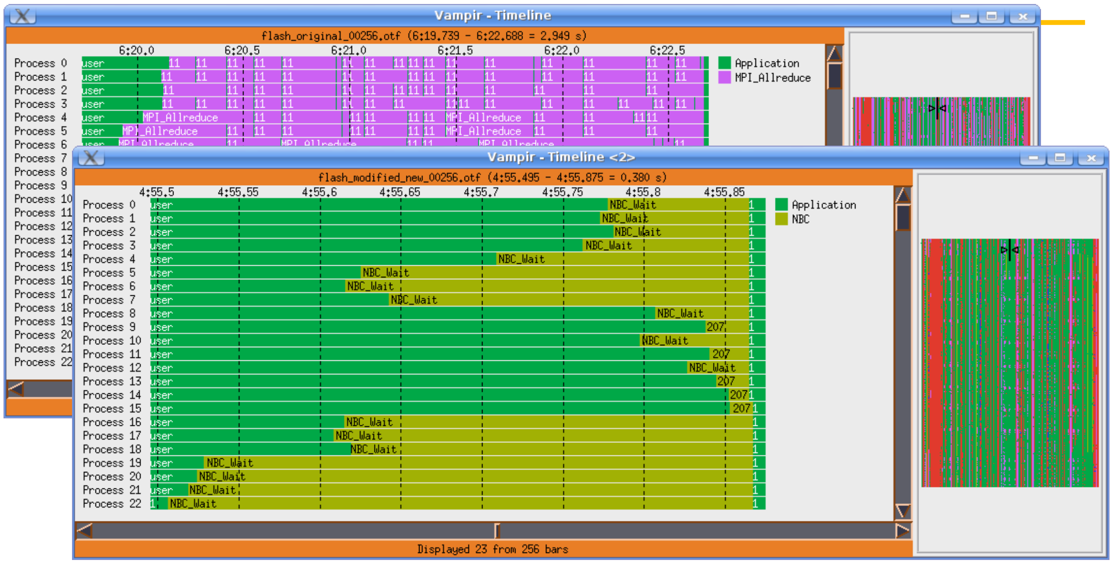
Propagated Delays in MPI_SendReceiveReplace

Bottlenecks in Communication



Unnecessary MPI_Barriers

Bottlenecks in Communication

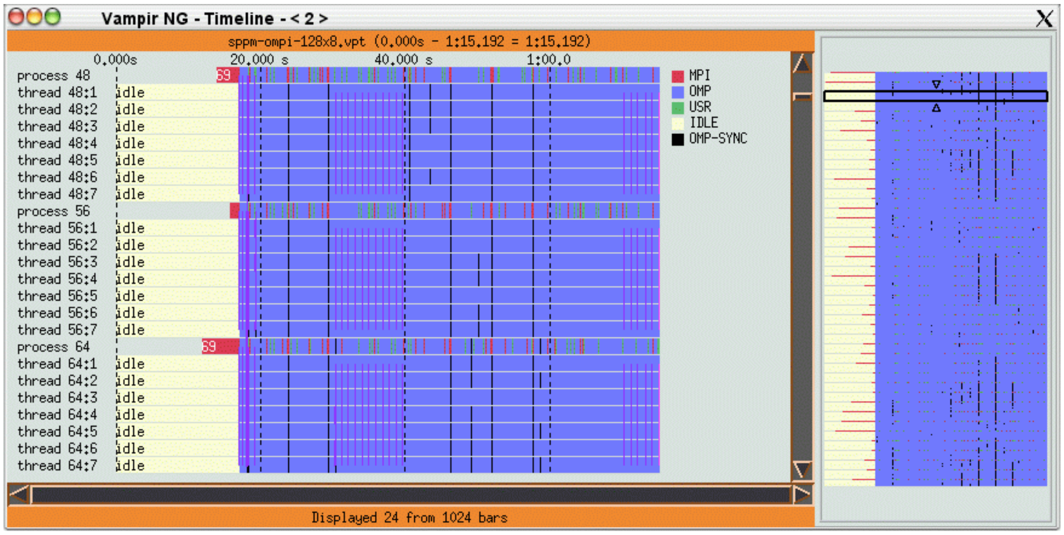


Patterns of successive MPI_Allreduce Calls

Further Bottlenecks

- Unbalanced computation
  - single late comer
- Strictly serial parts of program
  - idle processes/threads
- Very frequent tiny function calls
- Sparse loops

Further Bottlenecks

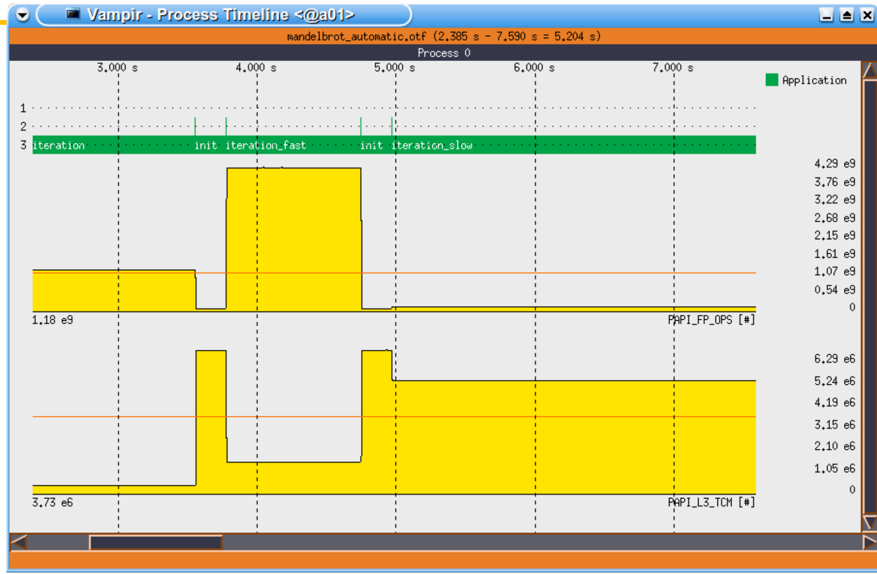


Idle OpenMP threads

Bottlenecks in Computation

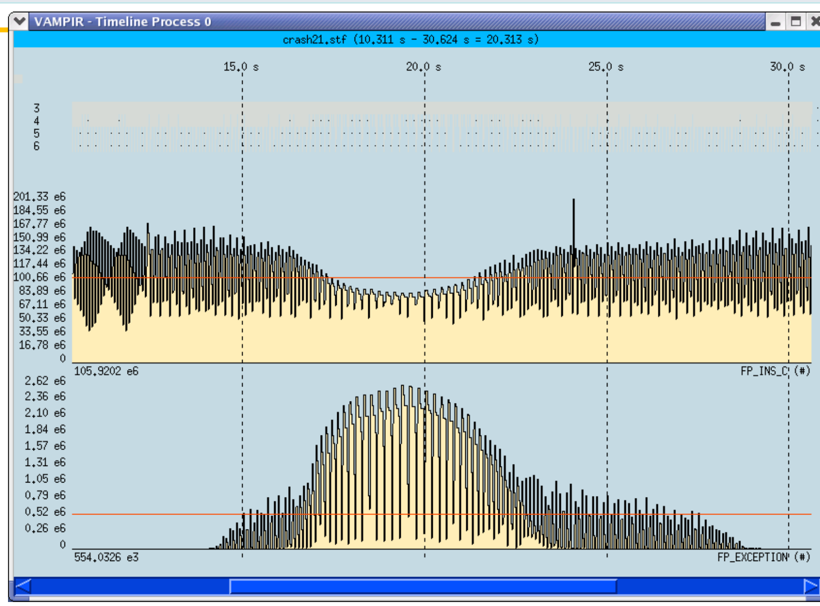
- Memory bound computation
  - inefficient L1/L2/L3 cache usage, TLB misses
  - detectable via HW performance counters
- I/O bound computation
  - slow input/output
  - sequential I/O with a single process
  - I/O load imbalance
- Exception handling

Bottlenecks in Computation



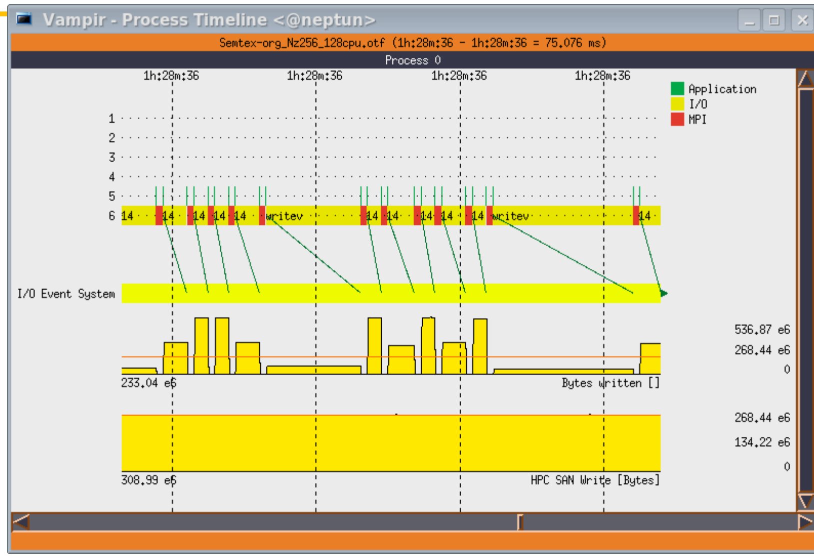
Low FP rate due to heavy cache misses

Bottlenecks in Computation



Low FP rate due to heavy FP exceptions

Bottlenecks in Computation

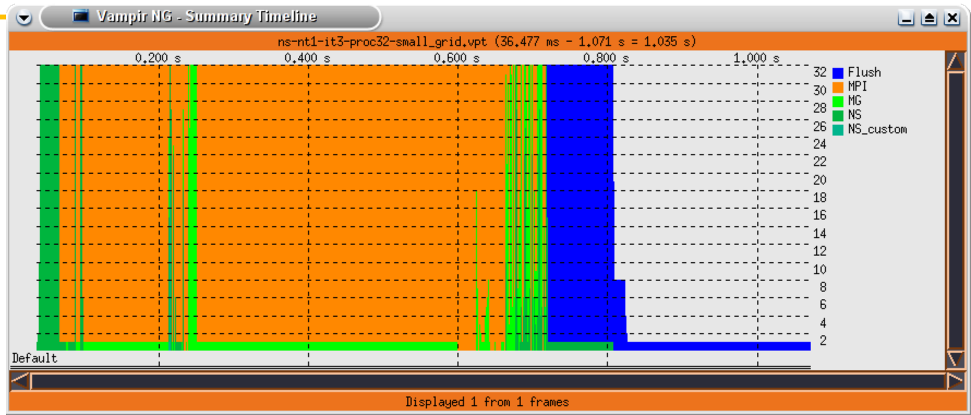


Irregular slow I/O operations

Effects due to Tracing Itself

- Measurement overhead
  - esp. grave for tiny function calls
  - solve with selective instrumentation
- Frequent/asynchronous trace buffer flushes
- Too many concurrent counters
- Heisenbugs

Effects due to Tracing Itself



- Buffer flushes are marked explicitly
- Harmless at the end of a trace

Overview

- Introduction
- Vampir Displays
- VampirTrace Instrumentation & Measurement
- Hands-on
  - First Steps
  - Buffer Management
  - Filtering and Grouping
  - PAPI Hardware Performance Counters
- Finding Performance Bottlenecks
- Conclusion & Outlook

Conclusions

- Performance analysis is very important
- Use tools!
- Do not spend effort in DIY solutions or printf-debugging
- Use tracing with some precautions
  - Overhead
  - Data volume
- Let us know about problems and feature wishes: vampirsupport@zih.tu-dresden.de

VAMPIR

Vampir and VampirTraces are

available at http://www.vampir.eu and

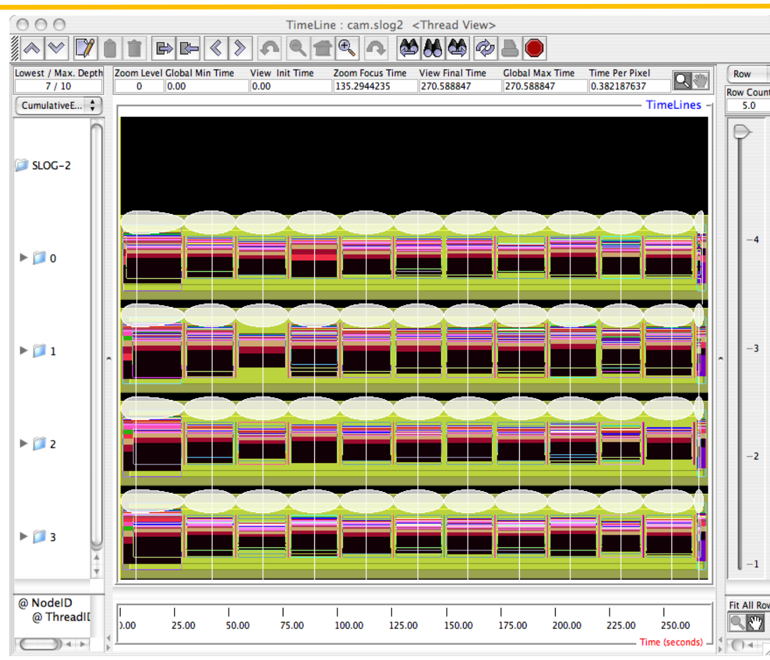
http://www.tu-dresden.de/zih/vampirtrace/ ,

support via vampirsupport@zih.tu-dresden.de

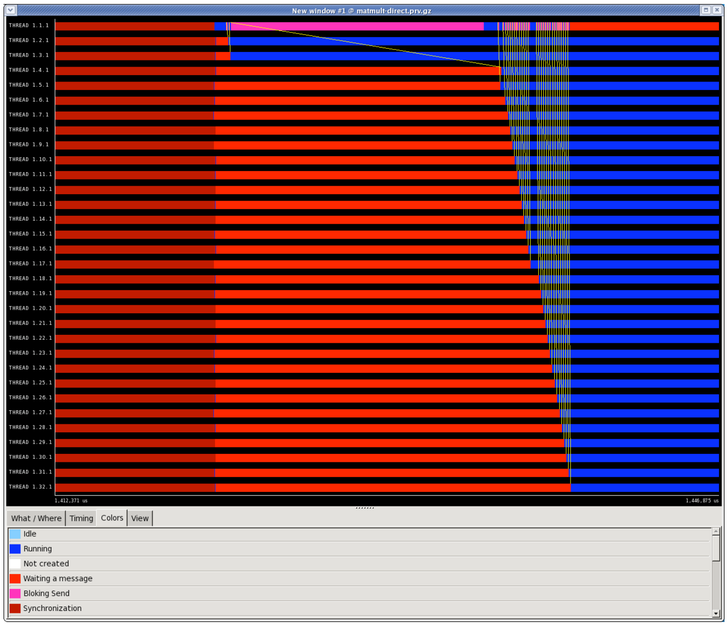
Jumpshot

- http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm
- Developed at Argonne National Laboratory as part of the MPICH project
  - Also works with other MPI implementations
  - Jumpshot is bundled with the TAU package
- Java-based tracefile visualization tool for postmortem performance analysis of MPI programs
- Latest version is Jumpshot-4 for SLOG-2 format
  - Scalable level of detail support
  - Timeline and histogram views
  - Scrolling and zooming
  - Search/scan facility

Jumpshot



ParaVer [BSC]



Part V: KOJAK/Scalasca

scalasca

Overview

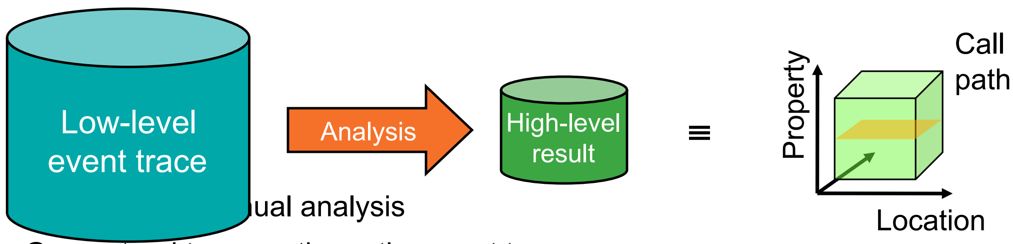
- Introduction
  - Motivation for automatic trace analysis
- Scalasca components and usage
  - instrumentation
  - measurement collection & automated analysis
  - analysis report exploration
- Demonstration
- Summary

Motivation

- Tracing offers critical insight into temporal behaviour of parallel execution unavailable from summarization
  - Inefficiencies manifest as wait states and imbalance
- Trace sizes proportional to number of processes/threads
  - as well as length of measurement and depth of detail
- Large-scale parallel traces must be carefully managed
  - minimization/elimination of disruptive file I/O
  - efficient parallel analysis of traces
  - effective hierarchical/graphical analysis presentation
- Simplification and ease-of-use
  - Automation of search for and classification of event patterns
  - Integration with trace visualizers to examine key instances

Automatic Trace Analysis

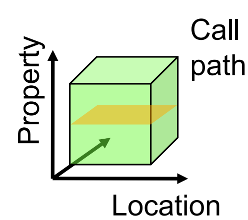
- Idea:
  - Automatic search for *patterns* of inefficient behaviour
  - Classification of behaviour
  - Quantification of significance



- ual analysis
- Guaranteed to cover the entire event trace

CUBE Result Browser

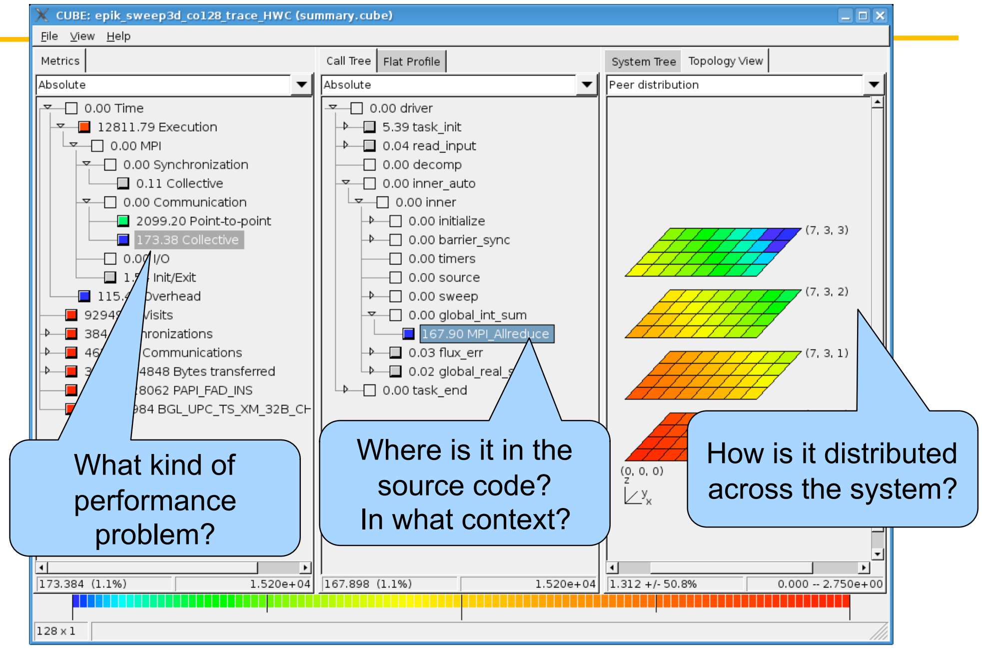
- Representation of results (severity matrix) along three hierarchical axes
  - Performance property
  - Call tree path
  - System location



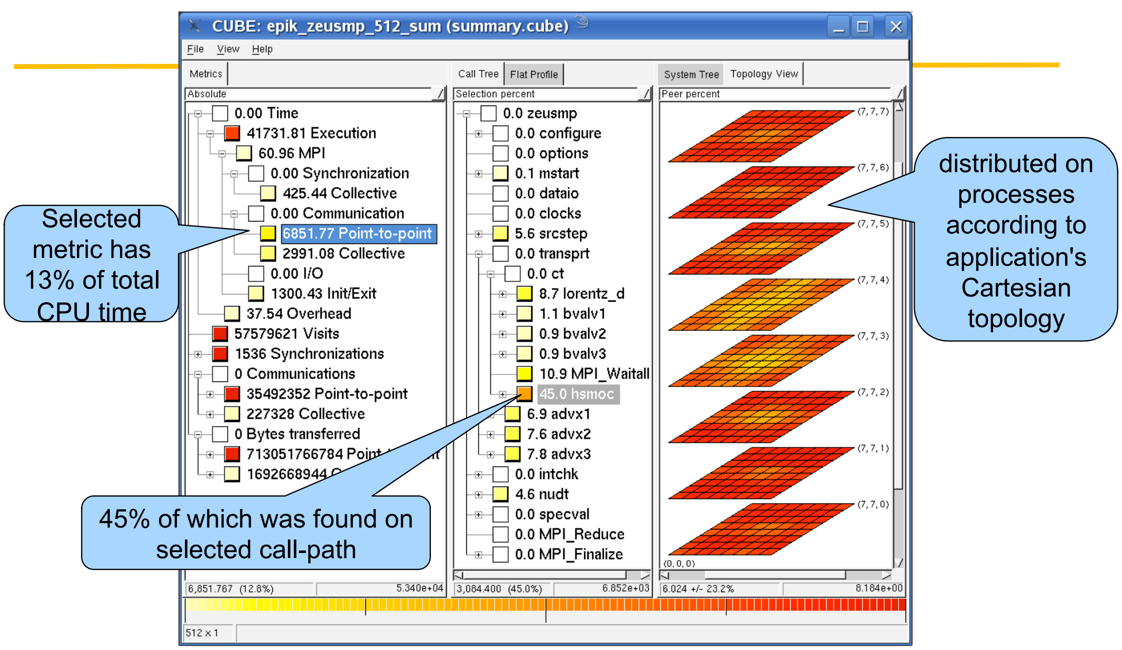
- Three coupled tree browsers

- Each node displays severity
  - As colour: for easy identification of hotspots
  - As value: for precise comparison
  - Inclusive value when closed or exclusive when expanded
  - Customizable via display mode

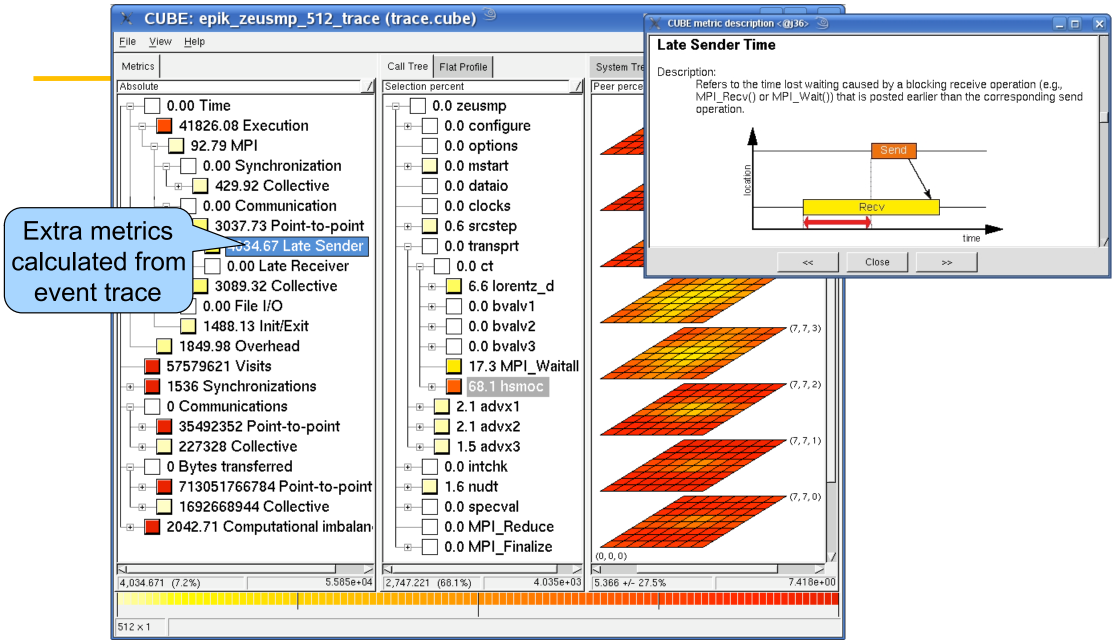
Basic Analysis Presentation



Summary Profile Analysis



Trace Pattern Analysis

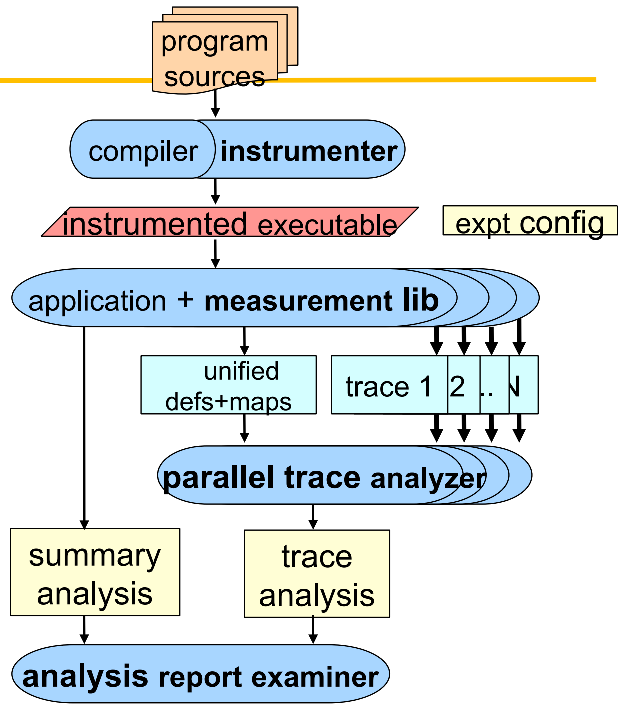


Analysis Methodology

- Instrumentation of application executable and libraries
  - automatic MPI, OpenMP and function instrumentation
  - complementary manual region and phase instrumentation
- Execution of instrumented executable under control of configurable measurement collection & analysis nexus
  - commence from scalable runtime summary
    - identify excess instrumentation and trace buffer requirements
  - target tracing where it is most productive (and practical)
    - analyze traces using same resources as measurement
- Interactive analysis report exploration and algebra
  - examine severities and their locations
  - combine, compare and process reports
- Refine and repeat as necessary

Scalasca Components

- Scalasca instrumenter = SKIN
- Scalasca measurement collector & analyzer = SCAN
- Scalasca analysis report examiner = SQUARE



Scalasca unified command: scalasca

- Run without action argument for basic usage info
  % scalasca
  usage: scalasca [-v][-n] {action}
  1. prepare application objects and executable for measurement:
     scalasca *-instrument* <compile-or-link-command> # ***skin***
  2. run application under control of measurement system:
     scalasca *-analyze* <application-launch-command> # ***scan***
  3. interactively explore measurement analysis report:
     scalasca *-examine* <experiment-archive|report> # ***square***
- Simply a convenience wrapper for action commands

Scalasca instrumenter: skin

- Usage: scalasca -instrument [opts] $CC ...
  - **scalasca -instrument -user** mpicc -fast -c bar.c
  - ***skin*** mpif90 -Openmp -o foobar -fast foo.c bar.o -lm
- Processes source modules during compile & augments link with measurement library
  - Configures automatic function instrumentation capability of native compiler (if available)
    - All functions in source module(s) are instrumented
  - **[–pomp]** option enables processing of POMP directives
    - Optional manual source annotation of functions & regions
    - Replaces automatic function instrumentation
  - [–user] activates EPIK user-annotation API

Scalasca collector & analyzer: scan

- Usage: scalasca -analyze [opts] <launch command>
  - ***scan*** *[opts]* [launcher [args]] [target [target-args]]
- Prepares & runs measurement collection, with follow-on trace analysis (if appropriate)
  - [-n] preview without executing launches
  - [-s] enables runtime summarization [default]
  - [-t] enables trace collection & automatic pattern analysis
  - determines NP and/or NT (number of processes & threads) and MODE=vn|co|dual|smp (where appropriate)
  - names default measurement experiment archive `epik_$(TARGET)_$(MODE)$(NP)x$(NT)_[sum|trace]`
  - [-f filter] specifies file listing functions not to be measured
  - [-m metric1:metric2:...] includes hardware counter metrics

Scalasca analysis report explorer: square

- Usage: scalasca -examine <epik_archive | cubefile >
  - **scalasca -examine** epik_sweep3d_co32_trace
  - ***square*** epik_sweep3d_co32_trace/summary.cube
- Prepares & presents final analysis report
  - Checks EPIK archive directory for cubefiles
  - Remaps primitive initial analysis report(s) into refined formal report(s) with enriched metrics & metric hierarchies
    - epitome.cube -> summary.cube
    - scout.cube -> trace.cube
  - Presents refined report in CUBE3 browser
    - Trace analysis shown in preference to summary analysis
    - Additional reports can be loaded via File/Open menu

EPIK experiment archive

- Directory created by measurement library
  - Measurement aborts if archive already exists!
- Contains all files related to measurement
  - Measurement & analysis logs (epik.log, scout.log, etc.)
  - Primitive analysis reports (epitome.cube, scout.cube)
  - Refined analysis reports (summary.cube, trace.cube)
  - Process trace datafiles (ELG/*)
  - Unified definitions & map data (epik.esd, epik.map)
  - Miscellaneous (epik.conf, epik.filt, epik.path)

EPIK measurement configuration

- ***epik_conf*** reports current configuration
  - logged in measurement archive as epik.conf
- Read from EPIK.CONF file(s)
  - System default: $SCALASCA_DIR/doc
  - Directory specified with EPIK_CONF environment variable [defaults to "."]
- Over-ridden by environment variables
  - with same names as configuration file variables
- Over-ridden by ***scan*** command-line settings

Trace collection & analysis issues

- Process rank trace too large for trace collection buffers
  - Results in intermediate trace buffer flushes
    (with remainder flushed at measurement finalization)
  - Serious measurement perturbation!
- Irrelevant functions encumber analysis
  - Undesirable complexity and processing slowdown
  - Parallel trace analyzer requires memory more than twice largest rank
    (uncompressed) trace size to load entire trace
- Options
  - enlarge trace buffer size: ELG_BUFFER_SIZE
    - ***cube3_score*** utility provides estimate from summary
  - remove selected function instrumentation
  - specific function measurement filter (if supported!)

Selective instrumentation/measurement

- Unimportant functions can be determined from summary analysis report
  - form leaves of callpath-tree (w/o MPI)
  - negligible proportion of (exclusive) execution time
  - high proportion of (exclusive) visit count
  - ***cube3_score -r*** provides region breakdown & classification
    - MPI, USR (no MPI), COM (combined/intermediate)
- Eliminating pure user (USR) regions reduces overheads
  - runtime processing, storage & analysis
- Makes them “invisible” in the analysis
  - logically become part of their calling functions
    (as if they were in-lined by an optimizing compiler!)

Scalasca runtime summarization

- Event measurements accumulated and summarized for each call-path during runtime execution
- Summary report produced at finalization
- Provides overview of measured execution
  - contains call-path Visit frequency, Time, and MPI message statistics
  - *plus* optional hardware counter metrics
  - size independent of length of execution
- Scales to long execution measurements

Scalasca trace analysis

- Trace analysis based on parallel replay
  – enables scalability to thousands of processes
  – however, only suited to relatively brief measurements!
- Extends summary metric analysis
  – Summary can help configure selective tracing
- Allows execution performance properties to be more accurately determined and refined
- Can be combined with complementary runtime summary analysis
  – avoiding storage/processing overhead of hardware counter metrics in traces via direct summarization

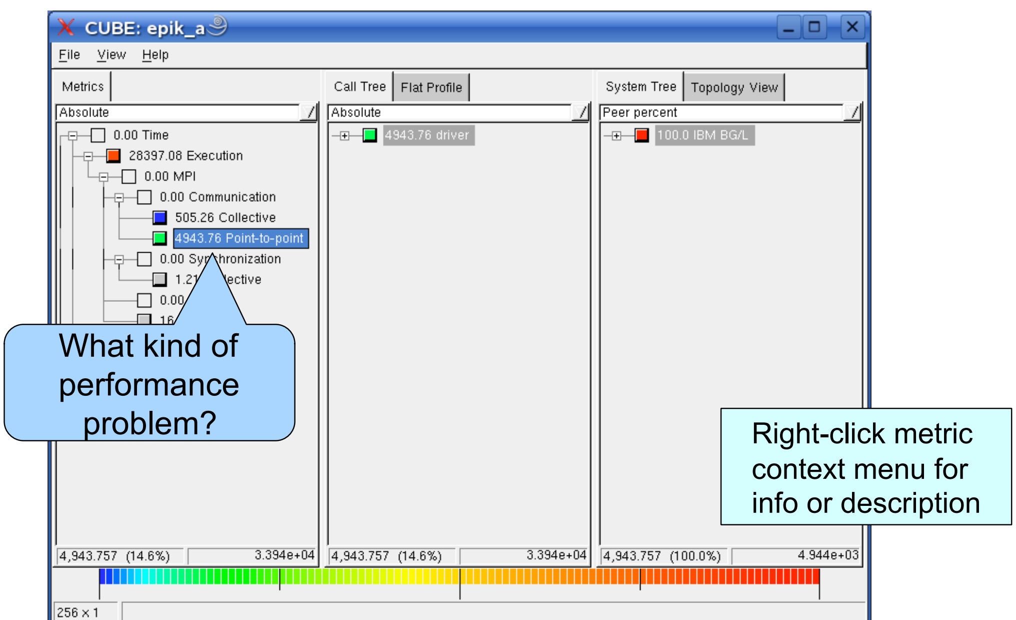
Measurement support

- OpenMP compilers
  – GCC
  – IBM XL
  – Intel
  – Pathscale
  – PGI
  – Sun Studio
  – ...
- Supported functionality varies by language, version & system
- MPI libraries
  – MPICH 1 & 2
  – OpenMPI
  – Intel-MPI
  – IBM POE & BlueGene
  – Cray XT
  – Sun HPC ClusterTools
  – SGI MPToolkit
  – SiCortex MPI
  – Scali-MPI
  – HP-MPI
  – LAM

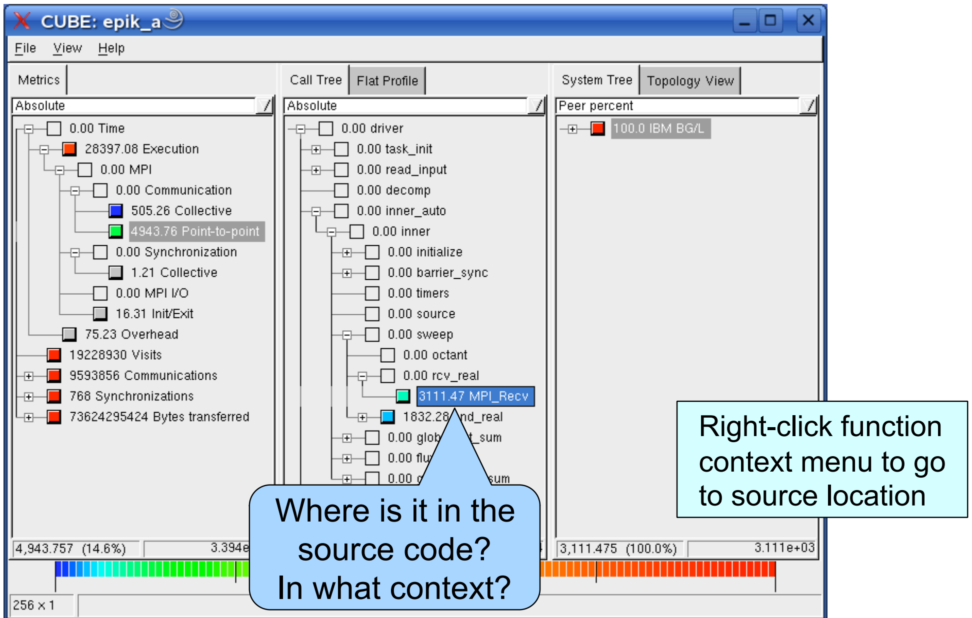
Basic use of Scalasca

- Automatic function instrumentation
  - Supported by most but not all compilers!
- Summary measurement experiment
- Summary analysis report exploration
- Trace collection & analysis experiment
- Trace pattern analysis report exploration

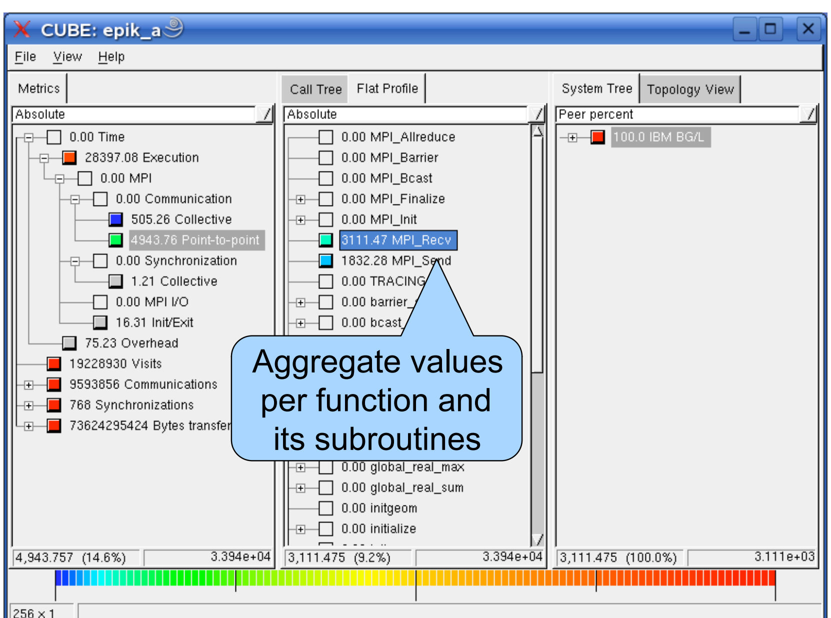
CUBE metrics dimension



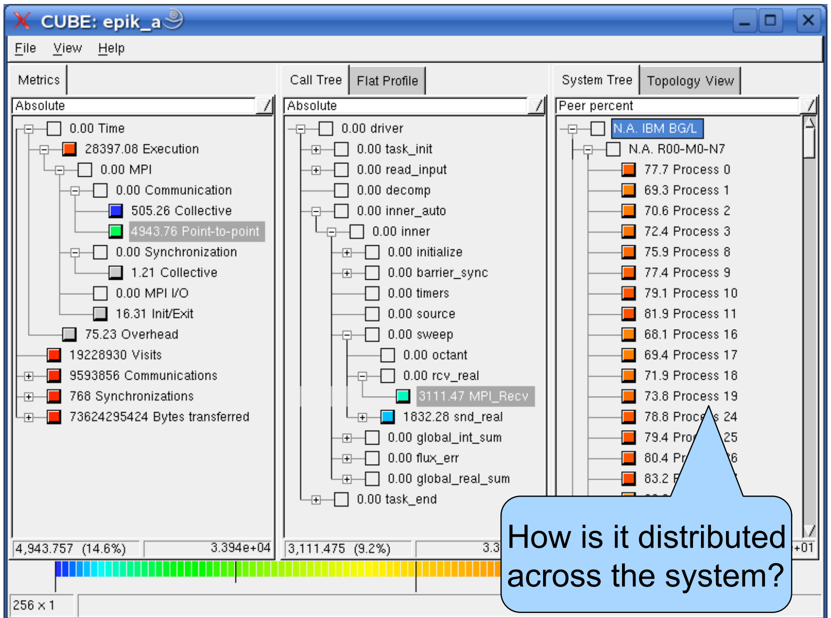
CUBE call tree dimension



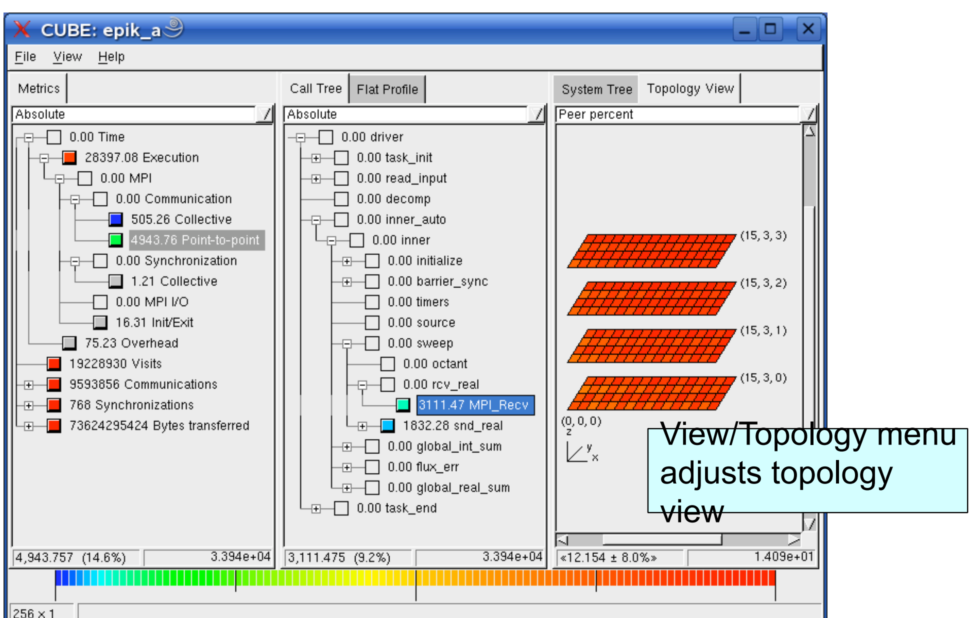
Alternative: Flat profile



System tree dimension



Alternative: Topology display



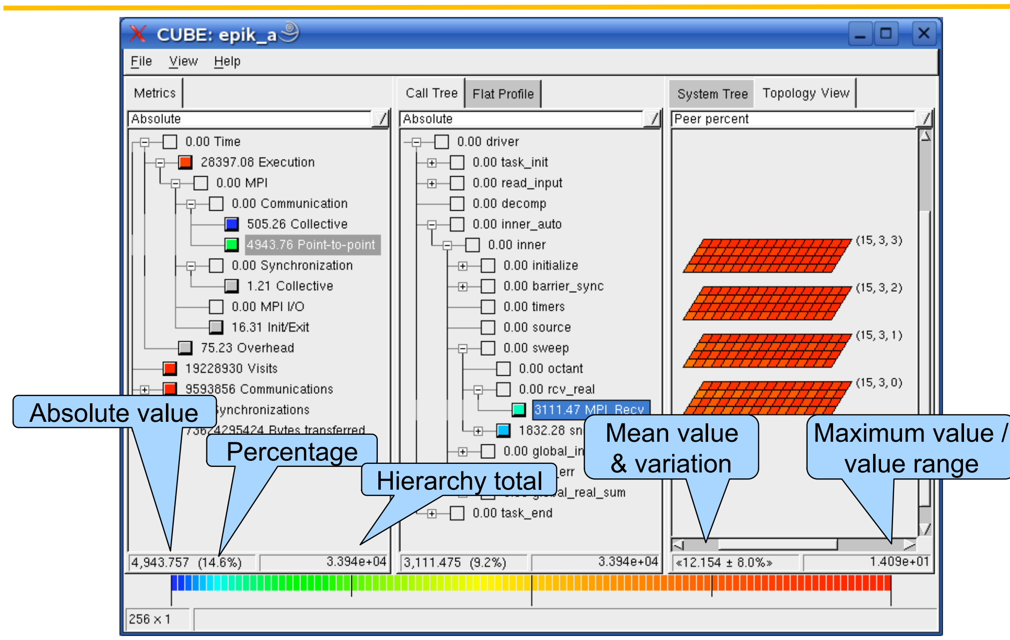
Topology display

- Topology information is recorded for
  - the hardware (supported on some systems)
  - MPI topologies (e.g., `MPI_Cart_create()`)
  - user-defined virtual topologies (under construction)

- Advantage
  - Better scalability than text-based system tree

- Restriction
  - Currently supports only 1D, 2D and 3D Cartesian topologies

Status fields



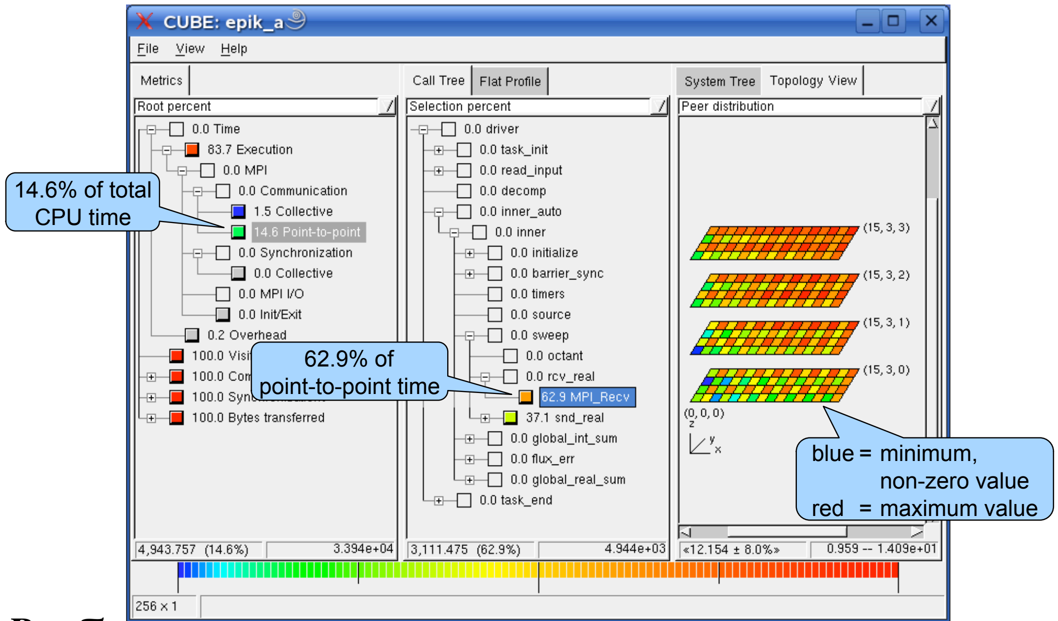
Display modes

- Absolute
  - Absolute values in seconds/number of occurrences
- Root percent
  - Percentage relative to the root node of the hierarchy
- External percent
  - Similar to “Root percent”, but relative to another data set
- Selection percent
  - Percentage relative to the node selected in the neighbouring column on the left

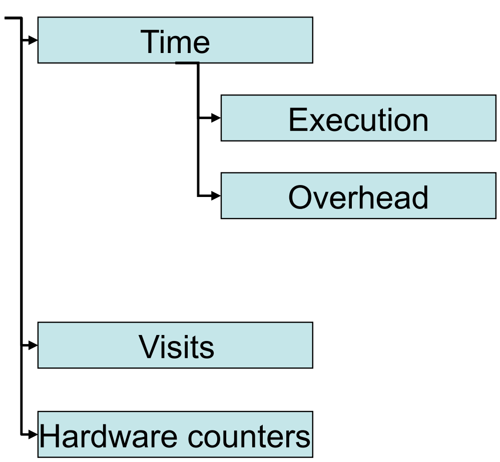
Display modes (system tree/topology only)

- Peer percent
  - Percentage relative to maximum of peer values (all values of the current leaf level)
- Peer distribution
  - Percentage relative to maximum and non-zero minimum of peer values

Display mode example

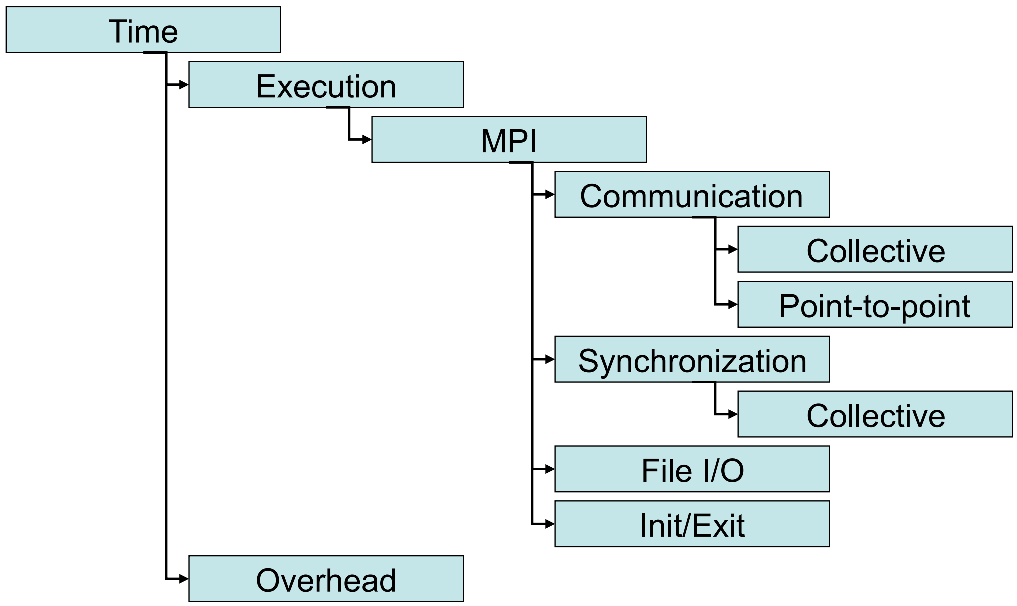


Generic metrics



- Time: Total CPU allocation time
- Execution: Execution time without overhead
- Overhead: Time spent in activities related to measurement (not including dilation per instrumented routine/region!)
- Visits: Number of times a routine/region was executed
- Hardware counters: Aggregated counter values for each routine/region

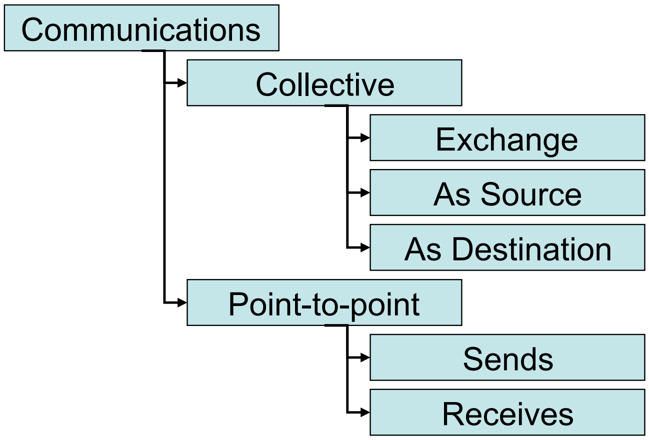
MPI Time hierarchy



MPI Time hierarchy (cont.)

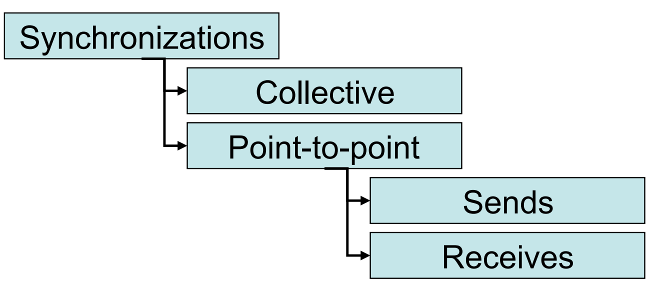
Time	Total CPU allocation time
Execution	Execution time without overhead
Overhead	Time spent in tasks related to measurement (not including dilation from instrumentation!)
MPI	Time spent in pre-instrumented MPI functions
Communication	Time spent in MPI communication calls, subdivided into collective and point-to-point
Synchronization	Time spent in MPI synchronization calls
File I/O	Time spent in MPI file I/O functions
Init/Exit	Time spent in `MPI_Init`() and `MPI_Finalize`()

MPI Communications hierarchy



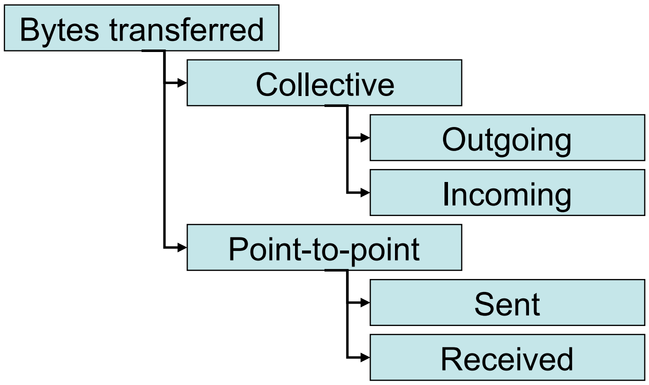
- Provides the number of calls to an MPI communication function of the corresponding class
- Zero-sized message transfers are considered *synchronization*!

MPI Synchronizations hierarchy



- Provides the number of calls to an MPI synchronization function of the corresponding class
- MPI synchronizations include zero-sized message transfers!

MPI Bytes transferred hierarchy



- Provides the number of bytes transferred by an MPI communication function of the corresponding class

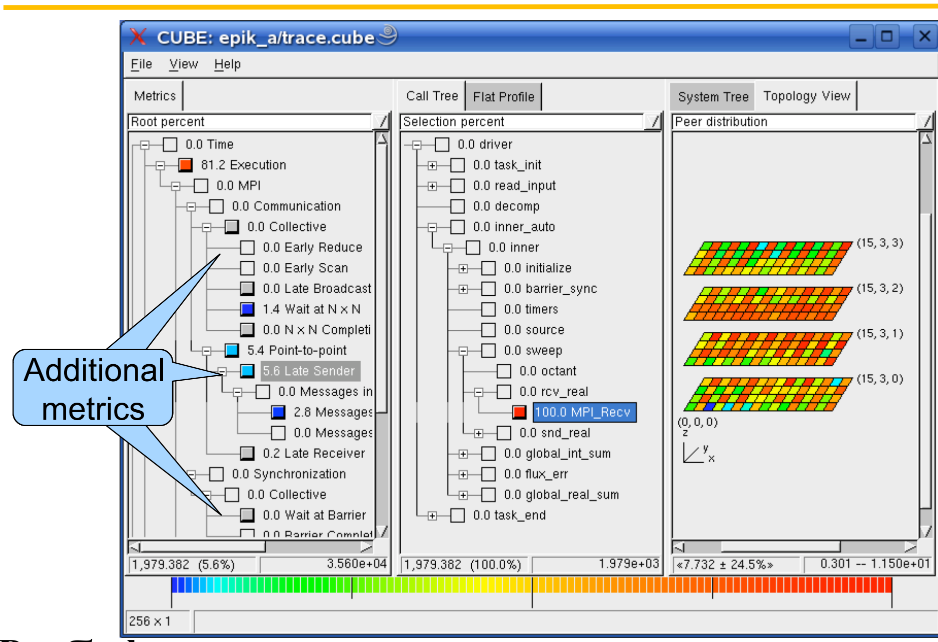
Combined trace collection & analysis

- Modify jobscript
  - Use “scan **-t**” (or set EPK_TRACE=1)
  - Trace experiment EPK_TITLE set to `$(TARGET)_$(MODE)$(NP)_trace`
  - Creates new experiment archive directory `./epik_$(EPK_TITLE)`
  - Trace unified & buffers flushed at measurement finalization
  - Automatic trace pattern analysis immediately follows
- Explore trace pattern analysis report using CUBE

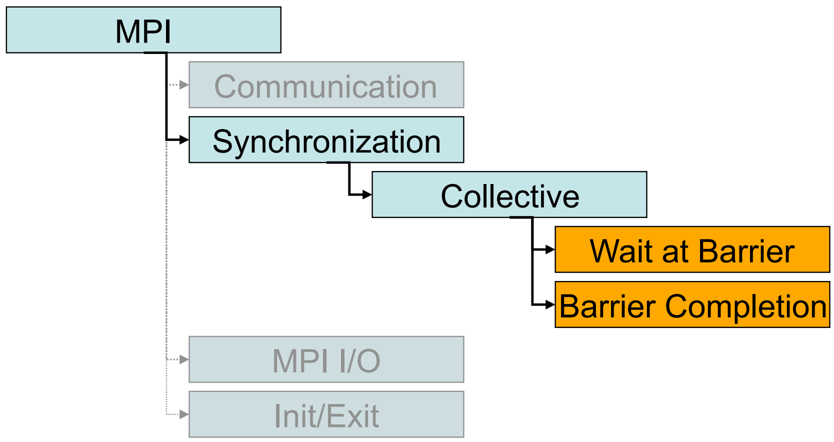
Trace analysis output example

```
SCOUT

Analyzing experiment archive ./epik_sweep3d_co32_trace

Reading definition files  ... done
Reading event trace files ... done
Preprocessing             ... done
Analyzing event traces    ... done
Writing report            ... done

Total processing time: 4.083s
Total number of events: 5206596
Max. memory usage: 15.453 MB
```

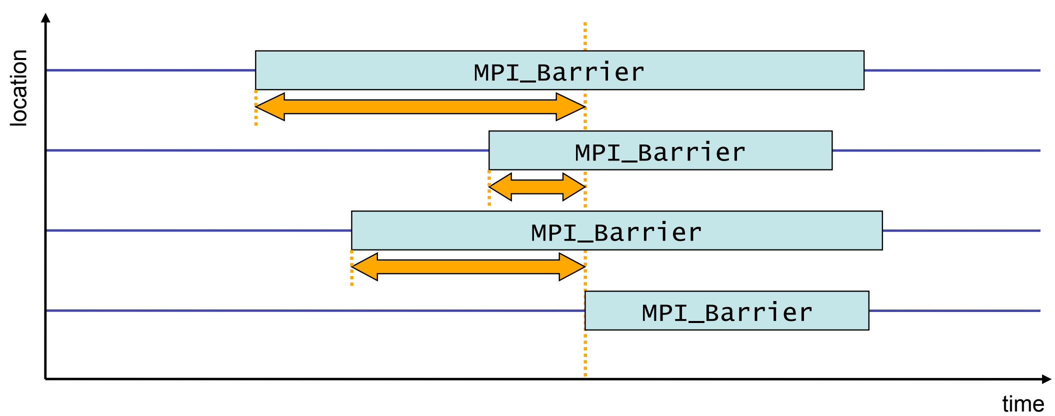

Trace analysis result



MPI collective synchronization time

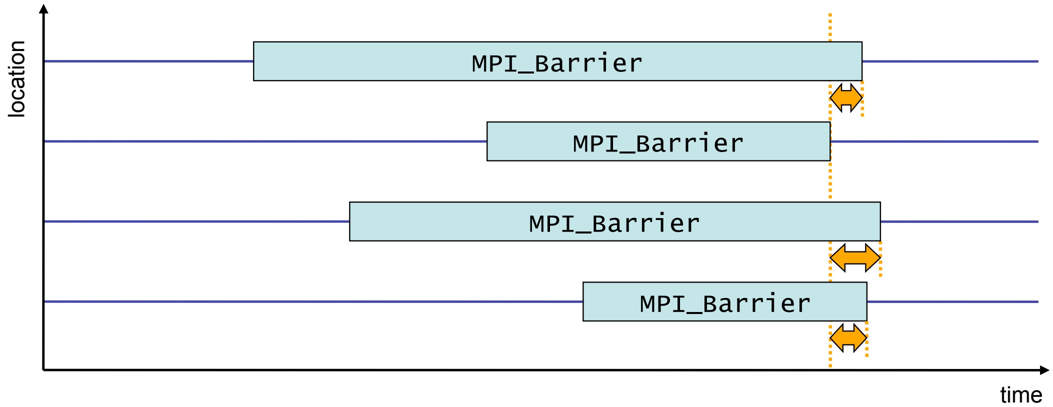


Wait at Barrier = Early Barrier



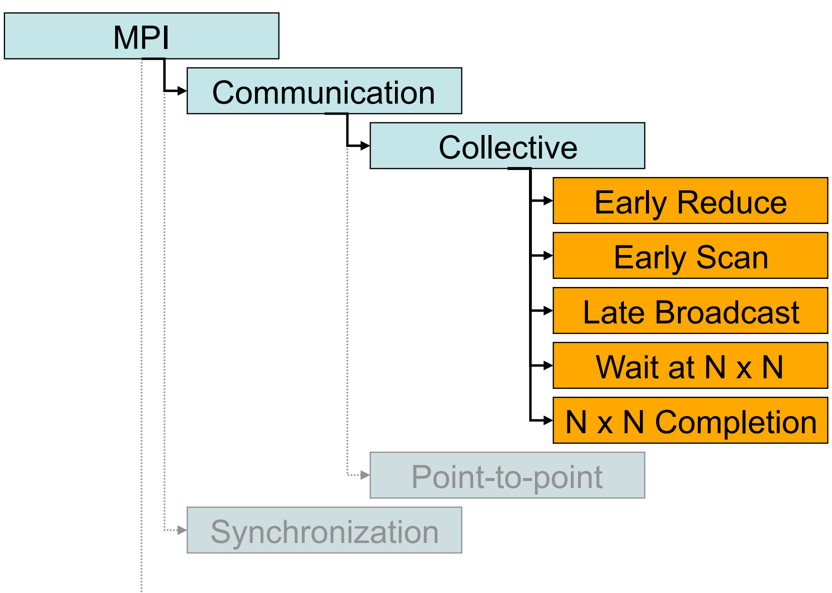
- Time spent waiting in front of a barrier call until the last process reaches the barrier operation
- Applies to: `MPI_Barrier()`

Barrier Completion

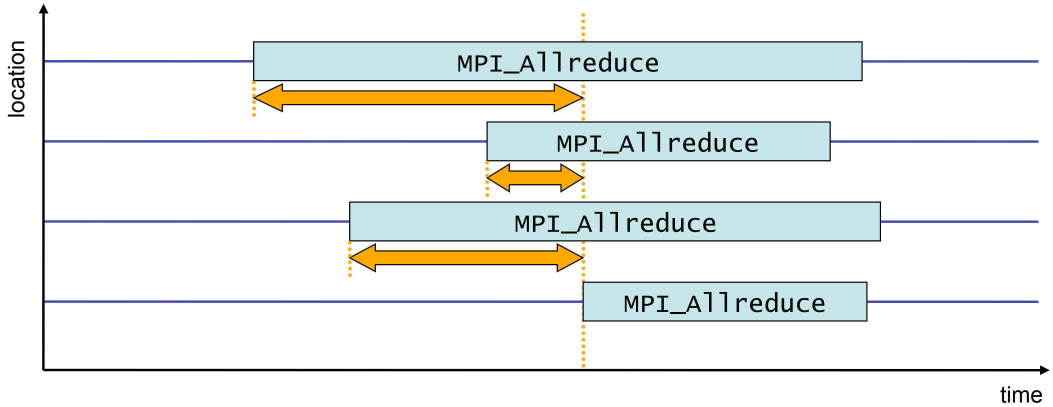


- Time spent in barrier after the first process has left the operation
- Applies to: `MPI_Barrier()`

MPI collective communication time

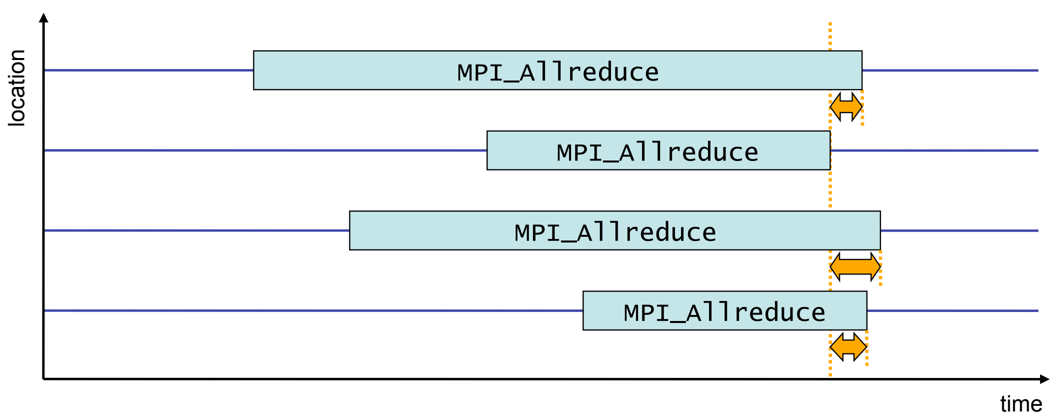


Wait at N x N = Early N x N



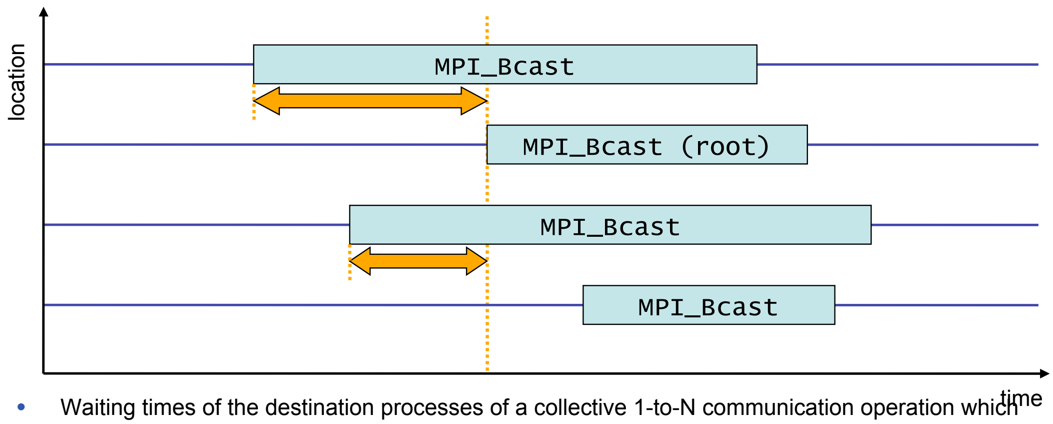
- Time spent waiting in front of a synchronizing collective operation call until the last process reaches the operation
- Applies to: `MPI_Allreduce`(), `MPI_Alltoall`(), `MPI_Alltoallv`(), `MPI_Allgather`(), `MPI_Allgatherv`(), `MPI_Reduce_scatter`()

N x N Completion



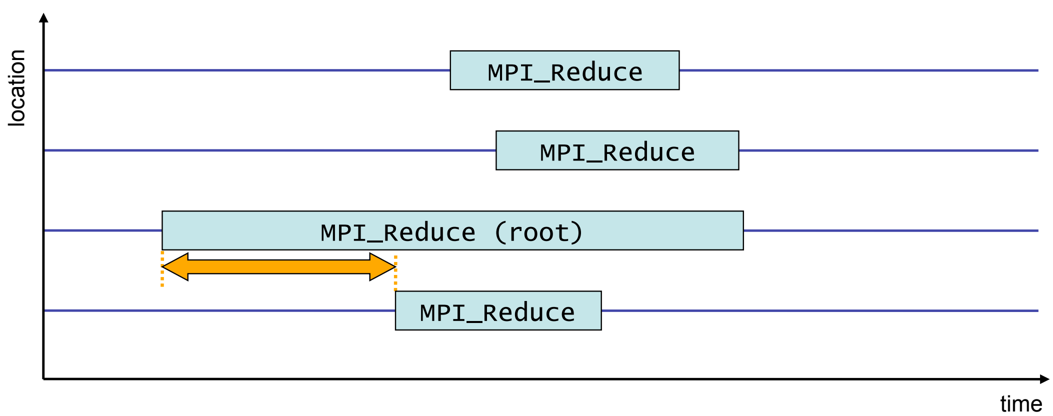
- Time spent in synchronizing collective operations after the first process has left the operation
- Applies to: `MPI_Allreduce`(), `MPI_Alltoall`(), `MPI_Alltoallv`(), `MPI_Allgather`(), `MPI_Allgatherv`(), `MPI_Reduce_scatter`()

Late Broadcast = Early Broadcast



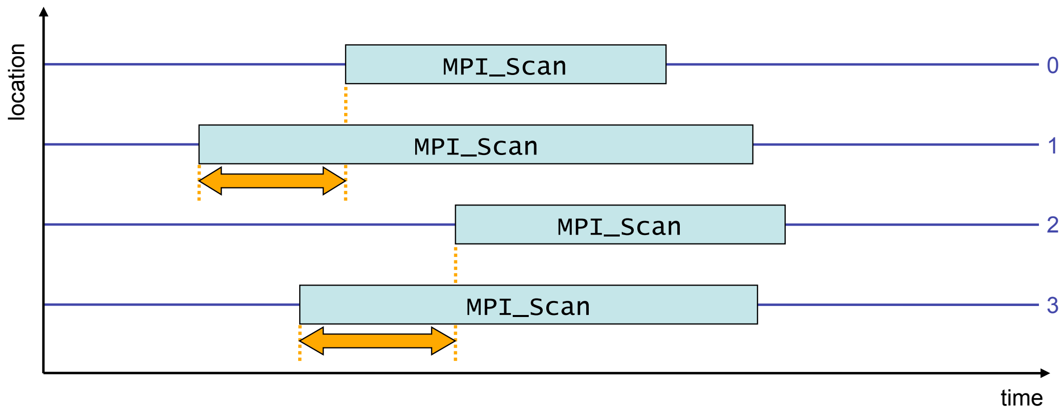
- Waiting times of the destination processes of a collective 1-to-N communication operation which enter the operation earlier than the source process (root)
  - Late Broadcast by source = Early Broadcast by destinations
- Applies to: `MPI_Bcast`(), `MPI_Scatter`(), `MPI_Scatterv`()

Early Reduce



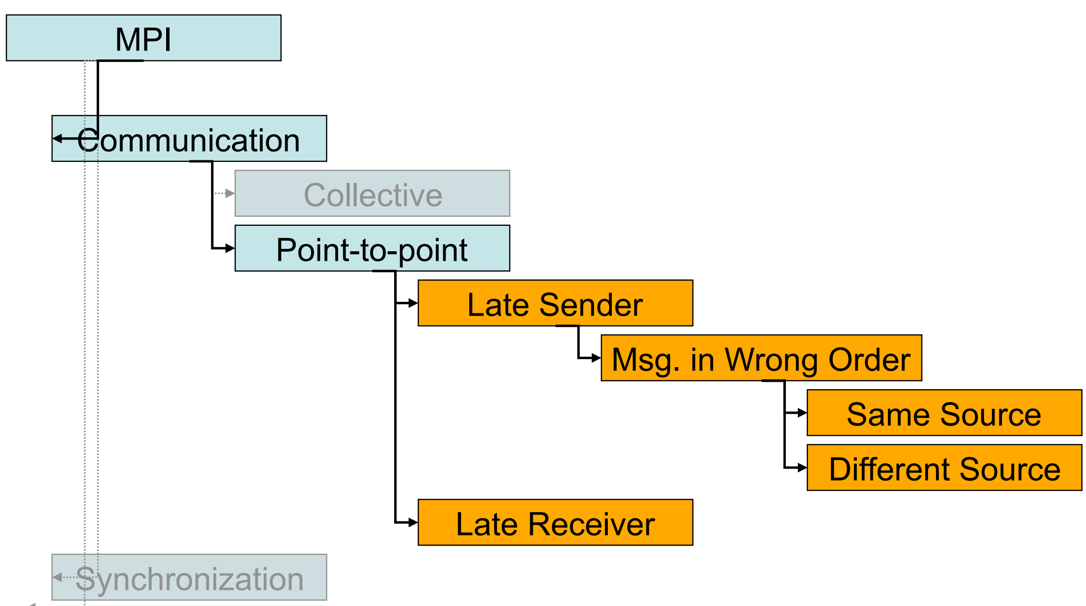
- Waiting time if the destination process (root) of a collective N-to-1 communication operation enters the operation earlier than its sending counterparts
- Applies to: `MPI_Reduce`(), `MPI_Gather`(), `MPI_Gatherv`()

Early Scan

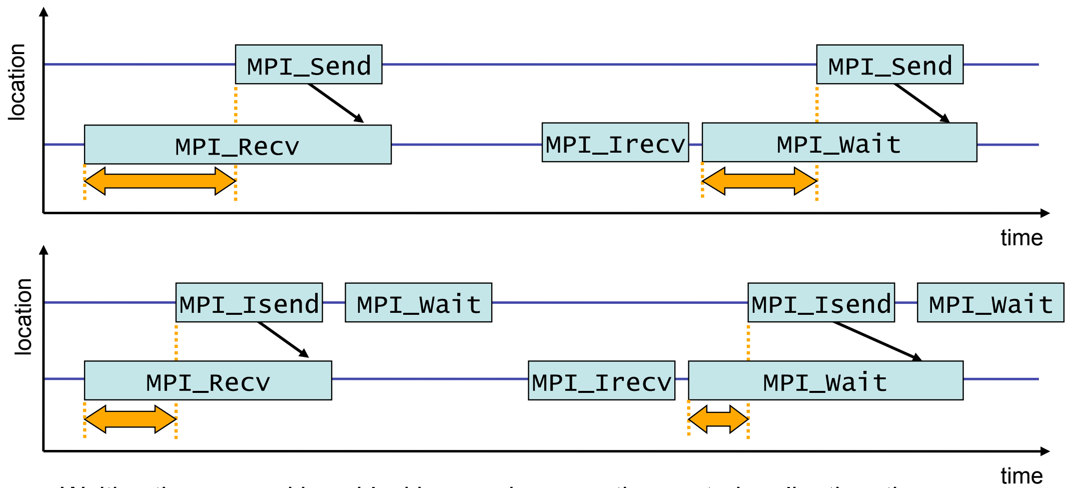


- Waiting time if process *n* enters a prefix reduction operation earlier than its sending counterparts (i.e., ranks 0..*n*-1)
- Applies to: `MPI_Scan`()

MPI point-to-point communication time

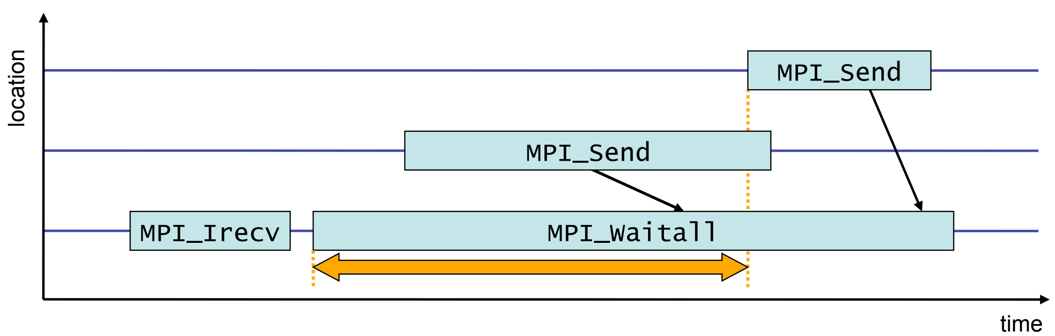


Late Sender = Early Receive



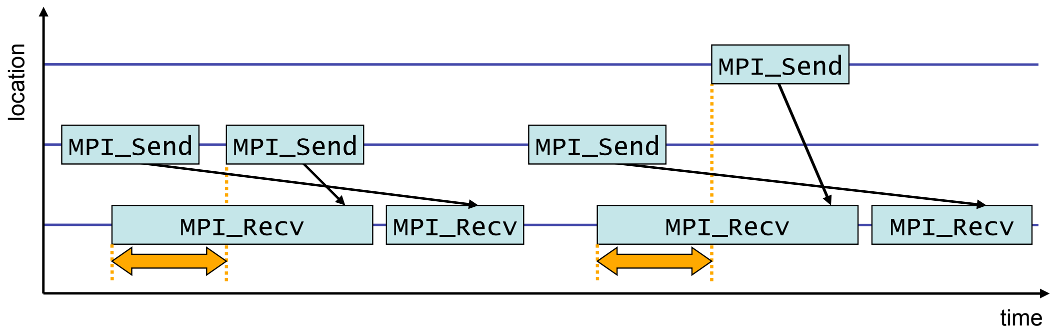
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send operation
- Applies to blocking as well as non-blocking communication

Late Sender = Early Receive (cont.)



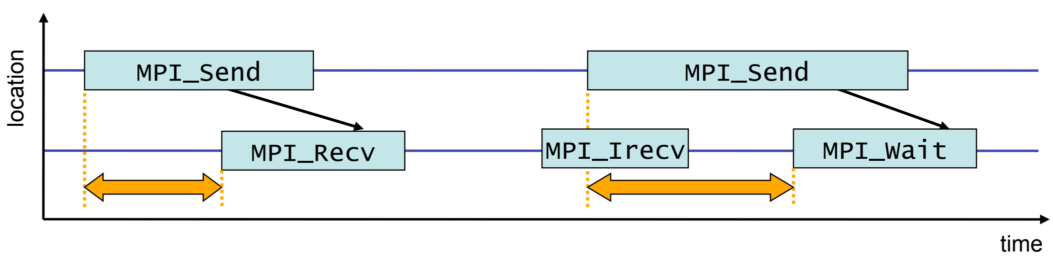
- While waiting for several messages, the maximum waiting time is accounted
- Applies to: `MPI_Waitall`(), `MPI_Waitsome`()

Late Sender, Messages in Wrong Order



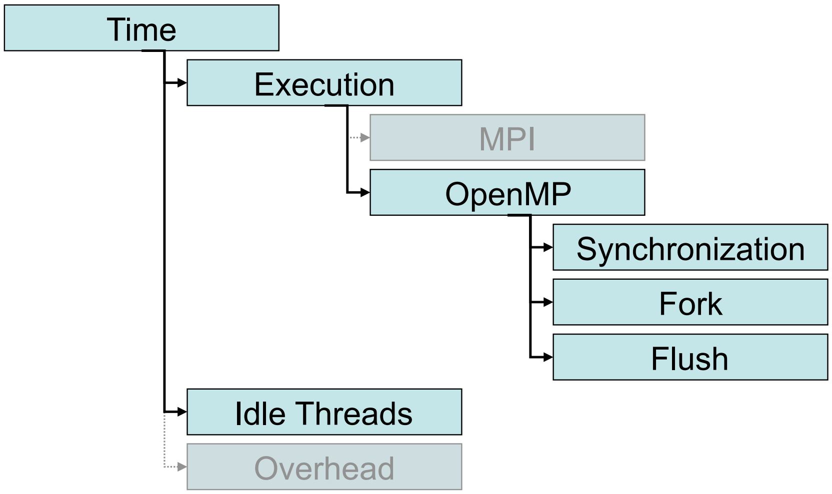
- Refers to Late Sender situations which are caused by messages received in wrong order
  - Early receive of message out of order
- Comes in two flavours:
  - Messages sent from same source location
  - Messages sent from different source locations

Late Receiver = Early Send



- Waiting time caused by a blocking send operation posted earlier than the corresponding receive operation
- Does not apply to non-blocking sends

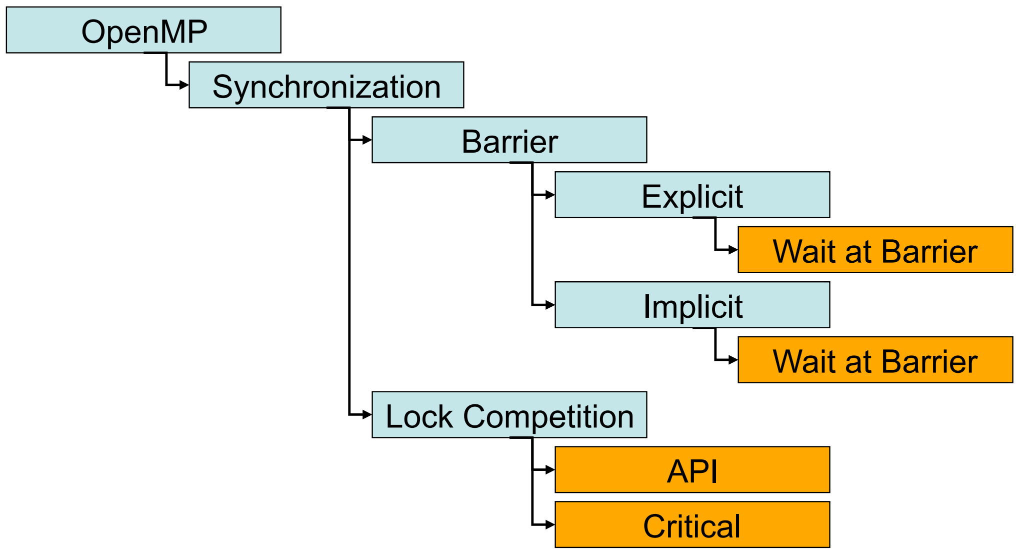
OpenMP Time hierarchy



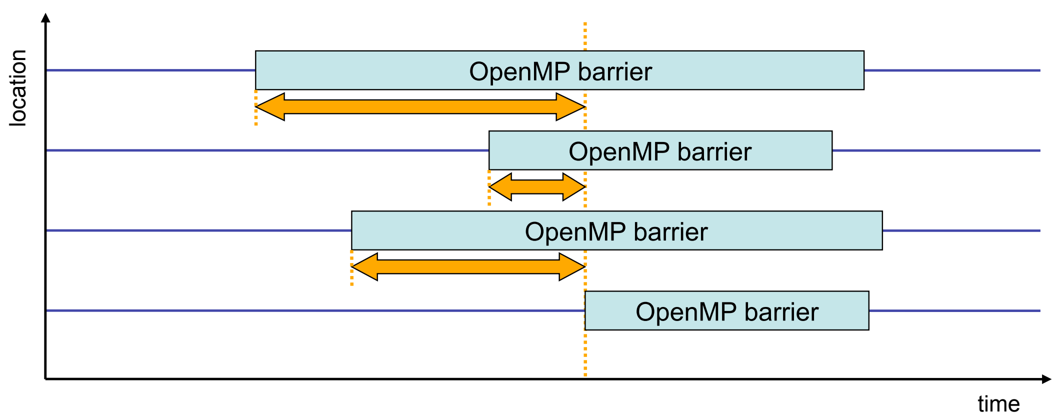
OpenMP Time hierarchy details

OpenMP	Time spent for all OpenMP-related tasks
Synchronization	Time spent synchronizing OpenMP threads
Fork	Time spent by master thread to create thread teams
Flush	Time spent in OpenMP flush directives
Idle Threads	Time spent idle on CPUs reserved for slave threads

OpenMP synchronization time

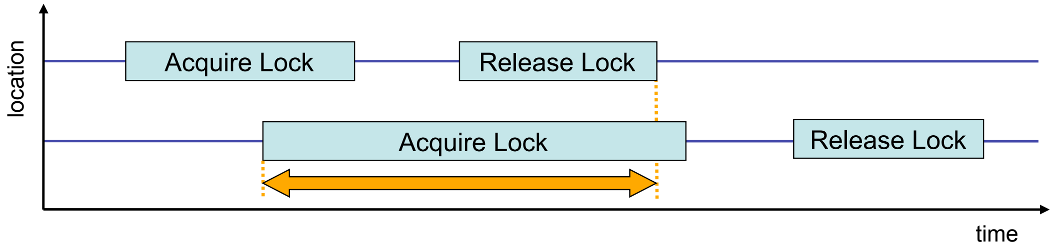


Wait at Barrier = Early Barrier



- Time threads spend waiting in front of a barrier call until the last thread reaches the barrier operation
- Applies to: Implicit/explicit barriers

Lock competition



- Time a thread spends waiting for a lock that is held by other threads until it is released and can be acquired by this thread
- Applies to: critical sections, OpenMP lock API

Other metrics

- LateReceivers/LateSenders
  - counts shown in hierarchies of Synchronizations & Communications below Sends & Receives respectively
- Computational Imbalance
  - load imbalance heuristic calculated as absolute difference from average exclusive execution time
- HWC metrics
  - shown as separate root metrics for each counter
  - only provided in summary reports

Intermediate use of Scalasca

- User-defined region instrumentation
  - EPIK annotation macros API
  - POMP annotation directives
    Selective instrumentation
- Summary collections & analysis experiment
- Trace collection & analysis experiment
- Analysis report effectiveness score
- Customisation of measurement collection
  - Sizing of measurement data structures (e.g., trace buffers)
  - Function filter configuration
  - Optional HWC metrics
- Analysis report algebra

Instrumentation/measurement configuration

- Selective instrumentation
  - Adjust build not to (auto-)instrument particular modules
  - Separate/preprocess sources for functions in same module
  - Entirely avoids instrumentation & overhead
- Selective measurement via function filtering
  - Supported for GCC, IBM & Intel compilers
  - Specify text file listing names of functions (one per line, shell wildcarding) to ignore with EPK_FILTER
  - Use linker/decorated function names [Fortran/C++]

cube3_score with (sorted) region breakdown

```
% cube3_score -r epik_smg2000_mano_64/summary.cube | sort ...
flt type    max_tbc      time        %  region
...
    MPI    2061936    293.30   23.89  MPI_Waitall
    COM    2346840     14.79    1.20  hypre_FinalizeCommunication
    COM    2346840     23.49    1.91  hypre_InitializeCommunication
    MPI    7495240     11.38    0.93  MPI_Irecv
    MPI    8149850     41.43    3.37  MPI_Isend
    USR    9426048     10.47    0.85  hypre_StructStencilElementRank
    USR    9426048     21.69    1.77  hypre_StructMatrixExtractPointerByIdx
    USR   11063016     16.80    1.37  hypre_MAlloc$AF10_5
    USR   11454432     25.82    2.10  hypre_MAlloc
    USR   11763336     26.90    2.19  hypre_CAlloc
    USR   23496576     38.16    3.11  hypre_Free

    ANY  162589938   1227.61  100.00  ALL     (254 regions)
    MPI   17649090    456.64   37.20    + MPI ( 13 regions) pure MPI
    COM    9905832    321.80   26.21    + COM ( 32 regions) combined
   MPI&USR
    USR  135034968    311.13   25.34    + USR (207 regions) pure User

max_tbc = est. maximum trace buffer capacity requirement (bytes/process)
   to store all events that would be generated in an equivalent trace
```


cube3_score with trial region filter

```
% cube3_score -r -f smg2000.filt epik_smg2000_mano_64/summary.cube | sort
flt type    max_tbc      time        %  region
...
  - MPI    2061936    293.30   23.89  MPI_Waitall
  - COM    2346840     14.79    1.20  hypre_FinalizeCommunication
  - COM    2346840     23.49    1.91  hypre_InitializeCommunication
  - MPI    7495240     11.38    0.93  MPI_Irecv
  - MPI    8149850     41.43    3.37  MPI_Isend
  + USR    9426048     10.47    0.85  hypre_StructStencilElementRank
  + USR    9426048     21.69    1.77  hypre_StructMatrixExtractPointerByIdx
  + USR   11063016     16.80    1.37  hypre_MAlloc$AF10_5
  + USR   11454432     25.82    2.10  hypre_MAlloc
  + USR   11763336     26.90    2.19  hypre_CAlloc
  + USR   23496576     38.16    3.11  hypre_Free

  - ANY  162589938   1227.61  100.00  ALL     (253 regions)
  - MPI   17649090    456.64   37.20    + MPI ( 13 regions) pure MPI
  - COM    9905832    321.80   26.21    + COM ( 32 regions) combined
   MPI&USR
  - USR  135034968    311.13   25.34    + USR (207 regions) pure User

  + FLT  103570824    182.11   14.83      FLT (  9 regions) filtered
  - FLT   59019114   1045.50   85.17  ALL-FLT (244 regions) remainder
```

Preparation of instrumented executable

- Auto-instrumentation of functions
  - Capability of most (but not all) compilers
  - Currently need separate Scalasca installations for each desired combination of MPI library & compiler suite
  - $(PREP) $(MPIFC) ...
  - $(PREP) $(MPICC) ...
  - $(PREP) $(MPICXX) ...
  - PREP="skin $(SKIN_OPTS)" for instrumented build
  - PREP="" for uninstrumented build for production
- Auto-instrumentation plus API for user-defined regions
  - #include "epik_user.inc" or "epik_user.h"
  - % skin -user $(MPIC) ...

Manual instrumentation options

- No instrumentation
  - $(MPIC) [`kconfig -cflags`]
- MPI library instrumentation
  - $(MPIC) [`kconfig -cflags`] `kconfig -libs`
- MPI library & EPIK user instrumentation
  - $(MPIC) `kconfig -cflags` `kconfig -libs` -D**EPIK**
- `kconfig -cflags` is optional for source modules without explicit EPIK API #include

EPIK instrumentation API dummy macros

- To use unmodified compile commands (without EPIK API include path) for sources with EPIK API calls, define dummy macros

```
#ifdef EPIK
#include "epik_user.inc" or "epik_user.h"
#else
#define EPIK_FUNC_REG(str)      /* undefined */
#define EPIK_FUNC_START()       /* undefined */
#define EPIK_FUNC_END()         /* undefined */
#define EPIK_USER_REG(id,str)   /* undefined */
#define EPIK_USER_START(id)     /* undefined */
#define EPIK_USER_END(id)       /* undefined */
#endif
```


EPIK instrumentation API

- Manual phase annotation
  - EPIK_FUNC_REG("Fortran function/subroutine")
  - EPIK_FUNC_START()
  - EPIK_USER_REG(tsloop, "<<time step>>")
  - EPIK_USER_START(tsloop)
  - EPIK_USER_END(tsloop)
  - EPIK_FUNC_END()
- Note matching of enter/start annotations
  - all possible exits must be annotated
  - regions must be correctly nested
  - C/C++ function names are automatically registered
  - Fortran function/routine names undefined if not preregistered

POMP instrumentation

- Uses pragma/comment directives to annotate regions

C/C++:

```
#pragma pomp inst init
#pragma pomp inst begin(tsloop)
  #pragma pomp inst altend(tsloop)
#pragma pomp inst end(tsloop)
```

Fortran:

```
!POMP$ INST INIT
!POMP$ INST BEGIN(tsloop)
  !POMP$ INST ALTEND(tsloop)
!POMP$ INST END(tsloop)
```

- Directives ignored unless activated with ***skin -pomp***
  - ***all directives in module instrumented***
- Current limitations
  - instrumentation inactive until “inst init”
  - no distinction of functions from other regions
  - last region exit must be marked “end”, all others as “altend”
  - doesn't support C99 `_Pragma` operator

Measurement configuration

- Example configuration
  - EPK_GDIR=/work/$USER # archive location
  - EPK_TITLE=app_$NP # experiment archive title
  - EPK_SUMMARY=1 # runtime summarisation
  - EPK_TRACE=0 # event trace collection
- New archive directory for each experiment
  - $EPK_GDIR/epik_$EPK_TITLE
  - contains intermediate data (e.g., trace files), log/config files and processed analyses
- Configured automatically (overridden) by ***scan*** args

Default EPIK.CONF configuration file extract

E P I K configuration
- EPK_TITLE=a # experiment archive title [scan -e]
- EPK_SUMMARY=1 # runtime summarization [scan -s]
- EPK_TRACE=0 # event trace collection [scan -t]
- EPK_FILTER= # file listing functions to skip
- EPK_METRICS= # colon-separated list of metrics [-m]

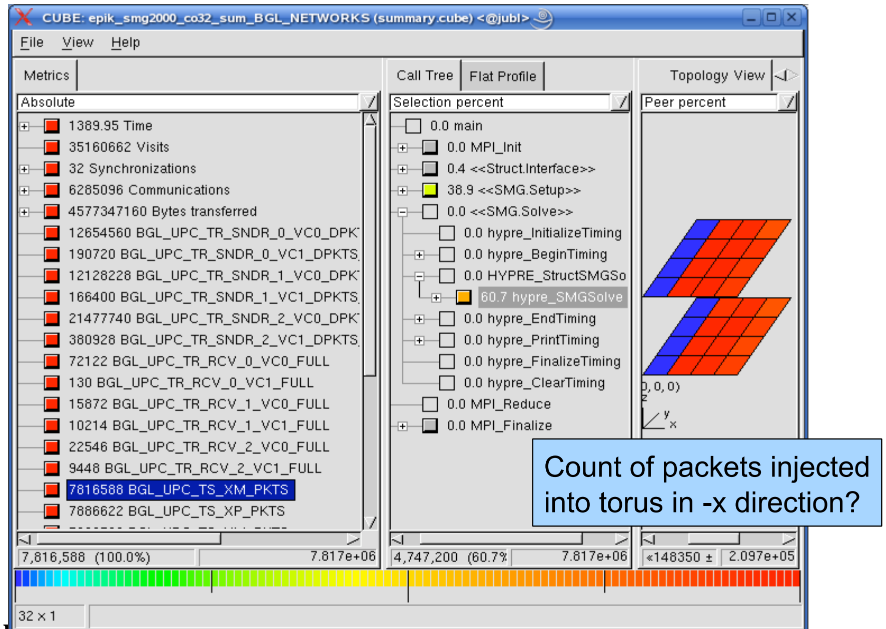
E P I S O D E configuration
- ESD_PATHS=1024 # max. recorded call-paths
- ESD_FRAMES=32 # max. call-stack frames
- ESD_BUFFER_SIZE=100000 # definitions bytes

E P I L O G configuration
- ELG_BUFFER_SIZE=10000000 # trace bytes

Hardware counter metrics

- Available counters (and their interpretation) are platform/processor-specific
  - considered separate root metrics in analyses
- Platform metrics specification
  - defines convenient groups of metrics
  - EPK_METRICS_SPEC=./METRICS.SPEC
- Group/list of counters to measure in experiment
  - EPK_METRICS=POWER4_DC # data-cache
  - EPK_METRICS=BGL_NETWORKS # torus & tree
  - EPK_METRICS=PM_CYC:PM_INST_CMPL
    or PAPI_TOT_CYC:PAPI_TOT_INS

Scalasca summary experiment with HWC metrics



CUBE algebra tools

- CUBE files can be compared/combined with some useful command line tools
- Note that these work directly on CUBE files and not on archive directories
  - Reads CUBE2 & CUBE3 files, but only writes CUBE3 files
- General usage:
  - cube3_*tool* [-o <output file>] <input file>
- If no output file is specified, *tool*.cube is generated

CUBE algebra tools (2)

- cube3_merge
  - combines multiple analysis reports into integrated report
  - merges metric, call-path & system trees
  - takes metric severities from first available report
  - e.g., combine measurements of sets of HWC metrics in summary report(s) with a (non-HWC) trace analysis report into a “holistic” analysis report

  % cube3_merge trace.cube summary_HWC[1234].cube
    - Metrics listed in order of appearance in input reports
    - User-defined hierarchies of measured & derived HWC metrics not yet supported by CUBE3!

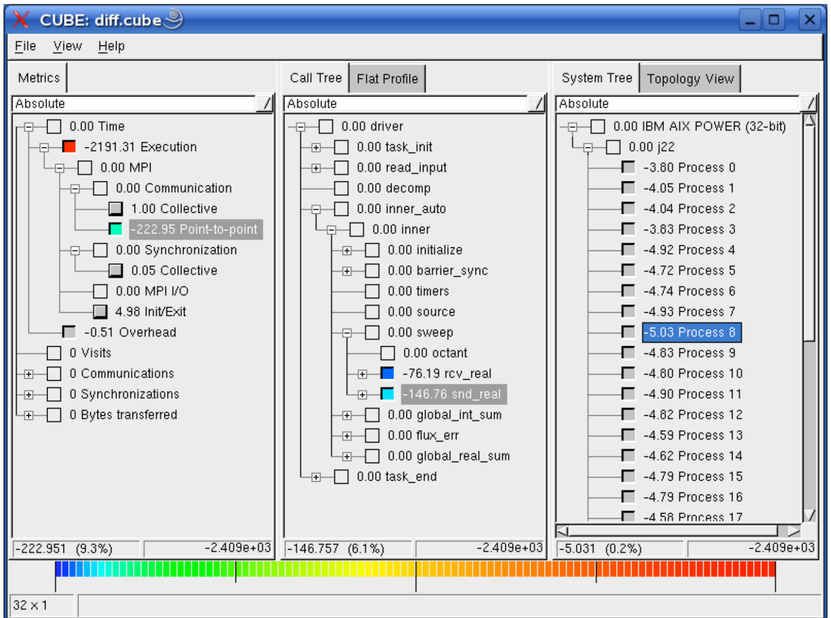
CUBE algebra tools (3)

- `cube3_mean`
  - Can eliminate “measurement noise” by averaging the results of several experiments

- `cube3_cut [-p prune] [-r root]`
  - Creates a new CUBE file without pruned subtrees and/or containing only the specified call tree node as new root(s)

- `cube3_diff`
  - Calculates the difference of two experiments
  - Useful to measure improvement/degradation due to a modification

Difference experiment:
JUMP – JUBL (different architectures)



Labs!

Lab: PAPI, TAU, Vampir, and Scalasca/KOJAK

Lab Instructions (for OCF systems)

Get workshop.tar.gz using:

```
% wget
  http://www.paratools.com/llnl10/workshop.tar.gz
```

Or

```
% cp /usr/global/tools/tau/training/workshop.tar.gz .
% tar zxf workshop.tar.gz
source /usr/global/tools/tau/training/tau.bashrc
```

OR

```
source /usr/global/tools/tau/training/tau.bashrc.impi
```

in your .login file and then follow the instructions in the README file.

For LiveDVD, see ~/workshop-point/README and follow.

Lab Instructions

To profile a code using TAU:

1. Change the compiler name to tau_cxx.sh, tau_f90.sh, tau_cc.sh:
   F90 = tau_f90.sh
2. Choose TAU stub makefile
   % export TAU_MAKEFILE=/usr/global/tools/tau/training/tau_latest/x86_64/lib/Makefile.tau-[options]
3. If stub makefile has -papi in its name, set the TAU_METRICS environment variable:
   % export TAU_METRICS=TIME:PAPI_L2_DCM:PAPI_TOT_CYC...
4. Build and run workshop examples, then run pprof/paraprof

Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science contracts
  - LLNL-LANL-SNL ASC/NNSA Level 3 contract
  - LLNL ParaTools contracts
- Department of Defense (DoD)
  - PET
- National Science Foundation (NSF)
  - POINT
- University of Oregon
  - A. Malony, W. Spear,
    S. Biersdorff, A. Nataraj
- University of Tennessee, Knoxville
  - Dr. David Cronk and Dr. Shirley Moore
- T.U. Dresden, GWT
  - Dr. Wolfgang Nagel and Dr. Holger Brunst
- Research Centre Juelich
  - Dr. Bernd Mohr, Dr. Felix Wolf