

# Performance Engineering of Parallel Applications

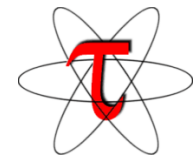
Philip Blood, Raghu Reddy  
Pittsburgh Supercomputing Center

# POINT Project

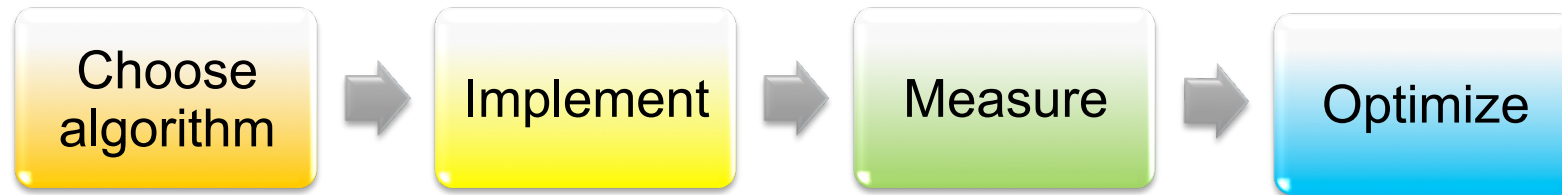
- “High-Productivity Performance Engineering (Tools, Methods, Training) for NSF HPC Applications”
  - NSF SDCl, Software Improvement and Support
  - University of Oregon, University of Tennessee, National Center for Supercomputing Applications, Pittsburgh Supercomputing Center
- POINT project
  - Petascale Productivity from Open, Integrated Tools
  - <http://www.nic.uoregon.edu/point>

# Parallel Performance Technology

- PAPI
  - University of Tennessee, Knoxville
- PerfSuite
  - National Center for Supercomputing Applications
- TAU Performance System
  - University of Oregon
- Kojak / Scalasca
  - Research Centre Juelich



# Code Development and Optimization Process



- Choice of algorithm most important consideration (serial and parallel)
- Highly scalable codes must be designed to be scalable from the beginning!
- Measurement may reveal need for new algorithm or completely different implementation rather than optimization
- Focus of this lecture: using tools to assess parallel performance

# A little background...

# Hardware Counters

- Counters: set of registers that count processor events, like floating point operations, or cycles (Opteron has 4 registers, so 4 types of events can be monitored simultaneously)
- **PAPI**: Performance **API**
- Standard API for accessing hardware performance counters
- Enable mapping of code to underlying architecture
- Facilitates compiler optimizations and hand tuning
- Seeks to guide compiler improvements and architecture development to relieve common bottlenecks

# Features of PAPI

- Portable: uses same routines to access counters across all architectures
- High-level interface
  - Using predefined standard events the same source code can access similar counters across various architectures without modification.
  - `papi_avail`
- Low-level interface
  - Provides access to all machine specific counters (requires source code modification)
  - Increased efficiency and flexibility
  - `papi_native_avail`
- Third-party tools
  - TAU, Perfsuite, IPM

# Example: High-level interface

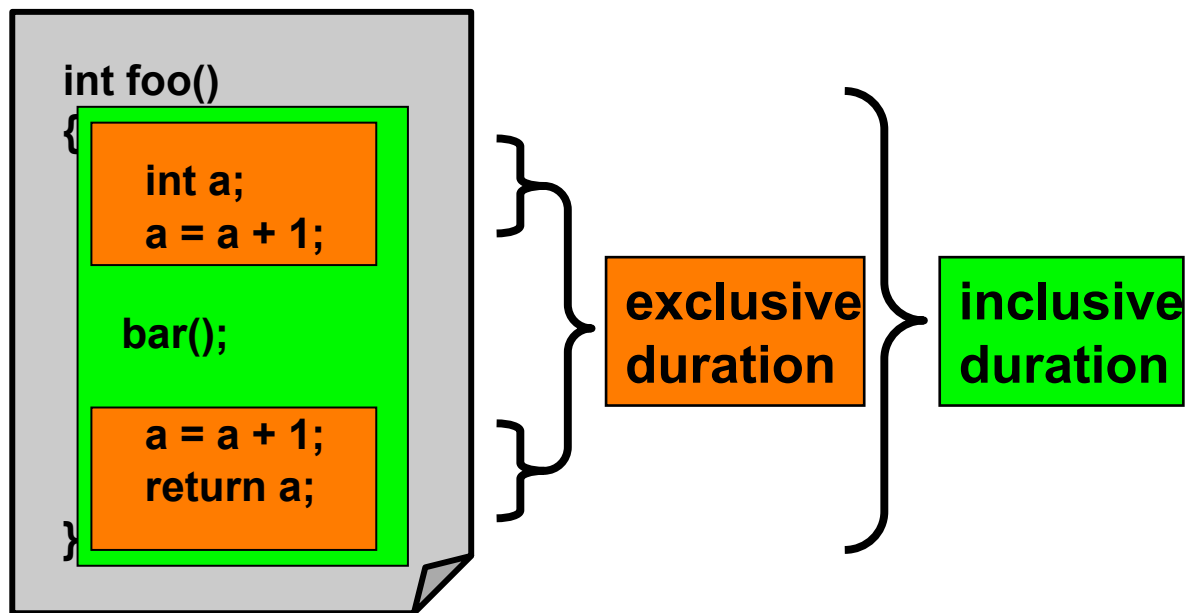
```
#include <papi.h>
#define NUM_EVENTS 2
main()
{
    int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long_long values[NUM_EVENTS];
    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
    /* Do some computation here */
    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
    /* Do some computation here */
    /* Stop counting events */
    if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
}
```

# Measurement Techniques

- When is measurement triggered?
  - Sampling (indirect, external, low overhead)
    - interrupts, hardware counter overflow, ...
  - Instrumentation (direct, internal, high overhead)
    - 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

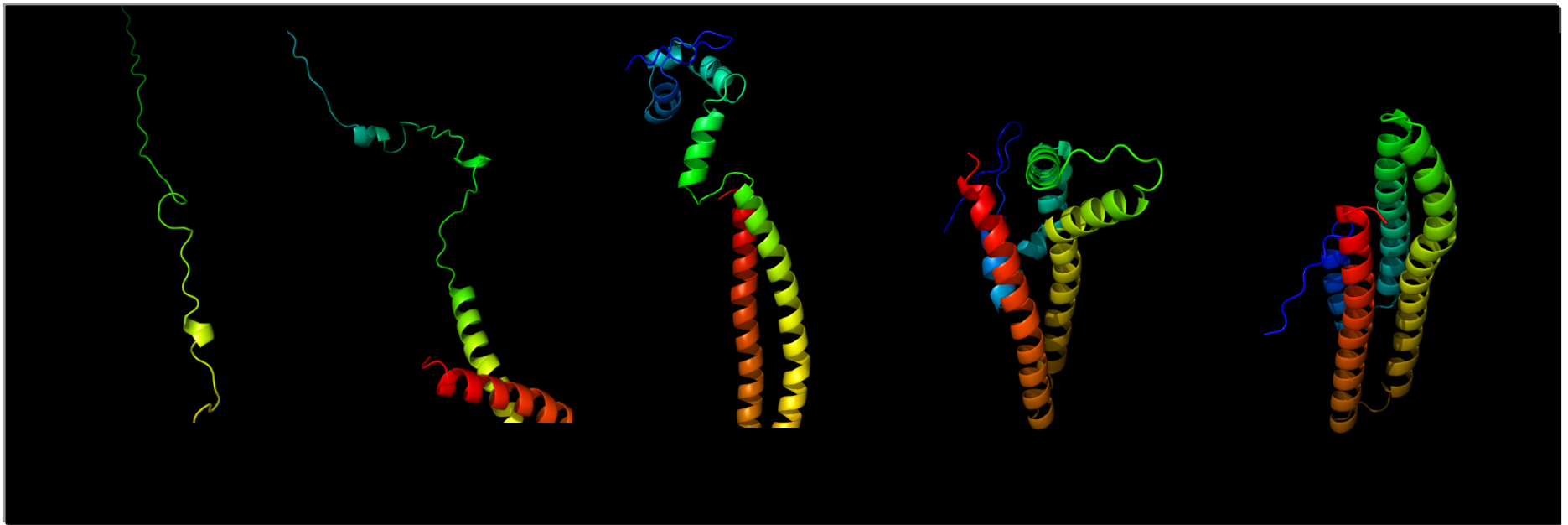
# Inclusive and Exclusive Profiles

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



# Applying Performance Tools to Improve Parallel Performance of the UNRES MD code

The UNRES molecular dynamics (MD) code utilizes a carefully-derived mesoscopic protein force field to study and predict protein folding pathways by means of molecular dynamics simulations.



<http://www.chem.cornell.edu/has5/>

<http://cbsu.tc.cornell.edu/software/protarch/index.htm>

# Structure of UNRES

- Two issues
  - Master/Worker code

```
if (myrank==0)
    MD=>...=>EELEC
else
    ERGASTULUM=>...=>EELEC
endif
```

- Significant startup time: must remove from profiling
  - Setup time: 300 sec
  - MD Time: 1 sec/step
  - Only MD time important for production runs of millions of steps
  - Could run for 30,000 steps to amortize startup!

# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Is There a Performance Problem?

- What does it mean for a code to perform “poorly”?
  - HPL on 4K cores can take a couple of hrs
  - Quantum calculations involving a few atoms may take a week
  - Depends on the work being done
- Where does performance need to be improved?
  - Serial performance problem?
  - Parallel performance problem?

# Detecting Performance Problems

- **Serial Performance: Fraction of Peak**
  - 20% peak (overall) is usually decent; After that you decide how much effort it is worth
  - Theoretical FLOP/sec peak =  $\text{FLOP/cycle} \times \text{cycles/sec}$
  - 80:20 rule
- **Parallel Performance: Scalability**
  - Does run time decrease by 2x when I use 2x cores?
    - **Strong scalability**
  - Does run time remain the same when I keep the amount of work per core the same?
    - **Weak scalability**

# IPM

- Very good tool to get an overall picture
  - Overall MFLOP
  - Communication/Computation ratio
- Pros
  - Quick and easy!
  - Minimal overhead (uses sampling rather than source code instrumentation)
- Cons
  - Harder to get at “nitty gritty” details
  - No OpenMP support

<http://ipm-hpc.sourceforge.net/>

# IPM Mechanics

On Ranger:

1) `module load ipm`

2) just before the `ibrun` command in the batch script add:  
`setenv LD_PRELOAD $TACC_IPM_LIB/libipm.so`

3) run as normal

4) to generate webpage

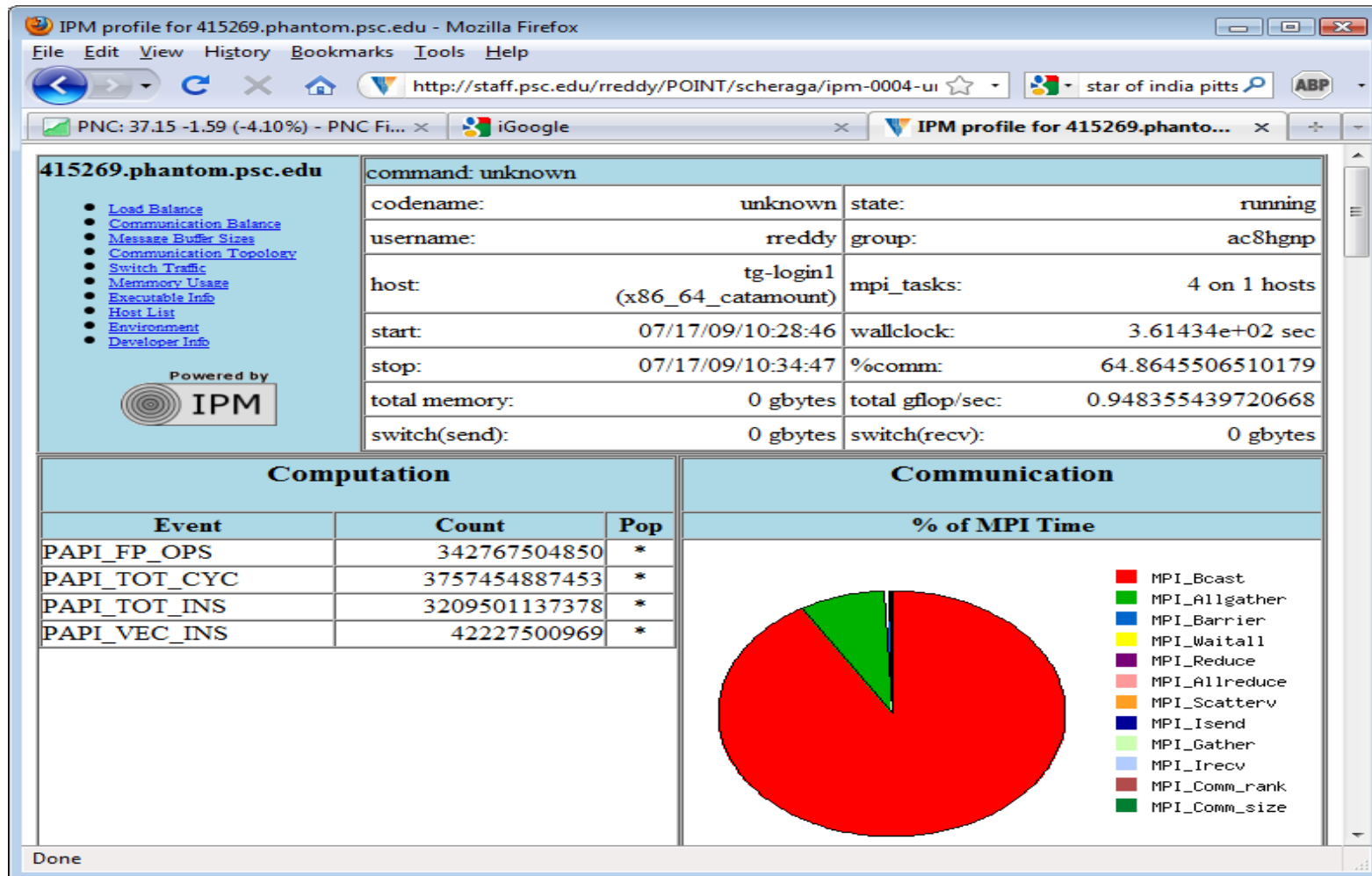
`module load ipm` (if not already)  
`ipm_parse -html <xml_file>`

You should be left with a directory with the html in. Tar it up, move to your local computer and open `index.html` with your browser.

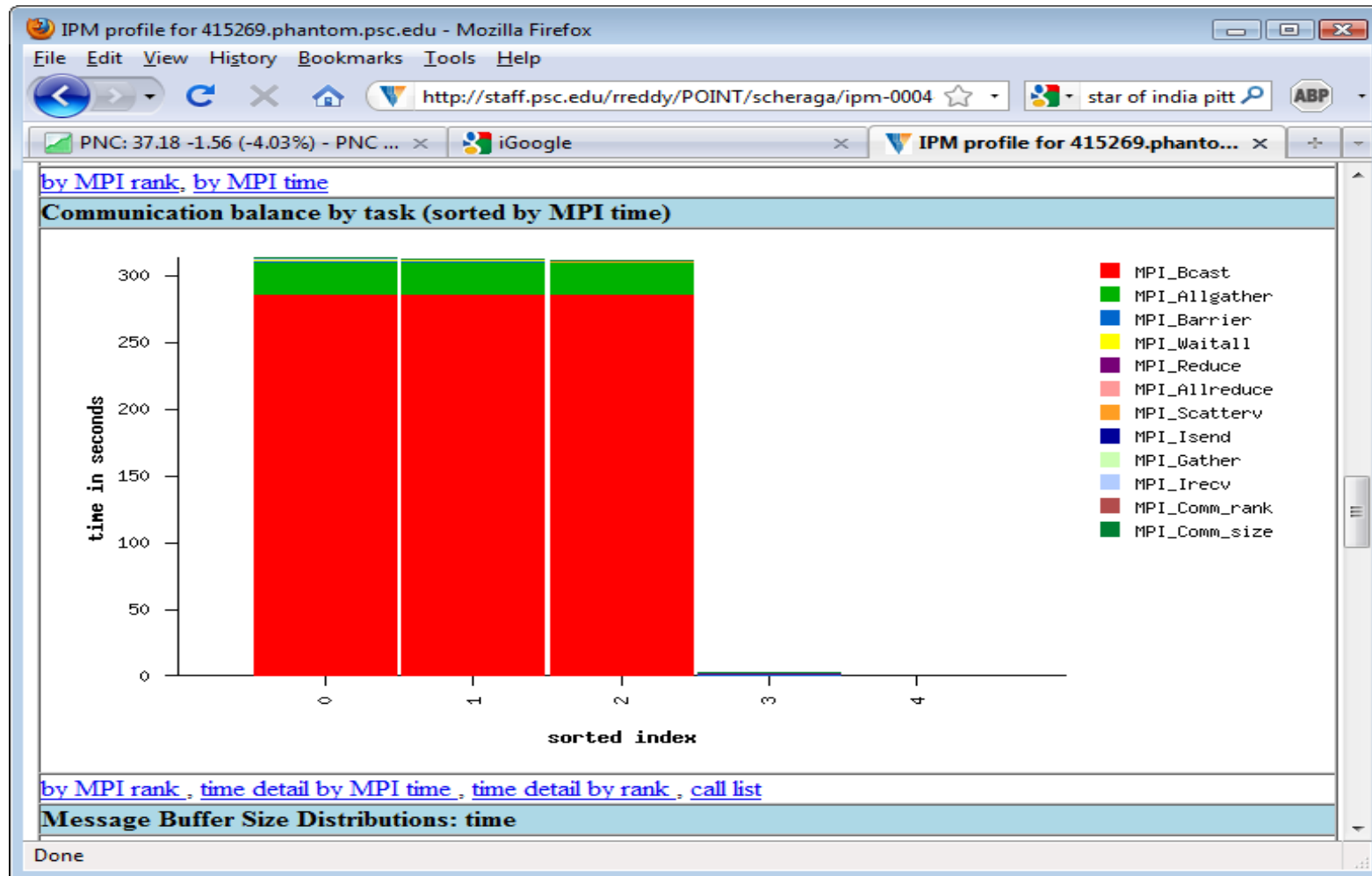
# IPM Overhead

- Was run with 500 MD steps (time in sec)
  - base: MD steps: 5.14637E+01
  - base-ipm: MD steps: 5.13576E+01
- Overhead is negligible

# IPM Results: Overall Picture



# IPM – Communication (overall)



# PerfSuite

- Similar to IPM: great for getting overall picture of application performance
- Pros
  - Easy: no need to recompile
  - Minimal overhead
  - Provides function-level information
  - Works with OpenMP
- Cons
  - Not available on all architectures: (x86, x86-64, em64t, and ia64)

<http://perfsuite.ncsa.uiuc.edu/>

# PerfSuite Mechanics: Overall performance

```
% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an output file (default is
plain text)

# First run: this will give you a summary of performance
information over total program execution (e.g. MFLOPS)
% psrun myprog

% psprocess myprog.12345.xml > myprog.txt
```

# First case provides hardware counter stats

Index	Description	Counter Value
1	Conditional branch instructions mispredicted.....	4831072449
4	Floating point instructions.....	86124489172
5	Total cycles.....	594547754568
6	Instructions completed.....	1049339828741

## Statistics

Graduated instructions per cycle.....	1.765
Graduated floating point instructions per cycle....	0.145
Level 3 cache miss ratio (data).....	0.957
Bandwidth used to level 3 cache (MB/s).....	385.087
% cycles with no instruction issue.....	10.410
% cycles stalled on memory access.....	43.139
MFLOPS (cycles).....	115.905
MFLOPS (wallclock).....	114.441

# UNRES: Serial Performance

## Processor and System Information (abbreviated output from PerfSuite)

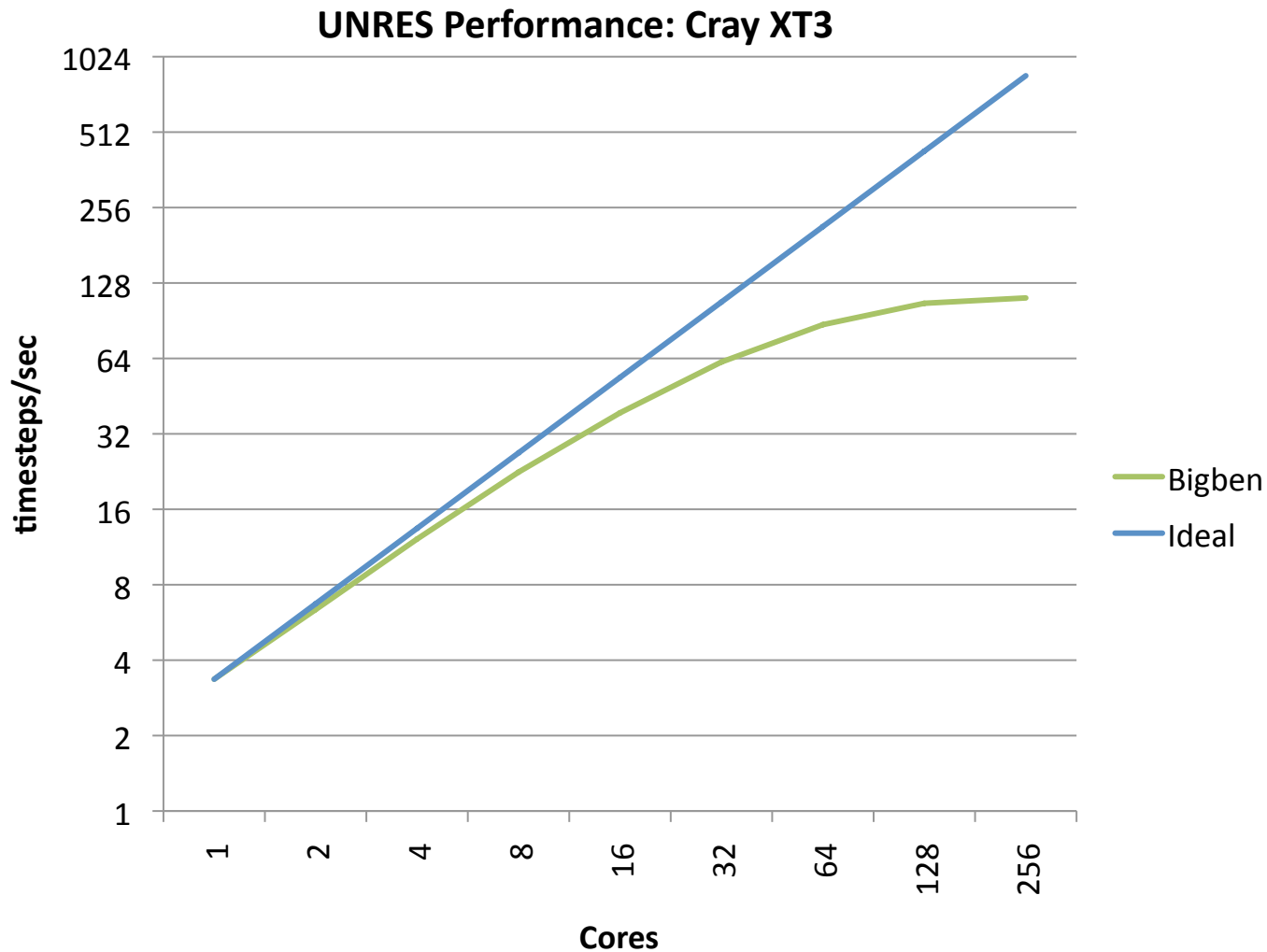
```
=====
Node CPUs          : 768
Vendor             : Intel
Family            : Itanium 2
Clock (MHz)       : 1669.001
```

## Statistics

```
=====
Floating point operations per cycle..... 0.597
MFLOPS (cycles)..... 995.801
CPU time (seconds)..... 1404.675
```

- Theoretical peak on Itanium2: 4 FLOP/cycle \*1669 MHz = 6676 MFLOPS
- UNRES getting 15% of peak--needs serial optimization on Itanium
- **Much better on Bigben (x86\_64): 1720 MFLOPS, 33% peak**
- Make sure compiler is inlining (-ipo needed for ifort, -Minline=reshape needed for pgf90)

# UNRES: Parallel Performance



# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Which Functions are Important?

- Usually a handful of functions account for 90% of the execution time
- Make sure you are measuring the production part of your code
- For parallel apps, measure at high core counts – insignificant functions become significant!

# PerfSuite Mechanics: Function breakdown

```
% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an output file (default is
# plain text)

# This will break down cycles spent in each function

% psrun -C -c papi_profile_cycles.xml myprog

% psprocess -e myprog myprog.67890.xml > myprog_functions.txt
```

## Second case gives contributions of functions

### Function Summary

Samples	Self %	Total %	Function
154346	76.99%	76.99%	pc_jac2d_blk3
14506	7.24%	84.23%	cg3_blk
10185	5.08%	89.31%	matxvec2d_blk3
6937	3.46%	92.77%	__kmp_x86_pause
4711	2.35%	95.12%	__kmp_wait_sleep
3042	1.52%	96.64%	dot_prod2d_blk3
2366	1.18%	97.82%	add_exchange2d_blk3

### Function:File:Line Summary

Samples	Self %	Total %	Function:File:Line
39063	19.49%	19.49%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:20
24134	12.04%	31.52%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:19
15626	7.79%	39.32%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:21
15028	7.50%	46.82%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:33
13878	6.92%	53.74%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:24
11880	5.93%	59.66%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:31
8896	4.44%	64.10%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:22
7863	3.92%	68.02%	matxvec2d_blk3:/home/rkufrin/apps/aspcg/matxvec2d_blk3.f:19
7145	3.56%	71.59%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:32

# PerfSuite Function Summary

## Function Summary

Samples	Self %	Total %	Function
2905589	51.98%	51.98%	eelecij
827023	14.79%	66.77%	egb
634107	11.34%	78.11%	setup_md_matrices
247353	4.42%	82.54%	escp
220089	3.94%	86.48%	etrbk3
183492	3.28%	89.76%	einvit
144851	2.59%	92.35%	banach
132058	2.36%	94.71%	ginv_mult
66182	1.18%	95.89%	multibody_hb
39495	0.71%	96.60%	etred3
38111	0.68%	97.28%	eelec

- Short runs include some startup functions amongst top functions

- To eliminate this perform a full production run with PerfSuite

- Can use PerfSuite and IPM during production runs due to low overhead—minimal impact on application performance

# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - **Instrument those functions**
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - **Instrument those functions**
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Instrument Key Functions

- Instrumentation: Insert functions into source code to measure performance
- Pro: Gives precise information about where things happen
- Con: High overhead and perturbation of application performance
- Thus essential to only instrument important functions

# TAU: Tuning and Analysis Utilities

- Useful for a more detailed analysis
    - Routine level
    - Loop level
    - Performance counters
    - Communication performance
  - A more sophisticated tool
    - Performance analysis of Fortran, C, C++, Java, and Python
    - Portable: Tested on all major platforms
    - Steeper learning curve
- <http://www.cs.uoregon.edu/research/tau/home.php>

# General Instructions for TAU

- Use a TAU Makefile stub (even if you don't use makefiles for your compilation)
- Use TAU scripts for compiling (tau\_cc.sh tau\_f90.sh)
- Example (most basic usage):

```
module load tau
```

```
setenv TAU_MAKEFILE <path>/Makefile.tau-papi-pdt-pgi
```

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

```
tau_f90.sh -o hello hello_mpi.f90
```

- Excellent “Cheat Sheet”!
    - Everything you need to know?! (Almost)
- <http://www.psc.edu/general/software/packages/tau/TAU-quickref.pdf>

# Using TAU with Makefiles

- Fairly simple to use with well written makefiles:

```
setenv TAU_MAKEFILE <path>/Makefile.tau-papi-mpi-pdt-pgi
```

```
setenv TAU_OPTIONS "-optVerbose -optKeepFiles -optPreProcess"
```

```
make FC=tau_f90.sh
```

- run code as normal
  - run pprof (text) or paraprof (GUI) to get results
  - **paraprof --pack file.ppk** (packs all of the profile files into one file, easy to copy back to local workstation)
- Example scenarios
  - Typically you can do cut and paste from here:  
<http://www.cs.uoregon.edu/research/tau/docs/scenario/index.html>

# Tiny Routines: High Overhead

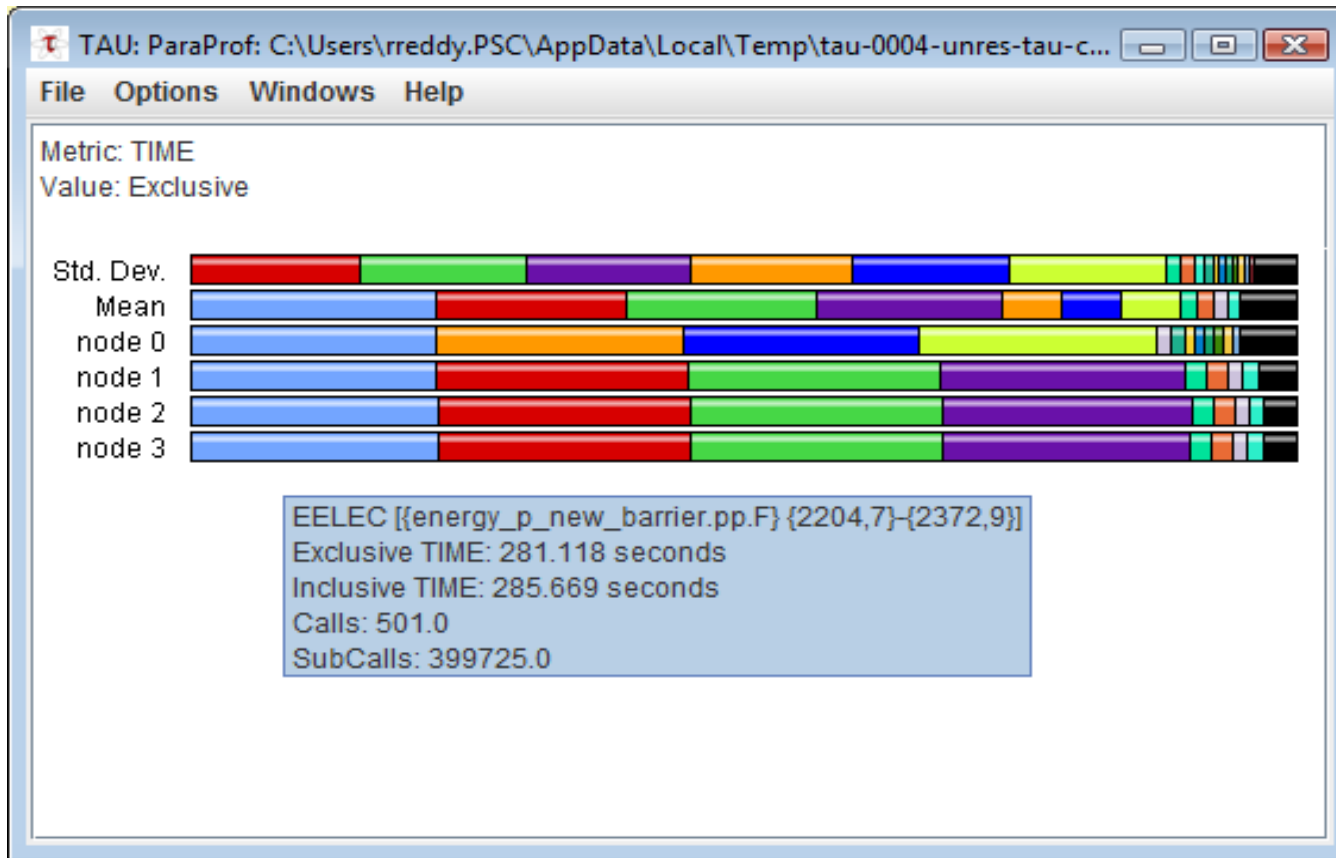
## Before:

```
double precision function scalar(u,v)
double precision u(3),v(3)
    scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
return
end
```

## After:

```
double precision function scalar(u,v)
double precision u(3),v(3)
    call TAU_PROFILE_TIMER(profiler, 'SCALAR [...]')
    call TAU_PROFILE_START(profiler)
    scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
    call TAU_PROFILE_STOP(profiler)
return
    call TAU_PROFILE_STOP(profiler)
end
```

# Reducing Overhead



**Overhead (time in sec):**

**MD steps base:**  
51.4 seconds

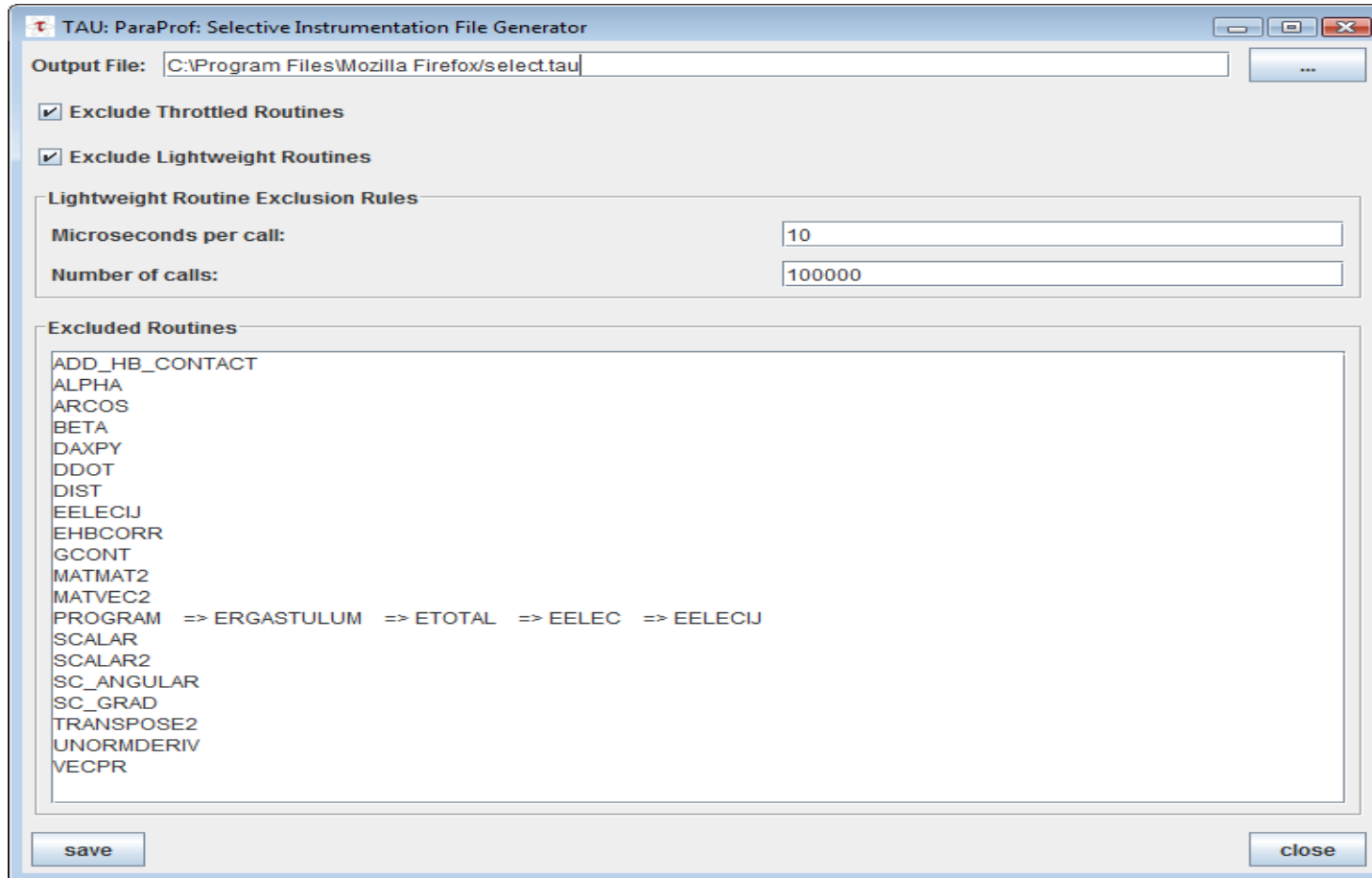
**MD steps with TAU:**  
315 seconds

**Must reduce overhead to  
get meaningful results:**

- In paraprof go to “File” and select “Create Selective Instrumentation File”

# Selective Instrumentation File

**TAU automatically generates a list of routines that you can save to a selective instrumentation file**



The screenshot shows a Windows-style dialog box titled "TAU: ParaProf: Selective Instrumentation File Generator". It contains the following elements:

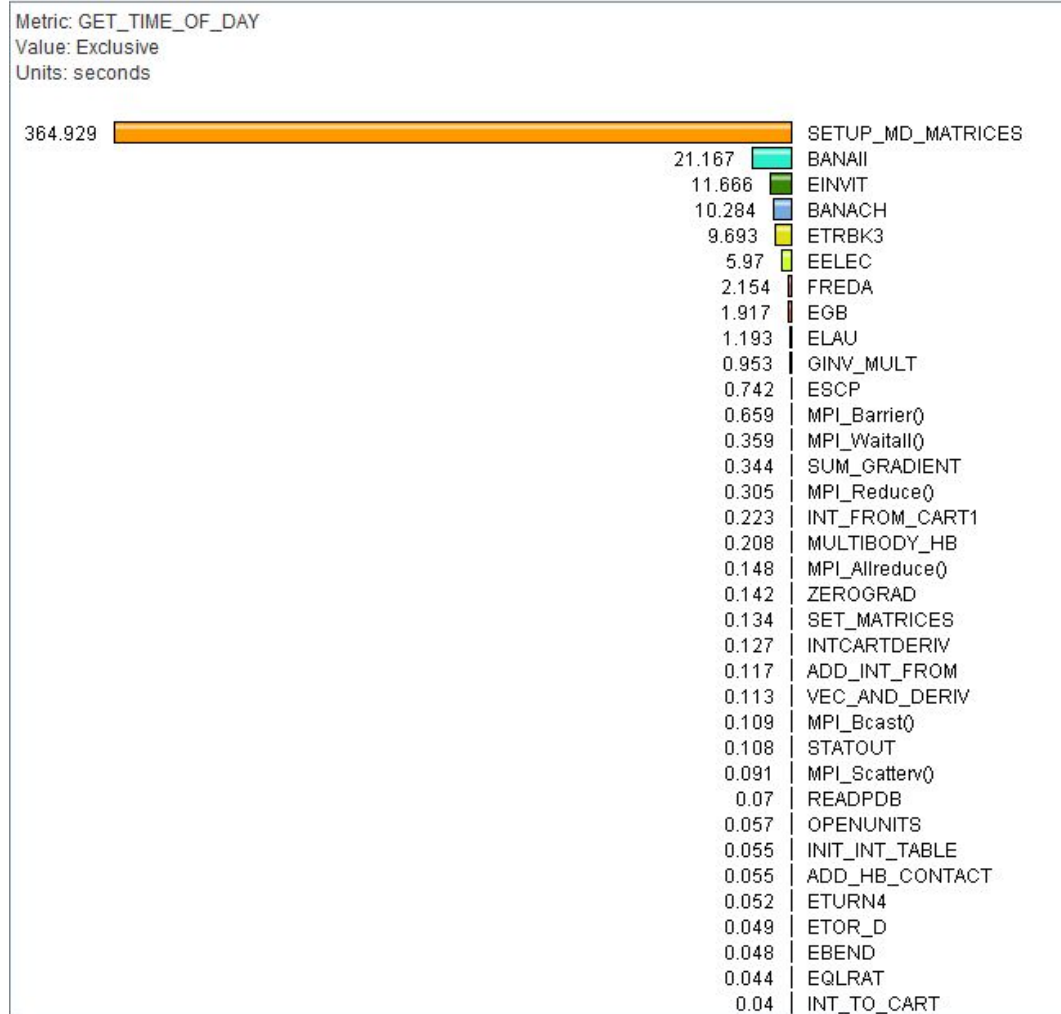
- Output File:** A text field with the path "C:\Program Files\Mozilla Firefox/select.tau" and a browse button ("...").
- Exclude Throttled Routines:** A checked checkbox.
- Exclude Lightweight Routines:** A checked checkbox.
- Lightweight Routine Exclusion Rules:**
  - Microseconds per call:** A text field containing "10".
  - Number of calls:** A text field containing "100000".
- Excluded Routines:** A large text area containing a list of routine names:

```
ADD_HB_CONTACT
ALPHA
ARCOS
BETA
DAXPY
DDOT
DIST
EELECIJ
EHBCORR
GCONT
MATMAT2
MATVEC2
PROGRAM => ERGASTULUM => ETOTAL => EELEC => EELECIJ
SCALAR
SCALAR2
SC_ANGULAR
SC_GRAD
TRANSPPOSE2
UNORMDERIV
VECPR
```
- Buttons:** "save" and "close" buttons at the bottom.

# Selective Instrumentation File

- Automatically generated file essentially eliminates overhead in instrumented UNRES
- In addition to eliminating overhead, use this to specify:
  - Files to include/exclude
  - Routines to include/exclude
  - Directives for loop instrumentation
  - Phase definitions
- Specify the file in TAU\_OPTIONS and recompile:  
**setenv TAU\_OPTIONS "-optVerbose -optKeepFiles  
-optPreProcess -optTauSelectFile=select .tau"**
- <http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01.html>

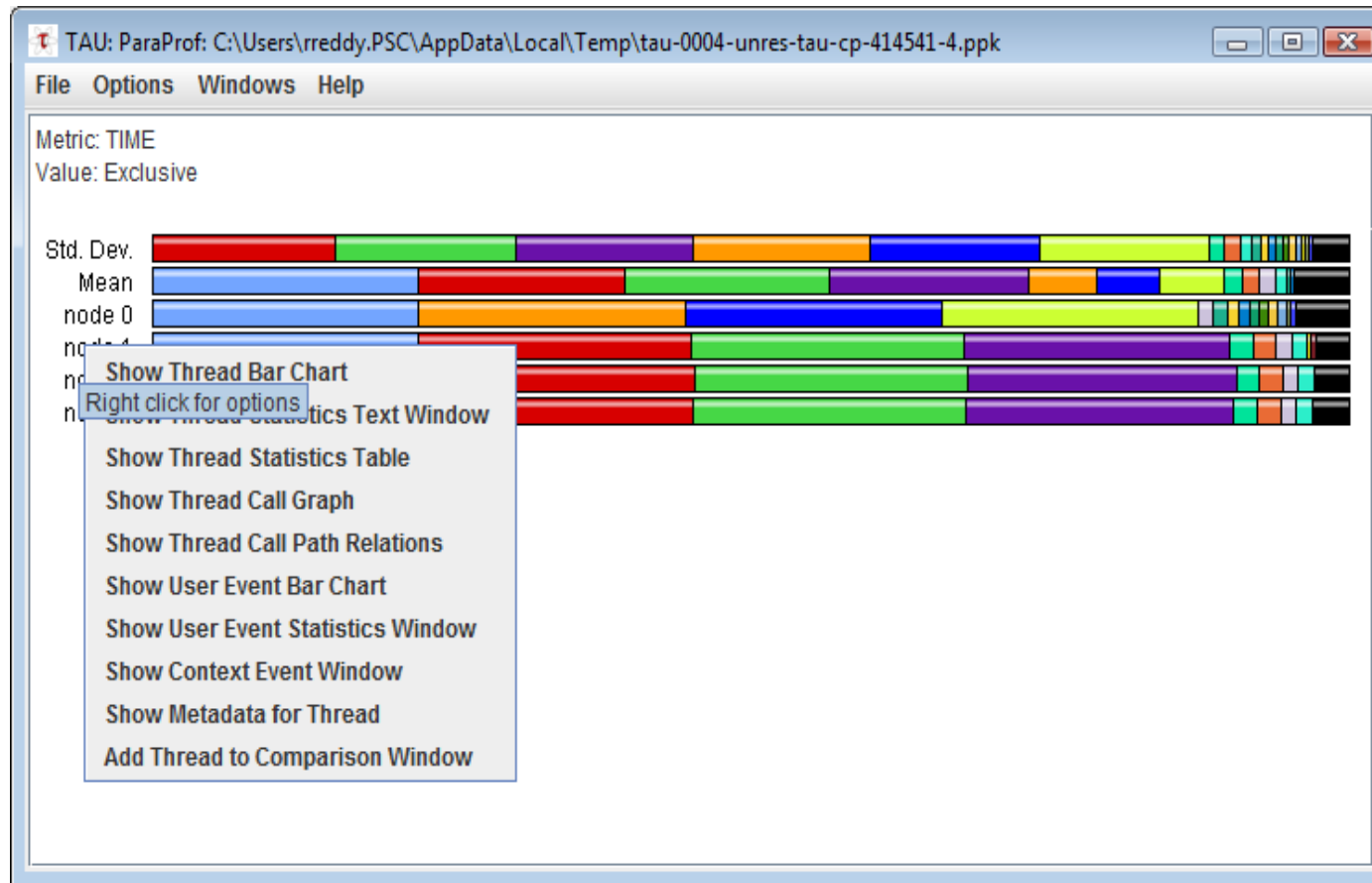
# Key UNRES Functions in TAU (with Startup Time)



## Getting a Call Path with TAU

- Why do I need this?
  - To optimize a routine, you often need to know what is above and below it
  - e.g. Determine which routines make significant MPI calls
  - Helps with defining phases: stages of execution within the code that you are interested in
- To get callpath info, do the following at runtime:
  - setenv TAU\_CALLPATH 1 (this enables callpath)
  - setenv TAU\_CALLPATH\_DEPTH 5 (defines depth)
- Higher depth introduces more overhead – keep as low as possible

# Getting Call Path Information



**Right click  
name of node  
and select  
“Show Thread  
Call Graph”**

# Phase Profiling: Isolate regions of code execution

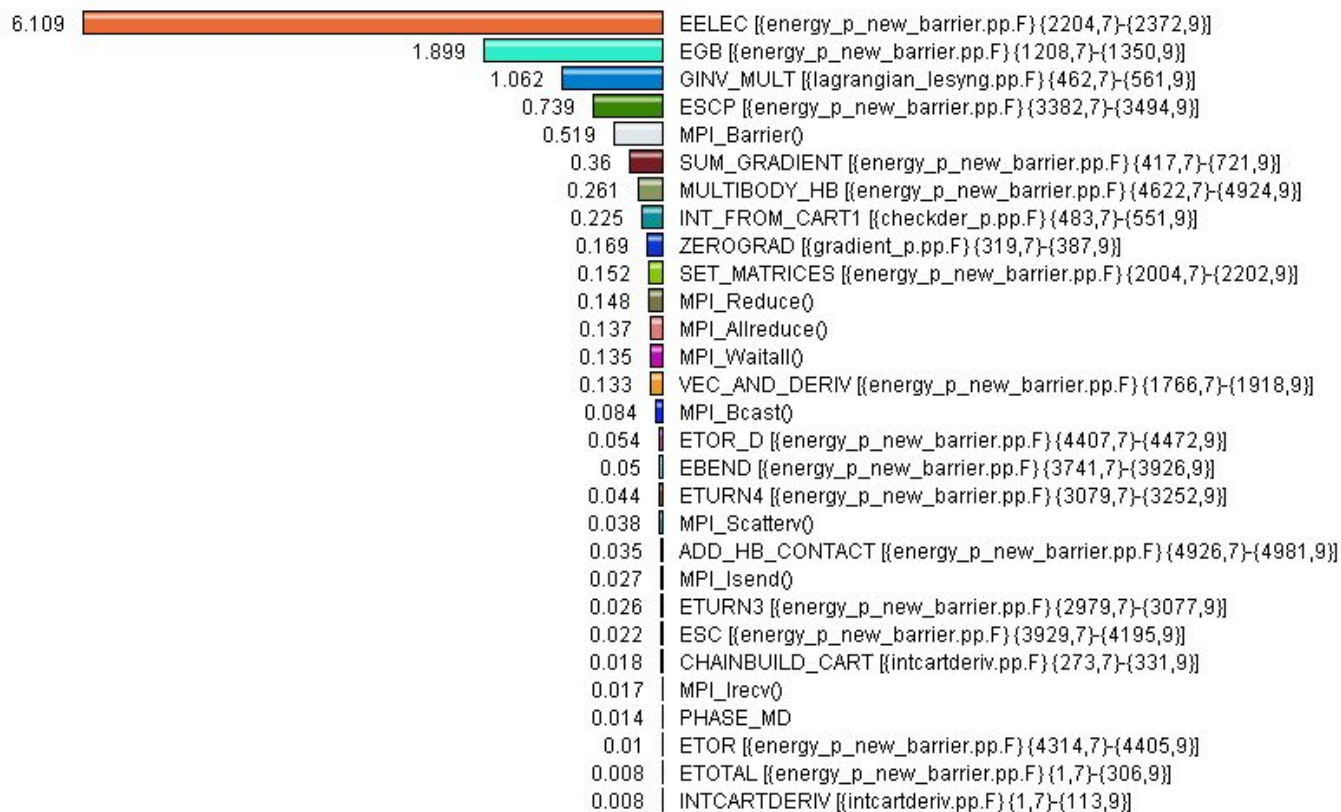
- Eliminated overhead, now we need to deal with startup time:
  - Choose a region of the code of interest: e.g. the main computational kernel
  - Determine where in the code that region begins and ends (call path can be helpful)
  - Then put something like this in selective instrumentation file:

```
static phase name="foo1_bar" file="foo.c" line=26 to line=27
```

- Recompile and rerun

# Key UNRES Functions (MD Time Only)

Phase: PHASE\_MD  
Metric: TIME  
Value: Exclusive  
Units: seconds



# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Hardware Counters

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

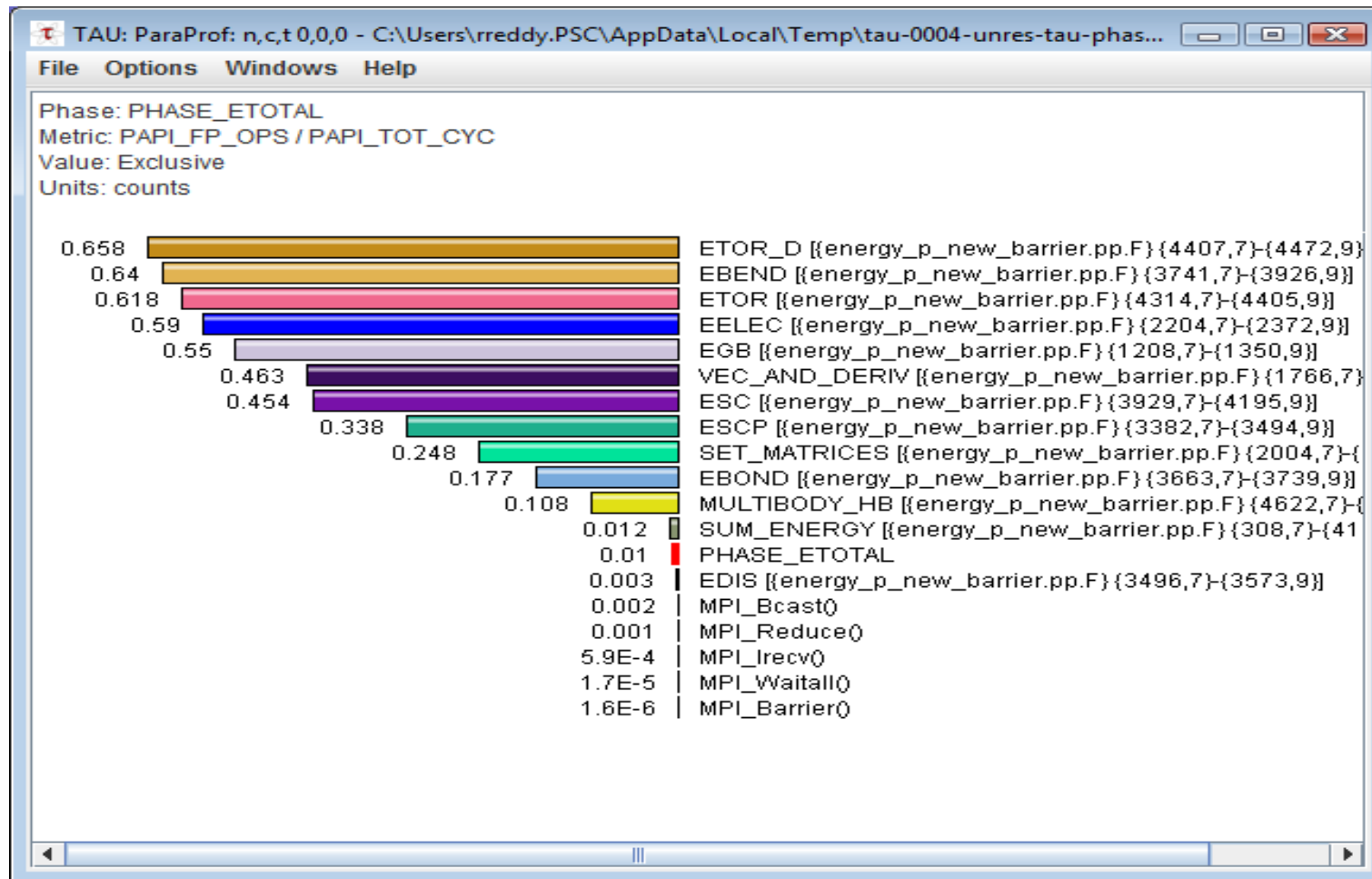
- 1. Whole program timing**
- 2. Cache behaviors**
- 3. Branch behaviors**
- 4. Memory and resource access patterns**
- 5. Pipeline stalls**
- 6. Floating point efficiency**
- 7. Instructions per cycle**
- 8. Subroutine resolution**
- 9. Process or thread attribution**

# Detecting Serial Performance Issues

- Identify hardware performance counters of interest
  - `papi_avail`
  - `papi_native_avail`
  - Run these commands on compute nodes! Login nodes will give you an error.
- Run TAU (perhaps with phases defined to isolate regions of interest)
- Specify PAPI hardware counters at run time:

```
setenv TAU_METRICS GET_TIME_OF_DAY:PAPI_FP_OPS:PAPI_TOT_CYC
```

# Perf of EELEC (peak is 2)

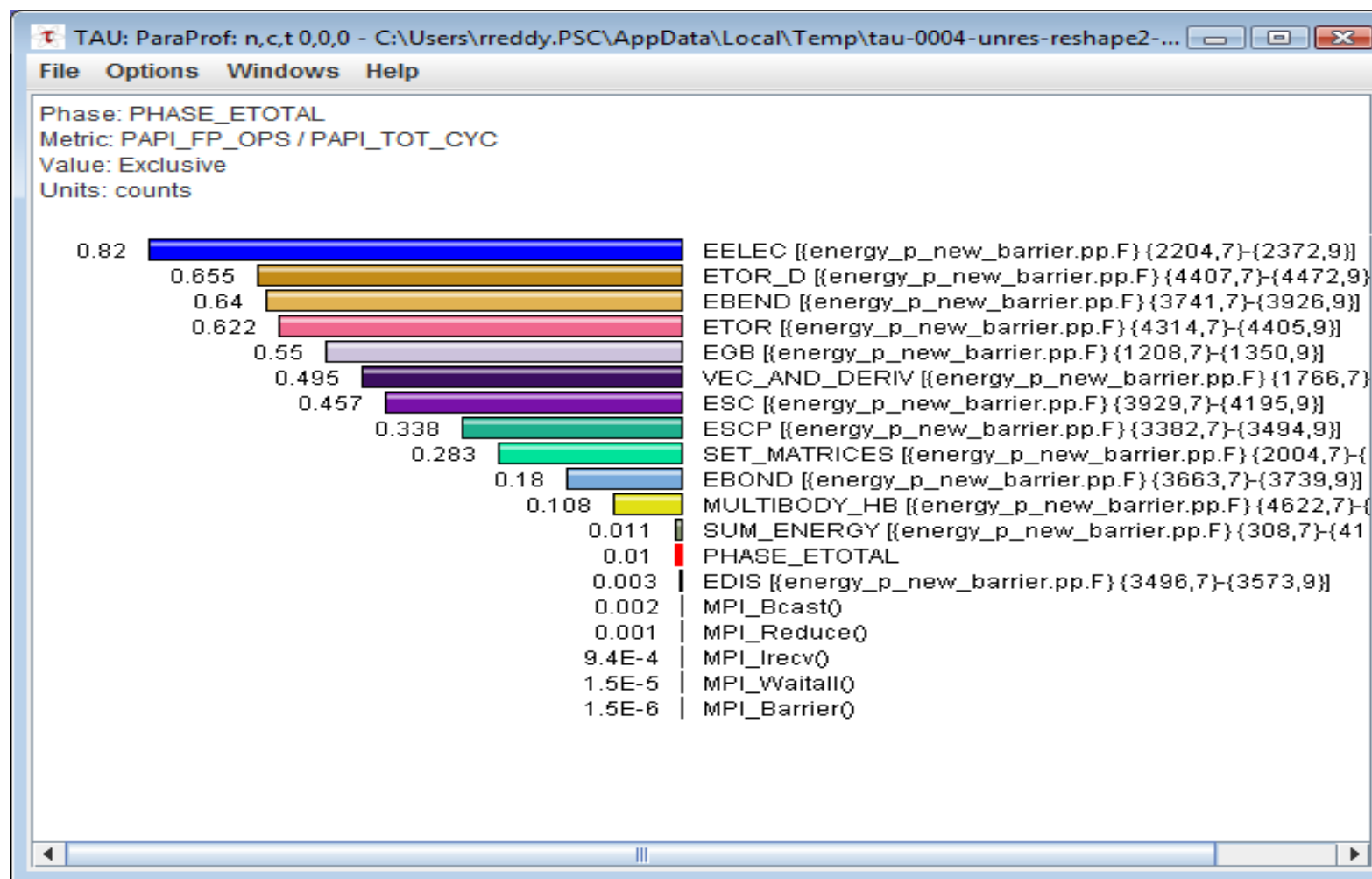


Go to: Paraprof  
manager  
Options->"Show  
derived metrics  
panel"

# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# EELEC – After forcing inlining



## Further Info on Serial Optimization

- Tools help you find issues, areas of code to focus on – solving issues is application and hardware specific
- Good resource on techniques for serial optimization:

“Performance Optimization of Numerically Intensive Codes” Stefan Goedecker, Adolfo Hoisie, SIAM, 2001.

CI-Tutor course: “Performance Tuning for Clusters” <http://ci-tutor.ncsa.illinois.edu/>

# Performance Engineering: Procedure

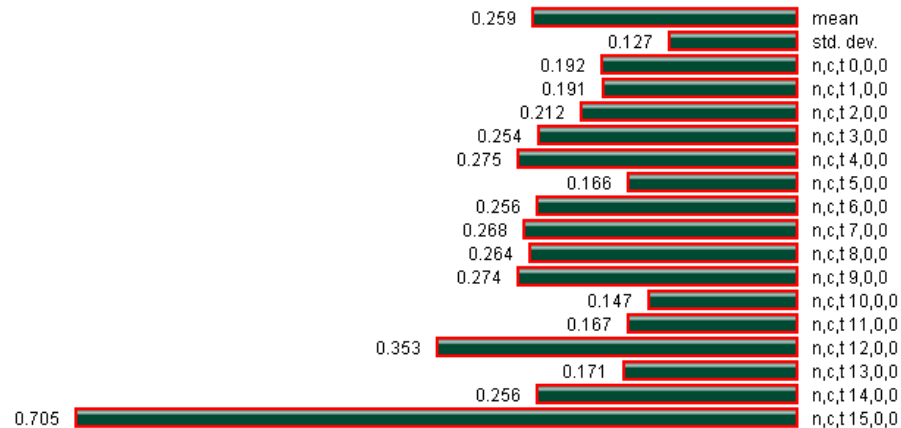
- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - **Identify load balancing issues, serial regions**
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Detecting Parallel Performance Issues: Load Imbalance

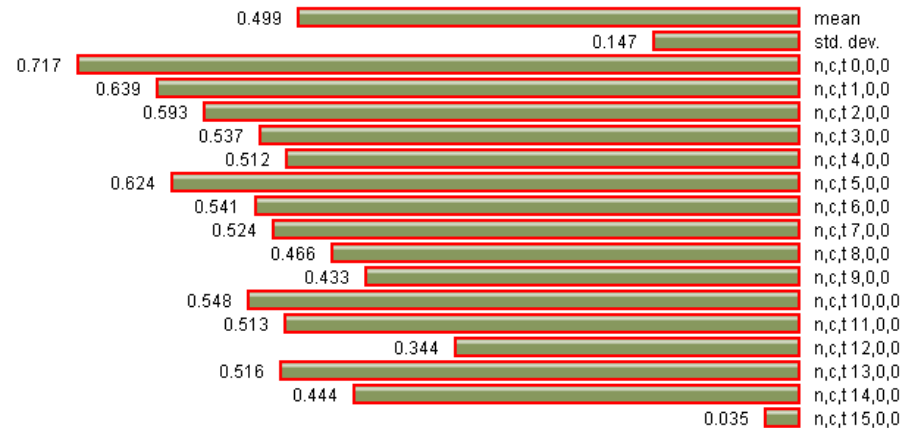
- Examine timings of functions in your region of interest
  - If you defined a phase, from paraprof window, right-click on phase name and select: 'Show profile for this phase'
- To look at load imbalance in a particular function:
  - Left-click on function name to look at timings across all processors
- To look at load imbalance across all functions:
  - In Paraprof window go to 'Options'
  - Uncheck 'Normalize' and 'Stack Bars Together'

# Load Imbalance

Phase: PHASE\_ETOTAL  
 Name: MULTIBODY\_HB [{energy\_p\_new\_barrier.pp.F} {4622,7}-{4924,9}]  
 Metric Name: TIME  
 Value: Exclusive  
 Units: seconds



Phase: PHASE\_ETOTAL  
 Name: MPI\_Barrier()  
 Metric Name: TIME  
 Value: Exclusive  
 Units: seconds



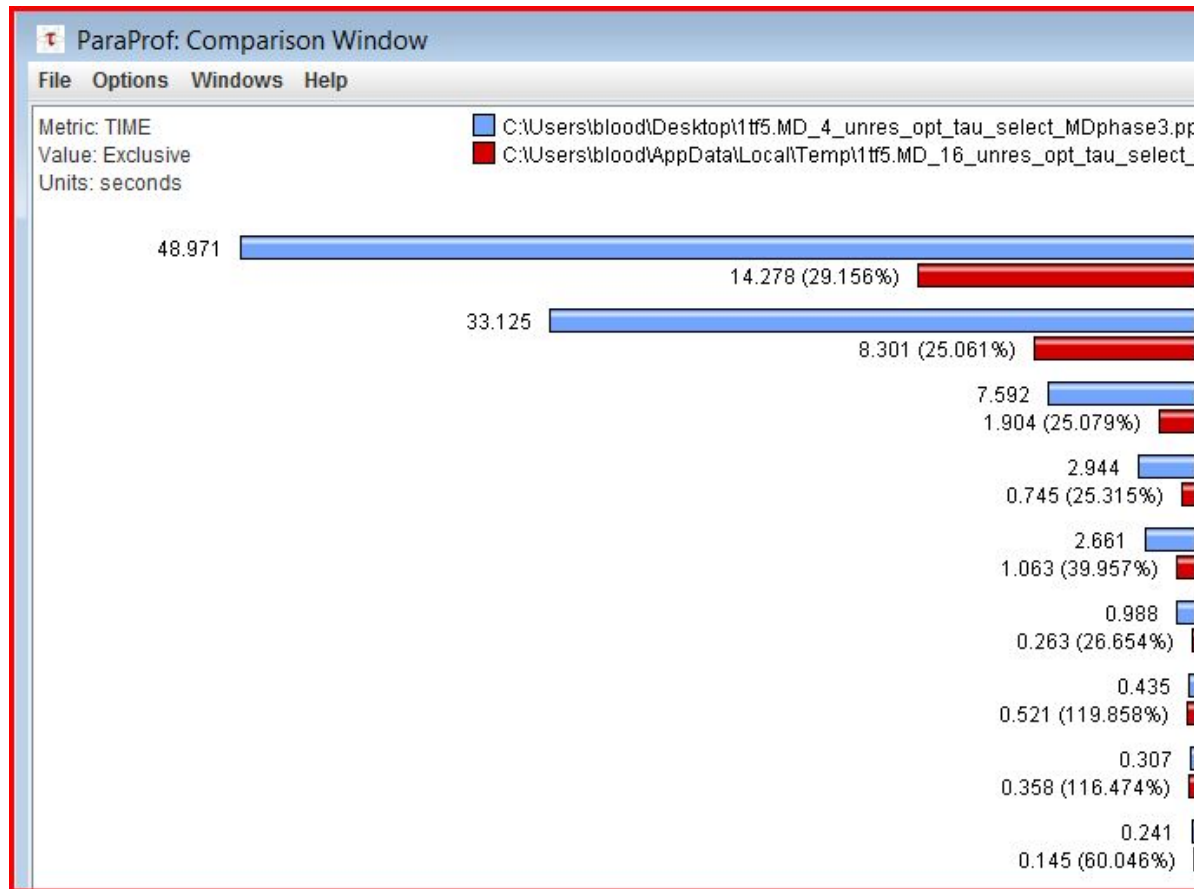
**Load imbalance on one processor causing other processors to idle in MPI\_Barrier**

May need to change how data is distributed, or even change underlying algorithm.

# Detecting Parallel Performance Issues: Serial Bottlenecks

- To identify scaling bottlenecks, do the following for each run in a scaling study (e.g. 2-64 cores):
  - 1) In Paraprof manager right-click “Default Exp” and select “Add Trial”. Find packed profile file and add it.
  - 2) If you defined a phase, from main paraprof window select: Windows -> Function Legend-> Filter->Advanced Filtering
  - 3) Type in the name of the phase you defined, and click ‘OK’
  - 4) Return to Paraprof manager, right-click the name of the trial, and select “Add to Mean Comparison Window”
- Compare functions across increasing core counts

# Function Scaling and Serial Bottlenecks



**Identify which functions need to scale better, or be parallelized, in order to increase overall scalability.**

**Find which communication routines are starting to dominate runtimes.**

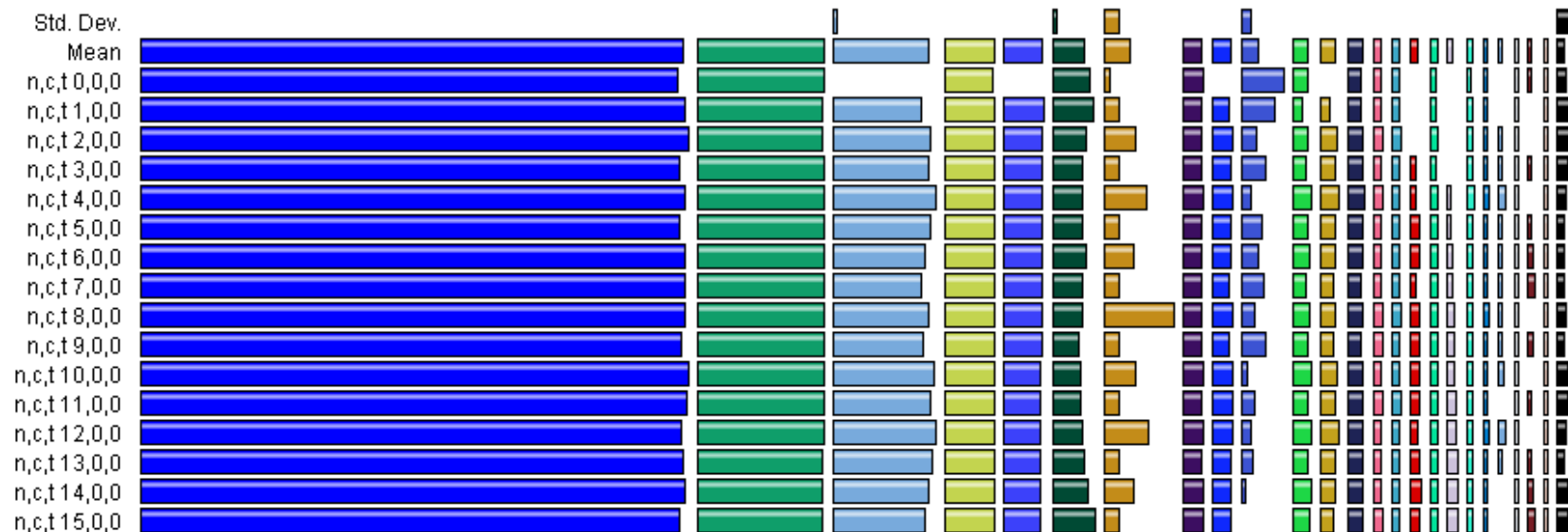
**Use call path information to find where those communication routines are being called**

# Major Serial Bottleneck and Load Imbalance in UNRES Eliminated

Phase: PHASE\_MD

Metric: TIME

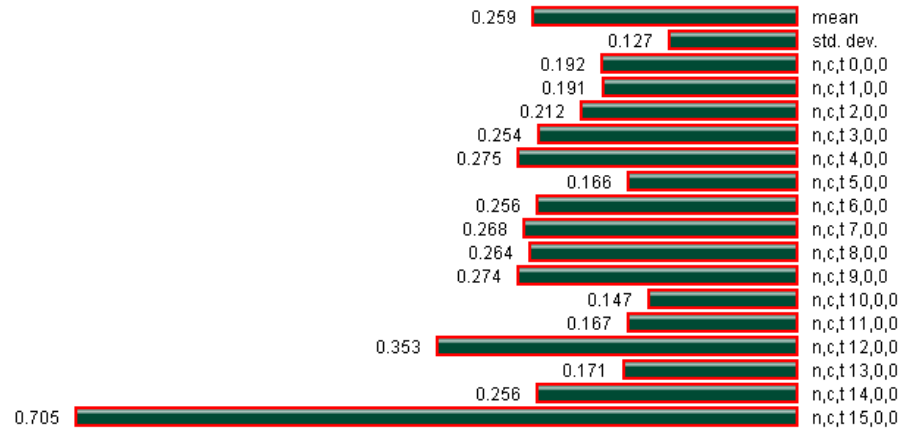
Value: Exclusive



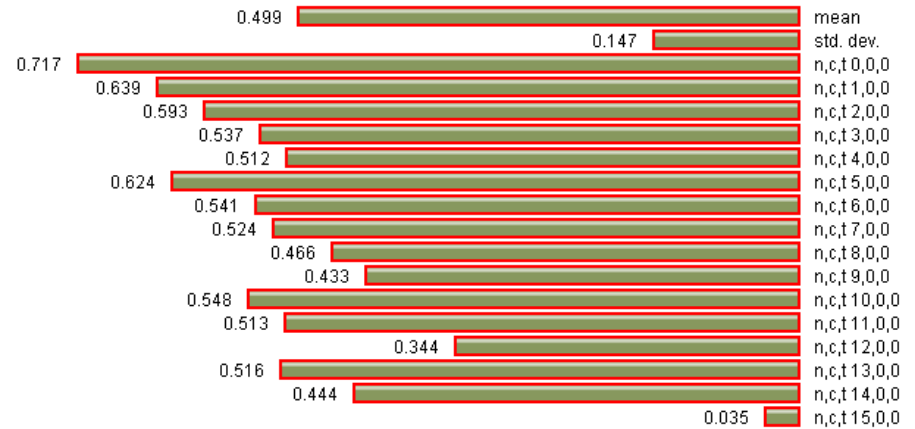
- Due to 4x faster serial algorithm the balance between computation and communication has shifted – communication must be more efficient to scale well
- Code is undergoing another round of profiling and optimization

# Next Iteration of Performance Engineering with Optimized Code

Phase: PHASE\_ETOTAL  
Name: MULTIBODY\_HB [{energy\_p\_new\_barrier.pp.F} {4622,7}-{4924,9}]  
Metric Name: TIME  
Value: Exclusive  
Units: seconds



Phase: PHASE\_ETOTAL  
Name: MPI\_Barrier()  
Metric Name: TIME  
Value: Exclusive  
Units: seconds



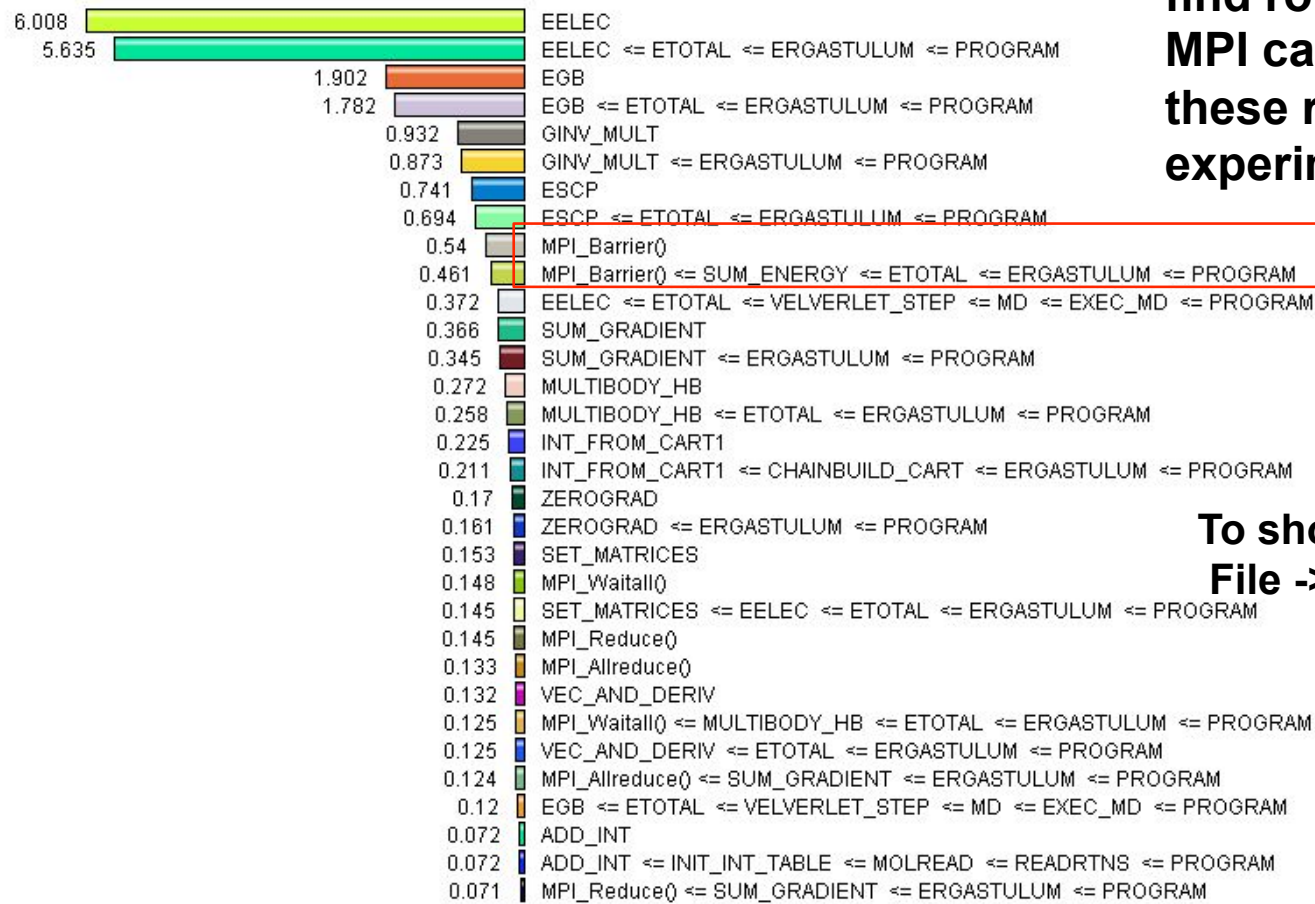
**Load imbalance on one processor apparently causing other processors to idle in MPI\_Barrier**

# Performance Engineering: Procedure

- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall parallel performance (scaling)
  - Identify functions where code spends most time (this may change at high core counts)
  - Instrument those functions
  - Identify load balancing issues, serial regions
  - Identify communication bottlenecks--use tracing to help identify cause and effect

# Use Call Path Information: MPI Calls

Metric: GET\_TIME\_OF\_DAY  
Value: Exclusive  
Units: seconds



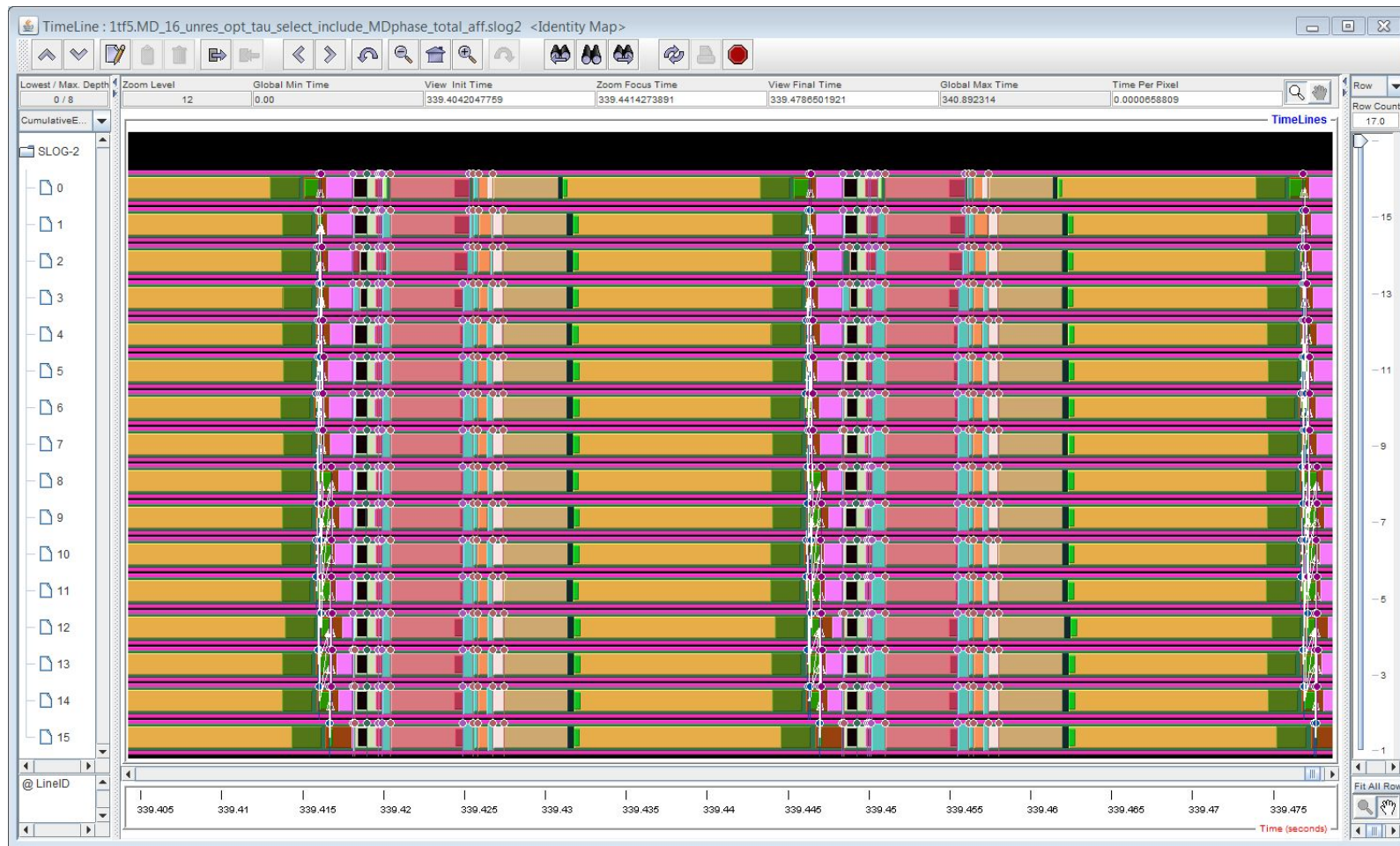
Use call path information to find routines from which key MPI calls are made. Include these routines in tracing experiment.

To show source locations select:  
**File -> Preferences**

# Generating a Trace

- At runtime: `setenv TAU_TRACE 1`
- Follow directions here to analyze:  
<http://www.psc.edu/general/software/packages/tau/TAU-quickref.pdf>
- Insight into causes of communication bottlenecks
  - Duration of individual MPI calls
  - Use of blocking calls
  - Posting MPI calls too early or too late
  - Opportunities to overlap computation and communication

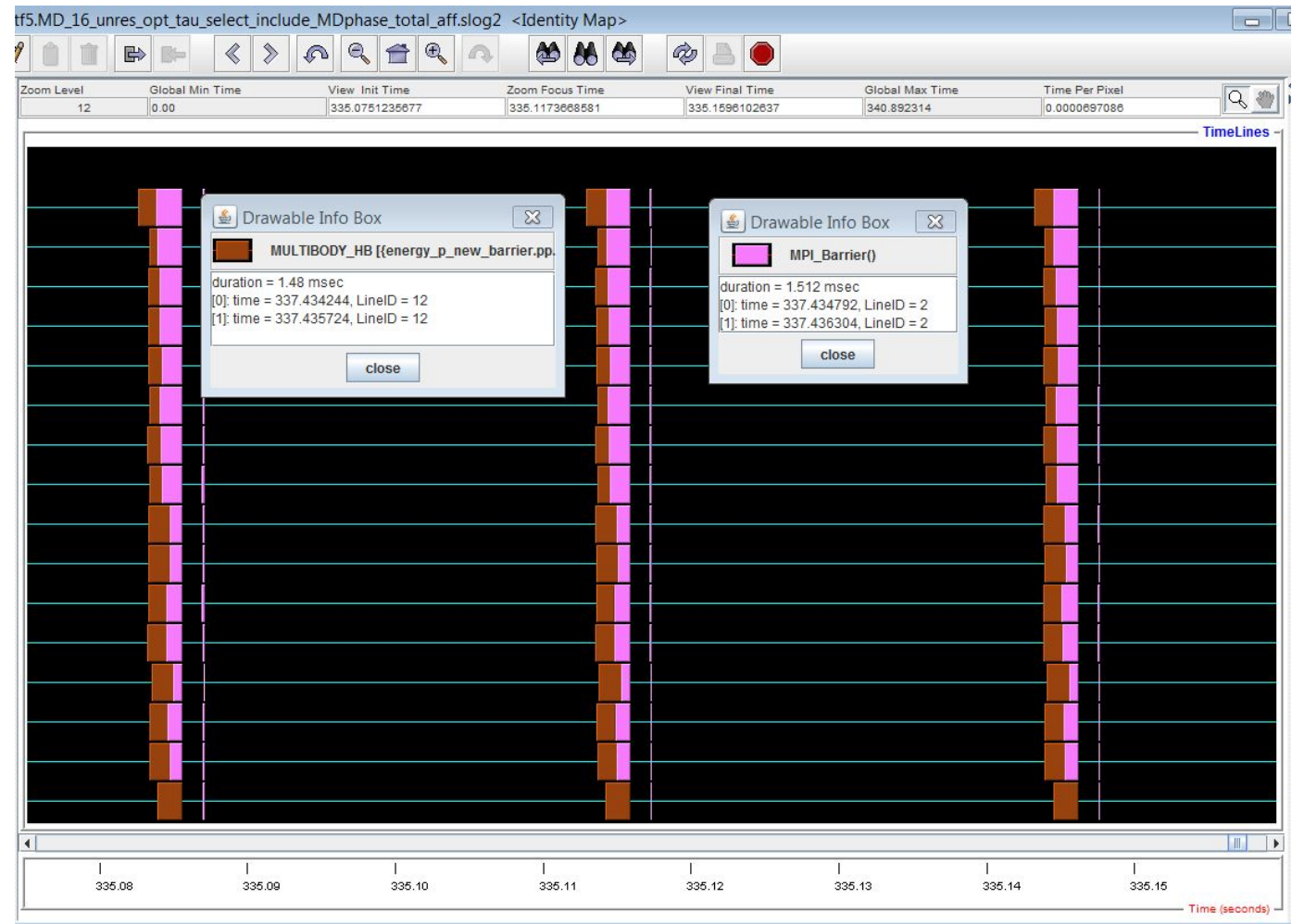
# TAU Trace of UNRES Timesteps in Jumpshot



# Time Resolved Examination of Load Imbalance

**Clear other functions  
to focus on  
problematic  
functions.**

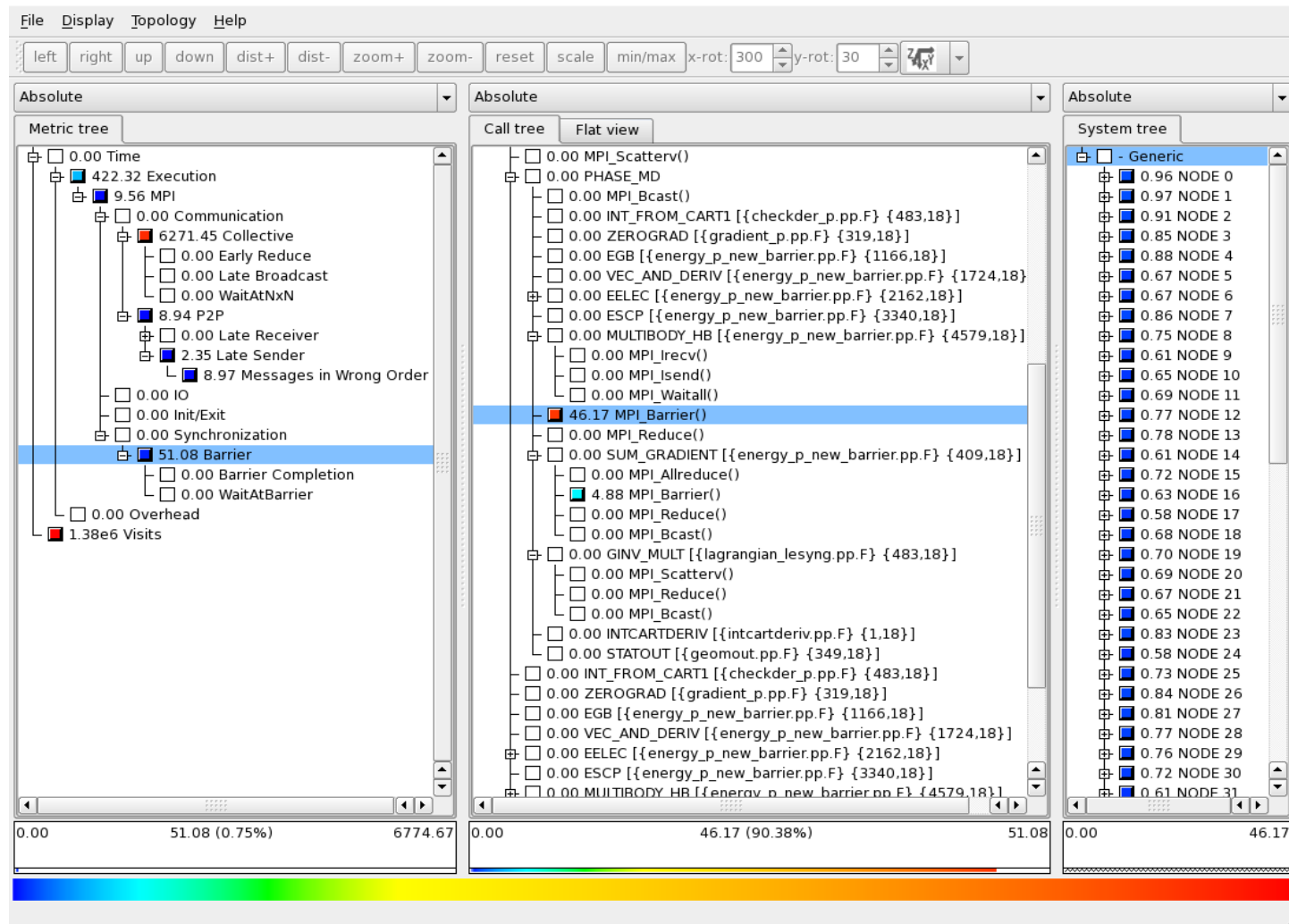
**In addition to node N  
taking much longer,  
Multibody\_HB always  
starts early on node 0  
and late on node N**



# Issues with Tracing

- At high processor counts the amount of data becomes overwhelming
- Very selective instrumentation is critical to manage data
- Also need to isolate the computational kernel and trace for minimum number of iterations to see patterns
- Complexity of manually analyzing traces on thousands of processors is an issue
- SCALASCA attempts to do automated analysis of traces to determine communication problems
- Vampir, Intel Trace Analyzer: cutting-edge trace analyzers (but not free)

# Automatic Trace Analysis with SCALASCA



## Some Take-Home Points

- Good choice of (serial and parallel) algorithm is most important
- Performance measurement can help you determine if algorithm and implementation is good
- Do compiler and MPI parameter optimizations first
- Check/optimize serial performance before investing a lot of time in improving scaling
- Choose the right tool for the job
- Know when to stop: 80:20 rule
- TeraGrid staff and tool developers collaborate with code developers to help with performance engineering of parallel codes

# Hands-On

- Find parallel performance issues in a production scientific application using TAU
- Exercises posted on Google groups:
  - If you have access to **Kraken** or **Ranger** look at:
    - UNRES\_Performance\_Profiling\_Exercises.pdf
  - If you have access to **QueenBee** look at:
    - LAMMPS\_Performance\_Profiling\_Exercises.pdf
- You are encouraged to adapt these to experiment with your own application