Performance Engineering of Parallel Applications

Philip Blood

Pittsburgh Supercomputing Center

blood@psc.edu

International Summer School on HPC Challenges in Computational Sciences

Boulder, CO

Acknowledgment

- Christian Feld, Jülich Supercomputing Centre
- Virtual Institute - High Productivity Supercomputing (VI-HPS)
- Raghu Reddy

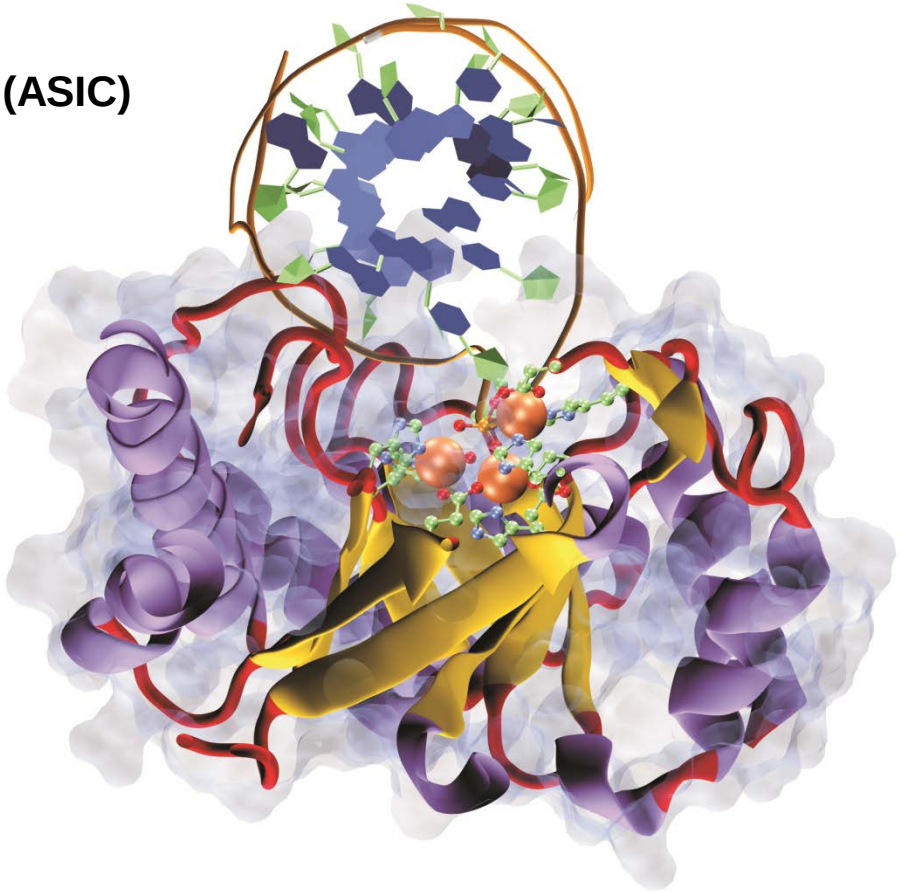
Outline for Performance Sessions

- Thursday:
  - Introduction to performance engineering (Phil Blood)
  - Performance profiling of scientific application with Score-P (Christian Feld)
  - Analysis of performance profiles with TAU Paraprof (Phil Blood)
- Friday:
  - Trace measurement using Score-P (Christian Feld)
  - Trace analysis with Scalasca (Christian Feld)

Fitting algorithms to hardware…and vice versa

Molecular dynamics simulations on Application Specific Integrated Circuit (ASIC)

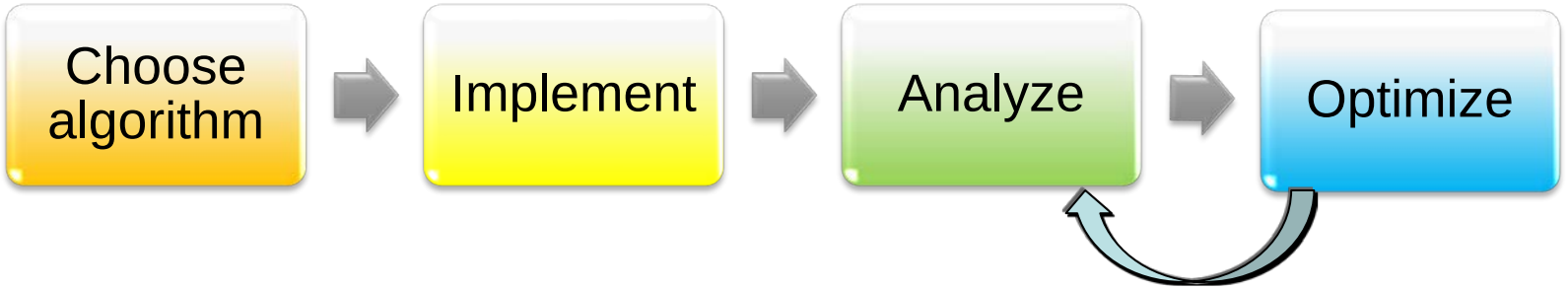
DE Shaw Research



Ivaylo Ivanov, Andrew McCammon, UCSD

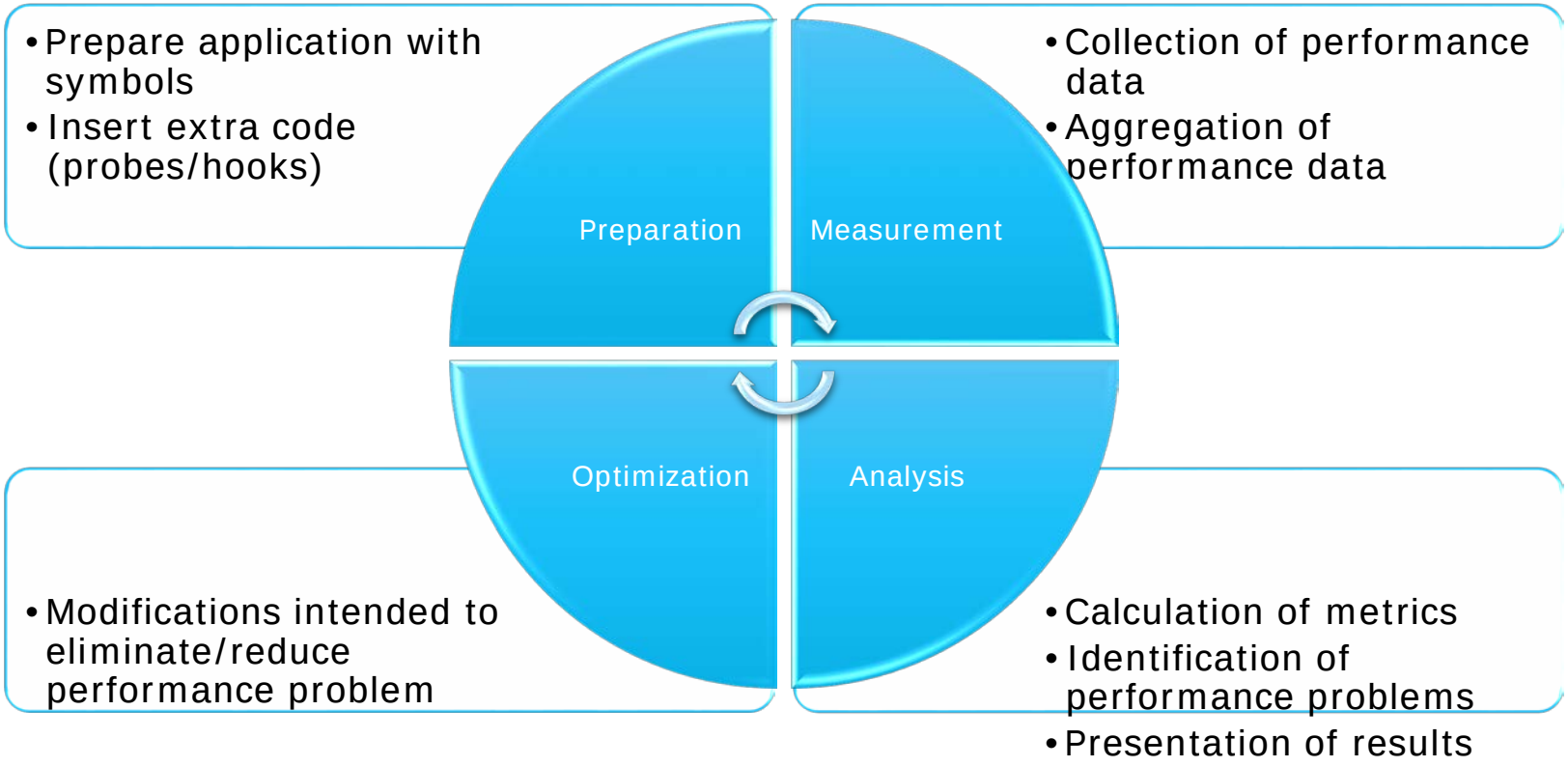
PSC
PITTSBURGH SUPERCOMPUTING CENTER

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 (or rewritten)!
- Analysis may reveal need for new algorithm or completely different implementation rather than optimization
- Focus of this lecture: using tools to assess parallel performance

Performance engineering workflow



Slide courtesy VI-HPS

A little background...

Hardware Counters

- Counters: set of registers that count processor events, like floating point operations, or cycles
- Opteron “Istanbul” had 6 counter registers, so 6 types of events could be monitored simultaneously
- **PAPI: P**erformance **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

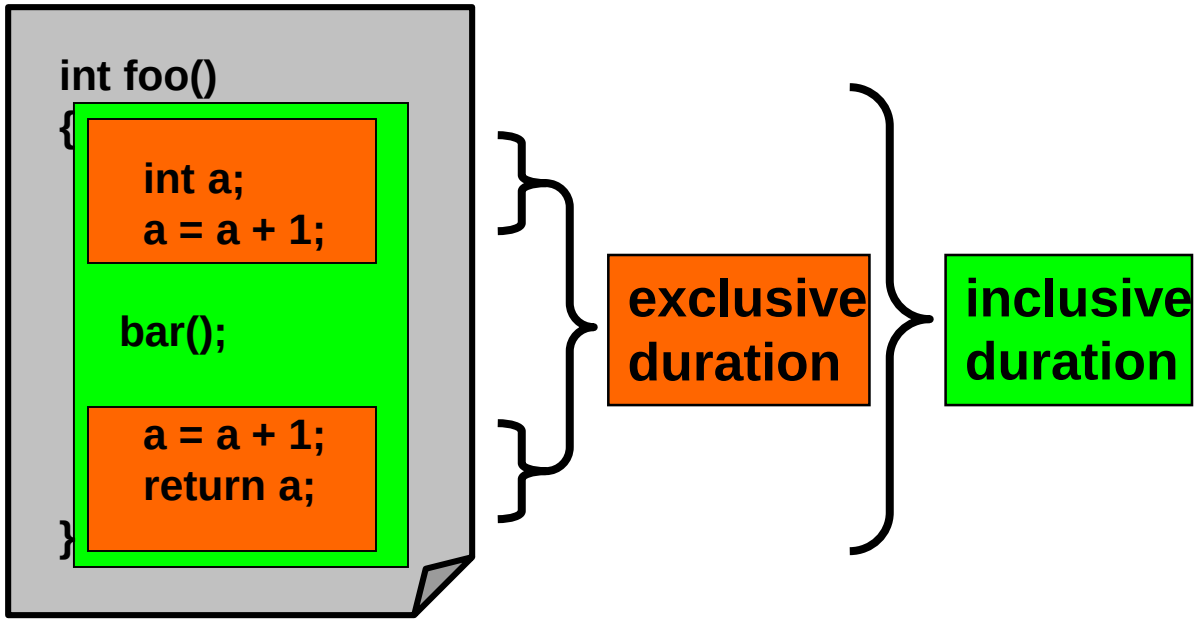
- Portable: uses same routines to access counters across all architectures
- High-level interface
  - Using predefined standard events the same source code can access similar counters across various architectures without modification.
  - **papi_avail**
- Low-level interface
  - Provides access to all machine specific counters (requires source code modification)
  - Increased efficiency and flexibility
  - **papi_native_avail**
- Third-party tools
  - TAU, HPC Toolkit
- Might require linux kernel patch
  - **Direct support in linux kernels ≥ 2.6.31 (use latest PAPI)**

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 data recorded?
  - 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

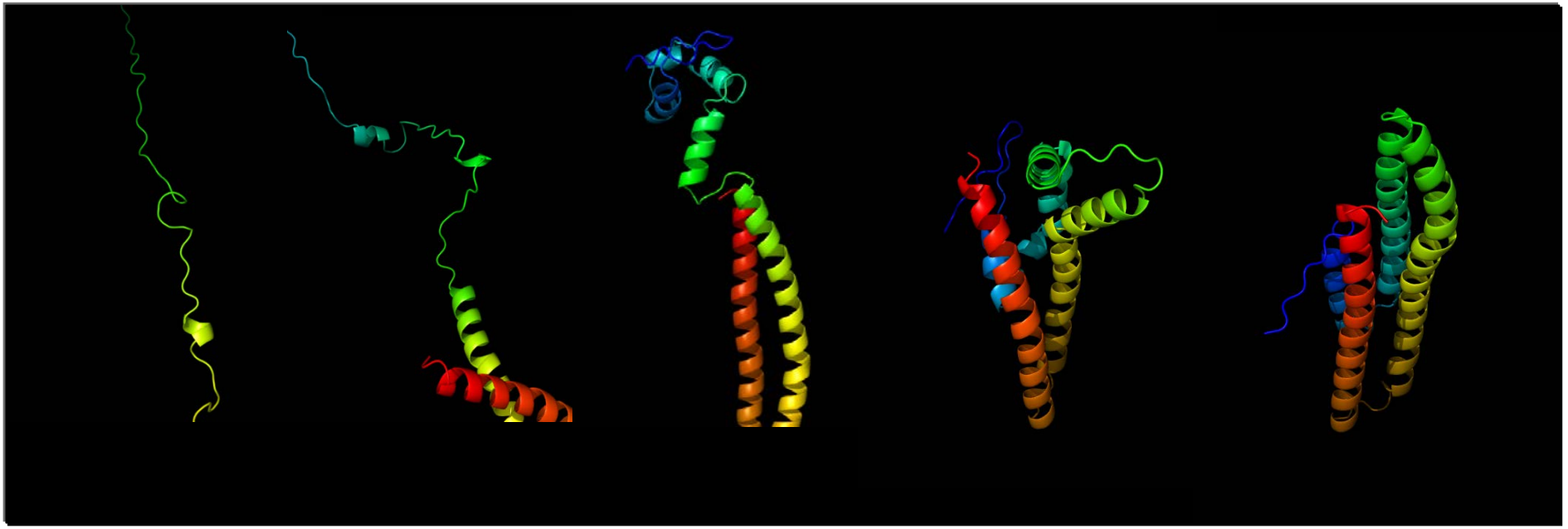
Inclusive and Exclusive Profiles

- 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

PSC
PITTSBURGH SUPERCOMPUTING CENTER

Structure of UNRES

- Two issues
  - Master/Worker code

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

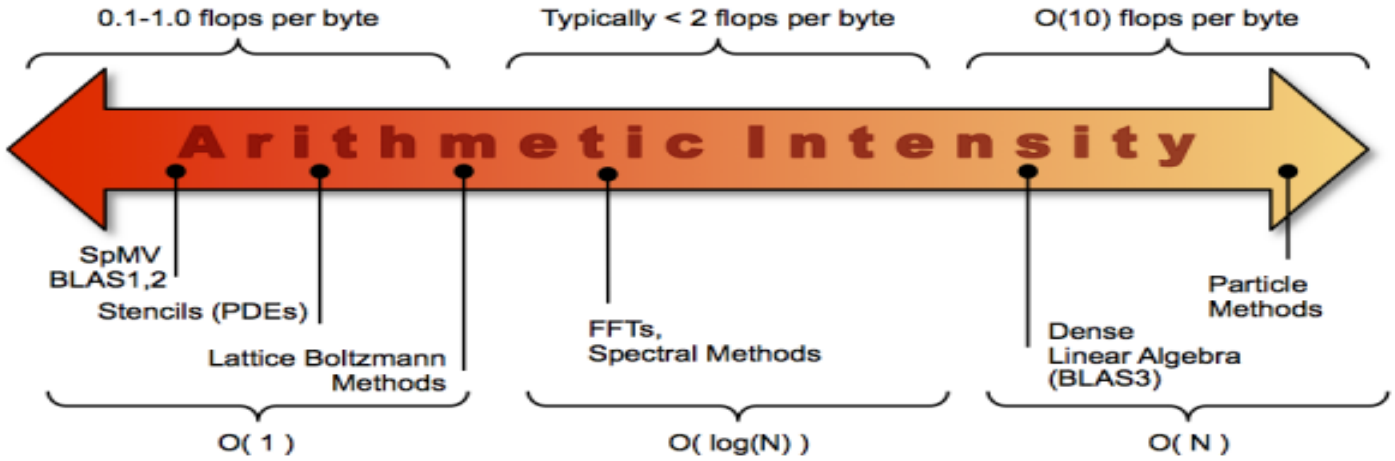
  - Significant startup time: must remove from profiling
    - Setup time: 300 sec
    - MD Time: 1 sec/step
    - Only MD time important for production runs of millions of steps
    - Could run for 30,000 steps to amortize startup!

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”?
  - Depends on the work being done
    - Traditional measure: Percentage of peak performance
  - What performance should I expect with my algorithm?
    - Roofline models: establish performance bounds for various numerical methods
    - Arithmetic Intensity: Ratio of total floating-point operations (FLOPs) to total data movement (bytes)



Source: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

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 = FLOP/cycle * cycles/sec
  - 80:20 rule
- Parallel Performance: Scalability
  - Does run time decrease by 2x when I use 2x cores? (total work remains constant)
    - **Strong scalability**
  - Does run time remain the same when I keep the amount of work per core the same?
    - **Weak scalability**

Use a Sampling Tool for Initial Performance Check

- HPC Toolkit
  - Powerful sampling based tool
  - No recompilation necessary
  - Function level information available
- PerfExpert: TACC-developed automated performance analysis built on HPC Toolkit
- Worth checking out:

http://hpctoolkit.org/

http://www.tacc.utexas.edu/perfexpert

UNRES: Serial Performance

Processor and System Information

==========

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

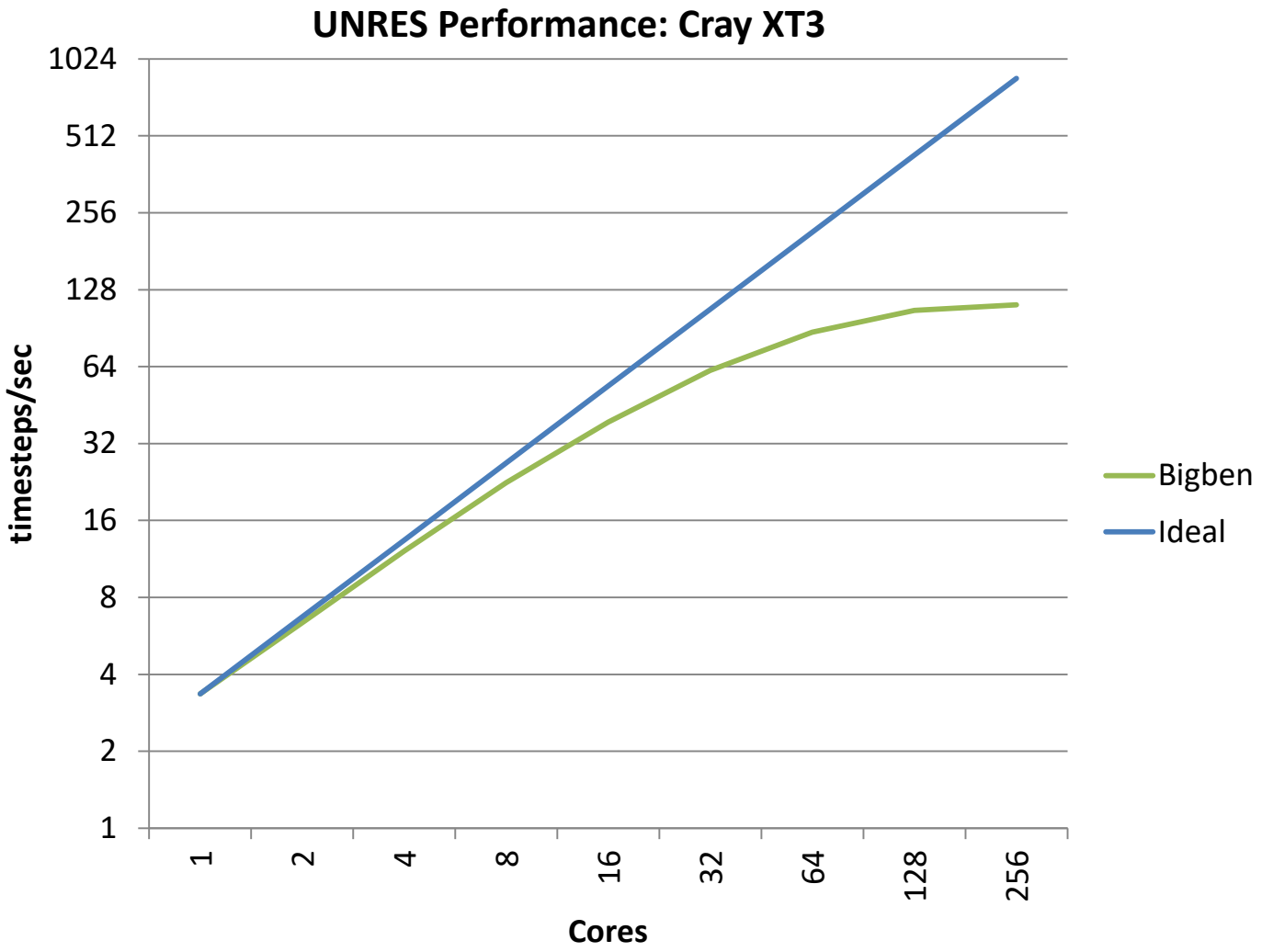
Statistics

==========

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 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!

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
```

UNRES 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 sampling tool**
- **Can use sampling tools 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

Digging Deeper: 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

Choose a tool: there are many!

- VI-HPS maintains a list and tool guide
  - http://www.vi-hps.org/tools/
- Will use TAU as an example in this presentation
- Focus on the general principles rather than specific details
- Christian Feld will take you through specific details using Score-P and Scalasca tools during hands-on session

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.cs.uoregon.edu/research/tau/tau_releases/tau-2.20.1/html/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:

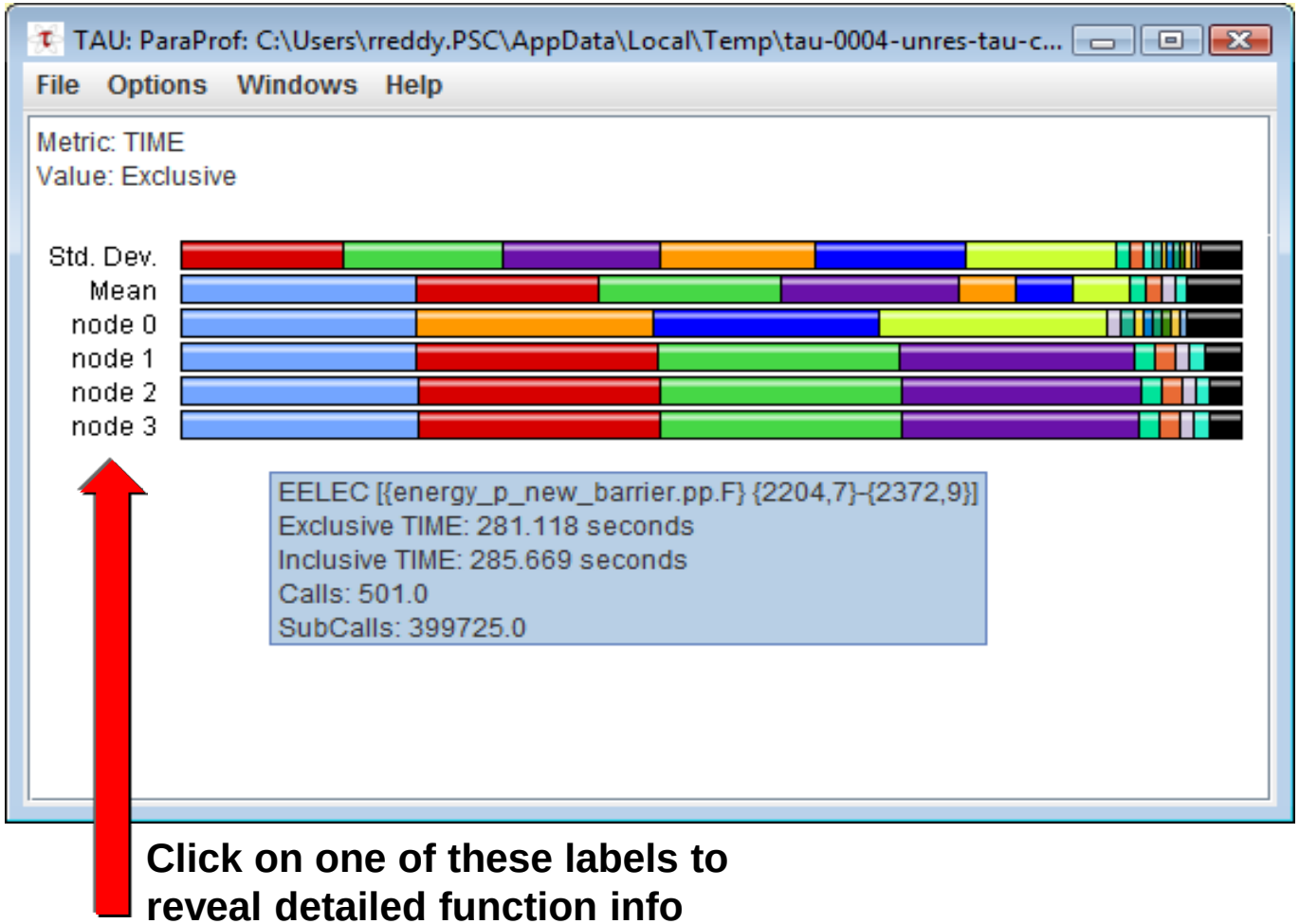
```
double precision function scalar(u,v)
double precision u(3),v(3)
     scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
return
end
```

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

ParaProf Profile Visualization Tool



Click on one of these labels to reveal detailed function info

Overhead (time in sec):

MD steps base:
51.4 seconds

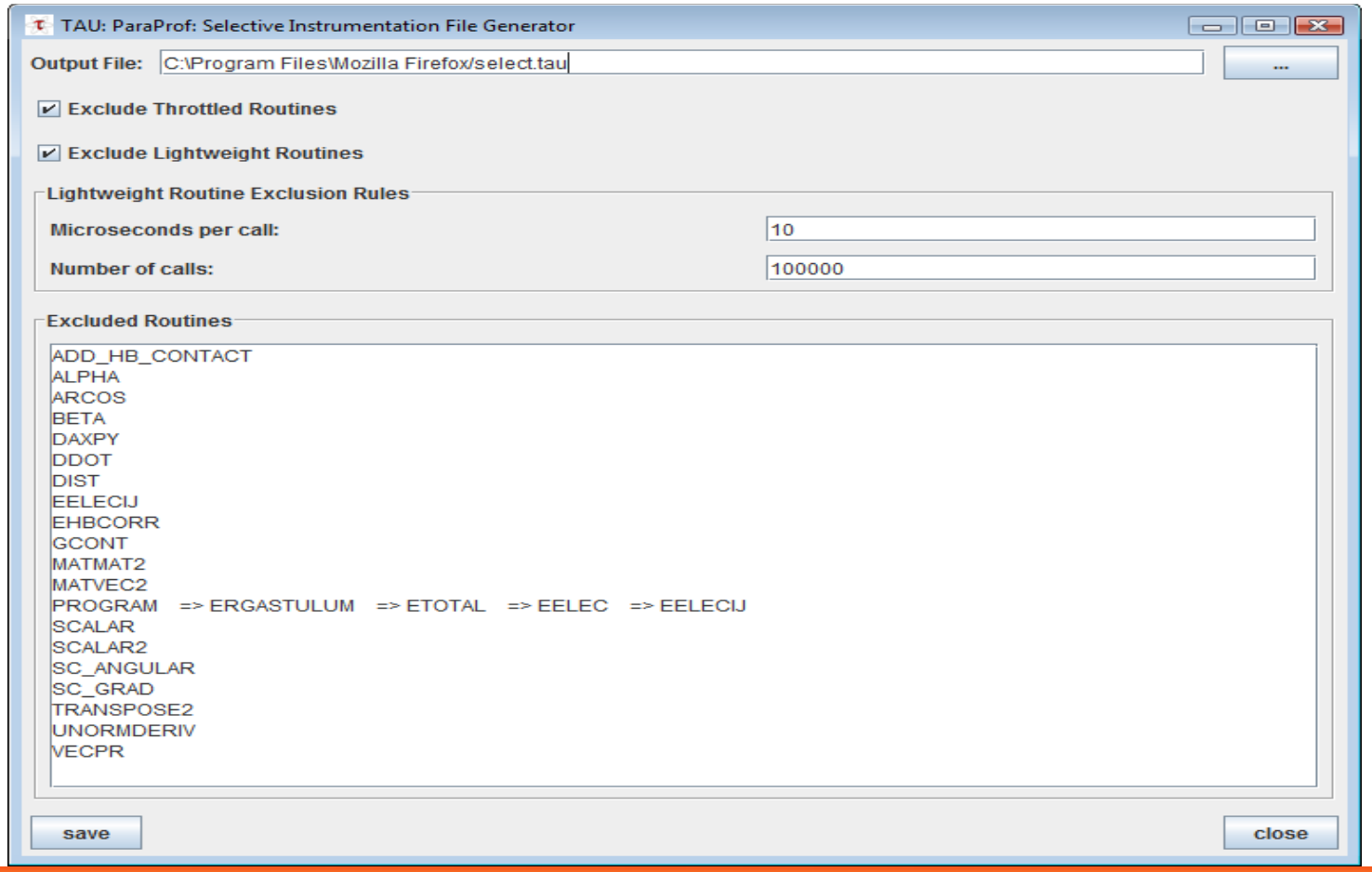
MD steps with TAU:
315 seconds

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



PSC
PITTSBURGH SUPERCOMPUTING CENTER

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

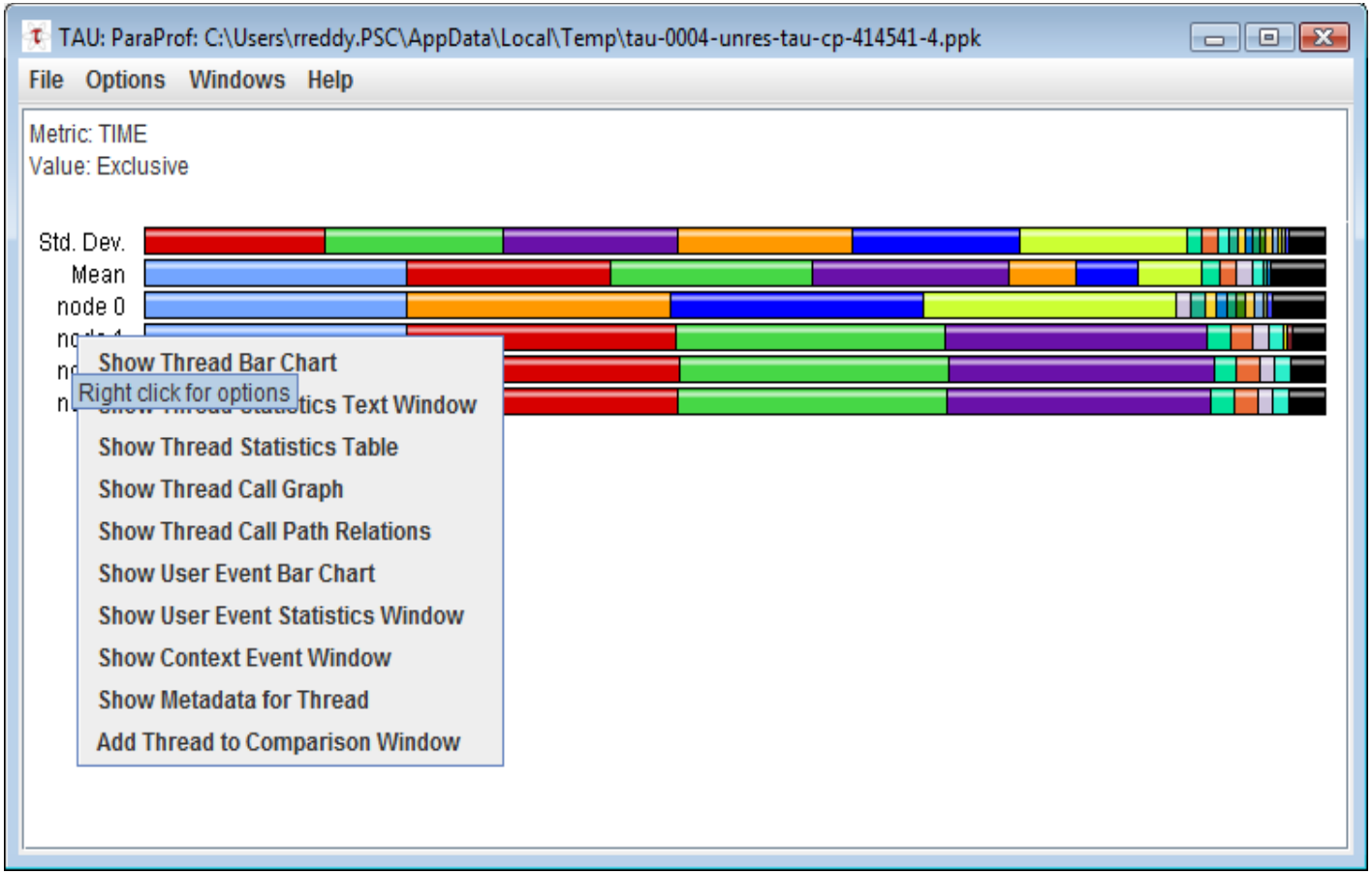
Getting a Call Path with TAU

- 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 in TAU

Getting Call Path Information



Right click name of node and select "Show Thread Call Graph"

PSC
PITTSBURGH SUPERCOMPUTING CENTER

Isolate regions of code execution

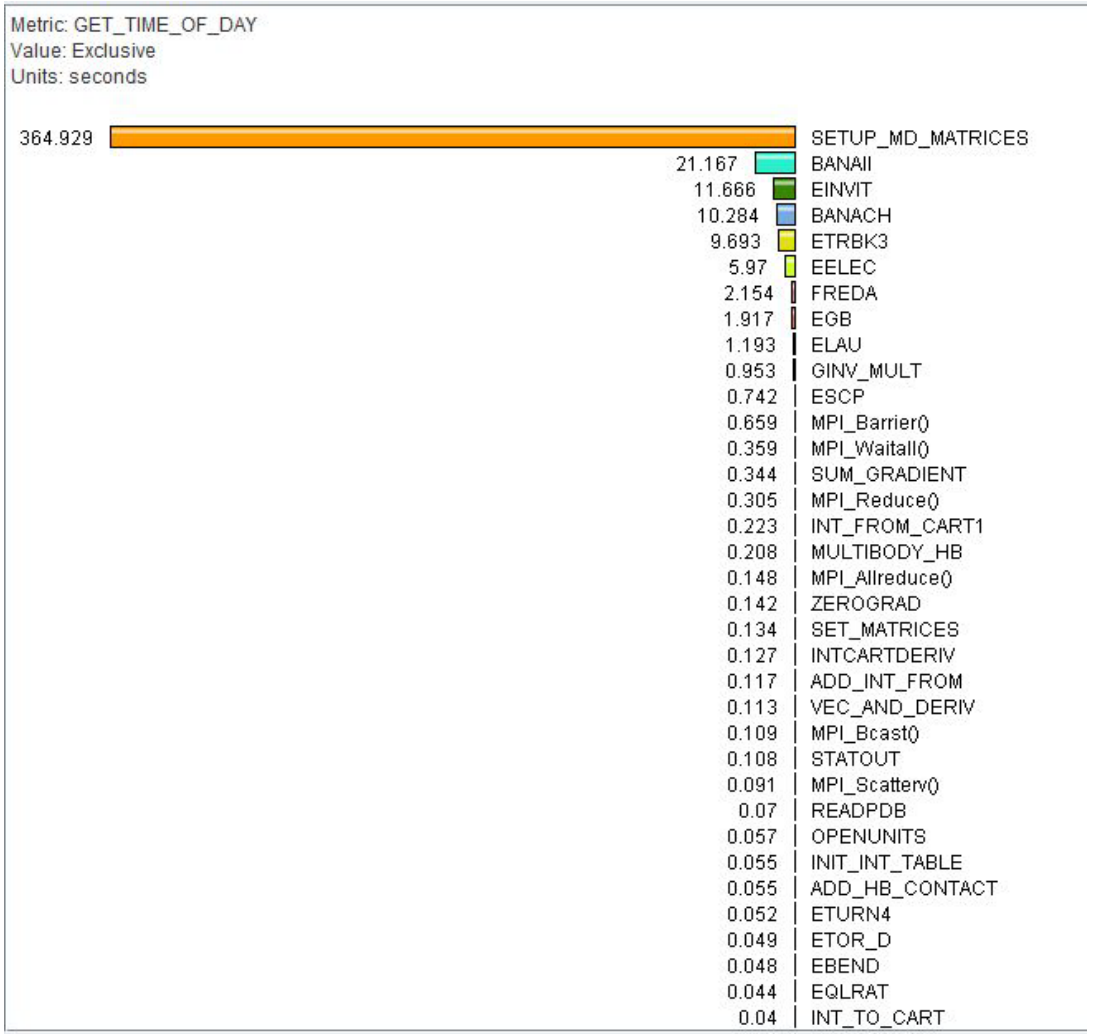
- Eliminated overhead, now we need to deal with startup time:
  - Choose a region of the code of interest: e.g. the main computational kernel
  - Determine where in the code that region begins and ends (call path can be helpful)
  - Then put something like this in selective instrumentation file:

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

  - Recompile and rerun

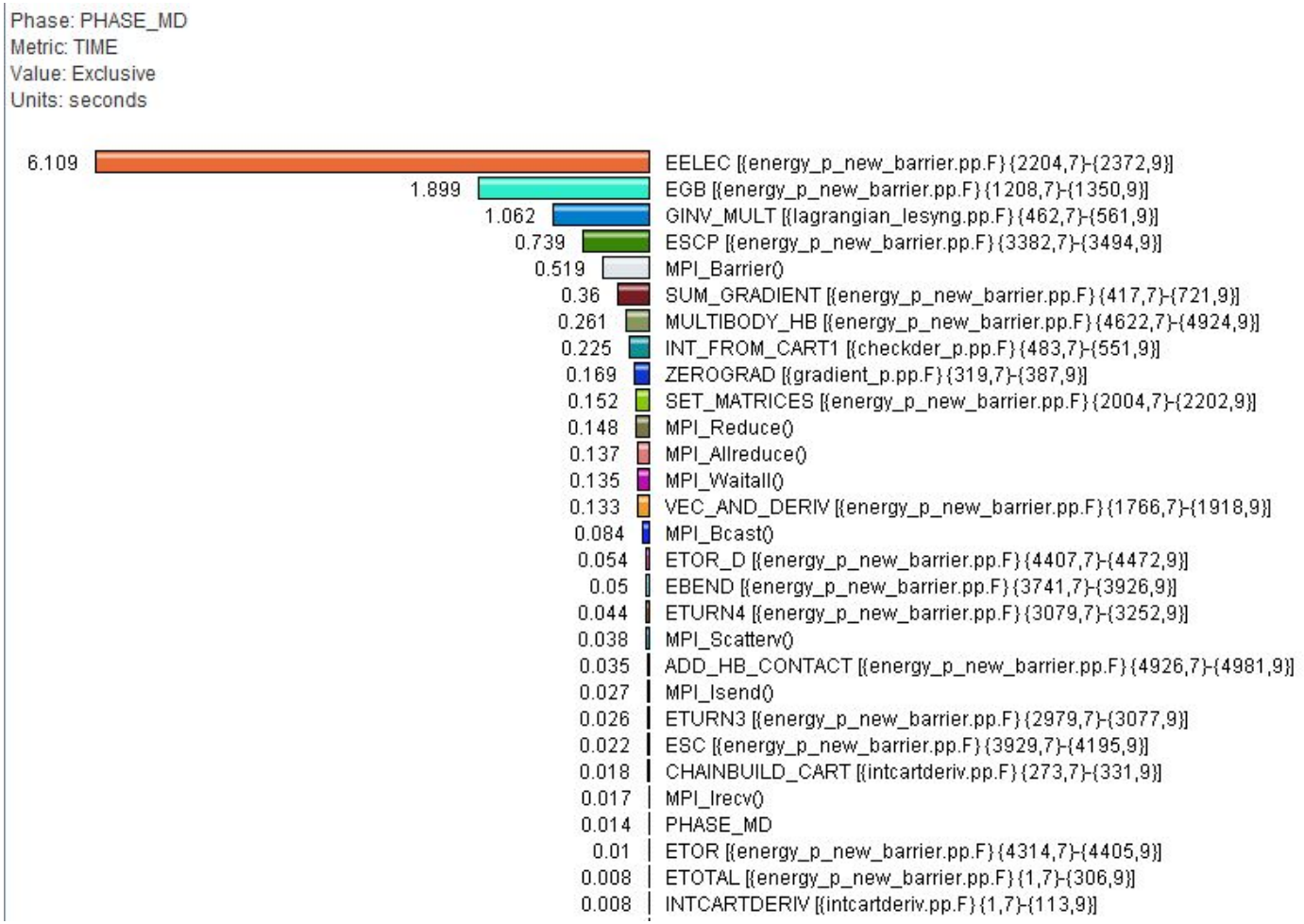
Key UNRES Functions in TAU (with Startup Time)

To get this view, left click on Mean, Max, Min, or Node labels on left hand side of main Paraprof window



PSC
PITTSBURGH SUPERCOMPUTING CENTER

Key UNRES Functions (MD Time Only)



PSC
PITTSBURGH SUPERCOMPUTING CENTER

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

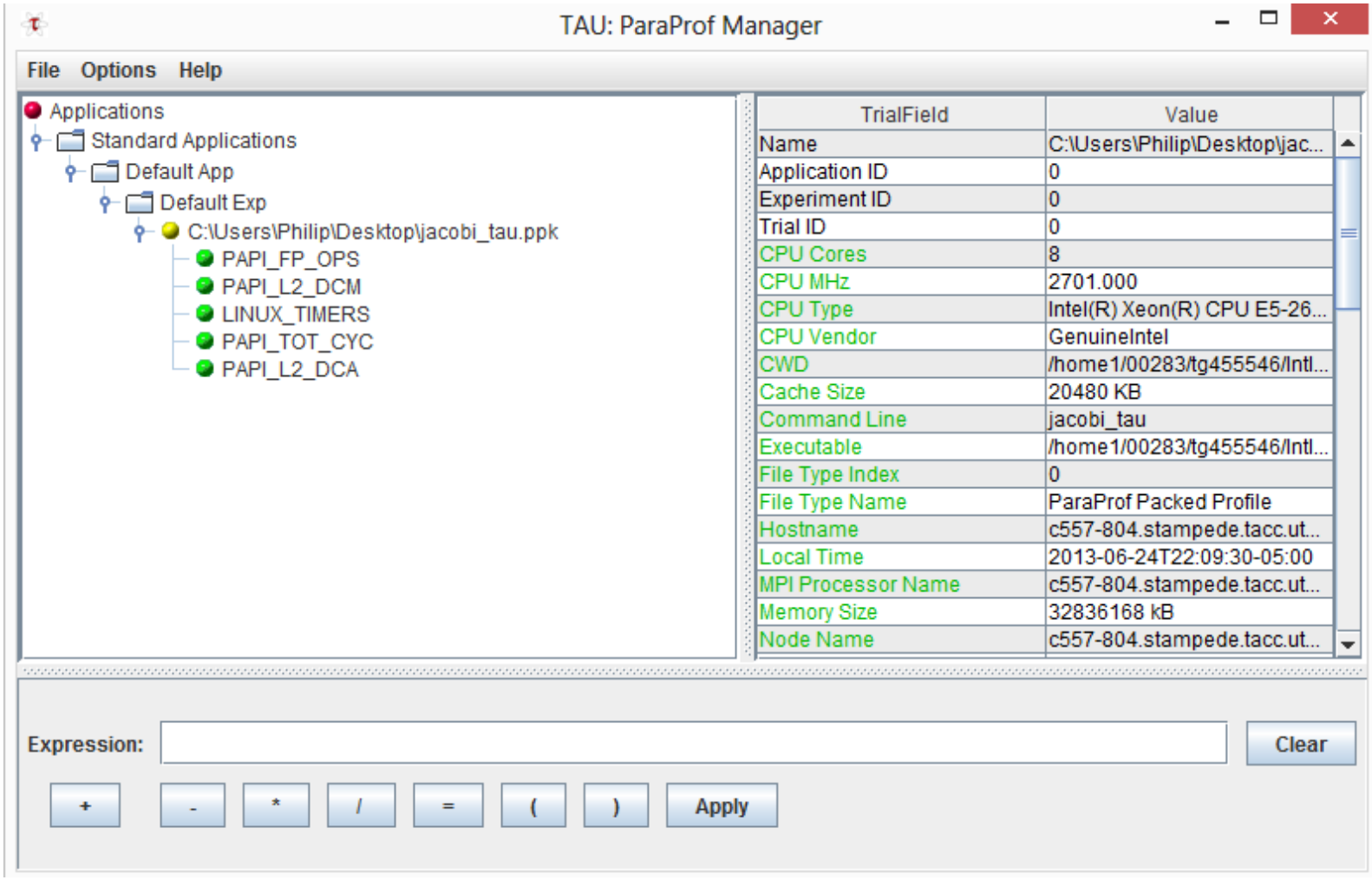
Detecting Serial Performance Issues

- Identify hardware performance counters of interest
  - papi_avail
  - papi_native_avail
  - Run these commands on compute nodes!
- Run TAU (perhaps isolating regions of interest)
- Specify PAPI hardware counters at run time

setenv TAU_METRICS GET_TIME_OF_DAY:PAPI_FP_OPS:PAPI_TOT_CYC

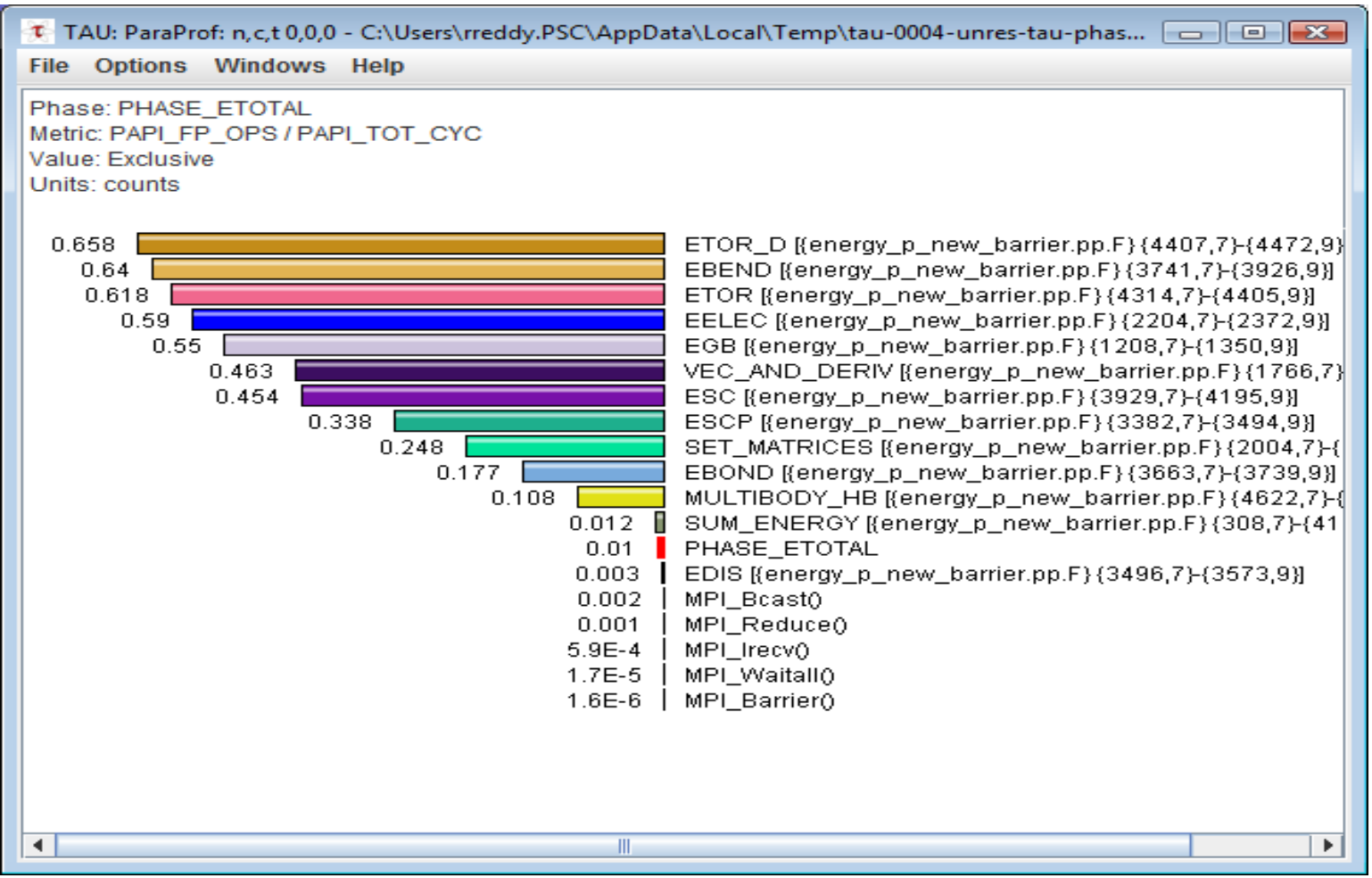
- Be careful! Definition (and accuracy) of PAPI hardware counter presets can vary between architectures

Create a Derived Metric in Paraprof Manager



PSC
PITTSBURGH SUPERCOMPUTING CENTER

Perf of EELEC (peak is 2)



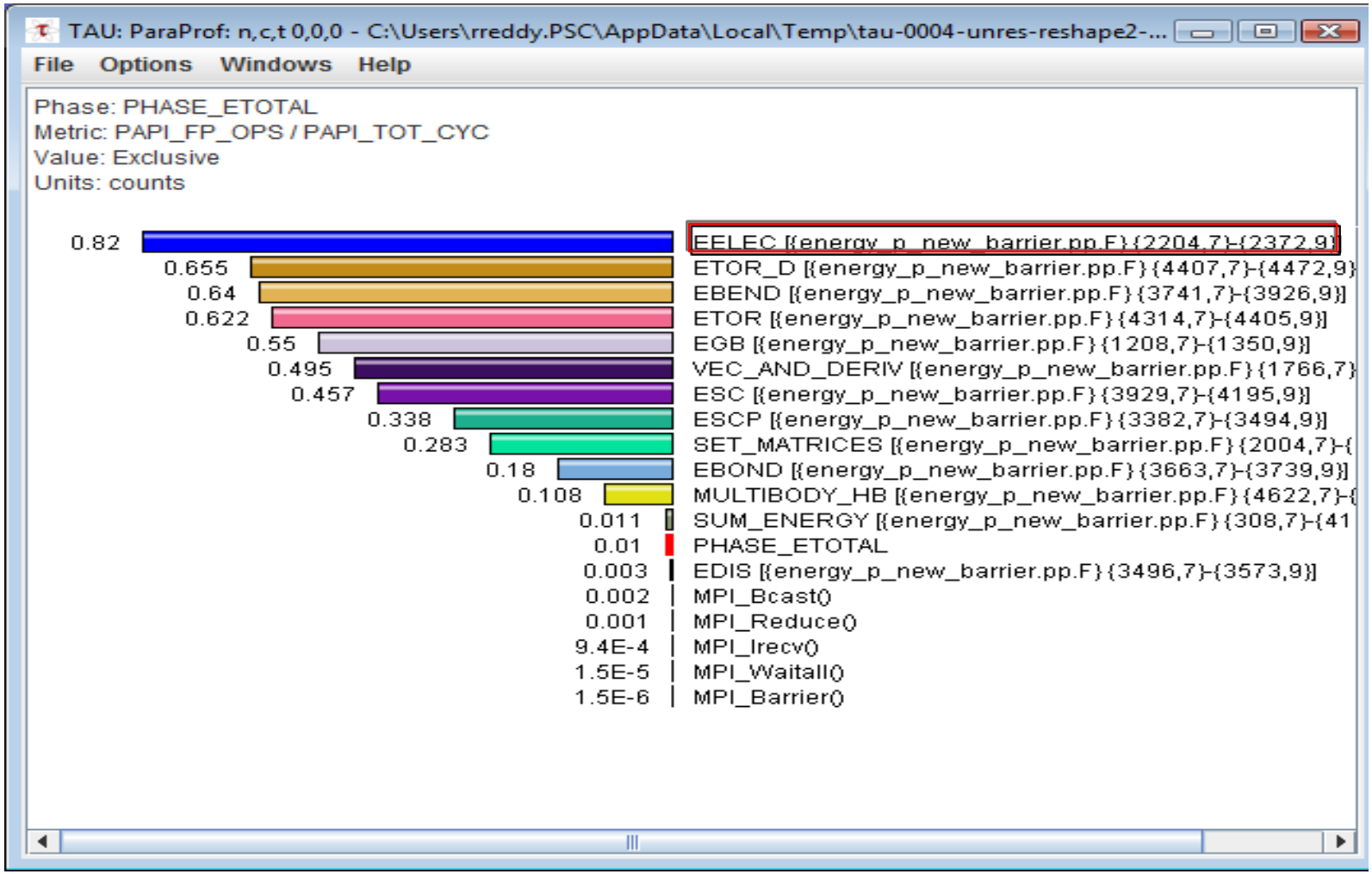
Go to: Paraprof manager Options->"Show derived metrics panel"

PSC PITTSBURGH SUPERCOMPUTING CENTER

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

Do compiler optimization first!
EELEC – After forcing inlining with compiler



PSC
PITTSBURGH SUPERCOMPUTING CENTER

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, Adolfy Hoisie, SIAM, 2001.
  - "Introduction to High Performance Computing for Scientists and Engineers", Georg Hager, Gerhard Wellein, CRC Press, 2010.
  - CI-Tutor course: "Performance Tuning for Clusters" http://ci-tutor.ncsa.illinois.edu/

Performance Engineering: Procedure

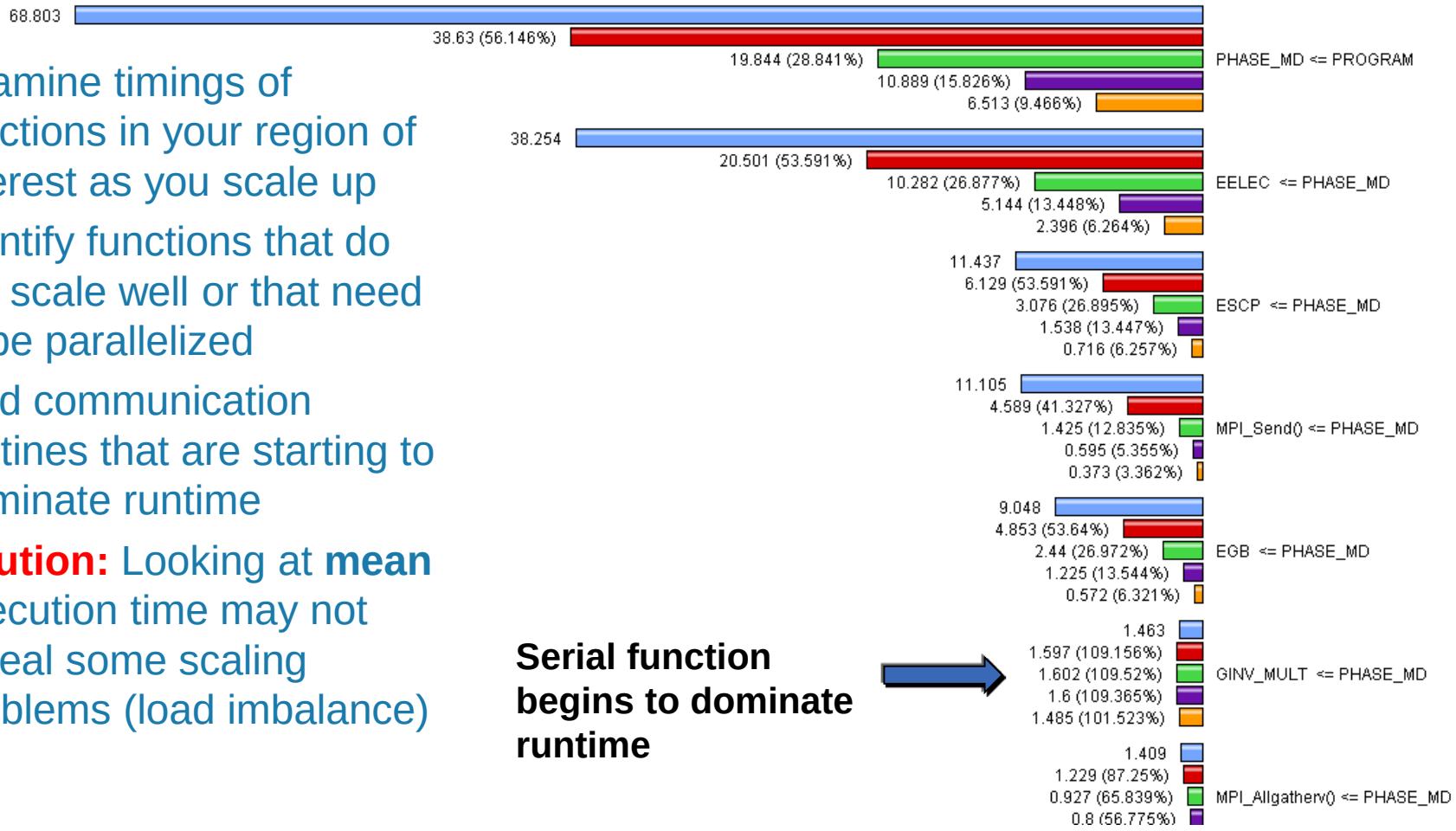
- Serial
  - Assess overall serial performance (percent of peak)
  - Identify functions where code spends most time
  - Instrument those functions
  - Measure code performance using hardware counters
  - Identify inefficient regions of source code and cause of inefficiencies
- Parallel
  - Assess overall 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

TAU Recipe #1: Detecting 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 ‘Apply’
  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

Serial Bottleneck Detection in UNRES: Function Scaling (2-32 cores)

- Examine timings of functions in your region of interest as you scale up
- Identify functions that do not scale well or that need to be parallelized
- Find communication routines that are starting to dominate runtime
- **Caution:** Looking at **mean** execution time may not reveal some scaling problems (load imbalance)

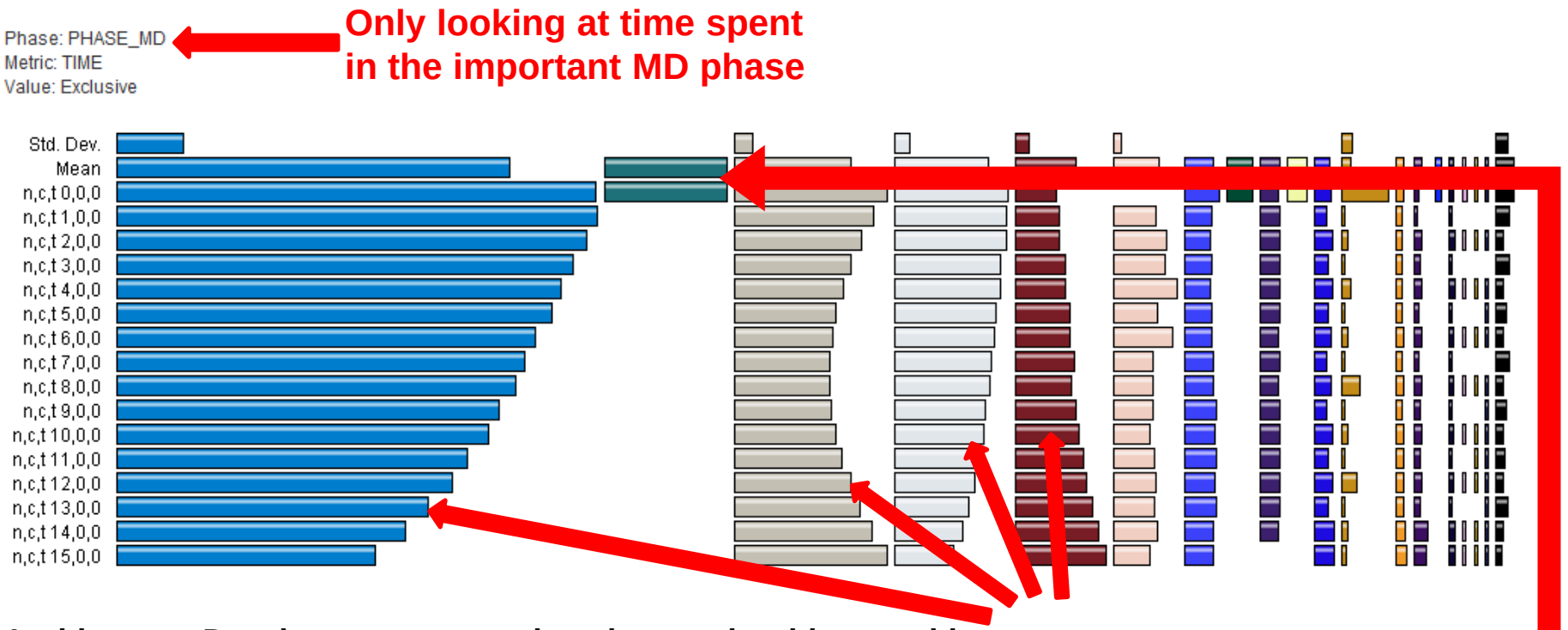


PSC
PITTSBURGH SUPERCOMPUTING CENTER

TAU Recipe #2: Detecting Parallel Load Imbalance

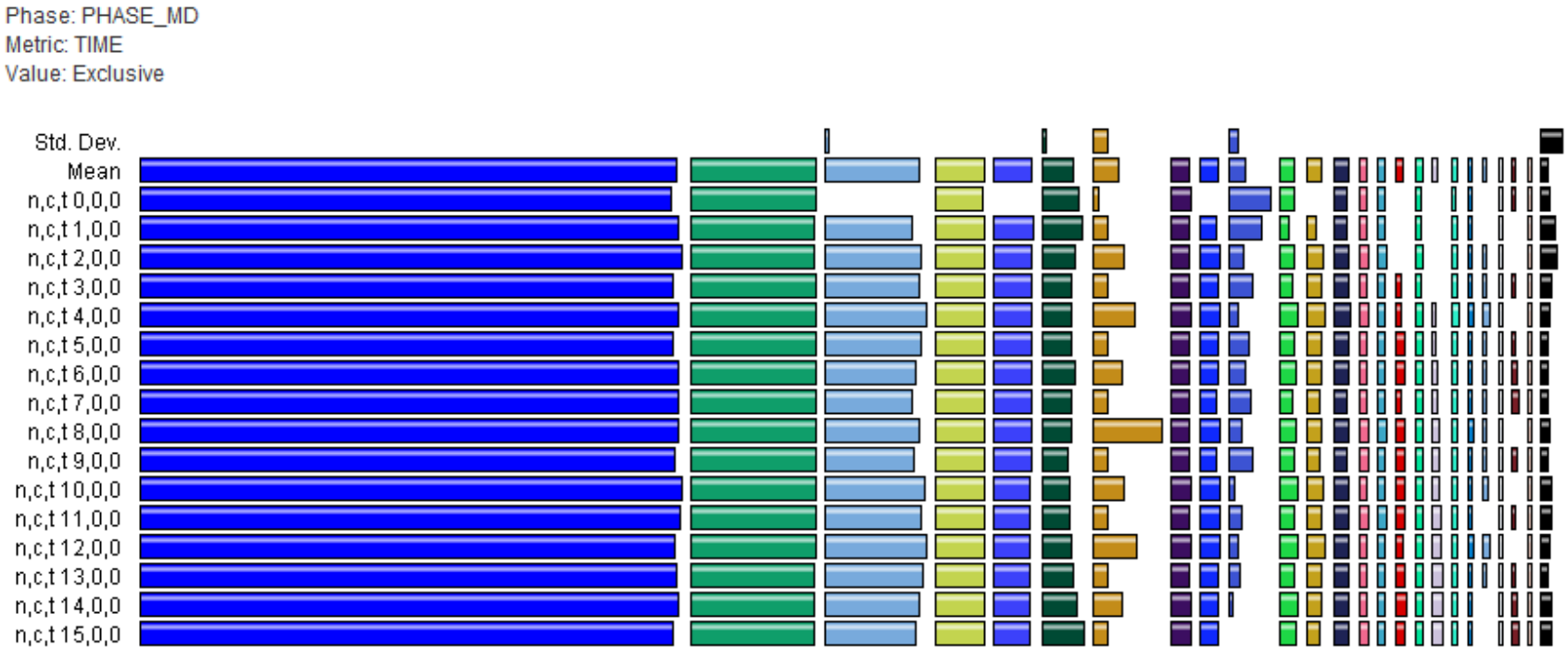
- Examine timings of functions in your region of interest
  - If you defined a phase, from paraprof window, right-click on phase name and select: ‘Show profile for this phase’
- To look at load imbalance in a **particular** function:
  - Left-click on function name to look at timings across all processors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

Load Imbalance Detection in UNRES



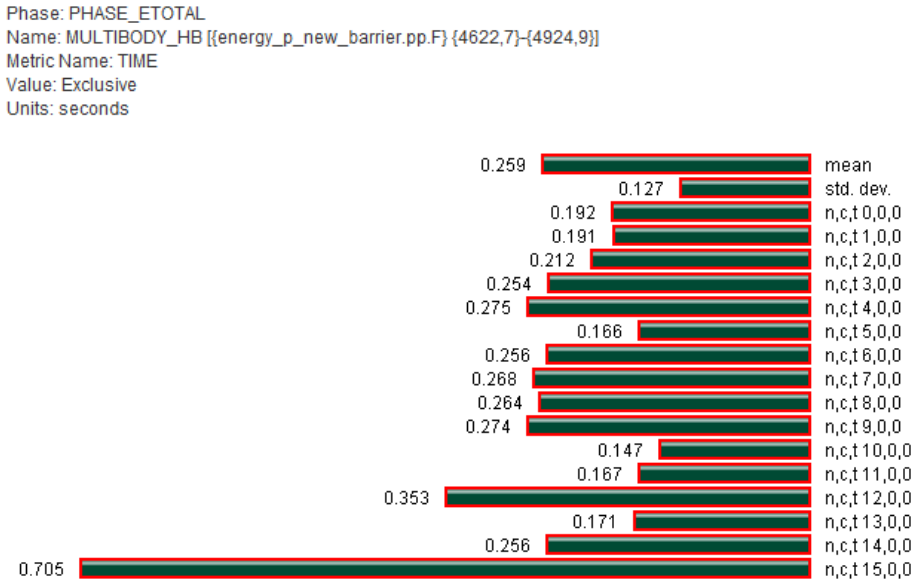
- **In this case: Developers unaware that chosen algorithm would create load imbalance**
- **Reexamined available algorithms and found one with much better load balance – also fewer floating point operations!**
- **Also parallelized serial function causing bottleneck**

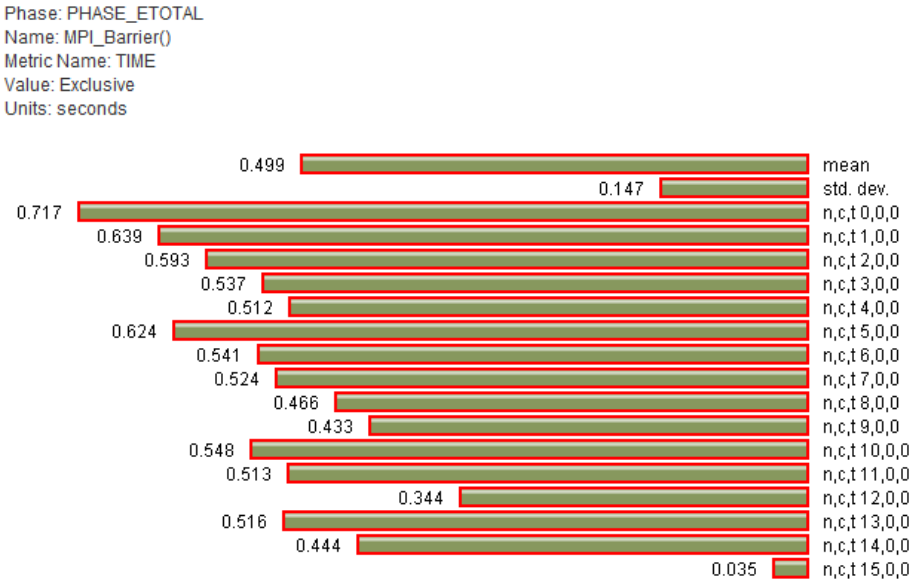
Major Serial Bottleneck and Load Imbalance in UNRES Eliminated



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

Next Iteration of Performance Engineering with Optimized Code





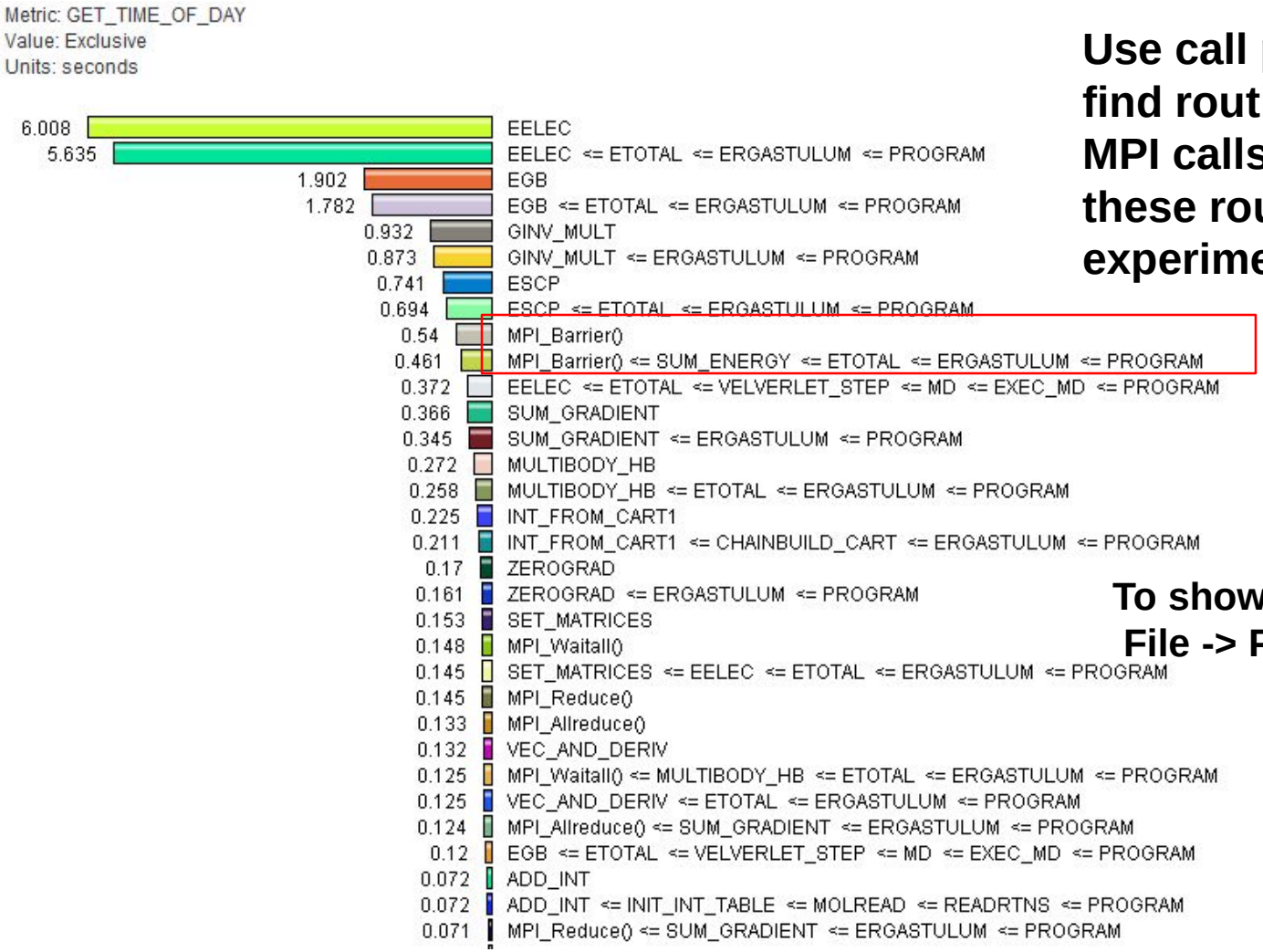
Load imbalance on one processor causing other processors to idle in MPI_Barrier

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

But beware investing too much effort for minimal gain!

PSC
PITTSBURGH SUPERCOMPUTING CENTER

Use Call Path Information: MPI Calls



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

PSC PITTSBURGH SUPERCOMPUTING CENTER

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

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
- XSEDE (and PRACE) staff collaborate with code developers to help with performance engineering of parallel codes (Extended Collaborative Support)

Questions?

blood@psc.edu