Profiling

Shawn T. Brown, PhD

Avid Angry Birds Transformers Fan

Performance Evaluation process

- Monitoring System
  - Observe both overall system performance and single-program execution characteristics.
    - Look to see if the system is doing well and what percentage of the resources your program is using.
    - Pro: easy   Con: not very detailed
- Profiling and Timing the code
  - Timing a whole programs (time command :/usr/bin/time)
  - Timing portions of the program (code modification)
  - Profiling

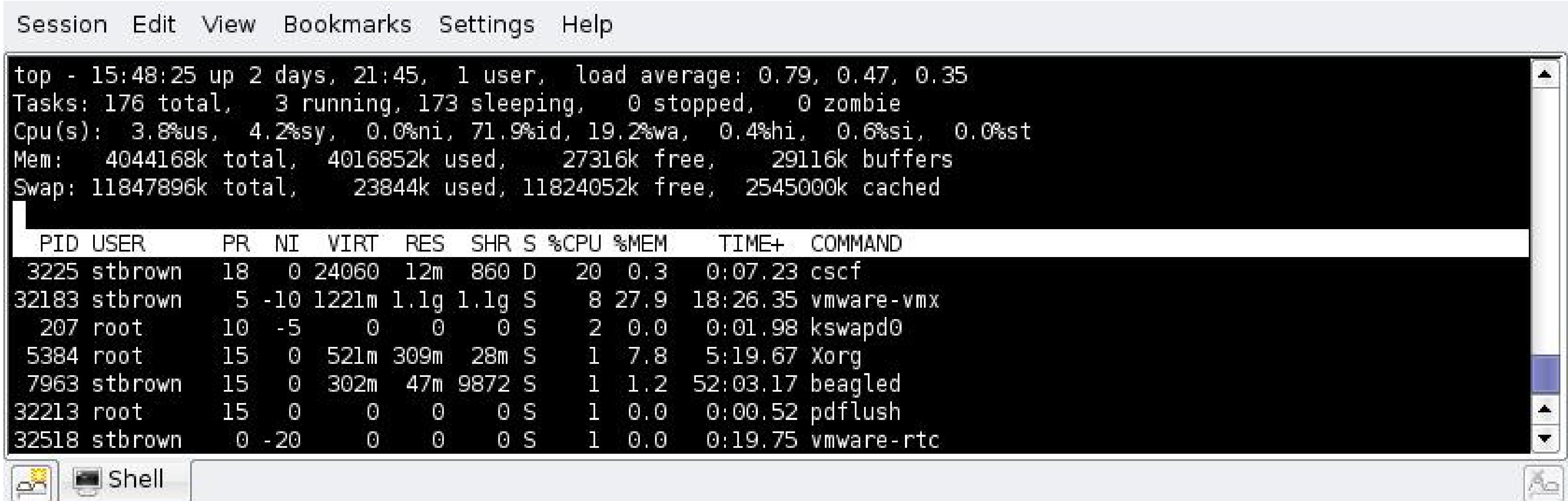
Measurement Techniques

- 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

Useful Monitoring Commands (Linux)

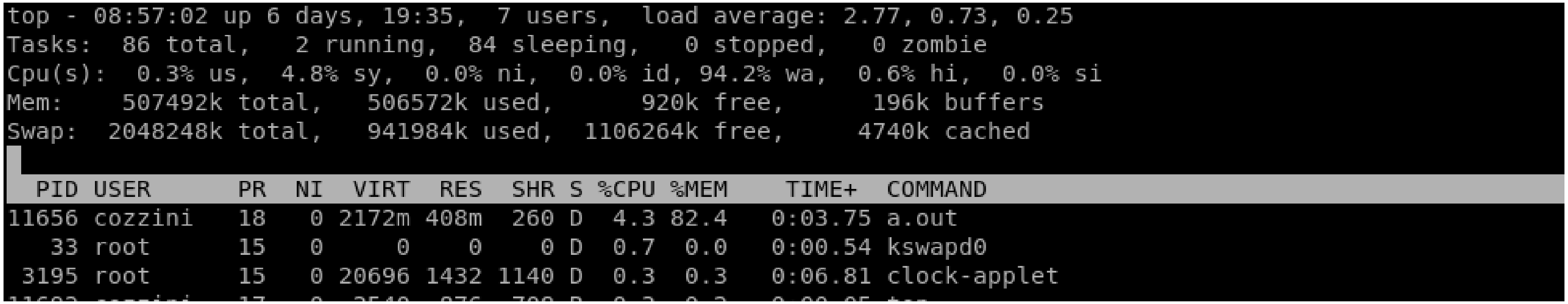
Uptime	returns information about system usage and user load
ps(1)	lets you see a “ snapshot” of the process table
top	process table dynamic display
free	memory usage
vmstat	memory usage monitor



```
top - 15:48:25 up 2 days, 21:45,  1 user,  load average: 0.79, 0.47, 0.35
Tasks: 176 total,   3 running, 173 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.8%us,  4.2%sy,  0.0%ni, 71.9%id, 19.2%wa,  0.4%hi,  0.6%si,  0.0%st
Mem:   4044168k total,  4016852k used,    27316k free,    29116k buffers
Swap: 11847896k total,    23844k used, 11824052k free,  2545000k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3225 stbrown   18   0 24060  12m  860 D   20  0.3   0:07.23 cscf
32183 stbrown    5 -10 1221m 1.1g 1.1g S    8 27.9  18:26.35 vmware-vmx
  207 root      10  -5     0    0    0 S    2  0.0   0:01.98 kswapd0
 5384 root      15   0  521m 309m  28m S    1  7.8   5:19.67 Xorg
 7963 stbrown   15   0  302m  47m 9872 S    1  1.2  52:03.17 beagled
32213 root      15   0     0    0    0 S    1  0.0   0:00.52 pdflush
32518 stbrown    0 -20     0    0    0 S    1  0.0   0:19.75 vmware-rtc
```

Swapping… A top disaster

- virtual or swap memory:
  - This memory, is actually space on the hard drive. The operatingsystem reserves a space on the hard drive for “ swap space” .
- time to access virtual memory VERY large:
- this time is done by the system not by your program !



```
top - 08:57:02 up 6 days, 19:35,  7 users,  load average: 2.77, 0.73, 0.25
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3% us,  4.8% sy,  0.0% ni,  0.0% id, 94.2% wa,  0.6% hi,  0.0% si
Mem:    507492k total,   506572k used,      920k free,      196k buffers
Swap:  2048248k total,   941984k used,  1106264k free,     4740k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11656 cozzini   18   0 2172m 408m  260 D  4.3 82.4   0:03.75 a.out
   33 root      15   0     0    0    0 D  0.7  0.0   0:00.54 kswapd0
 3195 root      15   0 20696 1432 1140 D  0.3  0.3   0:06.81 clock-applet
```

Monitoring your own code (time)

```
NAME
        time - time a simple command or give resource usage

SYNOPSIS
        time [options] command [arguments...]

DESCRIPTION
        The time command runs the specified program command with
        the given arguments. When command finishes, time writes a
        message to standard output giving timing statistics about
        this program ..
```

```
--------------->time ./a.out
[program output]
real 0m1.361s
user 0m0.770s
sys 0m0.590s
```

user time:	Cputime dedicated to your program
sys time:	time used by your program to execute system calls
real time:	total time (aka walltime)

Timing A Portion of the Code

- Most programming languages provide a means to access the systems own timing functions

- C function: clock

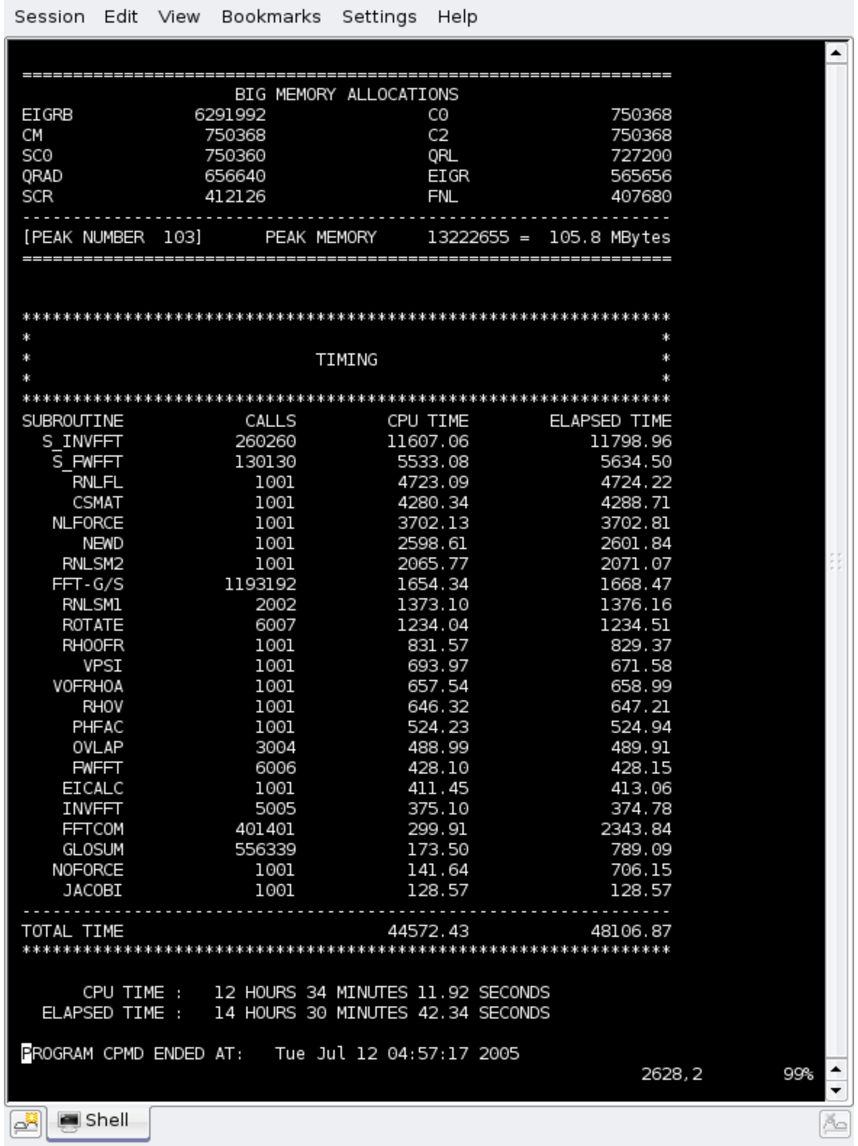
```
clock_t c0, c1;
c0 = clock();
          section to code..
c1= clock();
cputime = (c1 - c0)/(CLOCKS_PER_SEC );
```

- Fortran Subroutine: cpu_time

```
call cpu_time(t0)
          section to code..
call cpu_time(t1)
cputime = (t1 - t0)
```

It is good practice....

- Good application writers will take full advantage of these to give users insight into code performance.



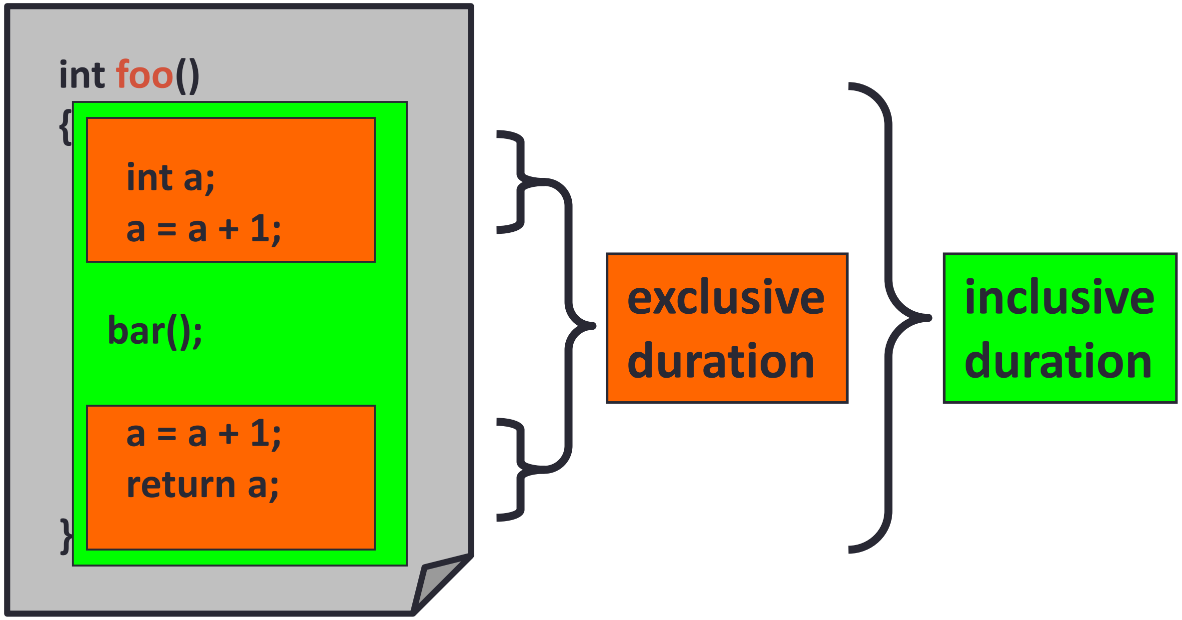
```
================================================================
                     BIG MEMORY ALLOCATIONS
EIGRB            6291992                C0                750368
CM                750368                C2                750368
SC0               750360                QRL               727200
QRAD              656640                EIGR              565656
SCR               412126                FNL               407680
----------------------------------------------------------------
[PEAK NUMBER  103]      PEAK MEMORY     13222655 =  105.8 MBytes
================================================================


****************************************************************
*                                                              *
*                            TIMING                            *
*                                                              *
****************************************************************
SUBROUTINE            CALLS         CPU TIME        ELAPSED TIME
  S_INVFFT           260260         11607.06            11798.96
   S_FWFFT           130130          5533.08             5634.50
     RNLFL             1001          4723.09             4724.22
     CSMAT             1001          4280.34             4288.71
   NLFORCE             1001          3702.13             3702.81
      NEWD             1001          2598.61             2601.84
    RNLSM2             1001          2065.77             2071.07
   FFT-G/S          1193192          1654.34             1668.47
    RNLSM1             2002          1373.10             1376.16
    ROTATE             6007          1234.04             1234.51
    RHOOFR             1001           831.57              829.37
      VPSI             1001           693.97              671.58
   VOFRHOA             1001           657.54              658.99
      RHOV             1001           646.32              647.21
     PHFAC             1001           524.23              524.94
     OVLAP             3004           488.99              489.91
     FWFFT             6006           428.10              428.15
    EICALC             1001           411.45              413.06
    INVFFT             5005           375.10              374.78
    FFTCOM           401401           299.91             2343.84
    GLOSUM           556339           173.50              789.09
   NOFORCE             1001           141.64              706.15
    JACOBI             1001           128.57              128.57
----------------------------------------------------------------
TOTAL TIME                          44572.43            48106.87
****************************************************************

      CPU TIME :   12 HOURS 34 MINUTES 11.92 SECONDS
  ELAPSED TIME :   14 HOURS 30 MINUTES 42.34 SECONDS

PROGRAM CPMD ENDED AT:   Tue Jul 12 04:57:17 2005
                                                             2628,2        99%
```

Profiling

- Profiling is an approach to performance analysis in which the amount of time spent in sections of code is measured (using either a sampling technique or on entry/exit of a code block) and presented as a histogram.
- Allows a developer to target key time consuming portions of codes.
- Profiling can be done at varied levels of granularity
    - Subroutine, code block, loop and source code line

Inclusive and Exclusive Profiles

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



```
int foo()
{
  int a;
  a = a + 1;

  bar();

  a = a + 1;
  return a;
}
```

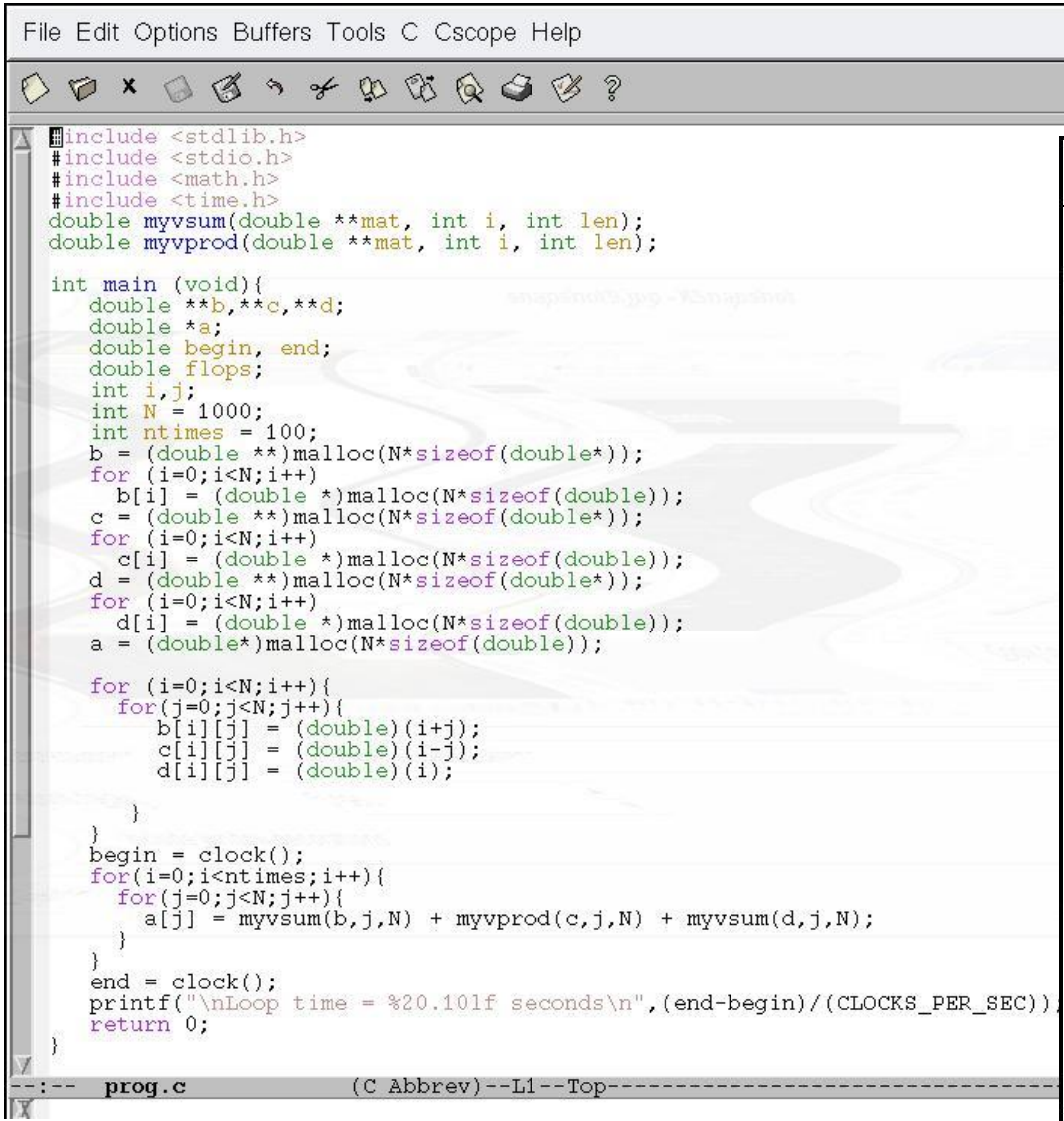
exclusive duration

inclusive duration

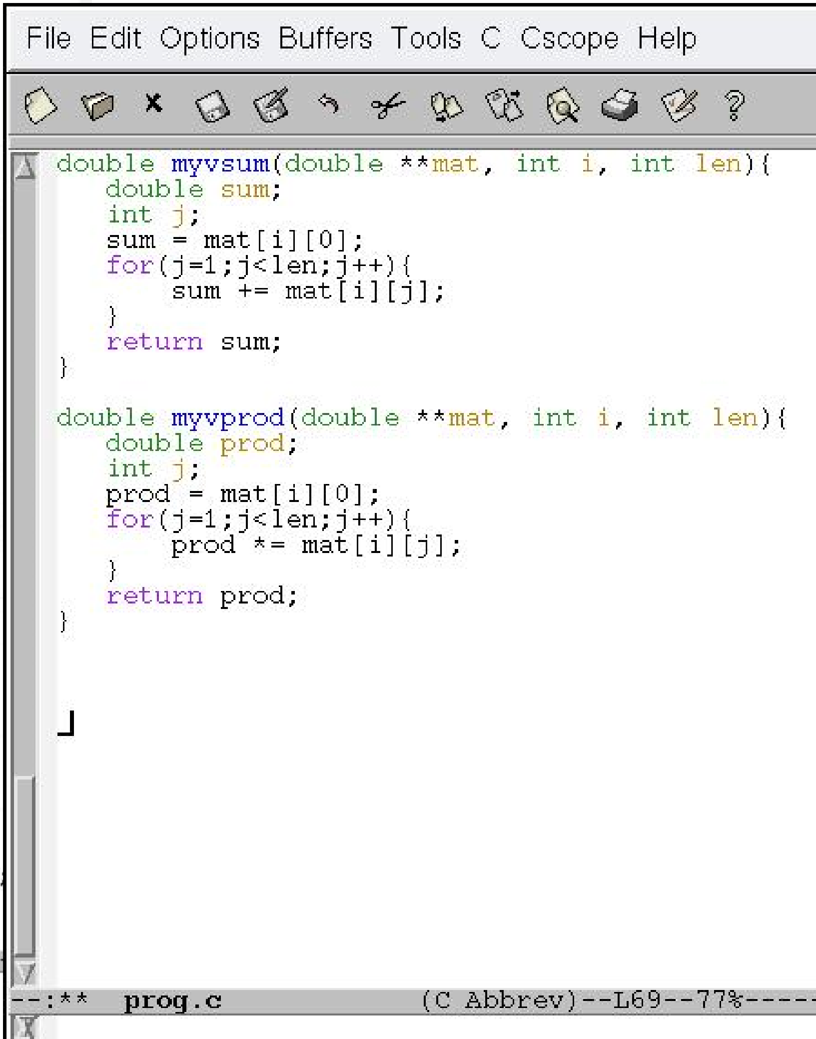
GCC profiling and gprof

- Simple gcc compiler flags can be used to get profiling information.
  - Great place to start
- GNU:
  - -p Generate extra code to write profile information suitable for analysis program prof
  - -pg Generate extra code to write profile information suitable for analysis by program gprof.
- Procedure
  - `gcc -pg prog.c -o prog`
  - `./prog`
  - `gprof prog.c gmon.out`

Example

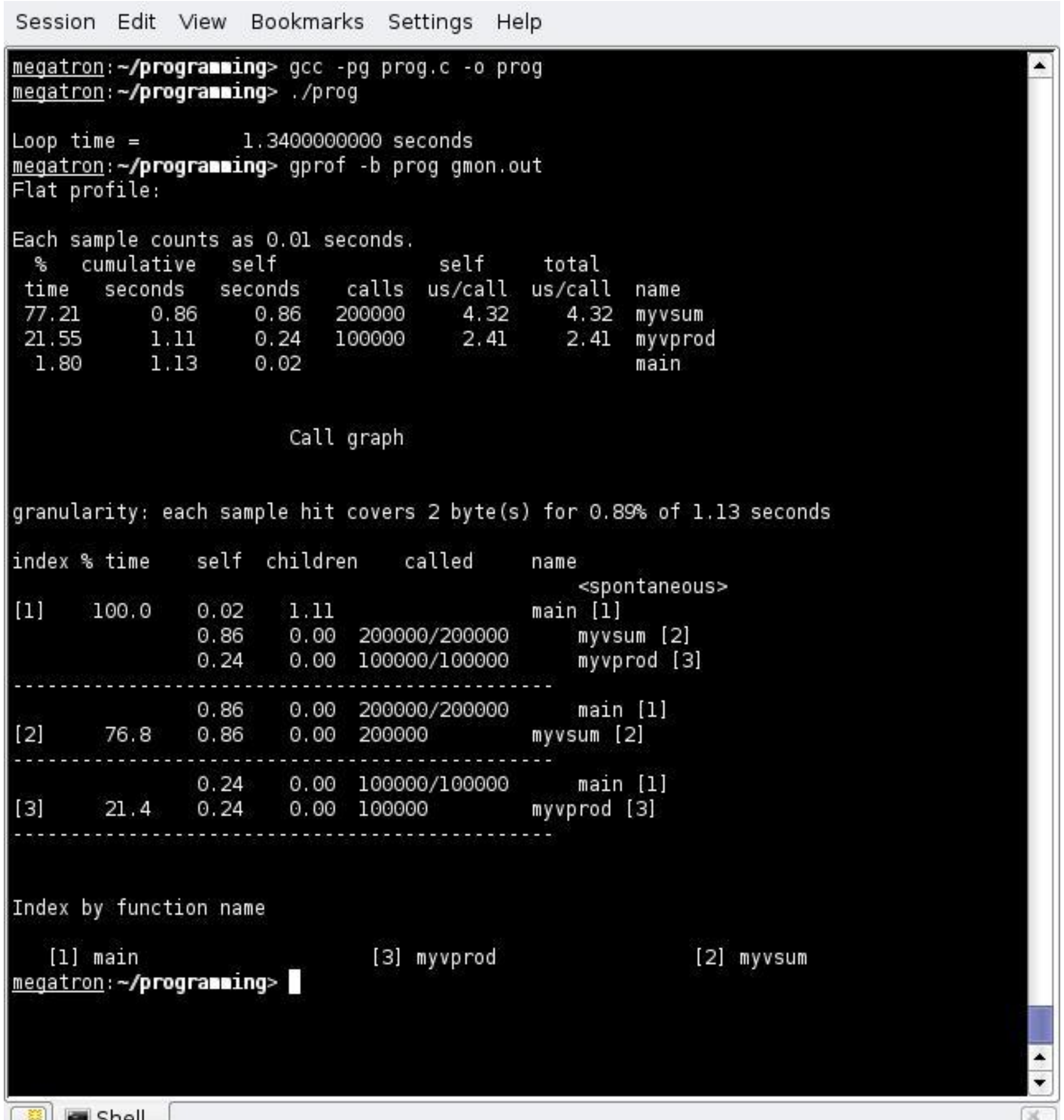


```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
double myvsum(double **mat, int i, int len);
double myvprod(double **mat, int i, int len);

int main (void){
   double **b,**c,**d;
   double *a;
   double begin, end;
   double flops;
   int i,j;
   int N = 1000;
   int ntimes = 100;
   b = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     b[i] = (double *)malloc(N*sizeof(double));
   c = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     c[i] = (double *)malloc(N*sizeof(double));
   d = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     d[i] = (double *)malloc(N*sizeof(double));
   a = (double*)malloc(N*sizeof(double));

   for (i=0;i<N;i++){
     for(j=0;j<N;j++){
        b[i][j] = (double)(i+j);
        c[i][j] = (double)(i-j);
        d[i][j] = (double)(i);

      }
   }
   begin = clock();
   for(i=0;i<ntimes;i++){
     for(j=0;j<N;j++){
       a[j] = myvsum(b,j,N) + myvprod(c,j,N) + myvsum(d,j,N);
     }
   }
   end = clock();
   printf("\nLoop time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_SEC));
   return 0;
}
```



```
double myvsum(double **mat, int i, int len){
   double sum;
   int j;
   sum = mat[i][0];
   for(j=1;j<len;j++){
       sum += mat[i][j];
   }
   return sum;
}

double myvprod(double **mat, int i, int len){
   double prod;
   int j;
   prod = mat[i][0];
   for(j=1;j<len;j++){
       prod *= mat[i][j];
   }
   return prod;
}
```

Example



```
megatron:~/programming> gcc -pg prog.c -o prog
megatron:~/programming> ./prog

Loop time =         1.3400000000 seconds
megatron:~/programming> gprof -b prog gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 77.21      0.86     0.86   200000     4.32     4.32  myvsum
 21.55      1.11     0.24   100000     2.41     2.41  myvprod
  1.80      1.13     0.02                             main


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.89% of 1.13 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02    1.11                 main [1]
                0.86    0.00  200000/200000      myvsum [2]
                0.24    0.00  100000/100000      myvprod [3]
-----------------------------------------------
                0.86    0.00  200000/200000      main [1]
[2]     76.8    0.86    0.00  200000         myvsum [2]
-----------------------------------------------
                0.24    0.00  100000/100000      main [1]
[3]     21.4    0.24    0.00  100000         myvprod [3]
-----------------------------------------------


Index by function name

   [1] main                    [3] myvprod                 [2] myvsum
megatron:~/programming> █
```

Hardware Counters

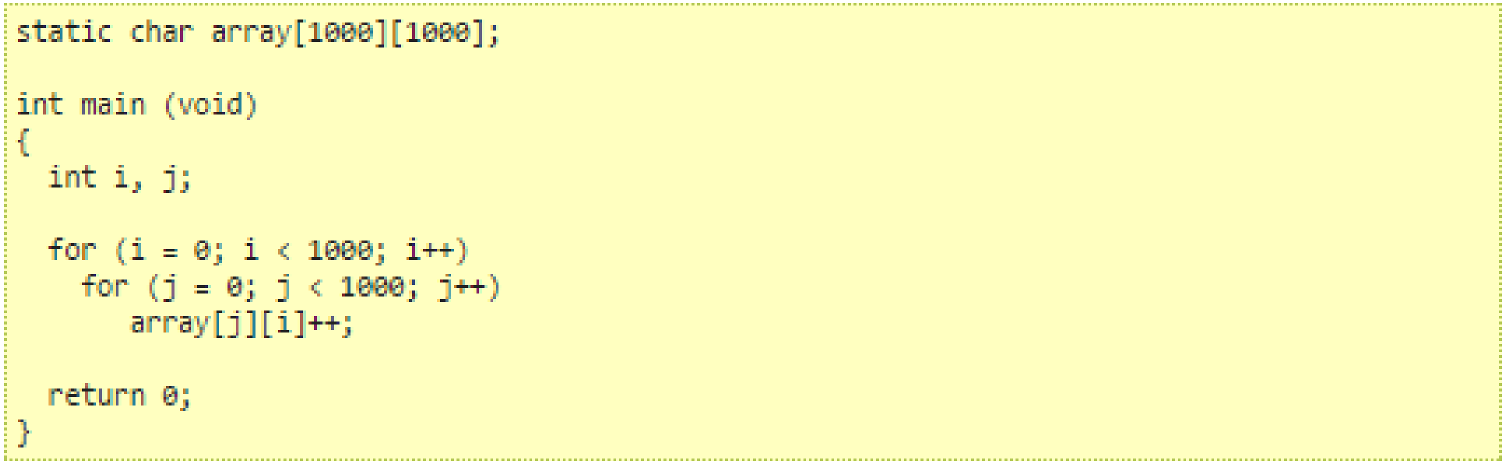
- Counters: set of registers that count processor events, like floating point operations, or cycles (Core i7 (Nahalem) has 7 counters per core, 3 fixed and 4 that can be assigned).

- **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

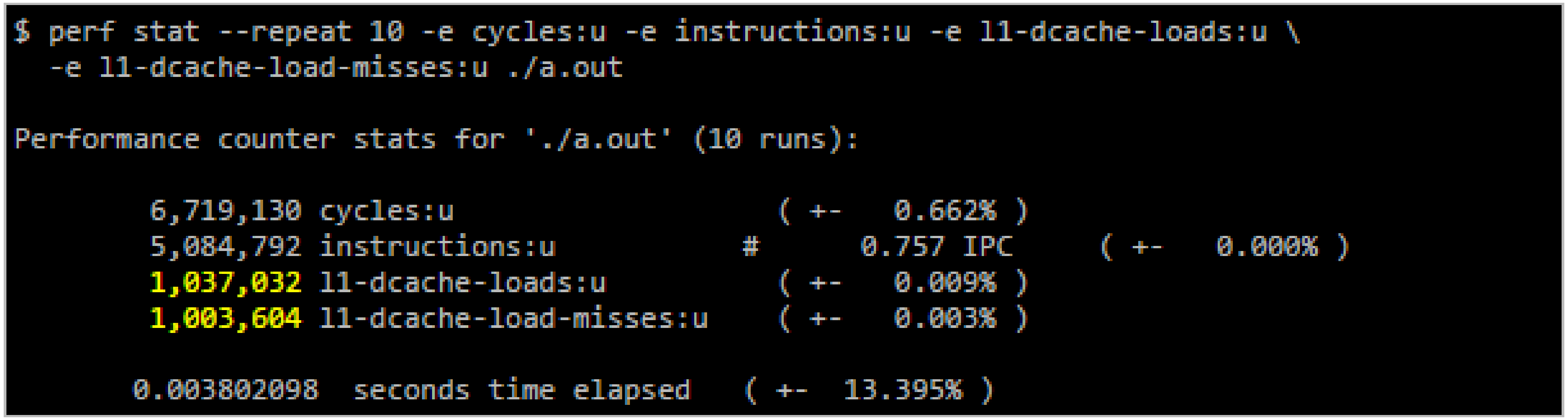
- Portable: uses same routines to access counters across all architectures
- High-level interface
  - Using predefined standard events the same source code can access similar counters across various architectures without modification.
  - papi_avail
- Low-level interface
  - Provides access to all machine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

Perf, a simple tool for accessing hardware counters



```
static char array[1000][1000];

int main (void)
{
  int i, j;

  for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
       array[j][i]++;

  return 0;
}
```

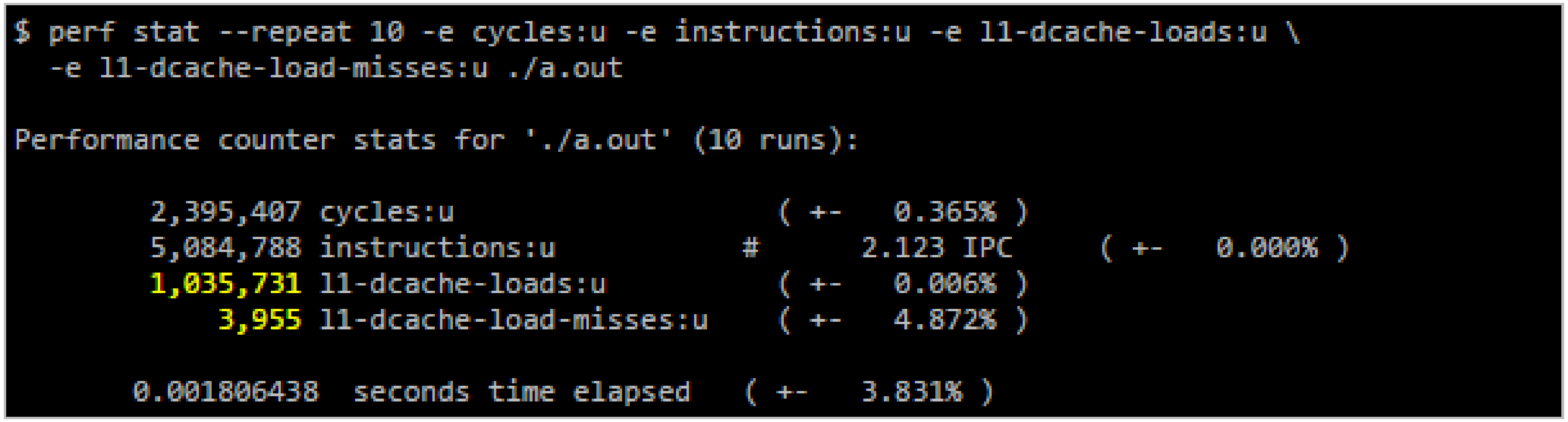
On hardware that supports enumerating cache hits and misses, you can run:



```
$ perf stat --repeat 10 -e cycles:u -e instructions:u -e l1-dcache-loads:u \
  -e l1-dcache-load-misses:u ./a.out

Performance counter stats for './a.out' (10 runs):

        6,719,130 cycles:u                    ( +-   0.662% )
        5,084,792 instructions:u            #      0.757 IPC     ( +-   0.000% )
        1,037,032 l1-dcache-loads:u           ( +-   0.009% )
        1,003,604 l1-dcache-load-misses:u     ( +-   0.003% )

       0.003802098  seconds time elapsed   ( +-  13.395% )
```

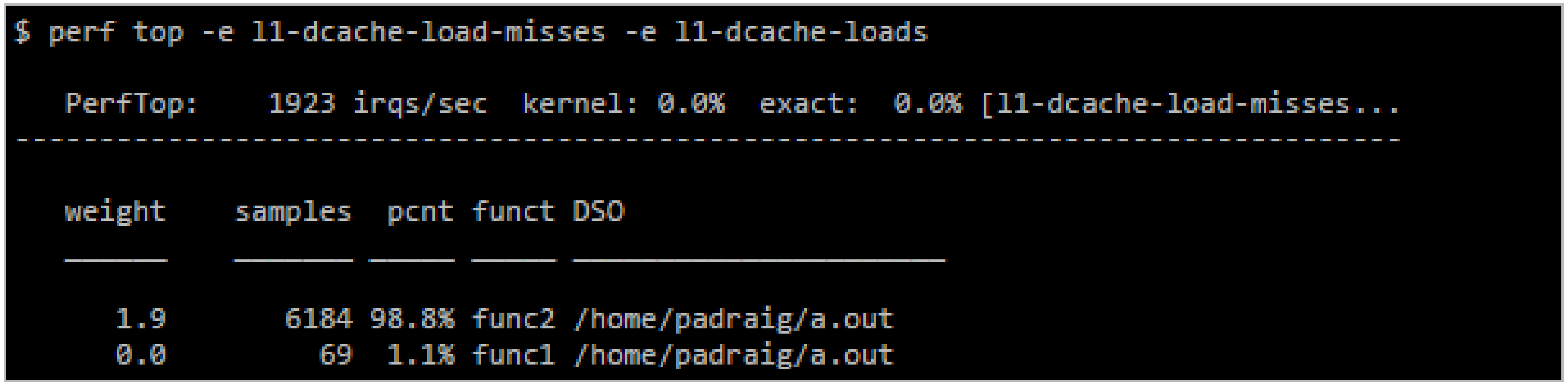
Perf, a simple tool for accessing hardware counters

Note the large ratio of cache misses.
Now if we change `array[j][i]++;` to `array[i][j]++;` and re-run perf-stat:



```
$ perf stat --repeat 10 -e cycles:u -e instructions:u -e l1-dcache-loads:u \
  -e l1-dcache-load-misses:u ./a.out

Performance counter stats for './a.out' (10 runs):

        2,395,407 cycles:u                  ( +-   0.365% )
        5,084,788 instructions:u          #      2.123 IPC     ( +-   0.000% )
        1,035,731 l1-dcache-loads:u         ( +-   0.006% )
            3,955 l1-dcache-load-misses:u   ( +-   4.872% )

       0.001806438  seconds time elapsed   ( +-   3.831% )
```

We can see the L1 cache is much more effective.
To identify hot spots to concentrate on you can use:

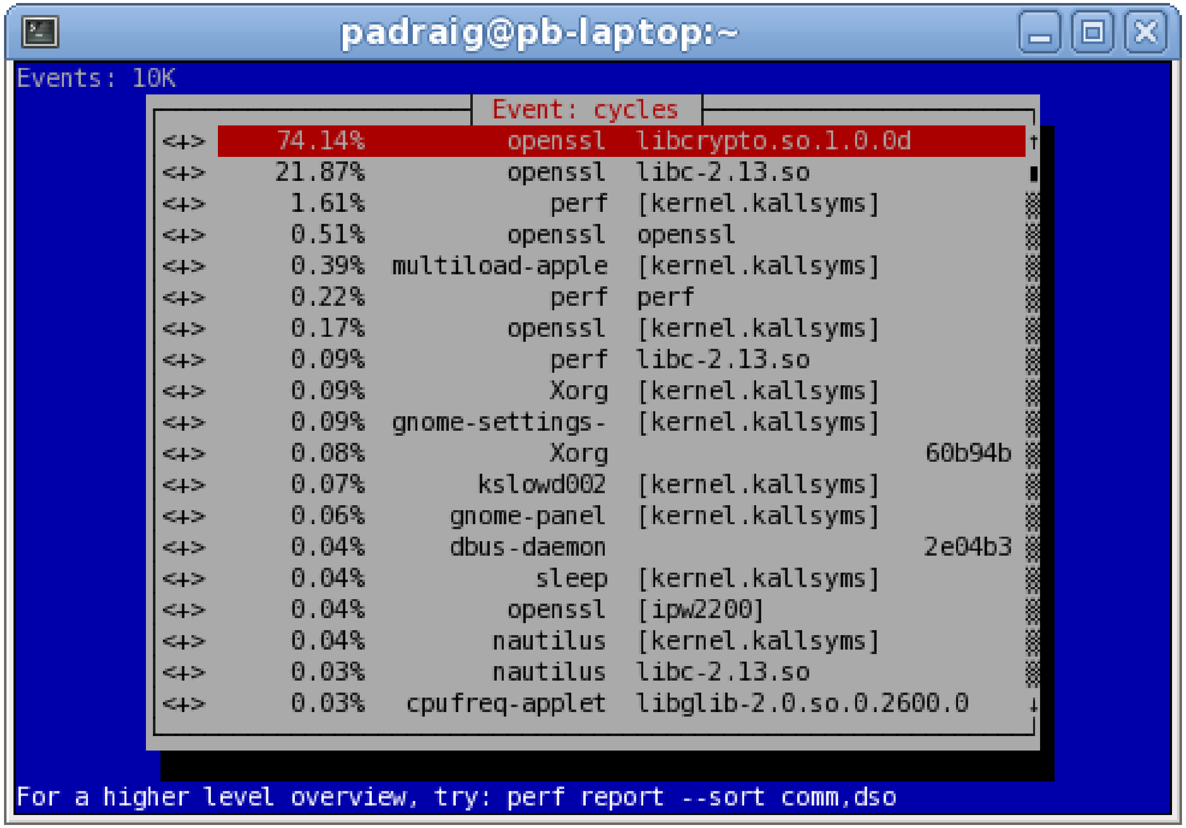


```
$ perf top -e l1-dcache-load-misses -e l1-dcache-loads

   PerfTop:    1923 irqs/sec  kernel: 0.0%  exact:  0.0% [l1-dcache-load-misses...
-----------------------------------------------------------------------------------

   weight    samples  pcnt funct DSO
   ______    _______ _____ _____ ______________________

      1.9       6184 98.8% func2 /home/padraig/a.out
      0.0         69  1.1% func1 /home/padraig/a.out
```

Perf, a simple tool for accessing hardware counters

```
$ perf record -a -g sleep 10  # record system for 10s
$ perf report --sort comm,dso # display report
```

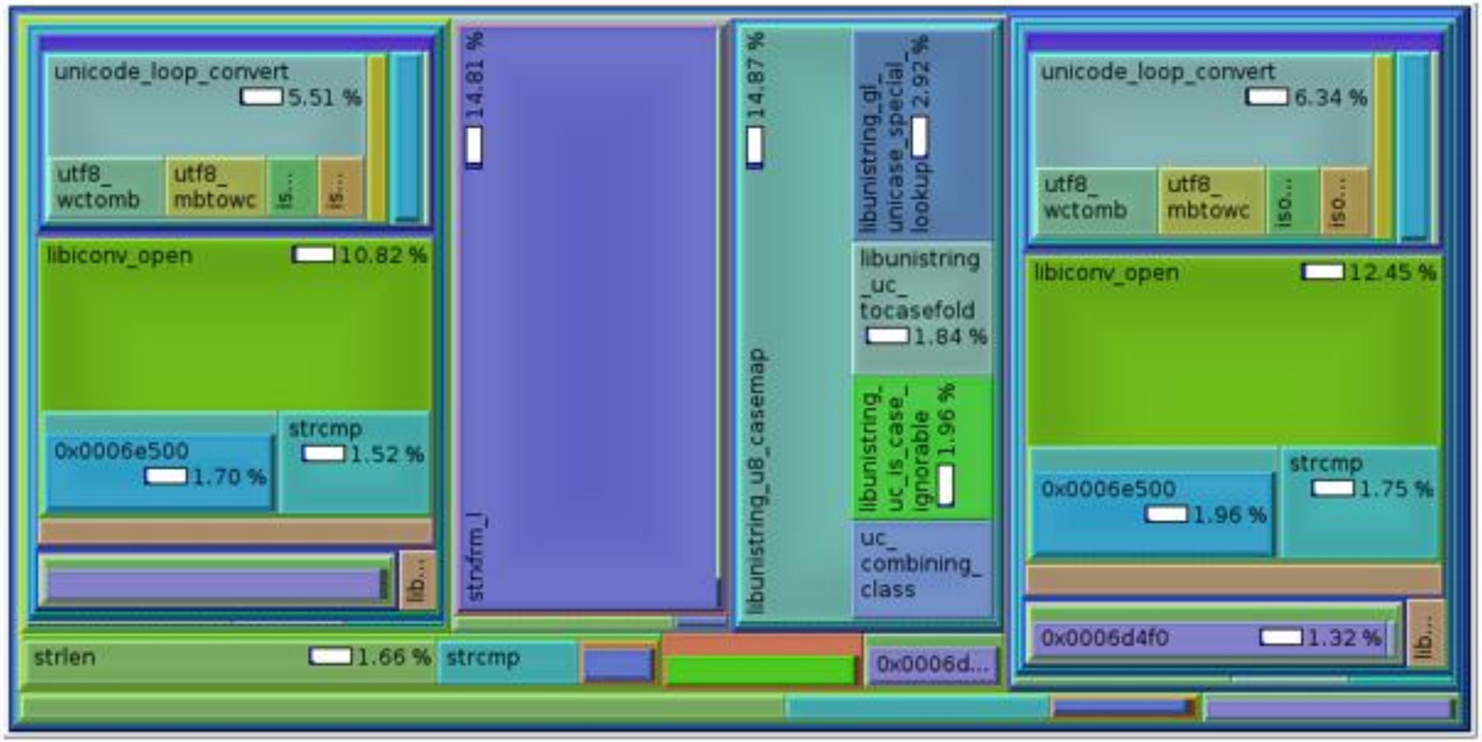
That will display this handy curses interface on basically any hardware platform, which you can use to drill down to the area of interest.



```
Events: 10K
                  Event: cycles
<+>   74.14%          openssl  libcrypto.so.1.0.0d
<+>   21.87%          openssl  libc-2.13.so
<+>    1.61%             perf  [kernel.kallsyms]
<+>    0.51%          openssl  openssl
<+>    0.39%   multiload-apple [kernel.kallsyms]
<+>    0.22%             perf  perf
<+>    0.17%          openssl  [kernel.kallsyms]
<+>    0.09%             perf  libc-2.13.so
<+>    0.09%             Xorg  [kernel.kallsyms]
<+>    0.09%   gnome-settings- [kernel.kallsyms]
<+>    0.08%             Xorg                     60b94b
<+>    0.07%        kslowd002  [kernel.kallsyms]
<+>    0.06%      gnome-panel  [kernel.kallsyms]
<+>    0.04%      dbus-daemon                     2e04b3
<+>    0.04%            sleep  [kernel.kallsyms]
<+>    0.04%          openssl  [ipw2200]
<+>    0.04%         nautilus  [kernel.kallsyms]
<+>    0.03%         nautilus  libc-2.13.so
<+>    0.03%   cpufreq-applet  libglib-2.0.so.0.2600.0

For a higher level overview, try: perf report --sort comm,dso
```

Using Valgrind for profiling

```
valgrind --tool=callgrind ./a.out
kcachegrind callgrind.out.*
```



Note kcachegrind is part of the "kdesdk" package on my fedora system, and can be used to read oprofile data (mentioned above) too.

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

TAU Parallel Profiling Tool

Performance Evaluation process

- Monitoring System
  - Observe both overall system performance and single-program execution characteristics.
    - Look to see if the system is doing well and what percentage of the resources your program is using.
    - Pro: easy Con: not very detailed
- Profiling and Timing the code
  - Timing a whole programs (time command :/usr/bin/time)
  - Timing portions of the program (code modification)
  - Profiling

Parallel Performance

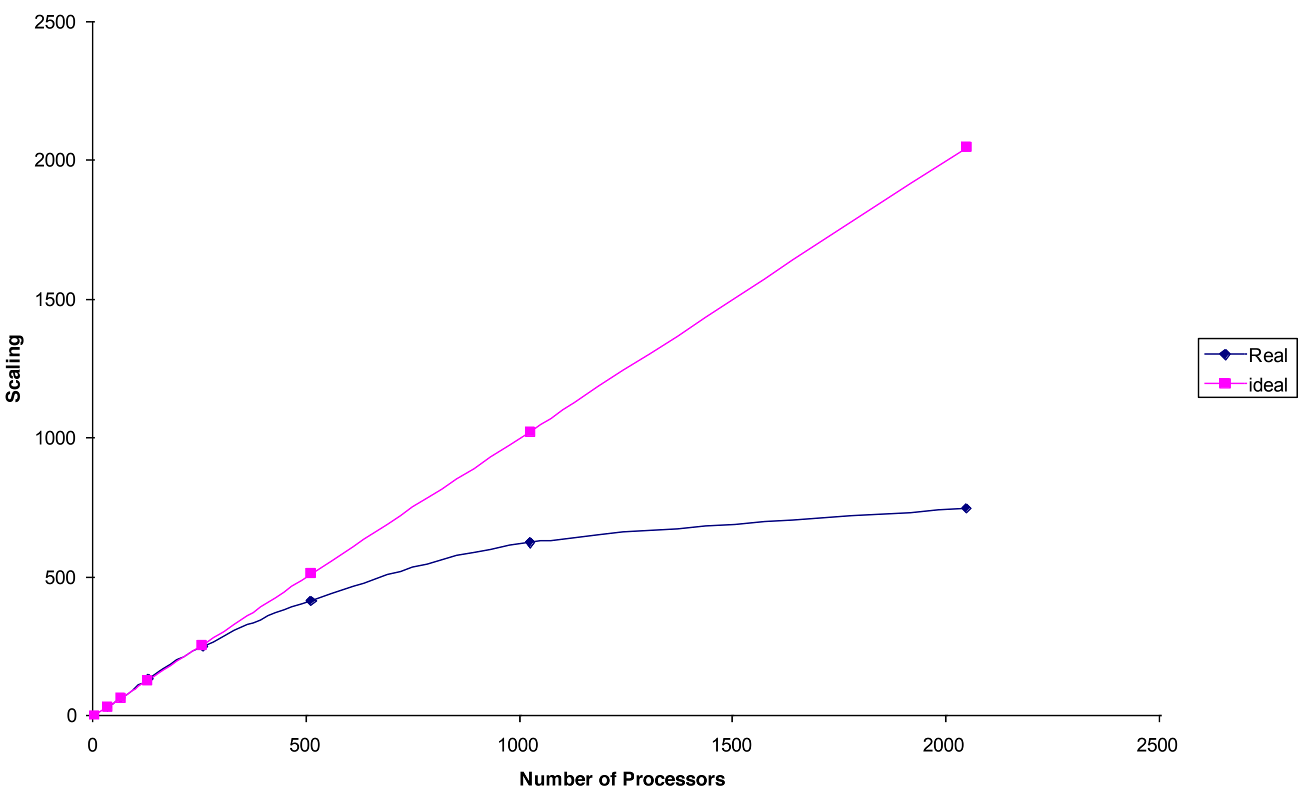
- The speed of the algorithm over multiple computing resources.
- Factors
  - Multi-core architecture
  - Network
  - Algorithm
  - Computational scaling
  - I/O subsystem

Parallel Performance

- Parallel performance is defined in terms of scalability

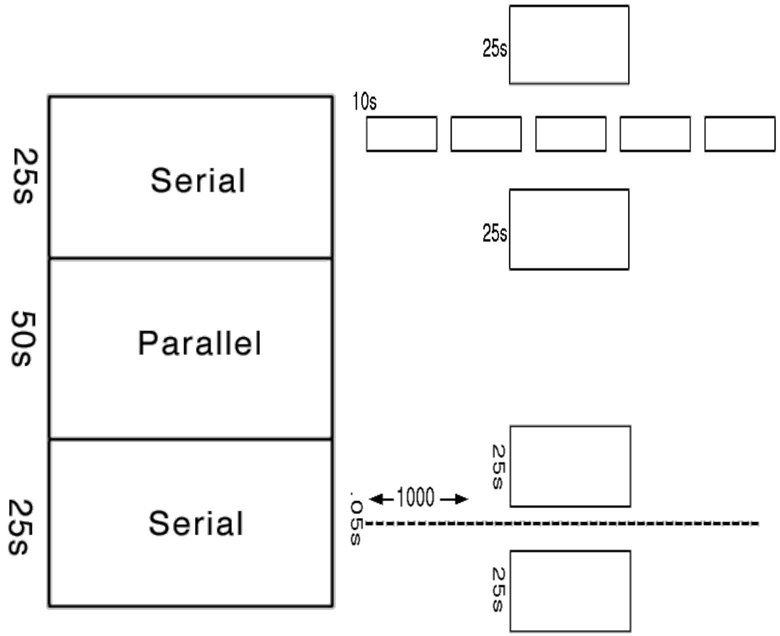
Strong Scalability
Can we get faster for a Problem size.

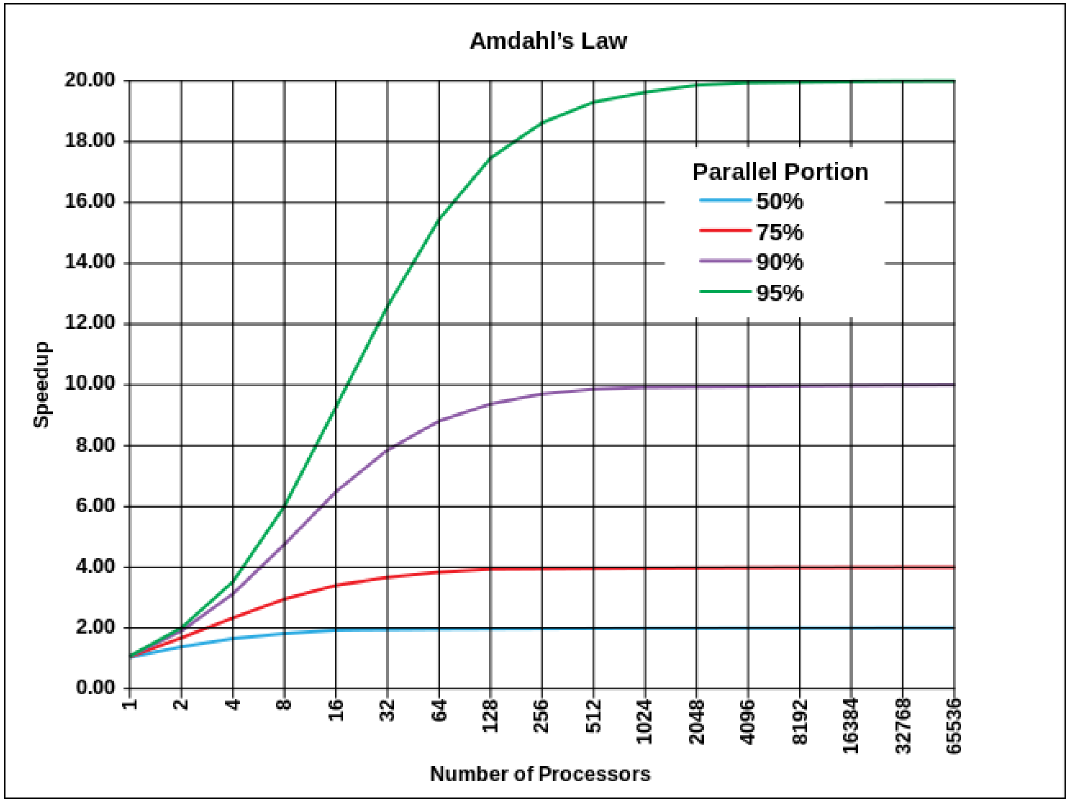
Scaling for LeanCP (32 Water Molecules at 70 Ry) on BigBen (Cray XT3)



Ahmdahl’s Law

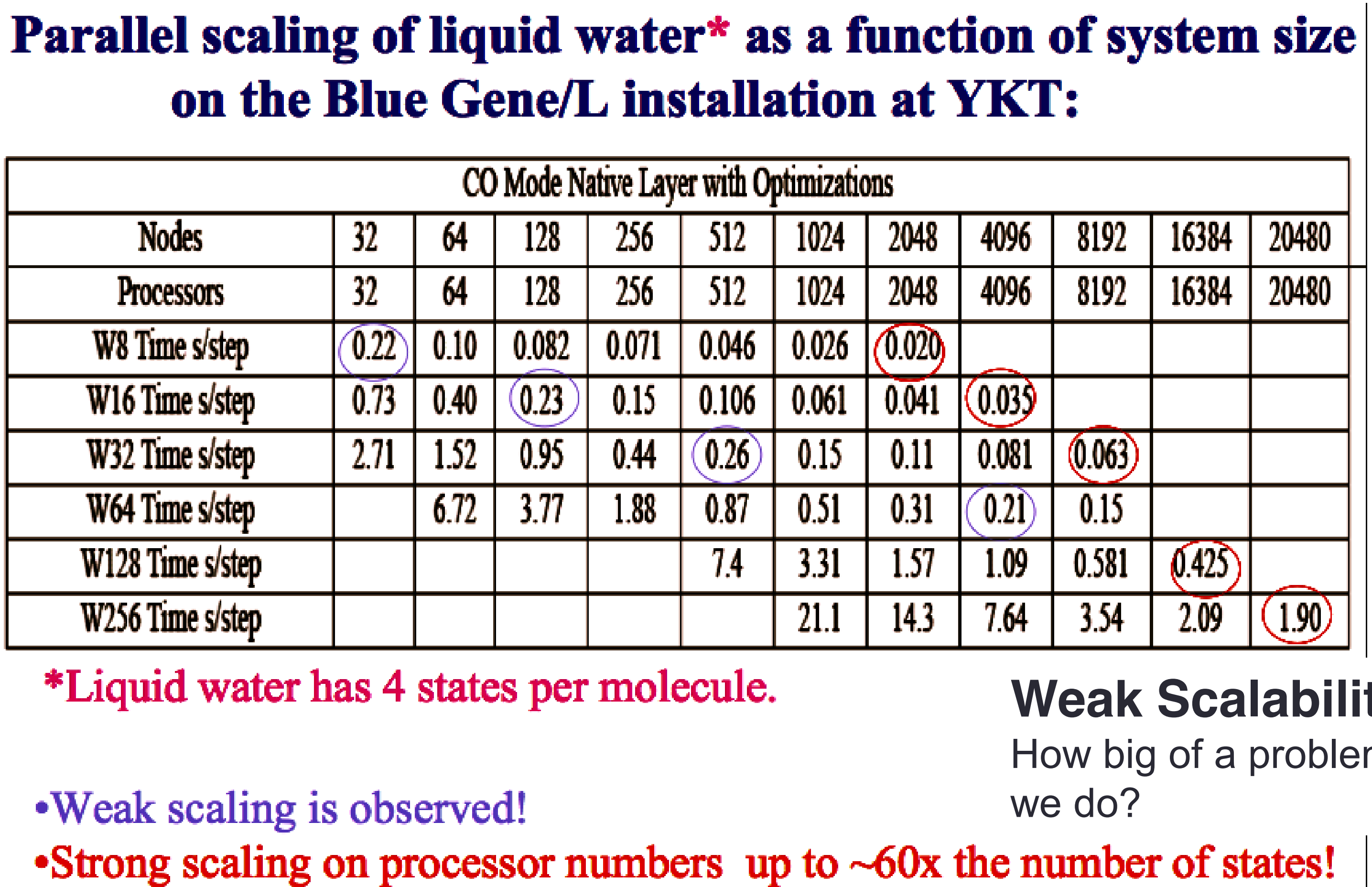
- The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.





Parallel Performance

- Parallel performance is defined in terms of scalability



Parallel scaling of liquid water* as a function of system size on the Blue Gene/L installation at YKT:

CO Mode Native Layer with Optimizations											
Nodes	32	64	128	256	512	1024	2048	4096	8192	16384	20480
Processors	32	64	128	256	512	1024	2048	4096	8192	16384	20480
W8 Time s/step	0.22	0.10	0.082	0.071	0.046	0.026	0.020				
W16 Time s/step	0.73	0.40	0.23	0.15	0.106	0.061	0.041	0.035			
W32 Time s/step	2.71	1.52	0.95	0.44	0.26	0.15	0.11	0.081	0.063		
W64 Time s/step		6.72	3.77	1.88	0.87	0.51	0.31	0.21	0.15		
W128 Time s/step					7.4	3.31	1.57	1.09	0.581	0.425	
W256 Time s/step						21.1	14.3	7.64	3.54	2.09	1.90

*Liquid water has 4 states per molecule.

- Weak scaling is observed!
- Strong scaling on processor numbers up to ~60x the number of states!

Weak Scalability

How big of a problem can we do?

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

TAU: Tuning and Analysis Utilities

- Profiling tool that is inserted in the compilation process to provide a very power and detailed level of performance measurement.
- 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

Why TAU?

- One stop shop for profiling parallel, serial, and shared memory performance.

- Graphical based reports make understanding the performance of your software more manageable.

- Build system makes it easy to quickly instrument your software and create a maintainable performance measurement system.

- Automatic profiling with PAPI, function level call paths, phasing and timers through integration with Makefiles.

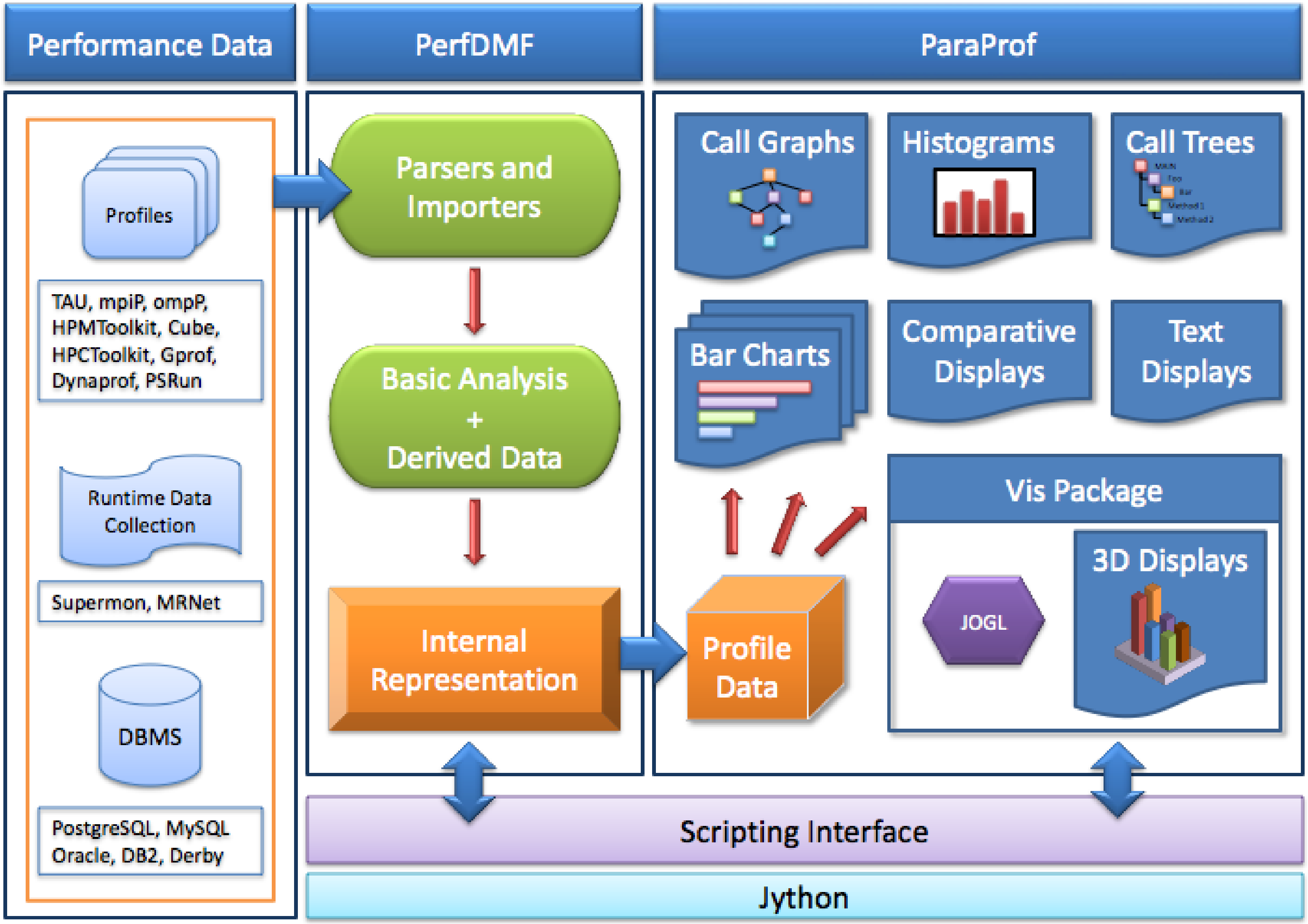
- Available on all major platforms (Windows, OS X, Linux) and works with C/C++, FORTRAN, and Python.

TAU Complications

- TAU works automatically at the source level with very sophisticated parsing … that doesn’t always work right.

- TAU can create a large amount of overhead when running, especially if small functions that are called millions of times are called.

- So much information…. Takes lots of experience to parse the information.



Performance Data

Profiles

TAU, mpiP, ompP, HPMToolkit, Cube, HPCToolkit, Gprof, Dynaprof, PSRun

Runtime Data Collection

Supermon, MRNet

DBMS

PostgreSQL, MySQL Oracle, DB2, Derby

PerfDMF

Parsers and Importers

Basic Analysis + Derived Data

Internal Representation

ParaProf

Call Graphs

Histograms

Call Trees

Bar Charts

Comparative Displays

Text Displays

Profile Data

Vis Package

JOGL

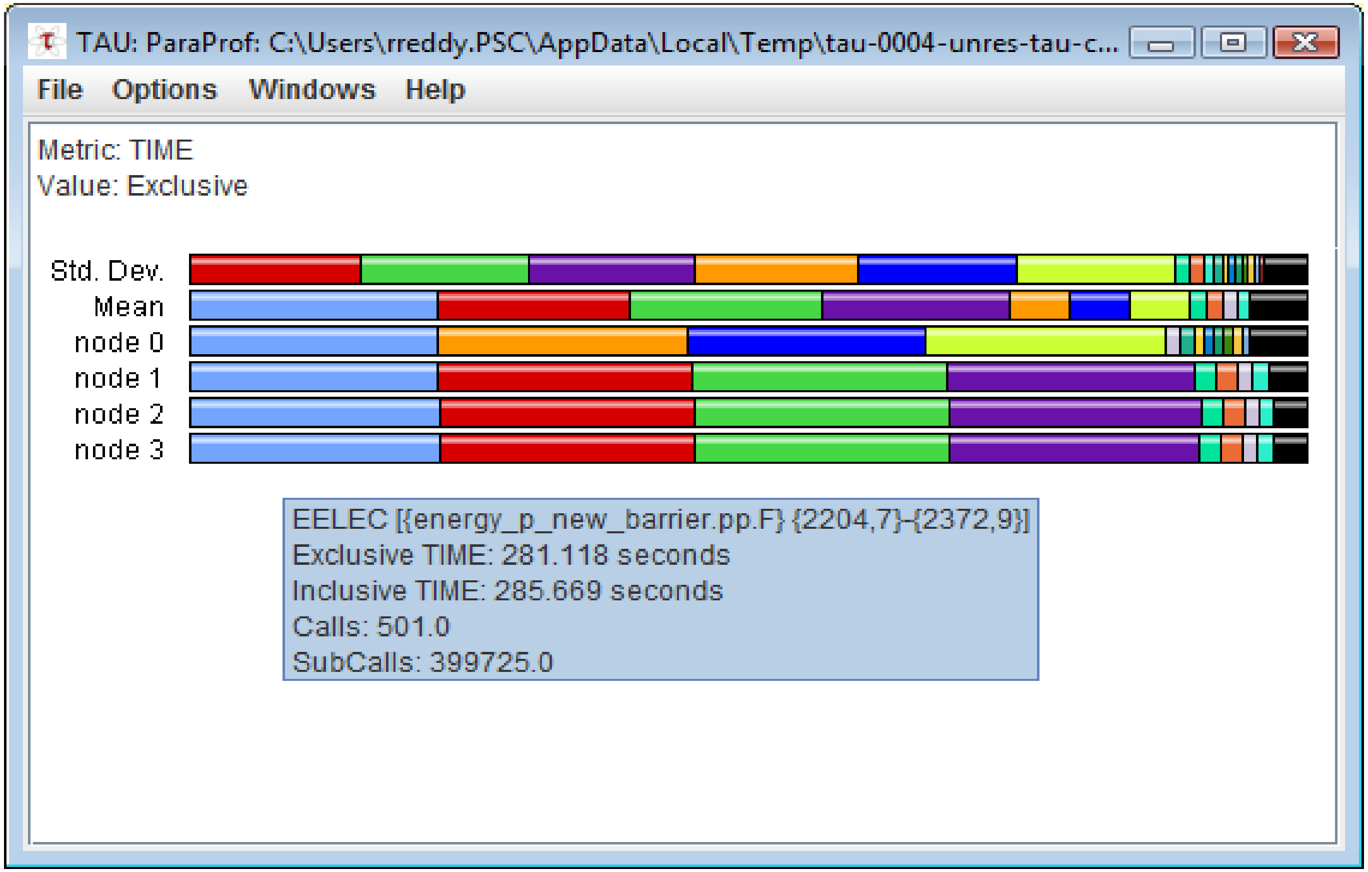
3D Displays

Scripting Interface

Jython

Reproduced from the TAU Tutorial Slides

What Does TAU Look like?



Metric: TIME
Value: Exclusive

EELEC [{energy_p_new_barrier.pp.F} {2204,7}-{2372,9}]
Exclusive TIME: 281.118 seconds
Inclusive TIME: 285.669 seconds
Calls: 501.0
SubCalls: 399725.0

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_cc.sh -o hello hello_mpi.c
```

- 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

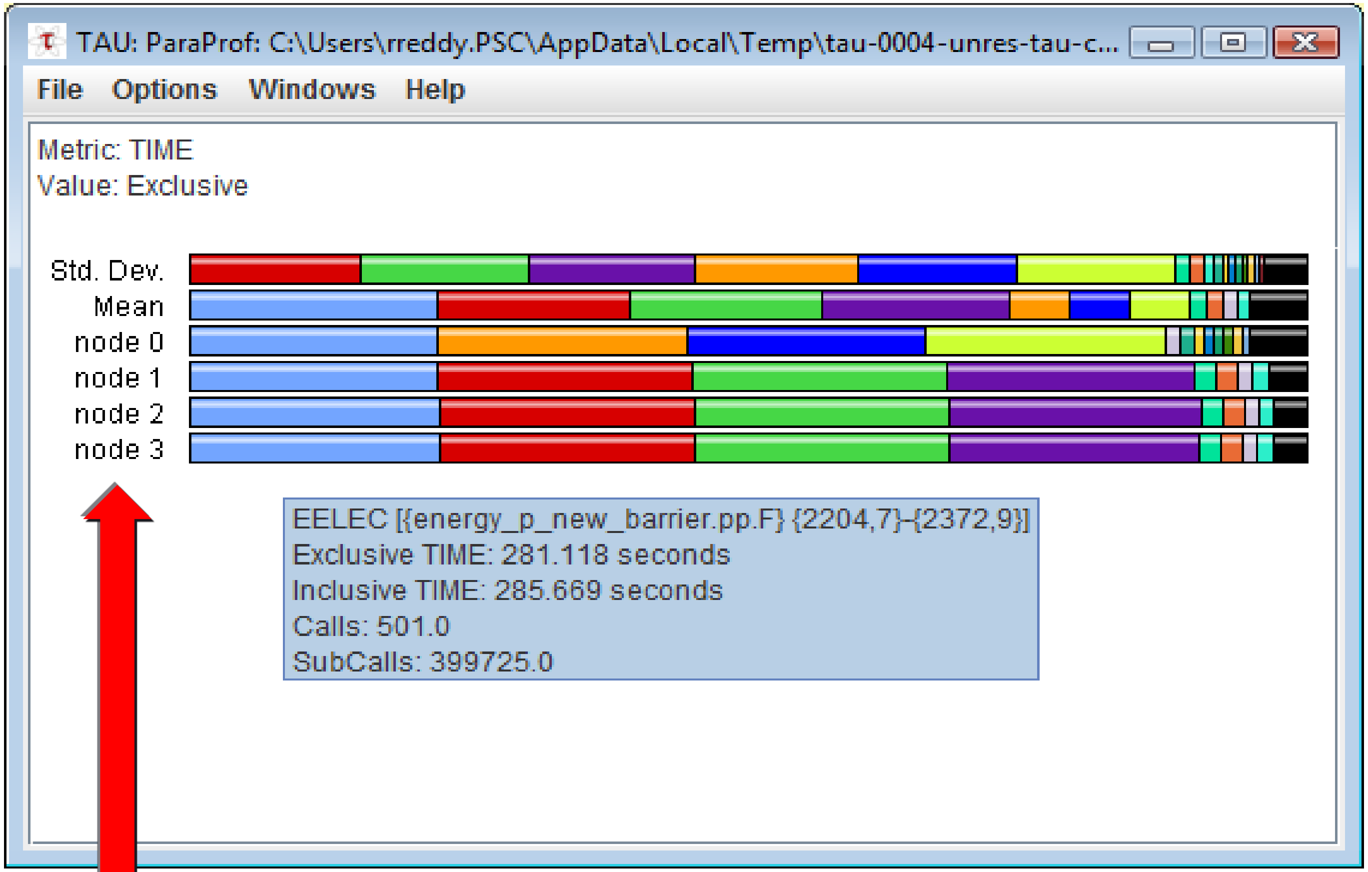
TAU API

- Additionally, TAU defines an API that allows developers to manually instrument their software for very low level control

```
#include <TAU.h>

int main (int argc, char **argv) {
    int ret; pthread_attr_t attr;
    pthread_t tid;
    TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char **)", TAU_DEFAULT);
    TAU_PROFILE_START(tautimer);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0);
    pthread_attr_init(&attr);
    printf("Started Main...\n"); // other statements
    TAU_PROFILE_STOP(tautimer); r
    eturn 0;
}
```

Roadblock: Reducing Overhead

ParaProf Profile Visualization Tool



Metric: TIME
Value: Exclusive

EELEC [{energy_p_new_barrier.pp.F} {2204,7}-{2372,9}]
Exclusive TIME: 281.118 seconds
Inclusive TIME: 285.669 seconds
Calls: 501.0
SubCalls: 399725.0

Click on one of these labels to reveal detailed function info

Overhead (time in sec):

MD steps base:
51.4 seconds

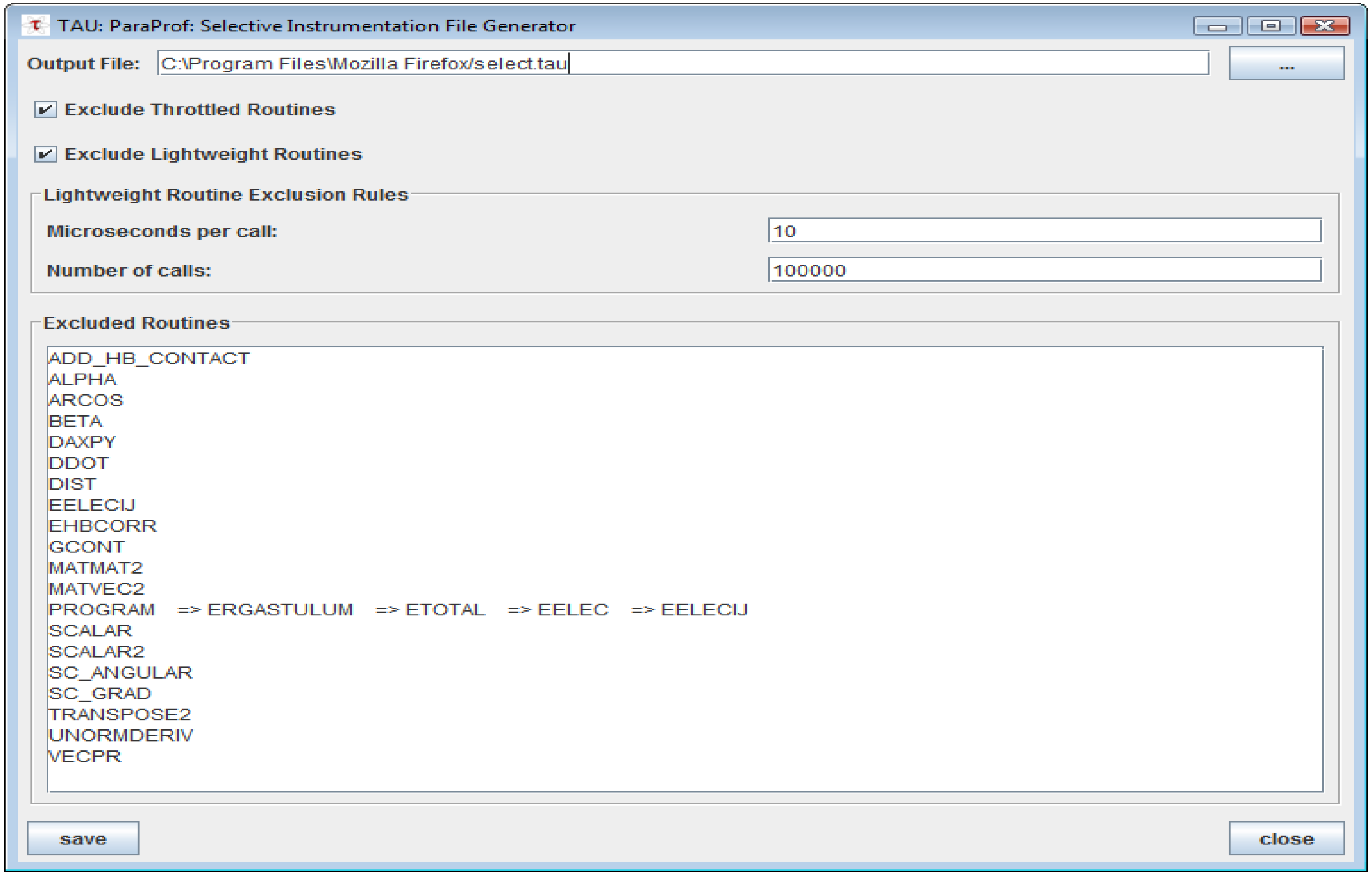
MD steps with TAU:
315 seconds

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



TAU: ParaProf: Selective Instrumentation File Generator

Output File: C:\Program Files\Mozilla Firefox\select.tau

☑ Exclude Throttled Routines

☑ Exclude Lightweight Routines

Lightweight Routine Exclusion Rules

Microseconds per call: 10

Number of calls: 100000

Excluded Routines

```
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
TRANSPOSE2
UNORMDERIV
VECPR
```

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

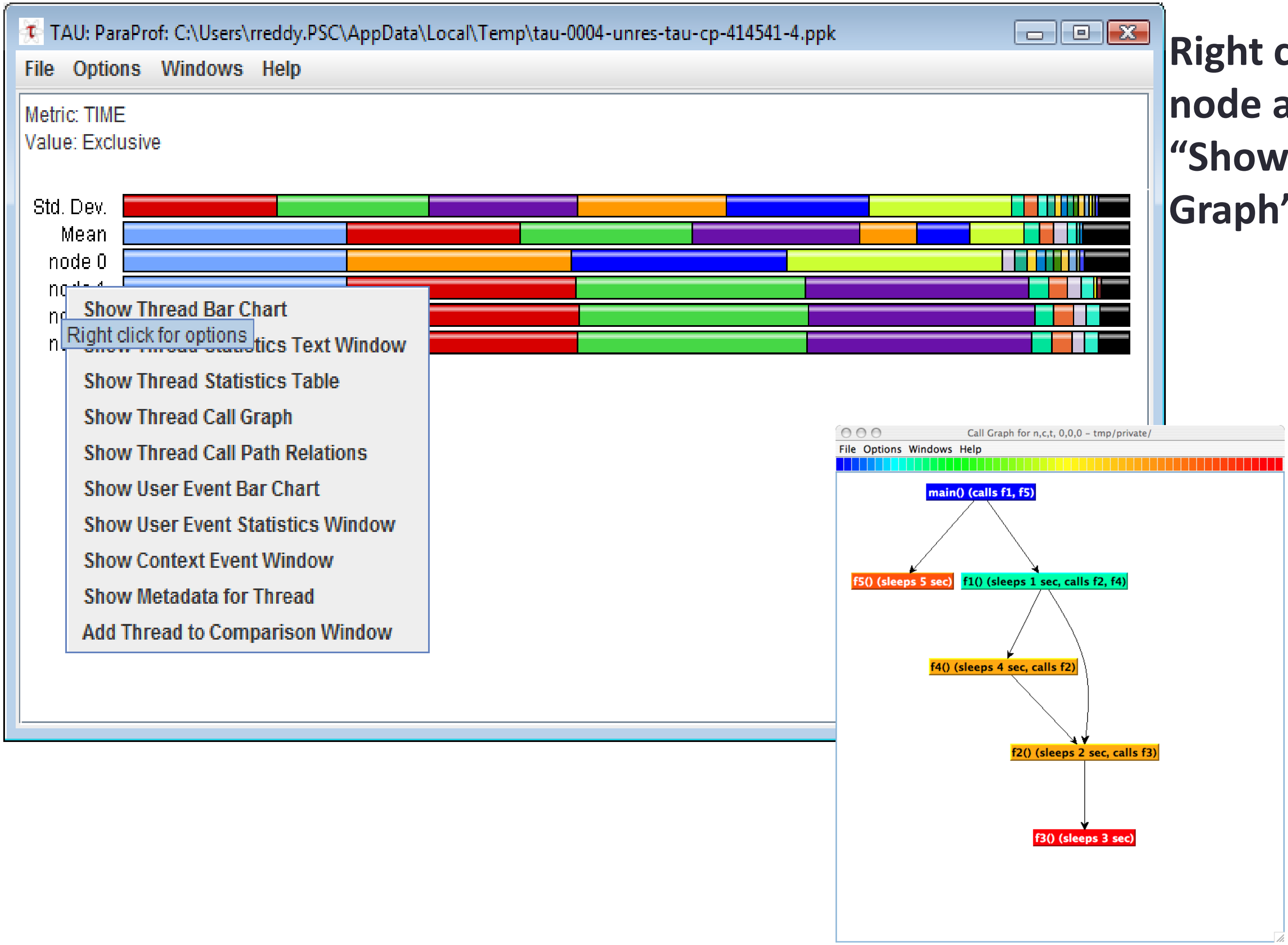
- Why do I need this?
  - To optimize a routine, you often need to know what is above and below it
  - e.g. Determine which routines make significant MPI calls
  - Helps with defining phases: stages of execution within the code that you are interested in

- 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

Getting Call Path Information



Metric: TIME
Value: Exclusive

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

Phase Profiling: Isolate regions of code execution

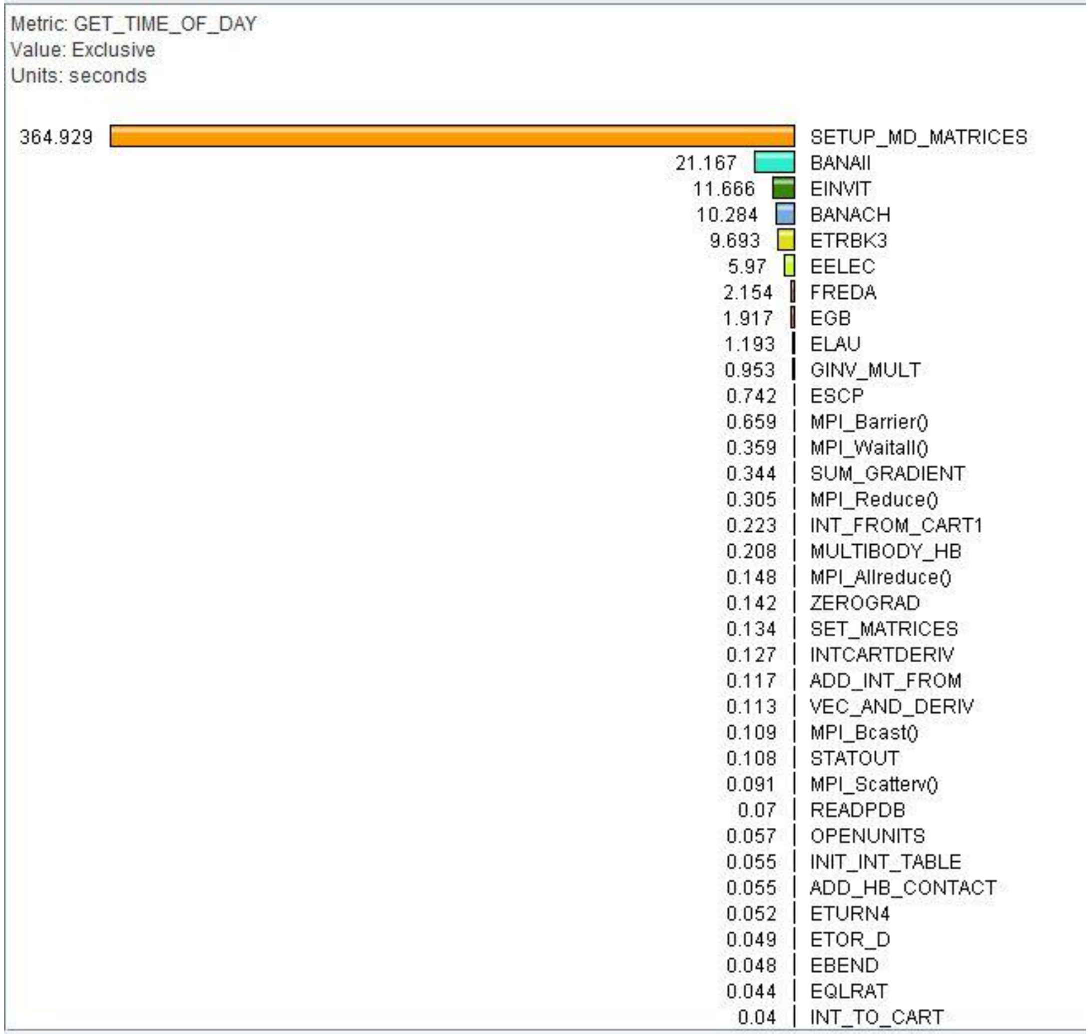
- Eliminated overhead, now we need to deal with startup time:
  - Choose a region of the code of interest: e.g. the main computational kernel
  - Determine where in the code that region begins and ends (call path can be helpful)
  - Then put something like this in selective instrumentation file:

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

  - Recompile and rerun

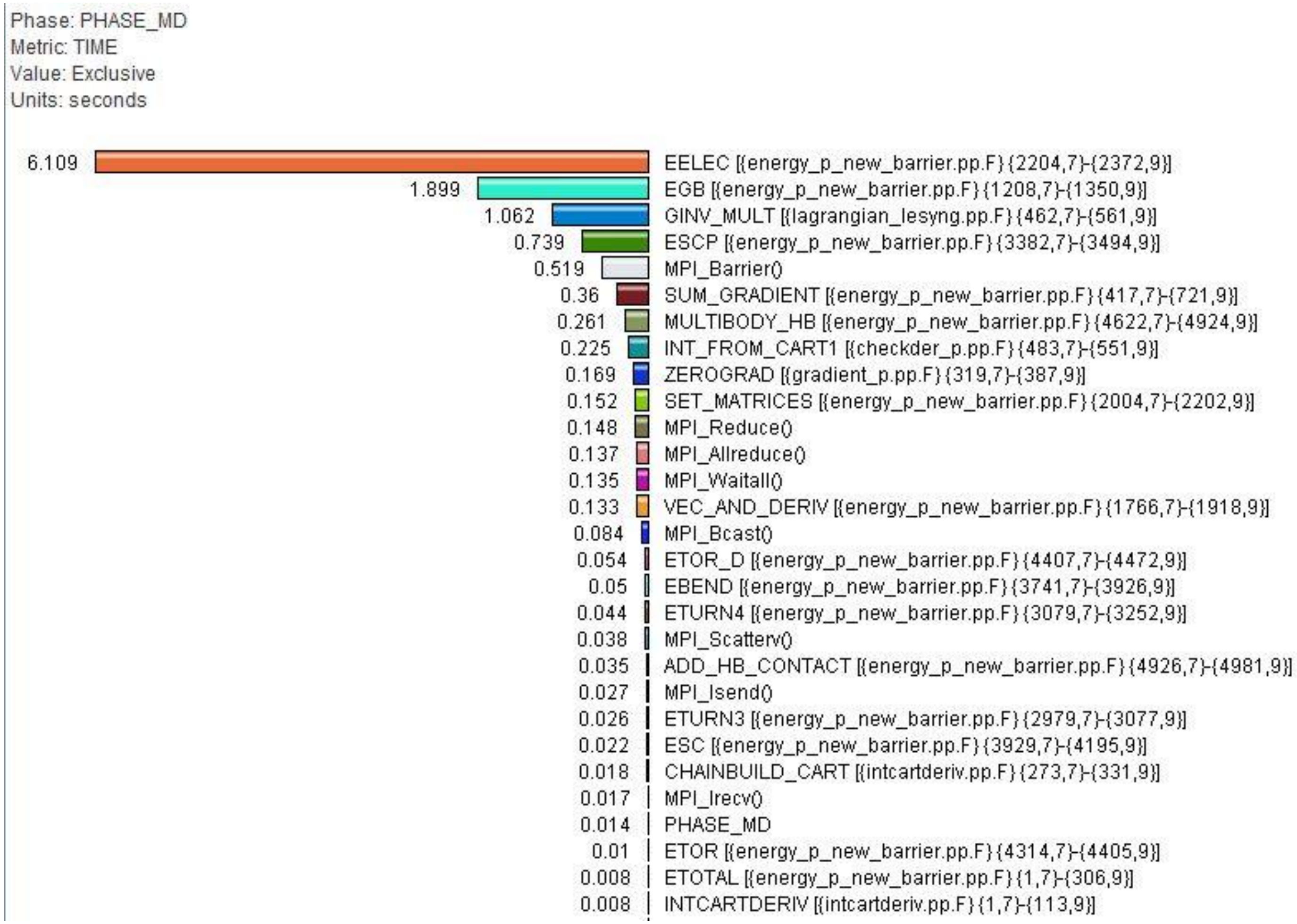
Key UNRES Functions in TAU (with Startup Time)

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



```
Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: seconds

364.929  SETUP_MD_MATRICES
 21.167  BANAII
 11.666  EINVIT
 10.284  BANACH
  9.693  ETRBK3
  5.97   EELEC
  2.154  FREDA
  1.917  EGB
  1.193  ELAU
  0.953  GINV_MULT
  0.742  ESCP
  0.659  MPI_Barrier()
  0.359  MPI_Waitall()
  0.344  SUM_GRADIENT
  0.305  MPI_Reduce()
  0.223  INT_FROM_CART1
  0.208  MULTIBODY_HB
  0.148  MPI_Allreduce()
  0.142  ZEROGRAD
  0.134  SET_MATRICES
  0.127  INTCARTDERIV
  0.117  ADD_INT_FROM
  0.113  VEC_AND_DERIV
  0.109  MPI_Bcast()
  0.108  STATOUT
  0.091  MPI_Scatterv()
  0.07   READPDB
  0.057  OPENUNITS
  0.055  INIT_INT_TABLE
  0.055  ADD_HB_CONTACT
  0.052  ETURN4
  0.049  ETOR_D
  0.048  EBEND
  0.044  EQLRAT
  0.04   INT_TO_CART
```

Key UNRES Functions (MD Time Only)



Phase: PHASE_MD
Metric: TIME
Value: Exclusive
Units: seconds

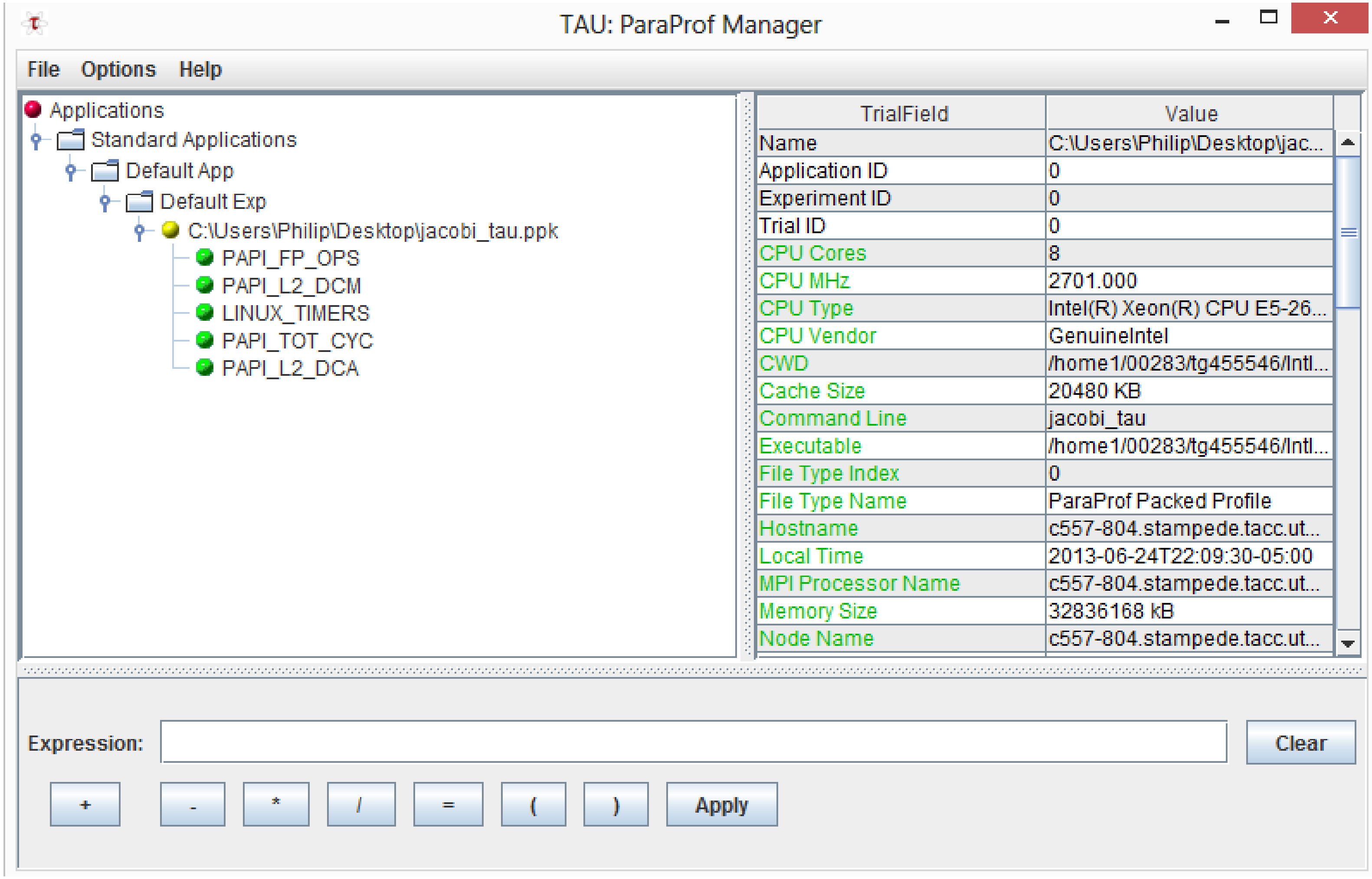
6.109	EELEC [{energy_p_new_barrier.pp.F} {2204,7}-{2372,9}]
1.899	EGB [{energy_p_new_barrier.pp.F} {1208,7}-{1350,9}]
1.062	GINV_MULT [{lagrangian_lesyng.pp.F} {462,7}-{561,9}]
0.739	ESCP [{energy_p_new_barrier.pp.F} {3382,7}-{3494,9}]
0.519	MPI_Barrier()
0.36	SUM_GRADIENT [{energy_p_new_barrier.pp.F} {417,7}-{721,9}]
0.261	MULTIBODY_HB [{energy_p_new_barrier.pp.F} {4622,7}-{4924,9}]
0.225	INT_FROM_CART1 [{checkder_p.pp.F} {483,7}-{551,9}]
0.169	ZEROGRAD [{gradient_p.pp.F} {319,7}-{387,9}]
0.152	SET_MATRICES [{energy_p_new_barrier.pp.F} {2004,7}-{2202,9}]
0.148	MPI_Reduce()
0.137	MPI_Allreduce()
0.135	MPI_Waitall()
0.133	VEC_AND_DERIV [{energy_p_new_barrier.pp.F} {1766,7}-{1918,9}]
0.084	MPI_Bcast()
0.054	ETOR_D [{energy_p_new_barrier.pp.F} {4407,7}-{4472,9}]
0.05	EBEND [{energy_p_new_barrier.pp.F} {3741,7}-{3926,9}]
0.044	ETURN4 [{energy_p_new_barrier.pp.F} {3079,7}-{3252,9}]
0.038	MPI_Scatterv()
0.035	ADD_HB_CONTACT [{energy_p_new_barrier.pp.F} {4926,7}-{4981,9}]
0.027	MPI_Isend()
0.026	ETURN3 [{energy_p_new_barrier.pp.F} {2979,7}-{3077,9}]
0.022	ESC [{energy_p_new_barrier.pp.F} {3929,7}-{4195,9}]
0.018	CHAINBUILD_CART [{intcartderiv.pp.F} {273,7}-{331,9}]
0.017	MPI_Irecv()
0.014	PHASE_MD
0.01	ETOR [{energy_p_new_barrier.pp.F} {4314,7}-{4405,9}]
0.008	ETOTAL [{energy_p_new_barrier.pp.F} {1,7}-{306,9}]
0.008	INTCARTDERIV [{intcartderiv.pp.F} {1,7}-{113,9}]

Detecting Serial Performance Issues

- Identify hardware performance counters of interest
  - papi_avail
  - papi_native_avail
  - Run these commands on compute nodes! Login nodes might 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
```

Create a Derived Metric in Paraprof Manager

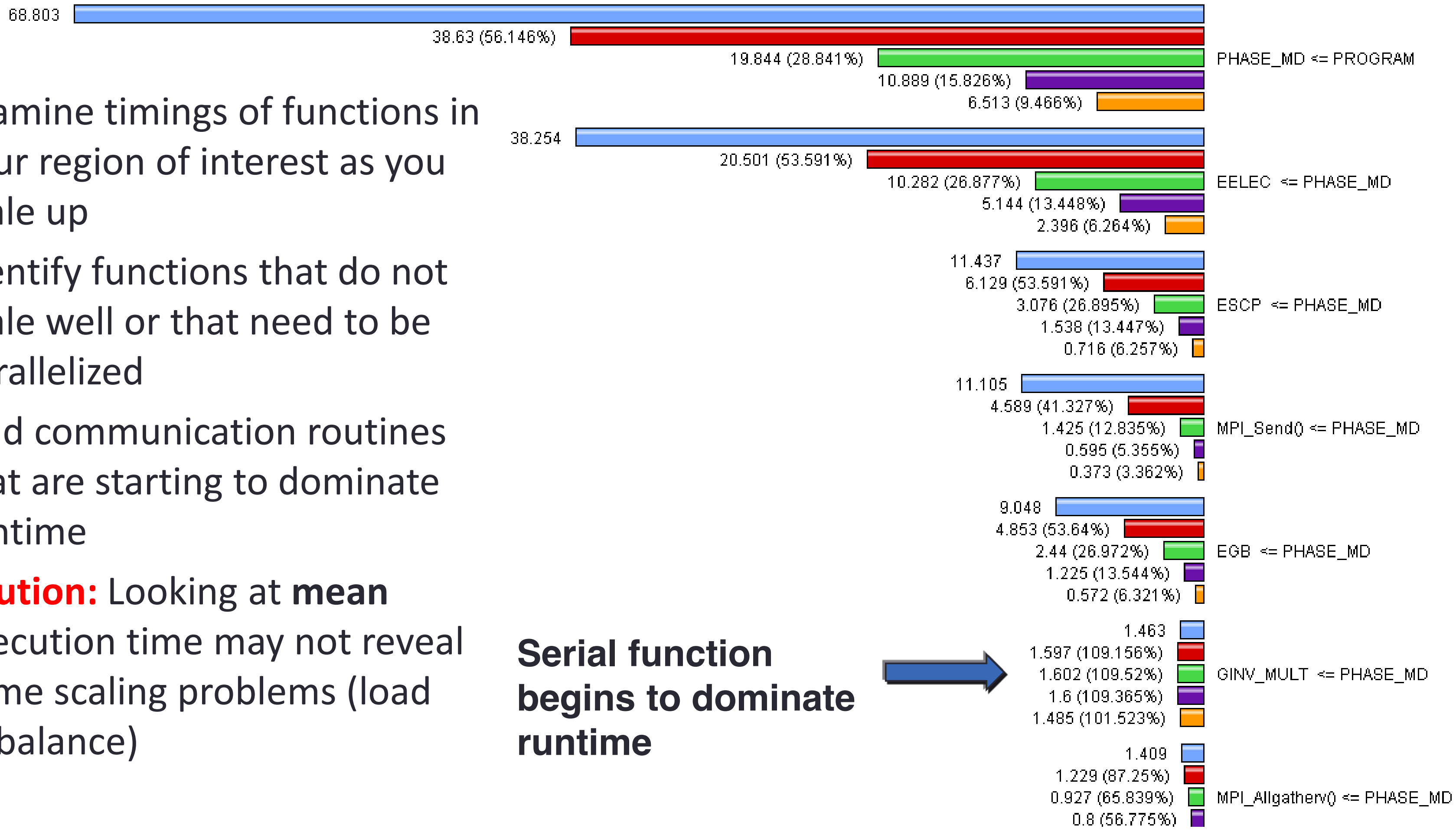


- Applications
  - Standard Applications
    - Default App
      - Default Exp
        - C:\Users\Philip\Desktop\jacobi_tau.ppk
          - PAPI_FP_OPS
          - PAPI_L2_DCM
          - LINUX_TIMERS
          - PAPI_TOT_CYC
          - PAPI_L2_DCA

TrialField	Value
Name	C:\Users\Philip\Desktop\jac...
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	8
CPU MHz	2701.000
CPU Type	Intel(R) Xeon(R) CPU E5-26...
CPU Vendor	GenuineIntel
CWD	/home1/00283/tg455546/Intl...
Cache Size	20480 KB
Command Line	jacobi_tau
Executable	/home1/00283/tg455546/Intl...
File Type Index	0
File Type Name	ParaProf Packed Profile
Hostname	c557-804.stampede.tacc.ut...
Local Time	2013-06-24T22:09:30-05:00
MPI Processor Name	c557-804.stampede.tacc.ut...
Memory Size	32836168 kB
Node Name	c557-804.stampede.tacc.ut...

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)



Serial function begins to dominate runtime

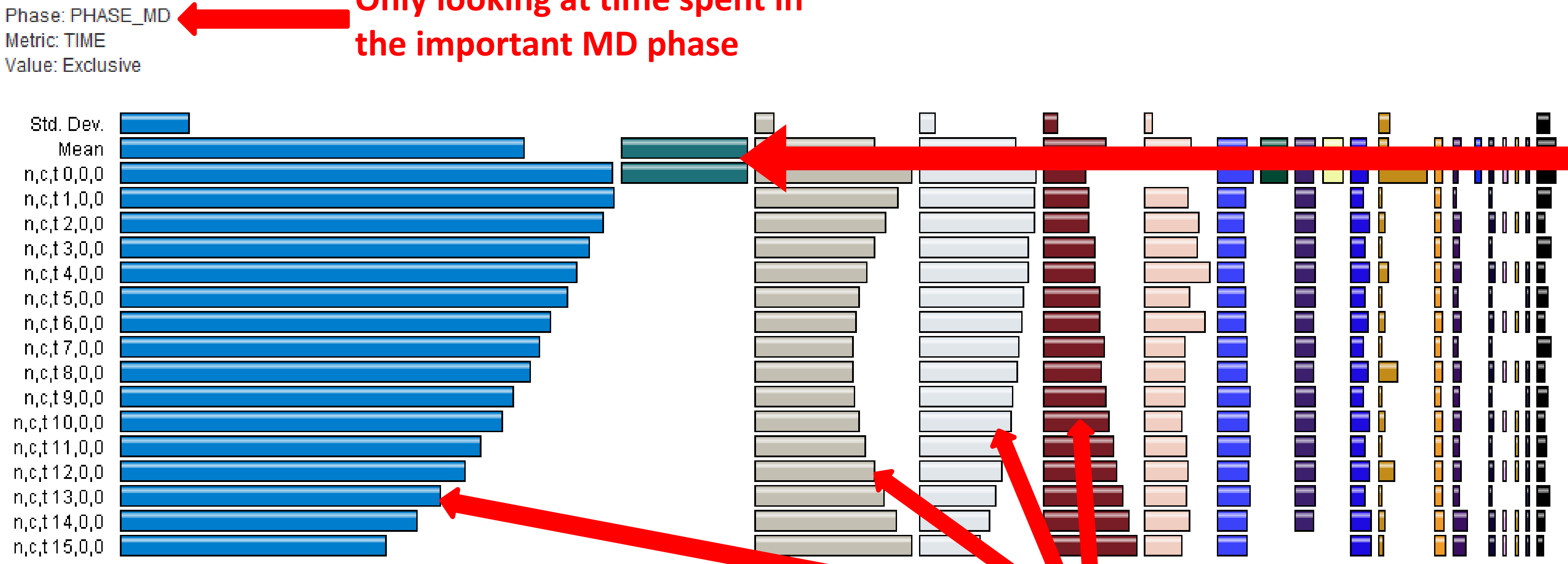
Detecting Parallel Performance Issues: Load Imbalance

- 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 Detection in UNRES



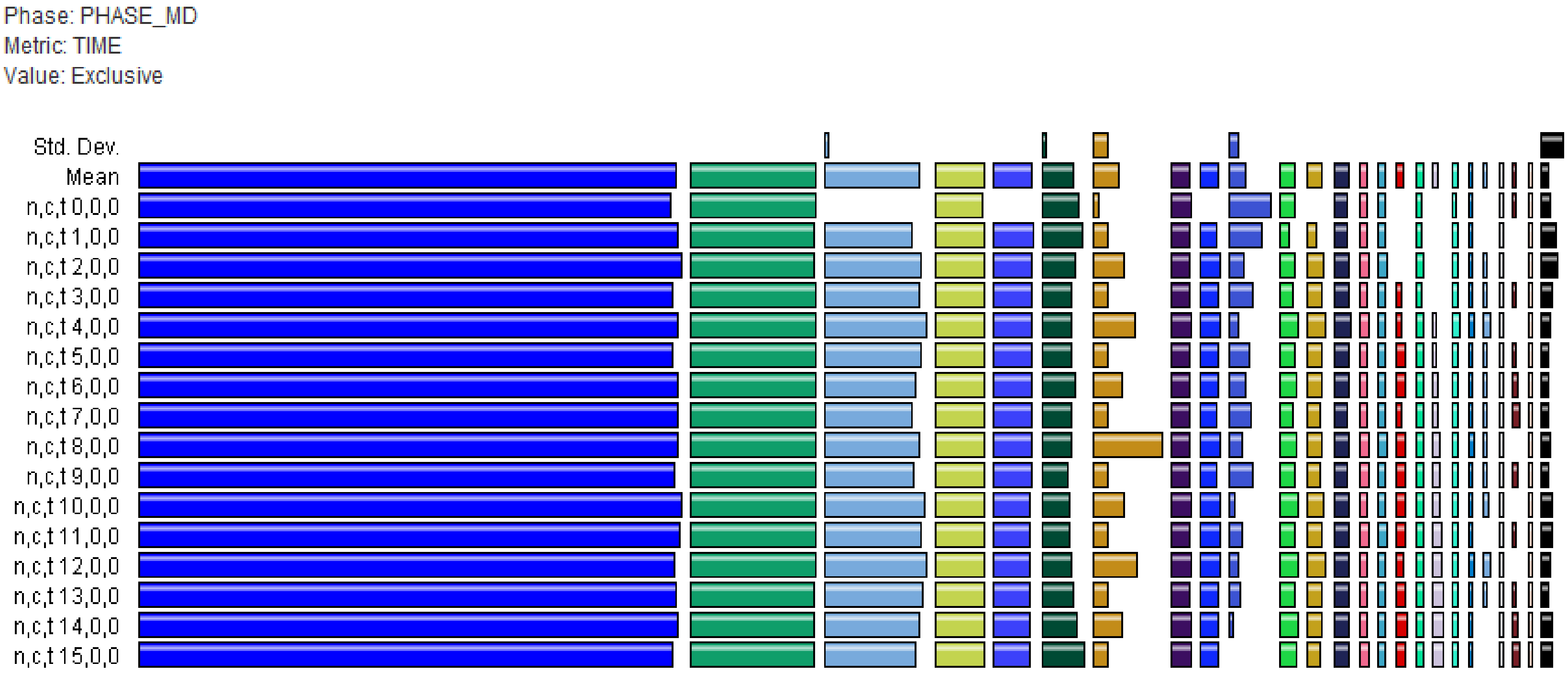
Phase: PHASE_MD
Metric: TIME
Value: Exclusive

Only looking at time spent in the important MD phase

- 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

Observe multiple causes of load imbalance, as well as the serial bottleneck

Major Serial Bottleneck and Load Imbalance in UNRES Eliminated

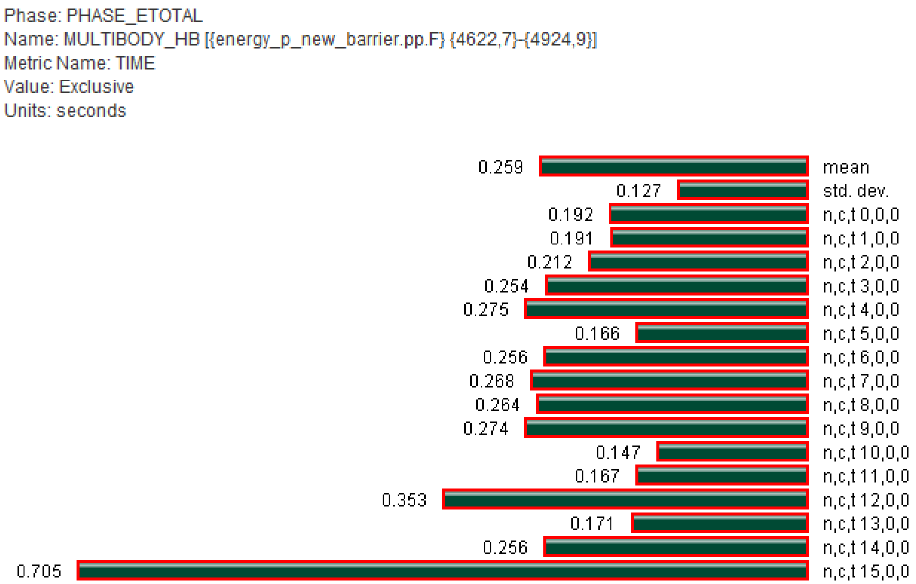


Phase: PHASE_MD
Metric: TIME
Value: Exclusive

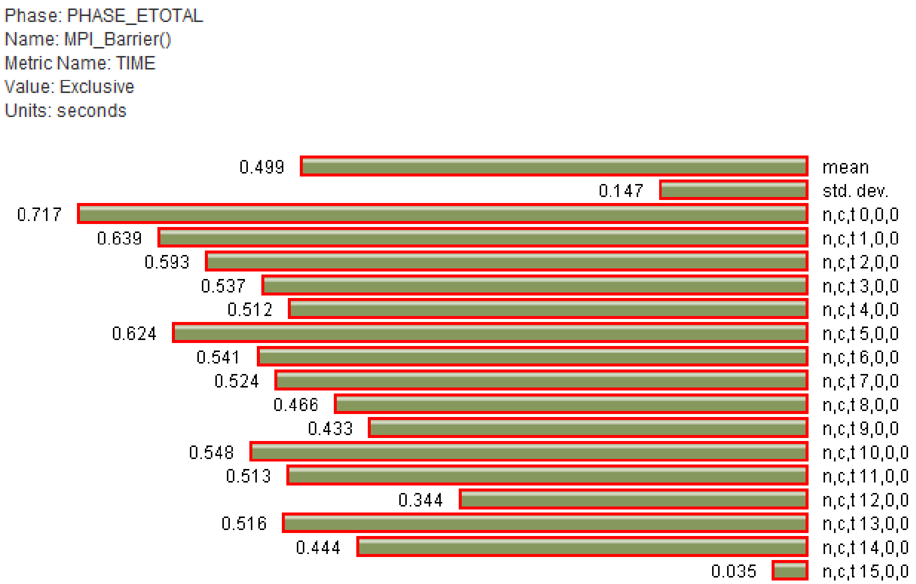
- **Due to 4x faster serial algorithm the balance between computation and communication has shifted – communication must be more efficient to scale well**

- **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

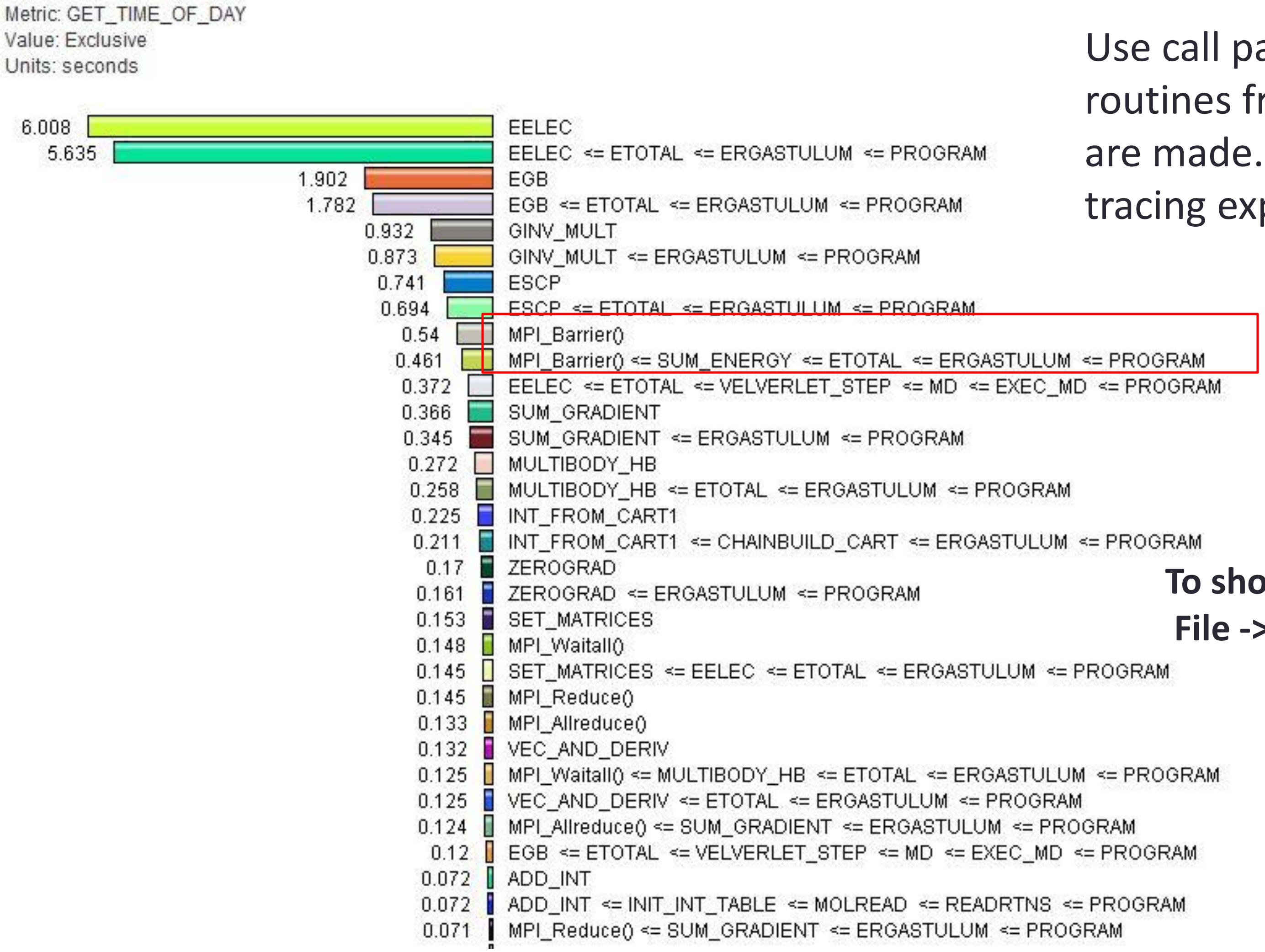


Phase: PHASE_ETOTAL
Name: MPI_Barrier()
Metric Name: TIME
Value: Exclusive
Units: seconds

Load imbalance on one processor 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!

Use Call Path Information: MPI Calls



Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: seconds

6.008	EELEC
5.635	EELEC <= ETOTAL <= ERGASTULUM <= PROGRAM
1.902	EGB
1.782	EGB <= ETOTAL <= ERGASTULUM <= PROGRAM
0.932	GINV_MULT
0.873	GINV_MULT <= ERGASTULUM <= PROGRAM
0.741	ESCP
0.694	ESCP <= ETOTAL <= ERGASTULUM <= PROGRAM
0.54	MPI_Barrier()
0.461	MPI_Barrier() <= SUM_ENERGY <= ETOTAL <= ERGASTULUM <= PROGRAM
0.372	EELEC <= ETOTAL <= VELVERLET_STEP <= MD <= EXEC_MD <= PROGRAM
0.366	SUM_GRADIENT
0.345	SUM_GRADIENT <= ERGASTULUM <= PROGRAM
0.272	MULTIBODY_HB
0.258	MULTIBODY_HB <= ETOTAL <= ERGASTULUM <= PROGRAM
0.225	INT_FROM_CART1
0.211	INT_FROM_CART1 <= CHAINBUILD_CART <= ERGASTULUM <= PROGRAM
0.17	ZEROGRAD
0.161	ZEROGRAD <= ERGASTULUM <= PROGRAM
0.153	SET_MATRICES
0.148	MPI_Waitall()
0.145	SET_MATRICES <= EELEC <= ETOTAL <= ERGASTULUM <= PROGRAM
0.145	MPI_Reduce()
0.133	MPI_Allreduce()
0.132	VEC_AND_DERIV
0.125	MPI_Waitall() <= MULTIBODY_HB <= ETOTAL <= ERGASTULUM <= PROGRAM
0.125	VEC_AND_DERIV <= ETOTAL <= ERGASTULUM <= PROGRAM
0.124	MPI_Allreduce() <= SUM_GRADIENT <= ERGASTULUM <= PROGRAM
0.12	EGB <= ETOTAL <= VELVERLET_STEP <= MD <= EXEC_MD <= PROGRAM
0.072	ADD_INT
0.072	ADD_INT <= INIT_INT_TABLE <= MOLREAD <= READRTNS <= PROGRAM
0.071	MPI_Reduce() <= SUM_GRADIENT <= ERGASTULUM <= PROGRAM

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