

# Profiling and debugging

Yaakoub El Khamra  
yaakoub@tacc.utexas.edu

Texas Advanced Computing Center

# Outline

## Debugging

- GDB
  - Basic use
  - Attaching to a running job
- DDT
  - Identify MPI problems using Message Queues
  - Catch memory errors

## Profiling

- Timers
- GPROF
- Advanced Tools
  - Gprof
  - PerfExpert
  - IPM
  - Tau ( and PAPI)

# Debugging

`gdb` and `ddt`

# Why use a debugger?

- You've got code -> you've got bugs
- Buffered output (printf / write may not help)
- Fast & Accurate
- Many errors are difficult to find without one!

# About GDB

GDB is the **GNU** Project **DeBugger**  
[www.gnu.org/software/gdb/](http://www.gnu.org/software/gdb/)

Looks inside a running program (SERIAL)

*From the GDB website:* GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

# Using GDB

Compile with debug flags: `gcc -g -O0 ./srcFile.c`

The **-g** flag generates the symbol table and provides the debugger with line-by-line information about the source code.

Execute debugger loading source dir: `gdb -d srcDir ./exeFile`

The **-d** option is useful when source and executable reside in different directories.

Use the **-q** option to skip the licensing message.

Type **help** at any time to see a list of the debugger options and commands.

# Two levels of control

- Basic:
  - Run the code and wait for it to crash.
  - Identify line where it crashes.
  - With luck the problem is obvious.
- Advanced:
  - Set breakpoints
  - Analyze data at breakpoints
  - Watch specific variables

# GDB basic commands

command	shorthand	argument	description
run/kill	r / k	NA	start/end program being debugged
continue	c	NA	continue running program from last breakpoint
step	s	NA	take a single step in the program from the last position
where	NA	NA	equivalent to backtrace
print	p	variableName	show value of a variable
list	l	srcFile.c:lineNumber	show the specified source code line
break	b	srcFile.c:lineNumber functionName	set a breakpoint by line number or function name
watch	NA	variableName	stops when the variable changes value

# GDB example

## divcrash.c

```
#include <stdio.h>
#include <stdlib.h>
int myDiv(int, int);

int main(void)
{
    int res, x = 5, y;

    for(y = 1; y < 10; y++){
        res = myDiv(x,y);
        printf("%d,%d,%d\n",x,y,res);
    }
    return 0;
}

int myDiv(int x, int y){
    return 1/( x - y);
}
```

## divcrash.f90

```
PROGRAM main

INTEGER :: myDiv
INTEGER :: res, x = 5, y

DO y = 1, 10
    res = myDiv(x,y)
    WRITE(*,*) x,y,res
END DO

END PROGRAM

FUNCTION myDiv(x,y)
    INTEGER, INTENT(IN) :: x, y
    myDiv = 1/(x-y)
    RETURN
END FUNCTION myDiv
```

# GDB example

Compile the program and start the debugger:

```
% pgcc -g -O0 ./divcrash.c  
% gdb ./a.out
```

Start the program:

```
(gdb) run
```

The debugger will stop program execution with the following message:

```
Program received signal SIGFPE, Arithmetic exception.  
0x00000000040051e in myDiv (x=5, y=5) at divcrash.c:28  
28 return 1/( x - y);
```

We can use **gdb** commands to obtain more information about the problem:

```
(gdb) where  
#0 0x00000000040051e in myDiv (x=5, y=5) at divcrash.c:28  
#1 0x0000000004004cf in main () at divcrash.c:19
```

# GDB example

In this case the problem is clear: a divide-by-zero exception happens in line 28 when variables **x** and **y** are the same.

This is related to the call to **myDiv** from line 19 that is within a for loop:

```
18: for(y = 1; y < 10; y++){  
19:   res = myDiv(x,y);
```

Eventually the loop sets the value of **y** equal to 5 (the value of **x**) producing the exception:

```
28: return 1/( x - y);
```

With the problem identified we can kill the program and exit the debugger :

```
(gdb) kill  
(gdb) quit
```

# Examining data

C	Fortran	Result
(gdb) p x	(gdb) p x	Print scalar data x value
(gdb) p V	(gdb) p V	Print all vector V components
(gdb) p V[i]	(gdb) p V(i)	Print element i of vector V
(gdb) p V[i]@n	(gdb) p V(i)@n	Print n consecutive elements starting with V <sub>i</sub>
(gdb) p M	(gdb) p M	Print all matrix M elements
(gdb) p M[i]	Not Available	Print row i of matrix M
(gdb) p M[i]@n	Not Available	Print n consecutive rows starting with row i
(gdb) p M[i][j]	(gdb) p M(i,j)	Print matrix element M <sub>ij</sub>
(gdb) p M[i][j]@n	(gdb) p M(i,j)@n	Print n consecutive elements starting with M <sub>ij</sub>

- No simple way to print columns in C or rows in Fortran
- Some debuggers print array slices (pgdbg, dbx), i.e. **p M(1:3,3:7)**

# Breakpoint control

- Stop the execution of the program
- Allow you to examine the execution state in detail
- Can be assigned to a line or function
- Can be set conditionally

command	argument	description
info	breakpoints/b/br	Prints to screen all breakpoints
breakpoint	srcFile:lineNumber if a < b	Conditional insertion of breakpoint
enable/disable	breakpointNumber	Enable/disable a breakpoint
delete	breakpointNumber	Delete a breakpoint
clear	srcFile:lineNumber functionName	Clear breakpoints at a given line or function

# Attaching GDB to a running program

Use `top` to find out the PID of the tasks run by your program (in the `top` listing PIDs appear on the left, job names on the right).

```
% top
```

Attach `gdb` to the relevant PID:

```
% gdb -p <PID>
```

or:

```
% gdb  
(gdb) attach <PID>
```

Once attached the debugger pauses execution of the program.

Same level of control than in a standard debugging session.

# Attaching GDB to a running program

Best way to debug production runs.  
Don't wait for your wall time to run out!

From the output of **qstat** obtain the blade where your code is running. In the *queue* field you will find an entry like

**development@i182-103.ta**

queue name

partial blade name:  
i182-103.tacc.utexas.edu

# GDB Summary

- Compile using debug flags:  
**% `icc -g -O0 ./srcFile.c`**
- Run indicating the directory where the source is:  
**% `gdb -d srcDir ./exeFile`**
- Main commands:
  - run/kill
  - continue/next/step
  - break/watch
  - print
  - where
  - help

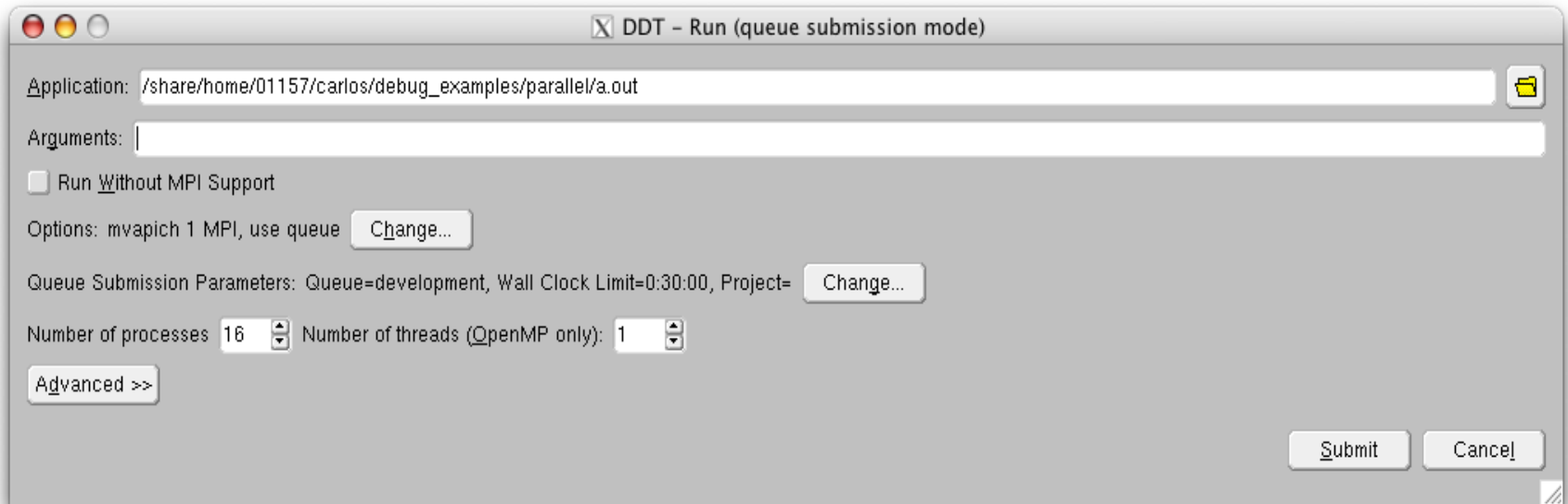
# DDT: Parallel Debugger with GUI

Allinea **D**istributed **D**ebugger **T**ool

[www.allinea.com](http://www.allinea.com)

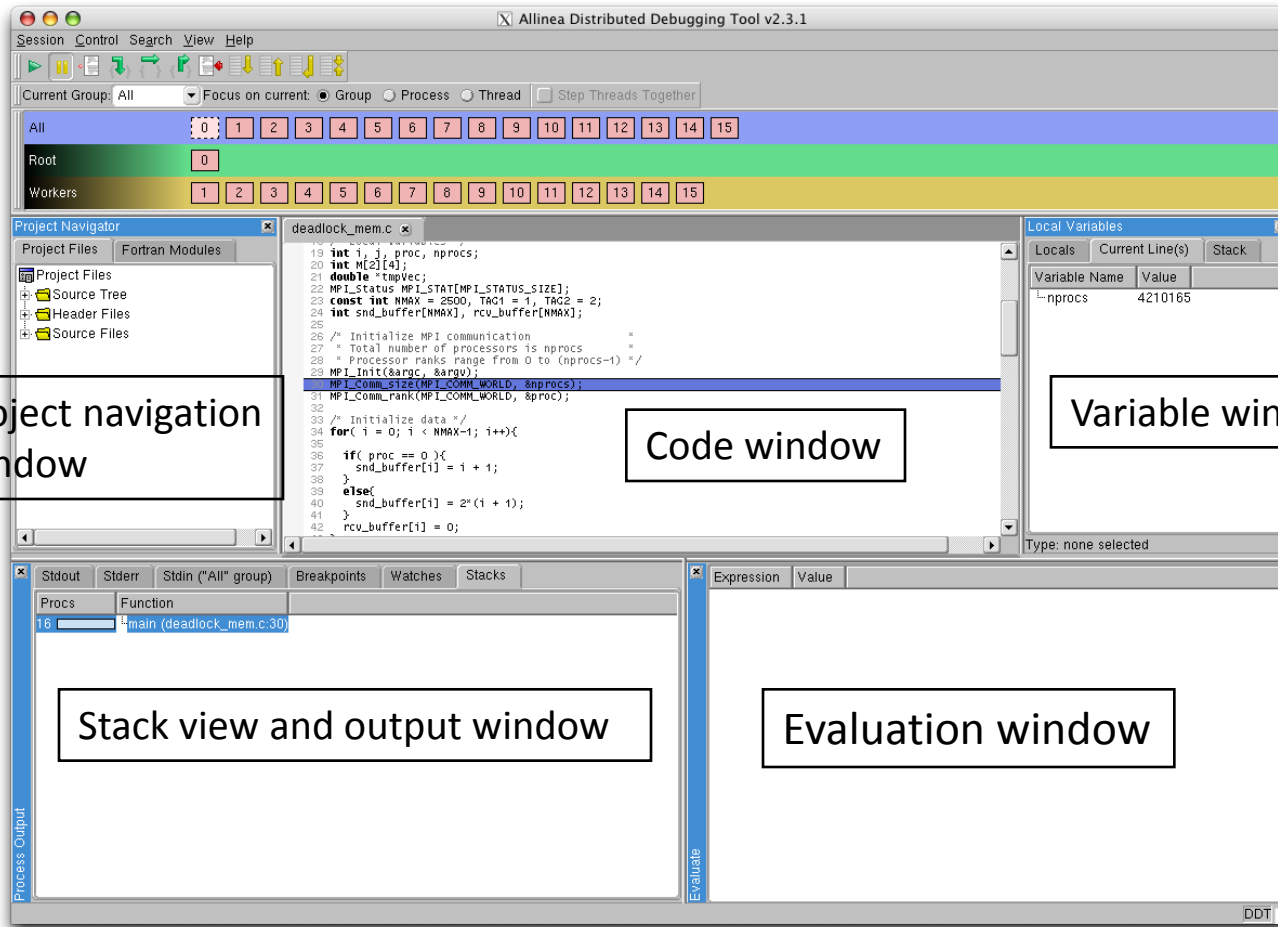
- Multiplatform
- Supports all MPI distributions
- Capable of debugging large scale OMP/MPI
- Comprehensive
  - Memory checking
  - MPI message tracking
- Useful Graphical User Interface

# Configure DDT: Job Submission



- General Options
- Queue Submission Parameters
- Processor and thread number
- Advanced Options

# DDT: The debug session



← Process controls

← Process groups window

Project navigation window

Code window

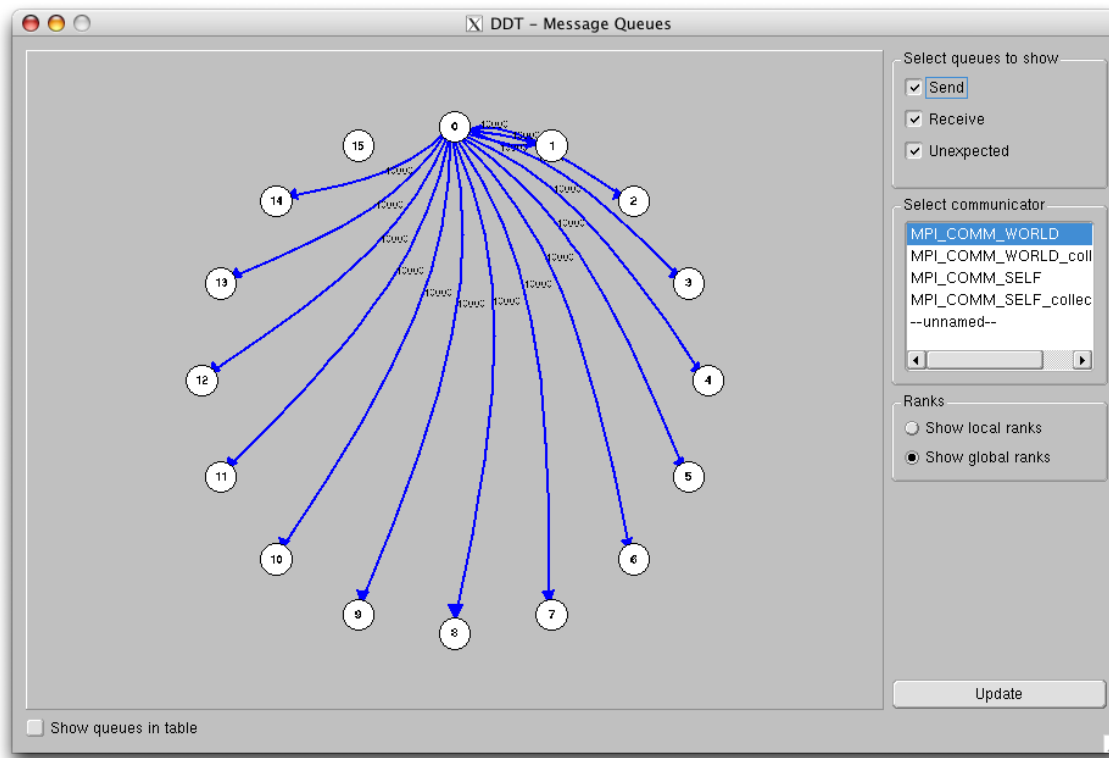
Variable window

Stack view and output window

Evaluation window

# DDT: Message Queues

Go to View -> Message Queues

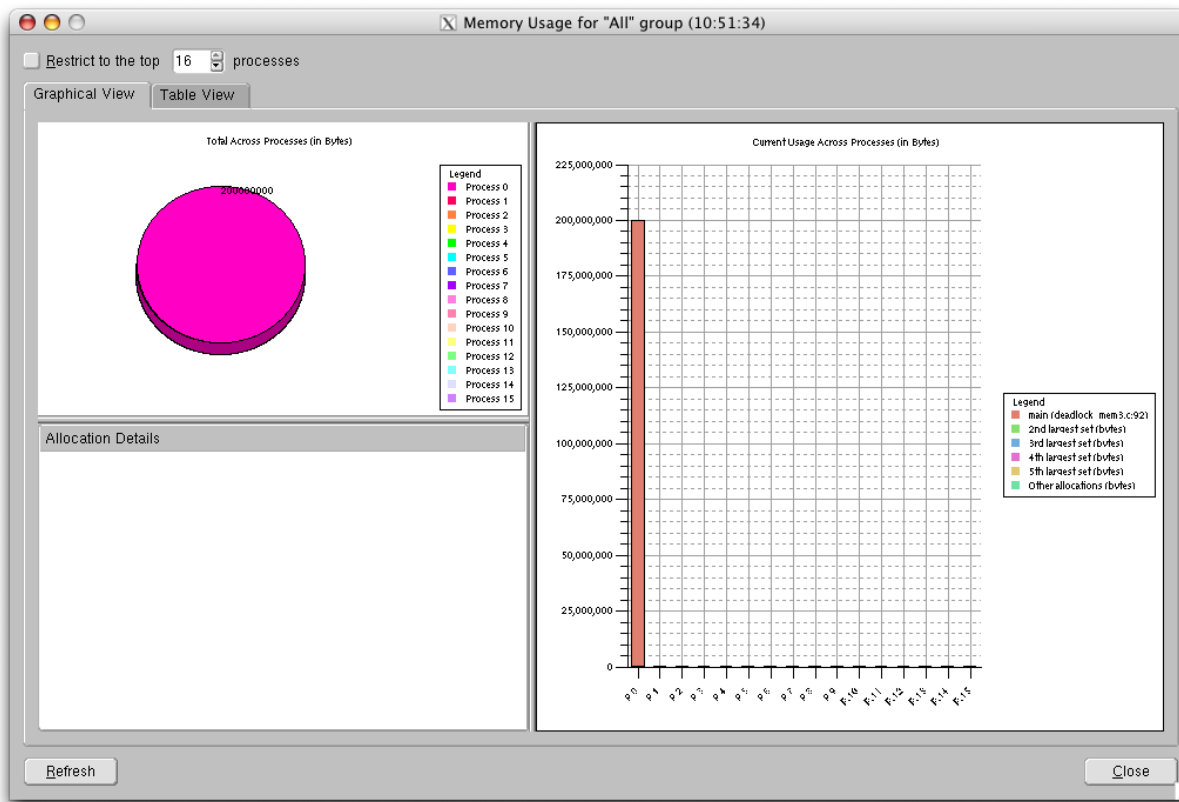


Uncompleted MPI messages appear in the *Unexpected queue*.

Extensive information on message size, sender/receiver available in table form.

# DDT: Memory Leaks

Go to View -> Current Memory Usage



Process 0 is using much more memory than the others.

This looks like a memory leak.

# DDT Summary

- ssh to Ranger allowing X11 forwarding:  
**% ssh -X username@ranger.tacc.utexas.edu**
- Compile with debugging flags:  
**% pgcc -g -O0 ./srcFile.c**
- Load the ddt module  
**% module load ddt**
- Run ddt  
**% ddt ./exeFile**
- Configure ddt properly before submission:
  - Options → MPI version
  - Queue Parameters → Wallclock/CPU/Project
  - Advanced → Memory Checking

# Profiling

timers & gprof

# Timers: Command Line

- The command **time** is available in most Unix systems.
- It is simple to use (no code instrumentation required).
- Gives total execution time of a process and all its children in seconds.

```
% /usr/bin/time -p ./exeFile
```

```
real 9.95
```

```
user 9.86
```

```
sys 0.06
```

Leave out the **-p** option to get additional information:

```
% time ./exeFile
```

```
% 9.860u 0.060s 0:09.95 99.9% 0+0k 0+0io 0pf+0w
```

# Timers: Code Section

```
INTEGER :: rate, start, stop
REAL    :: time

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
CALL SYSTEM_CLOCK(COUNT = start)

! Code to time here

CALL SYSTEM_CLOCK(COUNT = stop)
time = REAL( ( stop - start ) / rate )
```

```
#include <time.h>

double start, stop, time;
start = (double)clock()/CLOCKS_PER_SEC;

/* Code to time here */

stop = (double)clock()/CLOCKS_PER_SEC;
time = stop - start;
```

# About GPROF

GPROF is the **GNU** Project **PROF**iler.

[gnu.org/software/binutils/](http://gnu.org/software/binutils/)

- Requires recompilation of the code.
- Compiler options and libraries provide wrappers for each routine call and periodic sampling of the program.
- Provides three types of profiles
  - Flat profile
  - Call graph
  - Annotated source

# Types of Profiles

- Flat Profile
  - CPU time spend in each function (self and cumulative)
  - Number of times a function is called
  - Useful to identify most expensive routines
- Call Graph
  - Number of times a function was called by other functions
  - Number of times a function called other functions
  - Useful to identify function relations
  - Suggests places where function calls could be eliminated
- Annotated Source
  - Indicates number of times a line was executed

# Profiling with gprof

Use the **-pg** flag during compilation:

```
% gcc -g -pg ./srcFile.c
```

```
% icc -g -p ./srcFile.c
```

```
% pgcc -g -pg ./srcFile.c
```

Run the executable. An output file **gmon.out** will be generated with the profiling information.

Execute **gprof** and redirect the output to a file:

```
% gprof ./exeFile gmon.out > profile.txt
```

```
% gprof -l ./exeFile gmon.out > profile_line.txt
```

```
% gprof -A ./exeFile gmon.out > profile_annotated.txt
```

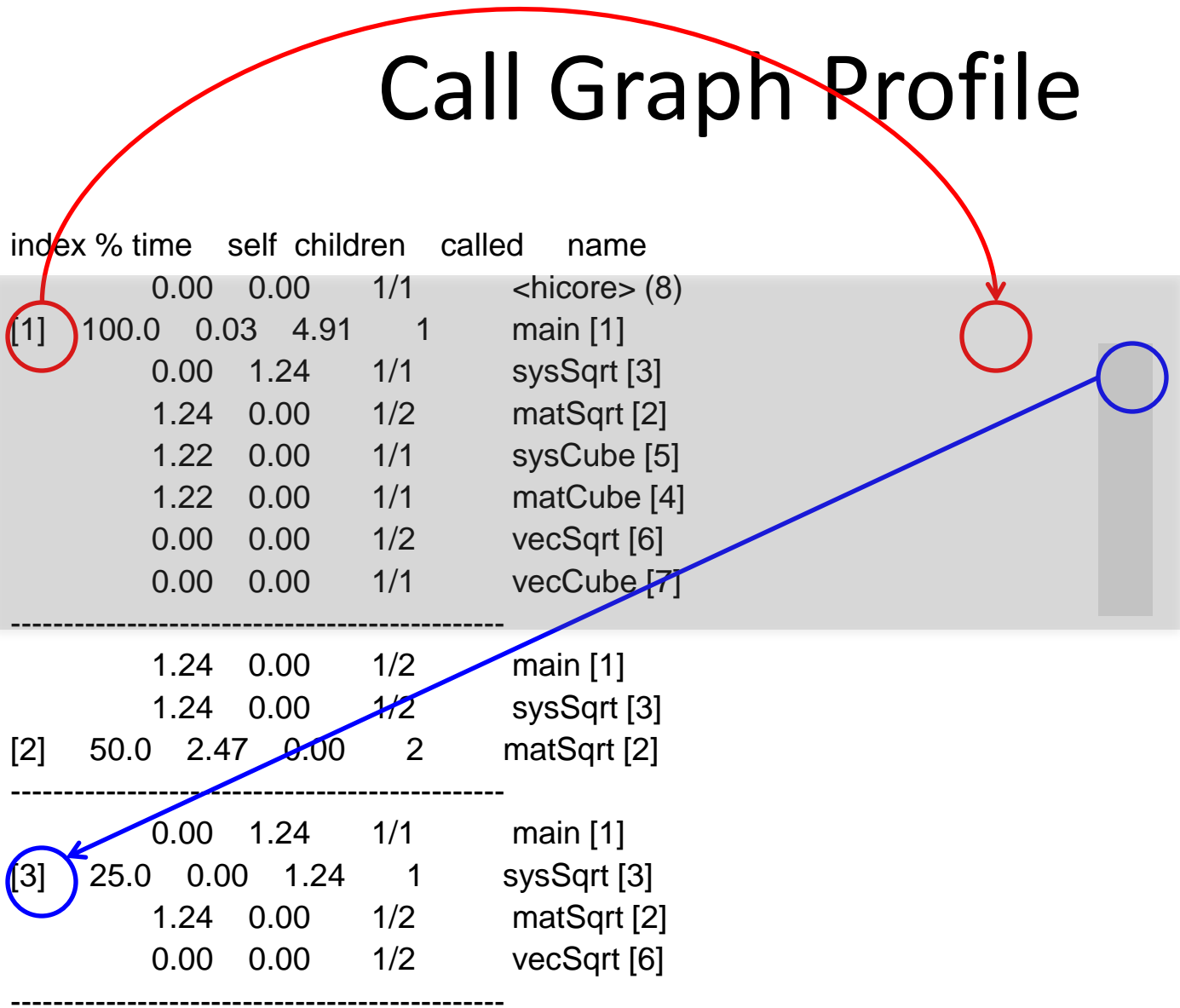
# Flat profile

In the flat profile we can identify the most expensive parts of the code (in this case, the calls to matSqrt, matCube, and sysCube).

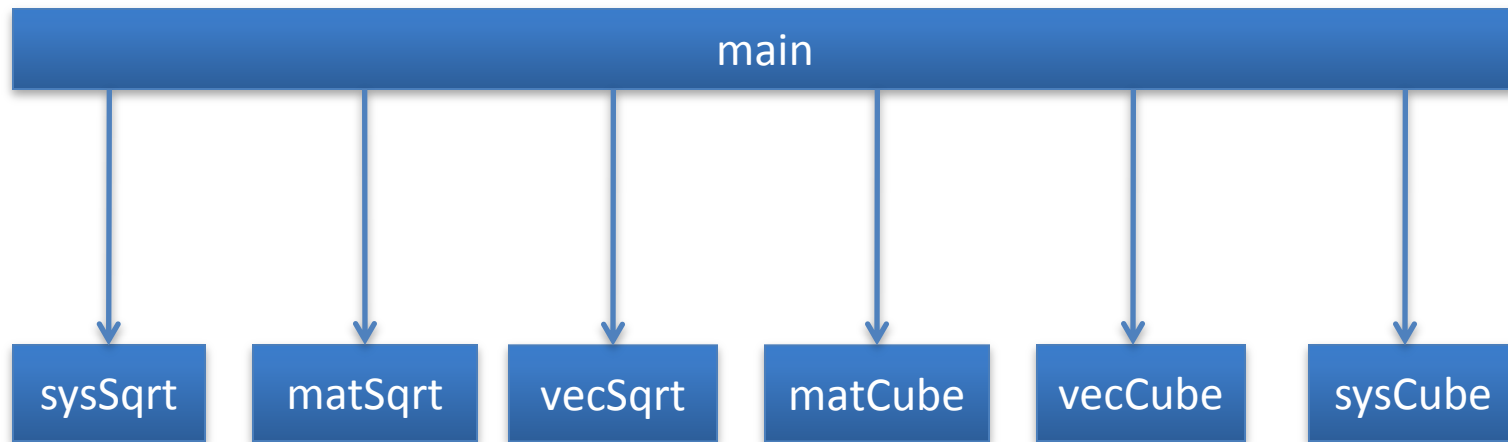
% cumulative time	seconds	self seconds	self calls	s/call	total s/call	name
50.00	2.47	2.47	2	1.24	1.24	matSqrt
24.70	3.69	1.22	1	1.22	1.22	matCube
24.70	4.91	1.22	1	1.22	1.22	sysCube
0.61	4.94	0.03	1	0.03	4.94	main
0.00	4.94	0.00	2	0.00	0.00	vecSqrt
0.00	4.94	0.00	1	0.00	1.24	sysSqrt
0.00	4.94	0.00	1	0.00	0.00	vecCube

# Call Graph Profile

index	% time	self	children	called	name
	0.00	0.00	1/1		<hicore> (8)
[1]	100.0	0.03	4.91	1	main [1]
	0.00	1.24	1/1		sysSqrt [3]
	1.24	0.00	1/2		matSqrt [2]
	1.22	0.00	1/1		sysCube [5]
	1.22	0.00	1/1		matCube [4]
	0.00	0.00	1/2		vecSqrt [6]
	0.00	0.00	1/1		vecCube [7]
-----					
	1.24	0.00	1/2		main [1]
	1.24	0.00	1/2		sysSqrt [3]
[2]	50.0	2.47	0.00	2	matSqrt [2]
-----					
	0.00	1.24	1/1		main [1]
[3]	25.0	0.00	1.24	1	sysSqrt [3]
	1.24	0.00	1/2		matSqrt [2]
	0.00	0.00	1/2		vecSqrt [6]
-----					



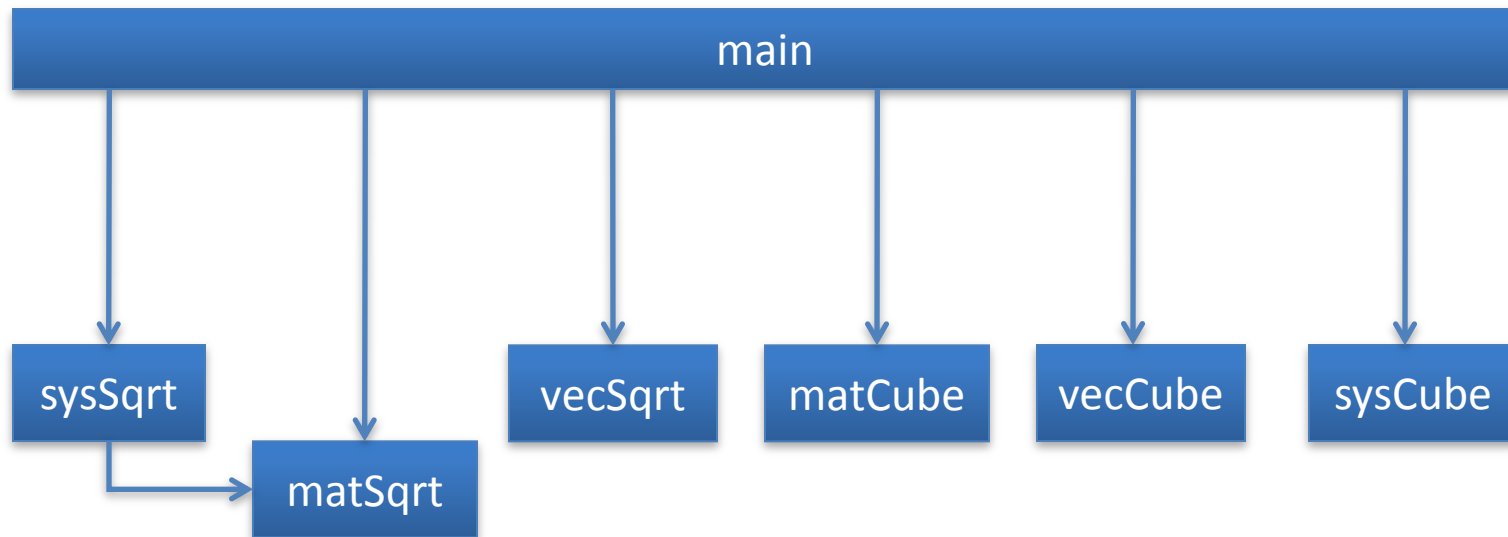
# Visual Call Graph



# Call Graph Profile

	index	% time	self	children	called	name
		0.00	0.00	1/1		<hicore> (8)
[1]	100.0	0.03	4.91	1		main [1]
		0.00	1.24	1/1		sysSqrt [3]
		1.24	0.00	1/2		matSqrt [2]
		1.22	0.00	1/1		sysCube [5]
		1.22	0.00	1/1		matCube [4]
		0.00	0.00	1/2		vecSqrt [6]
		0.00	0.00	1/1		vecCube [7]
-----						
		1.24	0.00	1/2		main [1]
		1.24	0.00	1/2		sysSqrt [3]
[2]	50.0	2.47	0.00	2		matSqrt [2]
-----						
		0.00	1.24	1/1		main [1]
[3]	25.0	0.00	1.24	1		sysSqrt [3]
		1.24	0.00	1/2		matSqrt [2]
		0.00	0.00	1/2		vecSqrt [6]

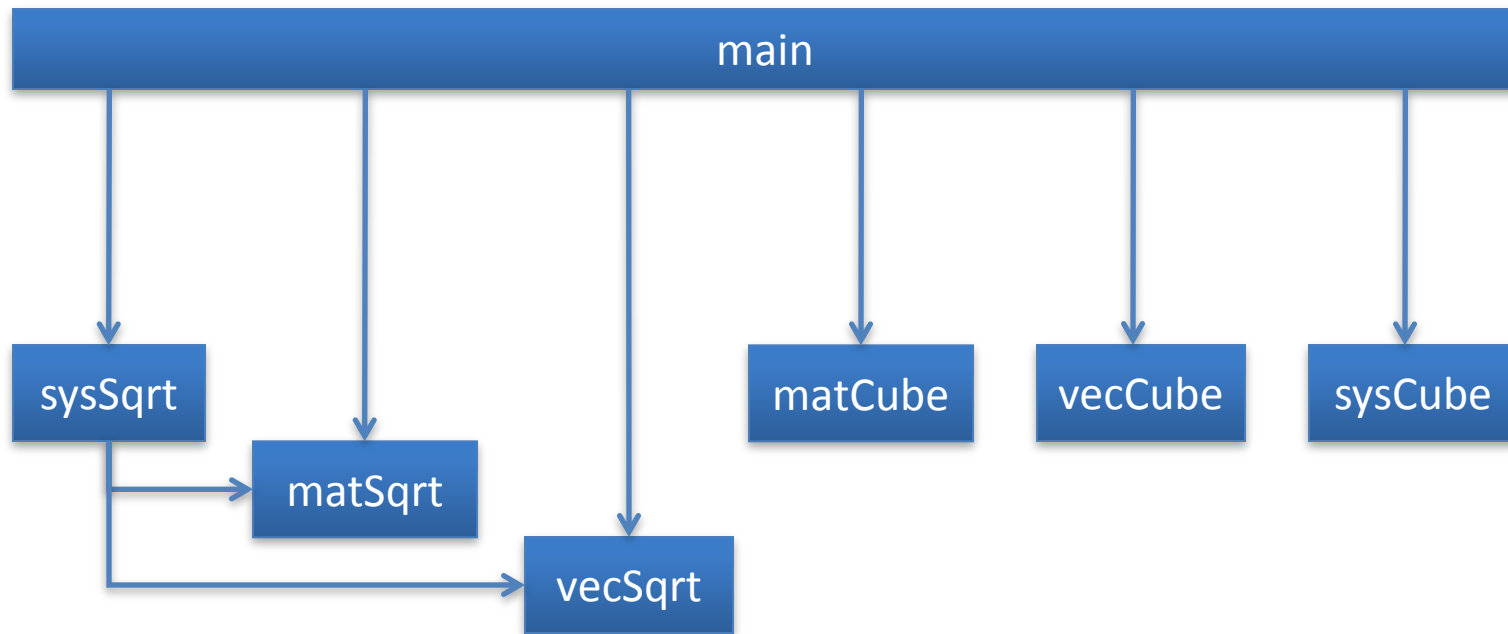
# Visual Call Graph



# Call Graph Profile

	index	% time	self	children	called	name
		0.00	0.00	1/1		<hicore> (8)
[1]	100.0	0.03	4.91	1		main [1]
		0.00	1.24	1/1		sysSqrt [3]
		1.24	0.00	1/2		matSqrt [2]
		1.22	0.00	1/1		sysCube [5]
		1.22	0.00	1/1		matCube [4]
		0.00	0.00	1/2		vecSqrt [6]
		0.00	0.00	1/1		vecCube [7]
-----						
		1.24	0.00	1/2		main [1]
		1.24	0.00	1/2		sysSqrt [3]
[2]	50.0	2.47	0.00	2		matSqrt [2]
-----						
		0.00	1.24	1/1		main [1]
[3]	25.0	0.00	1.24	1		sysSqrt [3]
		1.24	0.00	1/2		matSqrt [2]
		0.00	0.00	1/2		vecSqrt [6]
-----						

# Visual Call Graph



# PERF-EXPERT

# About PerfExpert

- Brand new tool, locally developed at UT
- Easy to use and understand
- Great for quick profiling and for beginners
- Provides recommendation on “what to fix” in a subroutine
- Collects information from PAPI using HPCToolkit
- No MPI specific profiling, no 3D visualization, no elaborate metrics
- Combines ease of use with useful interpretation of gathered performance data

# Profiling with PerfExpert: Compilation

- Load the java, papi, and perfexpert modules:
  - **module load java papi perfexpert**
- Compile the code with full optimization and with the -g flag:
  - **mpicc -g -O3 source.c**
  - **mpif90 -g -O3 source.f90**
- Copy the **PerfExpert.sge** submission script (for editing):
  - **cp \$TACC\_PERFEXPERT\_DIR/PerfExpert.sge ./**
- Edit the **PerfExpert.sge** script to ensure the correct executable name, correct directory, correct project name and so on.

# The PerfExpert.sge Job Script

- Edit the sample submission **PerfExpert.sge** script to match your code. You need to edit:
  - The project name
  - The “wayness” of the job and the number of cores
  - The directory where the code will run
  - Executable name
  - Any arguments for your code
- Make sure you request roughly **7 times** the normal runtime for your executable
- Submit your job
  - **qsub PerfExpert.sge**

# The PerfExpert.sge Job Script

```
#!/bin/bash
#$ -A RangerTechInsertion
#$ -V          # Inherit the submission environment
#$ -cwd       # Start job in submission directory
#$ -N PerfExpert # Job Name
#$ -j y       # combine stderr & stdout into stdout
#$ -o $JOB_NAME.$JOB_ID.out # Name of the output file
#$ -pe 1way 16 # Requests x cores/node, y cores total
#$ -q development # Queue name
#$ -l h_rt=00:02:00 # Run time (hh:mm:ss)
```

```
export MyExePath=./
export MyExeName=a.out
export MyCmdLine="myargs"
```

**You only need to modify these!**

##### do not modify anything below this line #####

**DO NOT CHANGE ANYTHING IN THIS SECTION**

# PerfExpert Analysis

- When your job is completed analyze the performance using:
  - [PerfExpert 0.1 ./hpctoolkit-<job name>/experiment.xml](#)
- This will give an assessment of performance in 6 critical areas:
  - Data accesses
  - Instruction accesses
  - Floating point instructions
  - Branch instructions
  - Data TLB
  - Instruction TLB
- Look for tips on how to improve performance in different areas at the PerfExpert site:
  - [www.tacc.utexas.edu/perfexpert/](http://www.tacc.utexas.edu/perfexpert/)



# PerfExpert Summary

- Load the papi, java, and perfexpert modules:  
`% module load papi java perfexpert`
- Copy the **PerfExpert.sge** submission script (for editing):  
`cp $TACC_PERFEXPERT_DIR/PerfExpert.sge ./`
- Edit the **PerfExpert.sge** script to ensure the correct executable name, correct directory, correct project name and so on.
- Submit your job:  
`% qsub PerfExpert.sge`
- To analyze results:  
`% PerfExpert <threshold> ./hpctoolkit-....`  
Typical value for threshold is 0.1

# **IPM: INTEGRATED PERFORMANCE MONITORING**

# IPM: Integrated Performance Monitoring

- “IPM is a portable profiling infrastructure for parallel codes. It provides a low-overhead performance summary of the computation and communication in a parallel program”
- IPM is a **quick, easy and concise** profiling tool
- The level of detail it reports is smaller than TAU, PAPI or HPCToolkit

# IPM: Integrated Performance Monitoring

- IPM features:
  - easy to use
  - has low overhead
  - is scalable
- Requires **no source code modification**, just adding the “-g” option to the compilation
- Produces XML output that is parsed by scripts to generate browser-readable html pages

# IPM: Integrated Performance Monitoring

- Available on Ranger for both intel and pgi compilers, with mvapich and mvapich2
- Create ipm environment with module command before running code: “`module load ipm`”
- In your job script, set up the following ipm environment before the ibrun command:

```
module load ipm
```

```
export LD_PRELOAD=$TACC_IPM_LIB/libipm.so
```

```
export IPM_REPORT=full
```

```
ibrun <my executable> <my arguments>
```

# IPM: Integrated Performance Monitoring

- `LD_PRELOAD=$TACC_IPM_LIB/libipm.so`
  - must be inside job script
- `IPM_REPORT`: `full`, `terse` or `none` are the levels of information
- `IPM_MPI_THRESHOLD`: Reports only routines using this percentage (or more) of MPI time.
  - e.g. “`IPM_MPI_THRESHOLD 0.3`” report subroutines that consume more than 30% of the total MPI time.
- Important details: “`module help ipm`”
- <http://www.cct.lsu.edu/~yye00>

# IPM: Text Output

```
##IPMv0.922#####  
#  
# command : /work/01125/yye00/ICAC/cactus_SandTank SandTank.par  
# host   : i101-309/x86_64_Linux   mpi_tasks : 32 on 2 nodes  
# start  : 05/26/09/11:49:06      wallclock : 2.758892 sec  
# stop   : 05/26/09/11:49:09      %comm     : 2.01  
# gbytes : 4.38747e+00 total      gflop/sec : 9.39108e-02 total  
#  
#####  
# region : *           [ntasks] =      32  
#  
#           [total]      <avg>         min         max  
# entries      32         1           1           1  
# wallclock    88.2742    2.75857    2.75816    2.75889  
# user         5.51634    0.172386  0.148009  0.200012  
# system       1.771     0.0553438 0.0536683 0.056717  
# %comm        2.00602    1.94539   2.05615  
# gflop/sec    0.0939108 0.00293471 0.00293338 0.002952  
# gbytes       4.38747   0.137109  0.136581  0.144985  
#
```

```
# PAPI_FP_OPS      2.5909e+08 8.09655e+06 8.09289e+06 8.14685e+06  
# PAPI_TOT_CYC     6.80291e+09 2.12591e+08 2.02236e+08 2.19109e+08  
# PAPI_VEC_INS     5.95596e+08 1.86124e+07 1.85964e+07 1.8756e+07  
# PAPI_TOT_INS     4.16377e+09 1.30118e+08 1.0987e+08 1.35676e+08  
#  
#           [time]      [calls]    <%mpi>    <%wall>  
# MPI_Allreduce    0.978938      53248     55.28     1.11  
# MPI_Comm_rank    0.316355      6002     17.86     0.36  
# MPI_Barrier      0.247135      3680     13.95     0.28  
# MPI_Allgather    0.16621       2848     9.39      0.19  
# MPI_Bcast        0.0217298     576      1.23     0.02  
# MPI_Allgather    0.0216982     672      1.23     0.02  
# MPI_Recv         0.0186796     32       1.05     0.02  
# MPI_Comm_size    0.000139921   2112     0.01     0.00  
# MPI_Send         0.000115622   32       0.01     0.00  
#####
```

# IPM: Integrated Performance Monitoring

1469389

- [Load Balance](#)
- [Communication Balance](#)
- [Message Buffer Sizes](#)
- [Communication Topology](#)
- [Switch Traffic](#)
- [Memory Usage](#)
- [Executable Info](#)
- [Host List](#)
- [Environment](#)
- [Developer Info](#)



```
command: /work/01125/yeye00/IPM/Benchmark/Defiant.exe perturb -runperturbed -da_grid_x 50 -da_grid_y 50 -da_grid_z 50 -seed_phi 3454345 -seed_k11 56756756 -seed_k22 235759 -seed_k33 234656 -seed_flowmask 3222111 -percentage_phi 0.1 -percentage_k11 0.1 -percentage_k22 0.1 -percentage_k33 0.1 -percentage_flowmask 0.15 -endtime 1.0 -ksp_type bicg -pc_type bjacobi
```

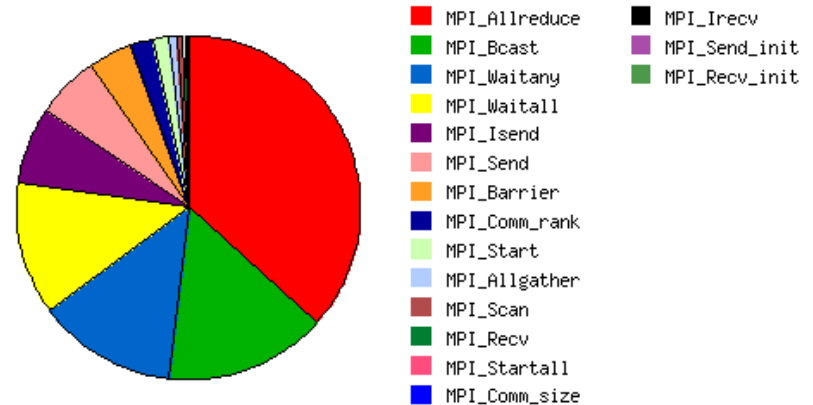
codename:	unknown	state:	running
username:	yeye00	group:	G-801077
host:	i115-108 (x86_64_Linux)	mpi_tasks:	64 on 4 hosts
start:	07/13/10/00:28:10	wallclock:	3.80580e+00 sec
stop:	07/13/10/00:28:13	%comm:	11.5752798360397
total memory:	10.85705 gbytes	total gflop/sec:	1.02192900310053
switch(send):	0 gbytes	switch(recv):	0 gbytes

## Computation

Event	Count	Pop
PAPI_FP_OPS	3889256866	*
PAPI_TOT_CYC	93055641837	*
PAPI_TOT_INS	82058705179	*
PAPI_VEC_INS	8293137711	*

## Communication

### % of MPI Time



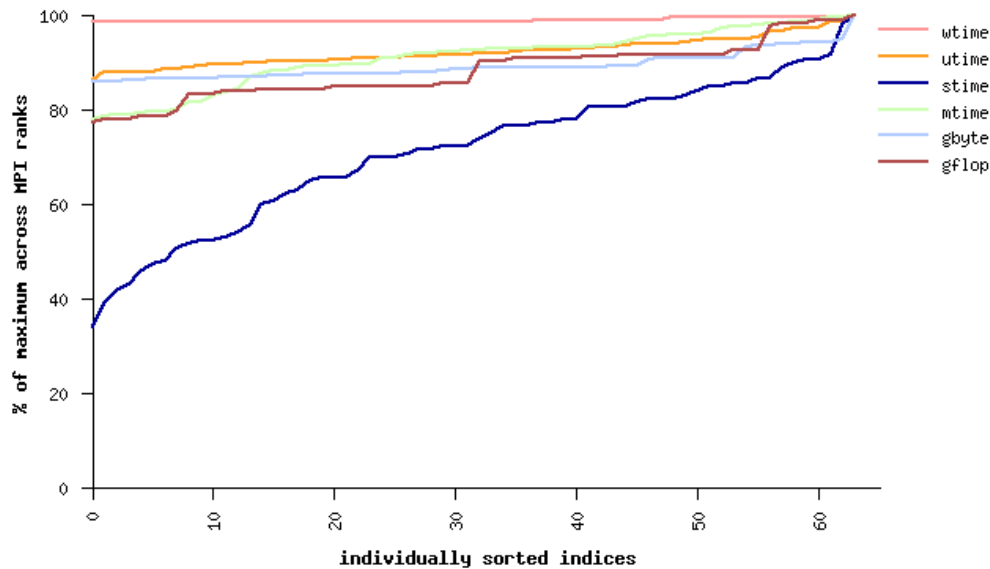
## HPM Counter Statistics

Event	Ntasks	Avg	Min(rank)	Max(rank)
PAPI_FP_OPS	*	60769638.53	53254674 (63)	68822066 (21)
PAPI_TOT_CYC	*	1453994403.70	1346848050 (23)	1646491906 (12)
PAPI_TOT_INS	*	1282167268.42	1134309464 (0)	1477795580 (12)
PAPI_VEC_INS	*	129580276.73	113002002 (63)	146915653 (21)

# IPM: Event Statistics

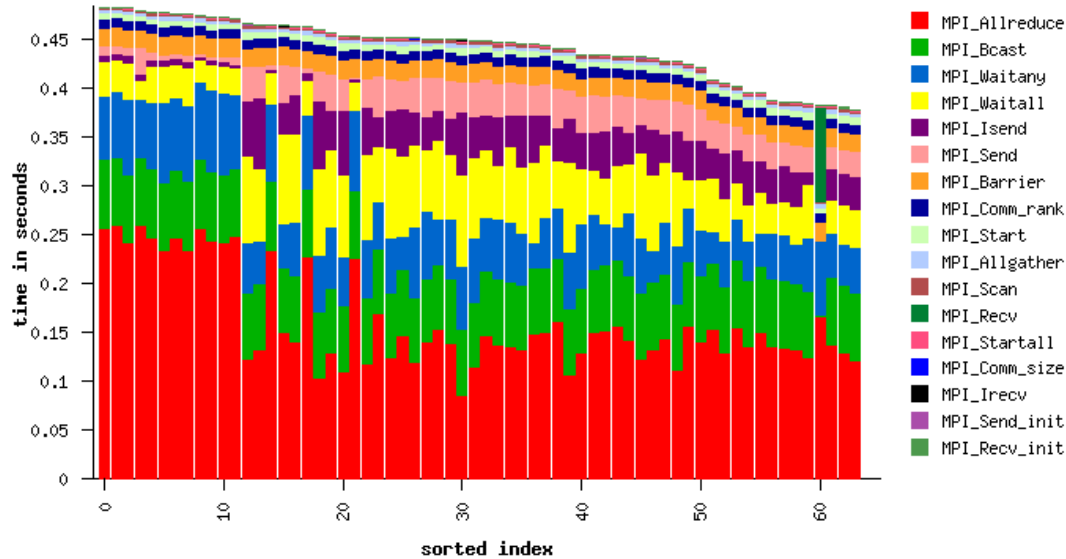
Communication Event Statistics (100.00% detail, 9.9012e-06 error)									
	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall		
MPI_Allreduce		8	79680	4.178	8.225e-06	8.882e-04	14.82		1.72
MPI_Bcast		4	1024	4.047	5.914e-08	6.413e-02	14.35		1.66
MPI_Allreduce		512	39936	3.803	1.660e-05	1.170e-01	13.49		1.56
MPI_Allreduce		4	25472	2.250	6.012e-07	1.552e-02	7.98		0.92
MPI_Barrier		0	64	1.176	1.814e-02	1.865e-02	4.17		0.48
MPI_Isend		8	630	1.028	3.427e-07	1.647e-02	3.65		0.42
MPI_Isend		4	4556	0.943	2.738e-07	1.833e-02	3.34		0.39
MPI_Send		14976	144	0.722	1.030e-03	7.308e-03	2.56		0.30
MPI_Comm_rank		0	106948	0.620	3.725e-08	9.872e-03	2.20		0.25
MPI_Waitany		0	6093	0.542	4.615e-07	1.358e-02	1.92		0.22
MPI_Waitany		1248	27462	0.519	5.183e-07	2.723e-04	1.84		0.21
MPI_Send		16224	144	0.517	4.283e-04	7.129e-03	1.83		0.21
MPI_Waitany		1352	20370	0.496	5.197e-07	5.783e-03	1.76		0.20
MPI_Start		0	269196	0.396	3.623e-07	3.685e-05	1.40		0.16
MPI_Send		13824	48	0.298	4.035e-03	7.227e-03	1.06		0.12
MPI_Waitany		1152	10980	0.243	5.383e-07	2.310e-04	0.86		0.10
MPI_Bcast		216	576	0.231	2.302e-06	5.843e-03	0.82		0.09
MPI_Allgather		4	9088	0.215	5.118e-07	1.793e-03	0.76		0.09
MPI_Waitall		184	11	0.210	1.633e-02	2.135e-02	0.74		0.09
MPI_Scan		4	384	0.144	2.259e-05	1.600e-03	0.51		0.06
MPI_Waitany		147	453	0.141	4.866e-07	1.406e-02	0.50		0.06
MPI_Waitany		4	448	0.132	4.345e-07	5.805e-03	0.47		0.05
MPI_Waitall		320	18	0.120	3.002e-06	1.284e-02	0.42		0.05
MPI_Send		17576	42	0.108	6.682e-05	6.406e-03	0.38		0.04
MPI_Waitall		72	6	0.103	1.547e-02	2.038e-02	0.36		0.04
MPI_Waitall		96	38	0.091	2.882e-06	1.563e-02	0.32		0.04
MPI_Waitany		624	140	0.088	1.126e-06	7.880e-03	0.31		0.04
MPI_Recv		8	9	0.085	8.373e-07	7.696e-02	0.30		0.03

Load balance by task: memory, flops, timings



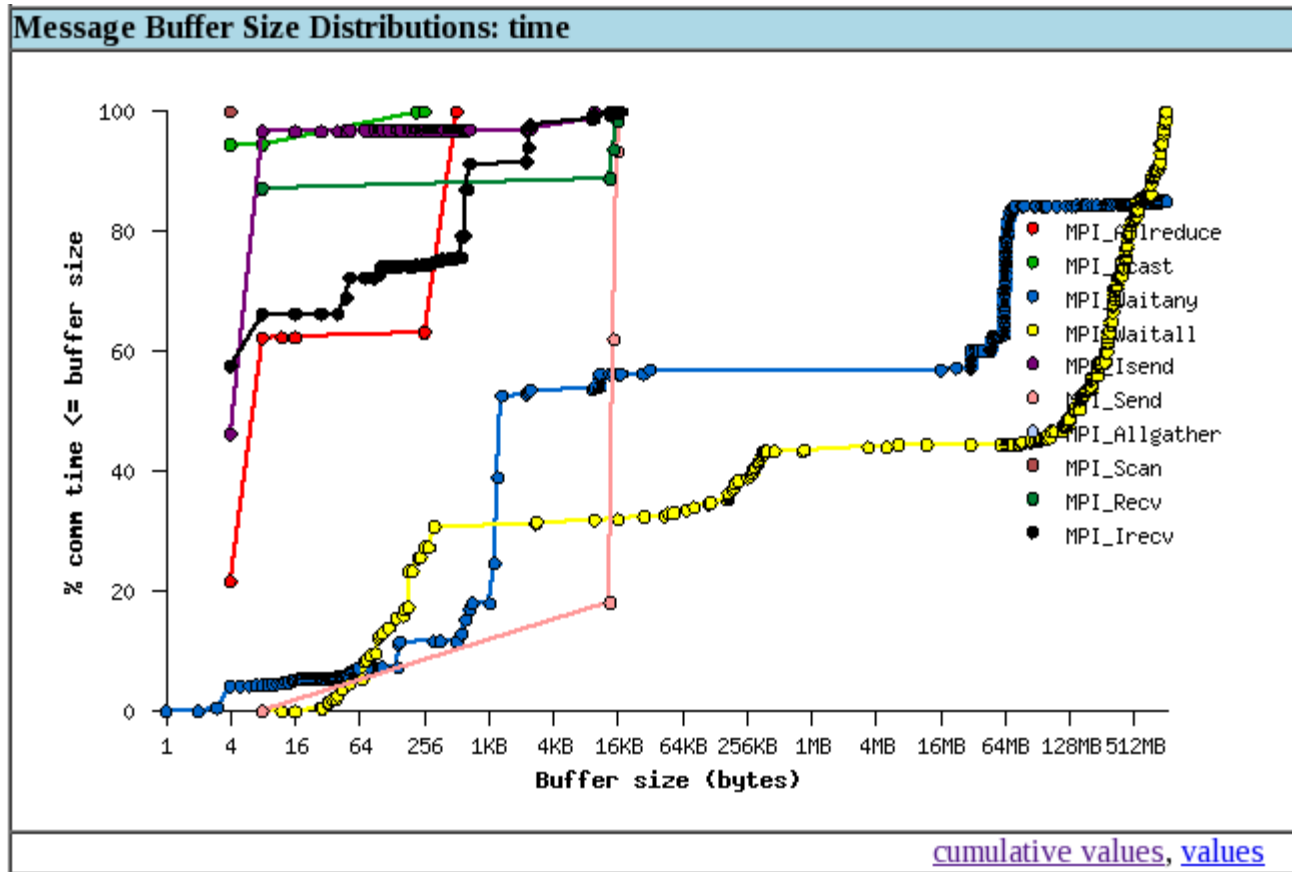
[by MPI rank, by MPI time](#)

Communication balance by task (sorted by MPI time)

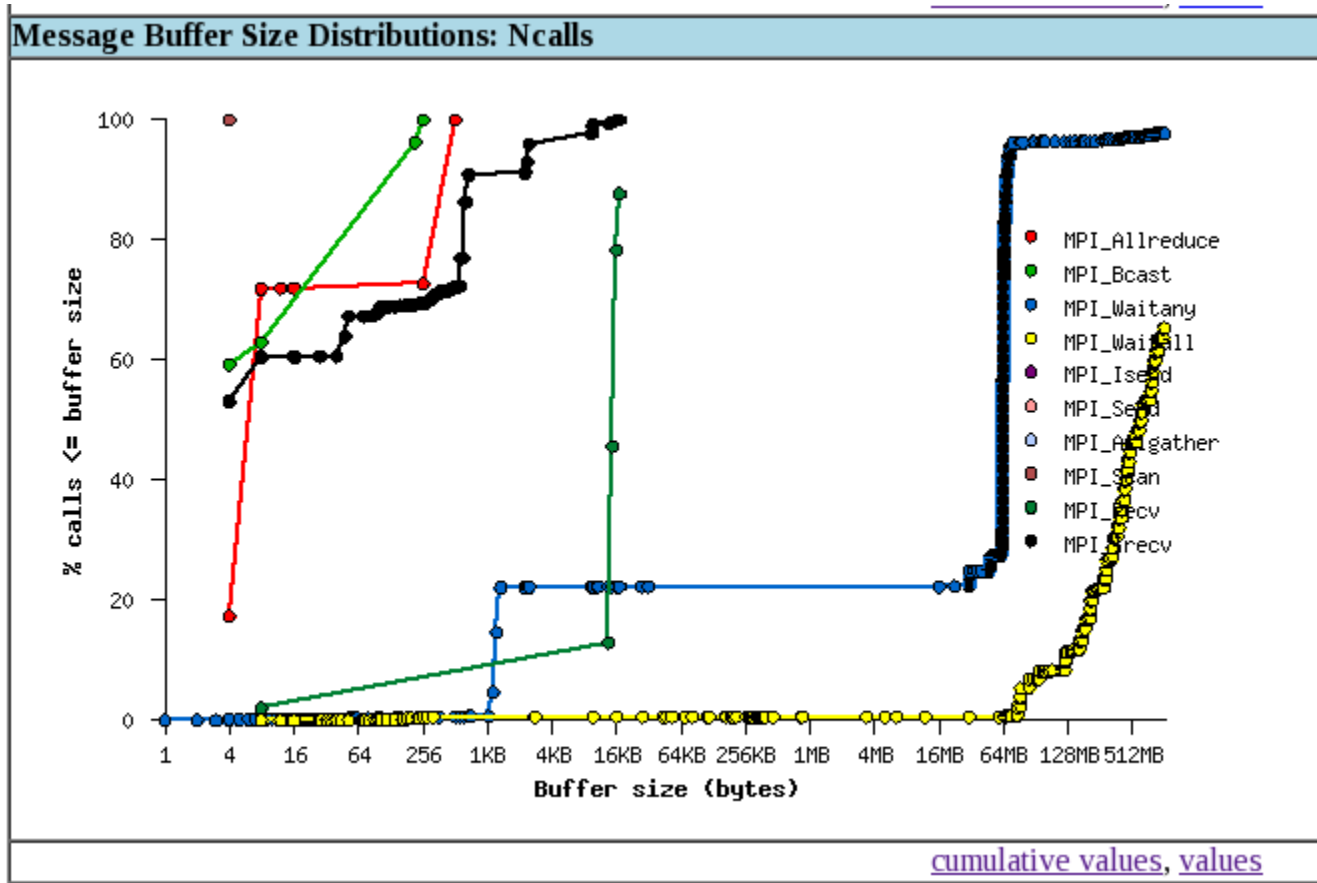


[by MPI rank, time detail by MPI time, time detail by rank, call list](#)

# IPM Buffer Size Distribution: % of Comm Time

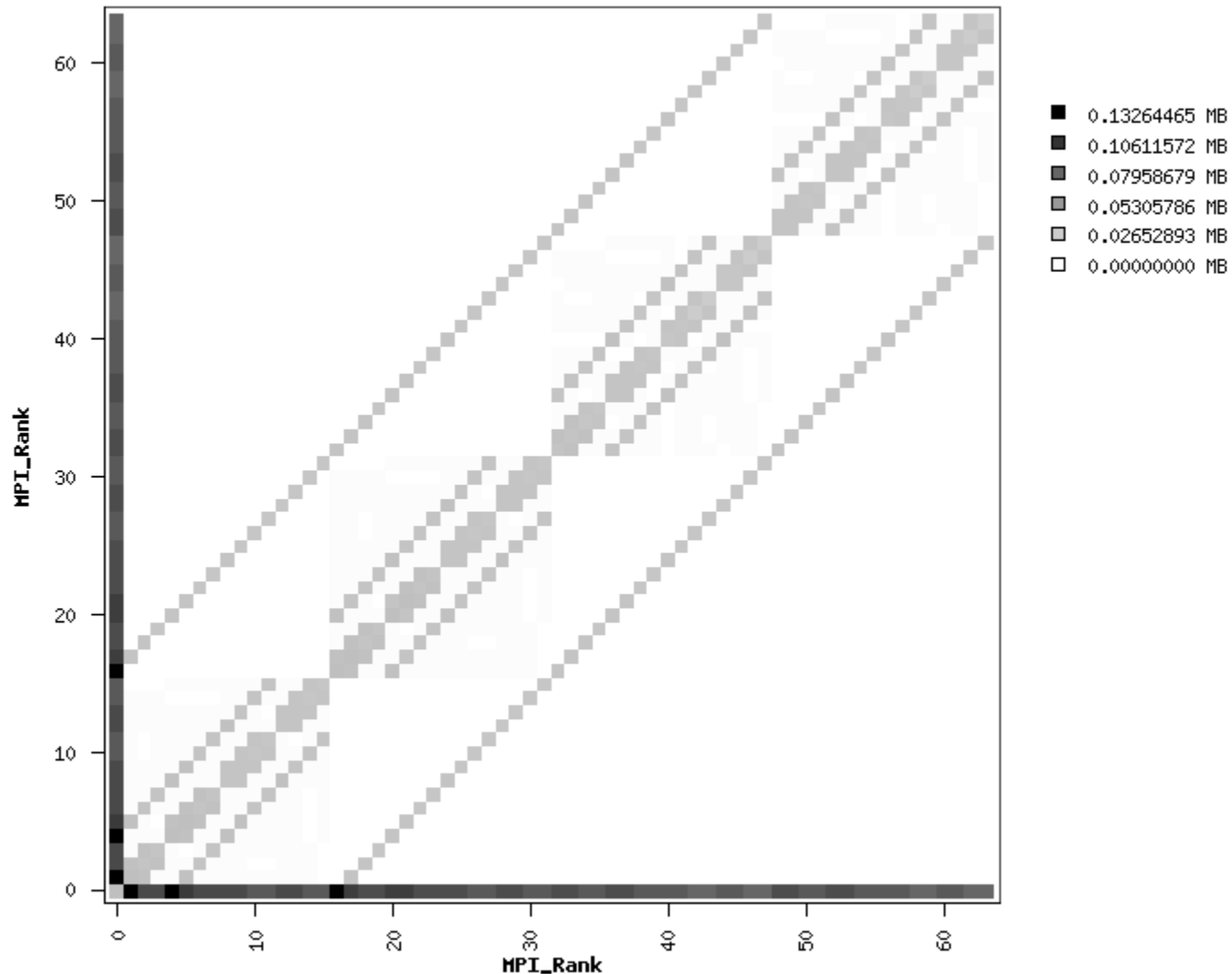


# Buffer Size Distribution: Ncalls



# Communication Topology

Communication Topology : point to point data flow



# IPM: Integrated Performance Monitoring

- When to use IPM?
  - To quickly find out **where your code is spending most of its time** (in both computation and communication)
  - For performing **scaling studies** (both strong and weak)
  - When you suspect you have **load imbalance** and want to verify it quickly
  - For a quick look at the **communication pattern**
  - To find out how much **memory** you are using **per task**
  - To find the **relative communication & compute time**

# IPM: Integrated Performance Monitoring

- When IPM is NOT the answer
  - When you already know where the performance issues are
  - When you need detailed performance information on exact lines of code
  - When want to find specific information such as cache misses

# Advanced Profiling Tools

: the next level

# Advanced Profiling Tools

- Can be intimidating:
  - Difficult to install
  - Many dependences
  - Require kernel patches } Not your problem!!
- Useful for serial and parallel programs
- Extensive profiling and scalability information
- Analyze code using:
  - Timers
  - Hardware registers (PAPI)
  - Function wrappers

# PAPI

PAPI is a Performance Application Programming Interface

[icl.cs.utk.edu/papi](http://icl.cs.utk.edu/papi)

- API to use hardware counters
- Behind Tau, HPCToolkit
- Multiplatform:
  - Most Intel & AMD chips
  - IBM POWER 4/5/6
  - Cray X/XD/XT
  - Sun UltraSparc I/II/III
  - MIPS
  - SiCortex
  - Cell
- Available as a module in Ranger

# About Tau

TAU is a suite of **T**uning and **A**nalysis **U**tilities

[www.cs.uoregon.edu/research/tau](http://www.cs.uoregon.edu/research/tau)

- 11+ year project involving
  - University of Oregon Performance Research Lab
  - LANL Advanced Computing Laboratory
  - Research Centre Julich at ZAM, Germany
- Integrated toolkit
  - Performance instrumentation
  - Measurement
  - Analysis
  - Visualization

# Using Tau

- Load the **papi** and **tau** modules
- Gather information for the profile run:
  - Type of run (profiling/tracing, hardware counters, etc...)
  - Programming Paradigm (MPI/OMP)
  - Compiler (Intel/PGI/GCC...)
- Select the appropriate **TAU\_MAKEFILE** based on your choices (**\$TACC\_TAU\_LIB/Makefile.\***)
- Set up the selected PAPI counters in your submission script
- Run as usual & analyze using **paraprof**
  - You can transfer the database to your own PC to do the analysis

# Tau: Example

Load the **papi** and **tau** modules:

```
% module load papi  
% module load tau
```

Say that we choose to do

- a profiling run with **multiple counters** for a
- **MPI** parallel code and use
- the **PDT instrumentator** with
- the **PGI compiler**

The **TAU\_MAKEFILE** to use for this combination is:

```
$TACC_TAU_LIB/Makefile.tau-multiplecounters-mpi-papi-pdt-pgi
```

So we set it up:

```
% setenv TAU_MAKEFILE $TACC_TAU_LIB/Makefile.tau-multiplecounters-mpi-papi-pdt-pgi
```

And we compile using the wrapper provided by **tau**:

```
% tau_cc.sh matmult.c
```

# Tau: Example (Cont.)

Next we decide which hardware counters to use:

- **GET\_TIME\_OF\_DAY** (time, profiling, similar to using **gprof**)
- **PAPI\_FP\_OPS** (Floating Point Operations Per Second)
- **PAPI\_L1\_DCM** (Data Cache Misses for the cache Level 1)

We set these as environmental variables in the command line or the submission script.

For csh:

```
% setenv COUNTER1 GET_TIME_OF_DAY  
% setenv COUNTER2 PAPI_FP_OPS  
% setenv COUNTER3 PAPI_L1_DCM
```

For bash:

```
% export COUNTER1 = GET_TIME_OF_DAY  
% export COUNTER2 = PAPI_FP_OPS  
% export COUNTER3 = PAPI_L1_DCM
```

The we send the job through the queue as usual.

# Tau: Example (Cont.)

When the program finishes running one new directory will be created for each hardware counter we specified:

- **MULTI\_GET\_TIME\_OF\_DAY**
- **MULTI\_PAPI\_FP\_OPS**
- **MULTI\_PAPI\_L1\_DCM**

Analyze the results with **paraprof**:

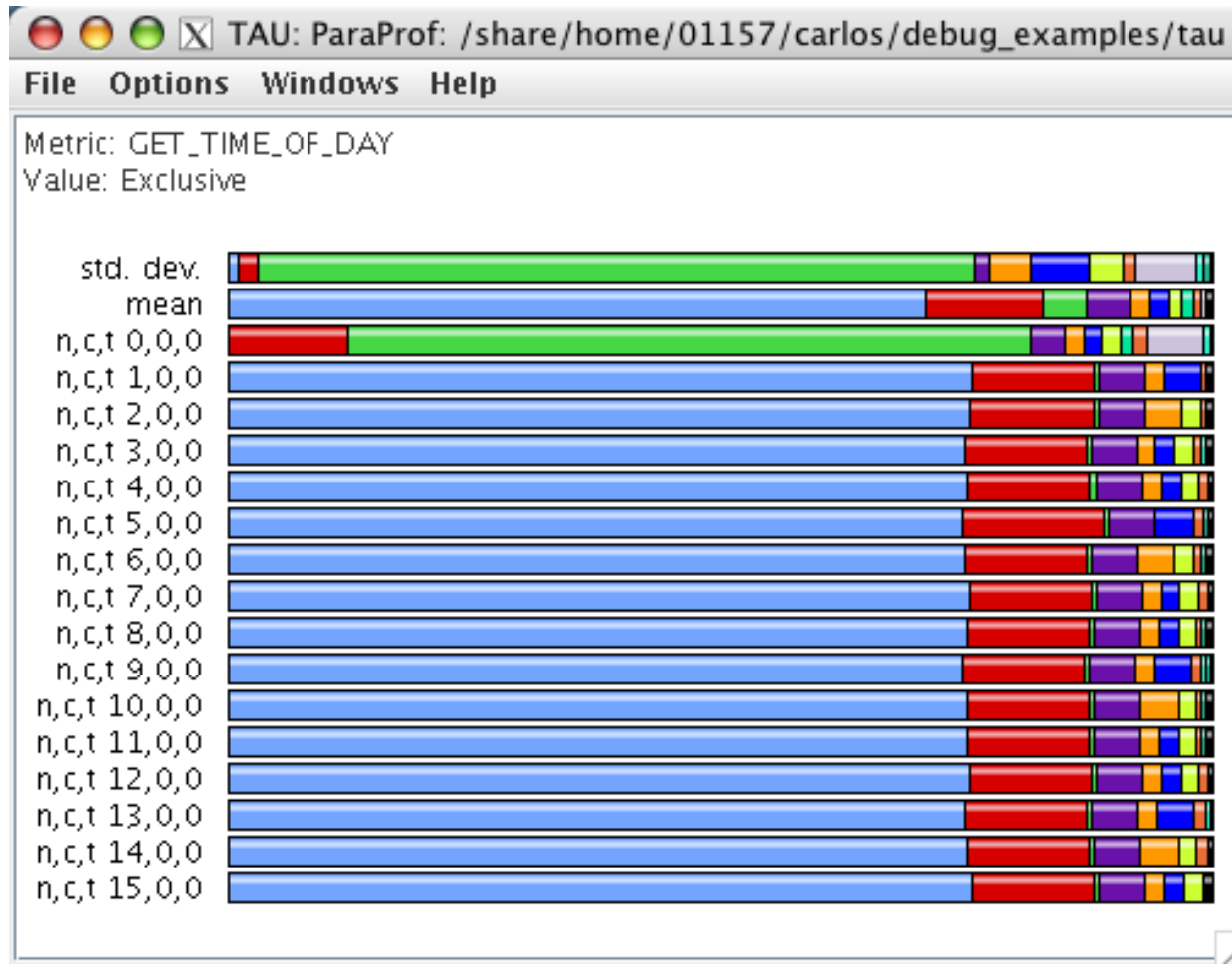
**% paraprof**

# TAU: ParaProf Manager

Counters we asked for

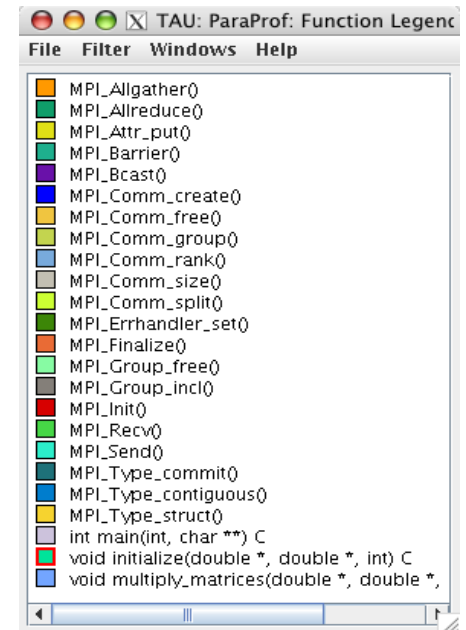
TrialField	Value
Name	tau/debug_examples/carlos/01157/h...
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	4
CPU MHz	2293.908
CPU Type	Quad-Core AMD Opteron(tm) Processo...
CPU Vendor	AuthenticAMD
CWD	/share/home/01157/carlos/debug_ex...
Cache Size	512 KB
Executable	/share/home/01157/carlos/debug_ex...
Hostname	i115-101.ranger.tacc.utexas.edu
Local Time	2009-03-18T14:16:48-05:00
MPI Processor Name	i115-101.ranger.tacc.utexas.edu
Memory Size	32878720 kB
Node Name	i115-101.ranger.tacc.utexas.edu
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.18.8.TACC.lustre.perfctr
OS Version	#4 SMP Tue Jul 22 07:16:12 CDT 2008
Starting Timestamp	1237403803413938
TAU Architecture	x86_64
TAU Config	-prefix = /opt/apps/pgi7_2/mvapich1_...
TAU Version	2.17
Timestamp	1237403808551516
UTC Time	2009-03-18T19:16:48Z
pid	12107
username	carlos

# Tau: Metric View



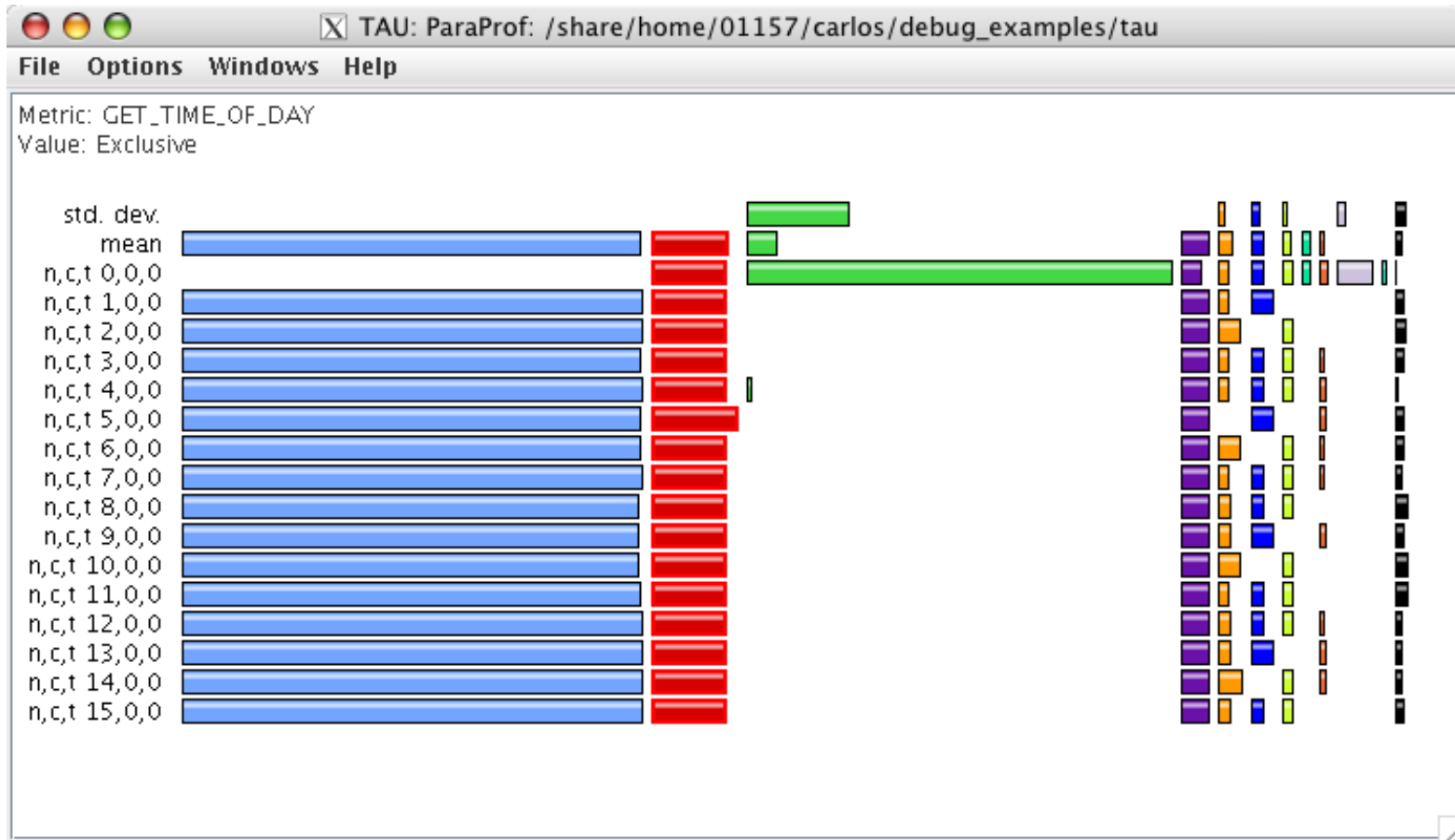
Information includes Mean and Standard Deviation

Windows->Function Legend



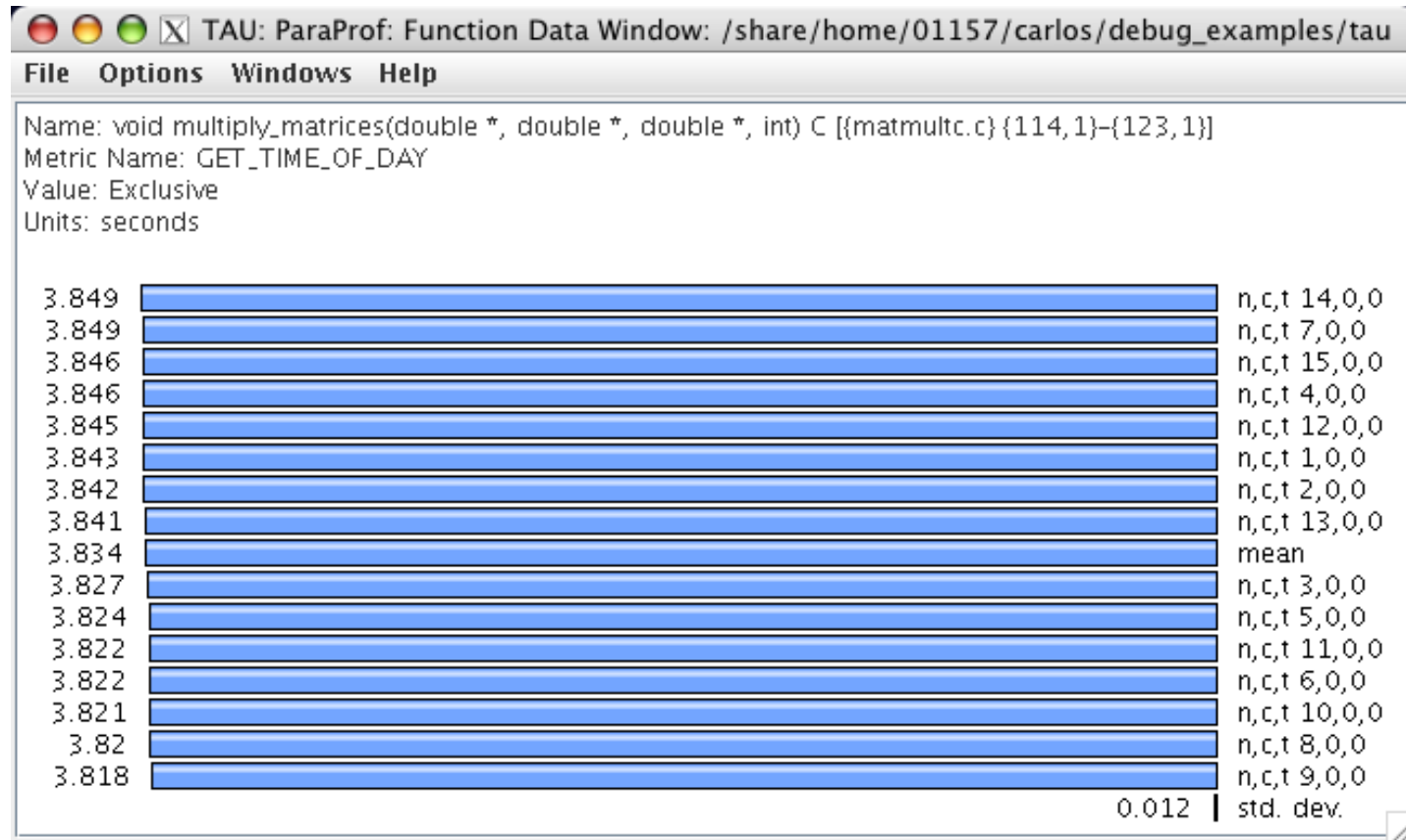
# Tau: Metric View

Unstack the bars for clarity: Options -> Stack Bars Together



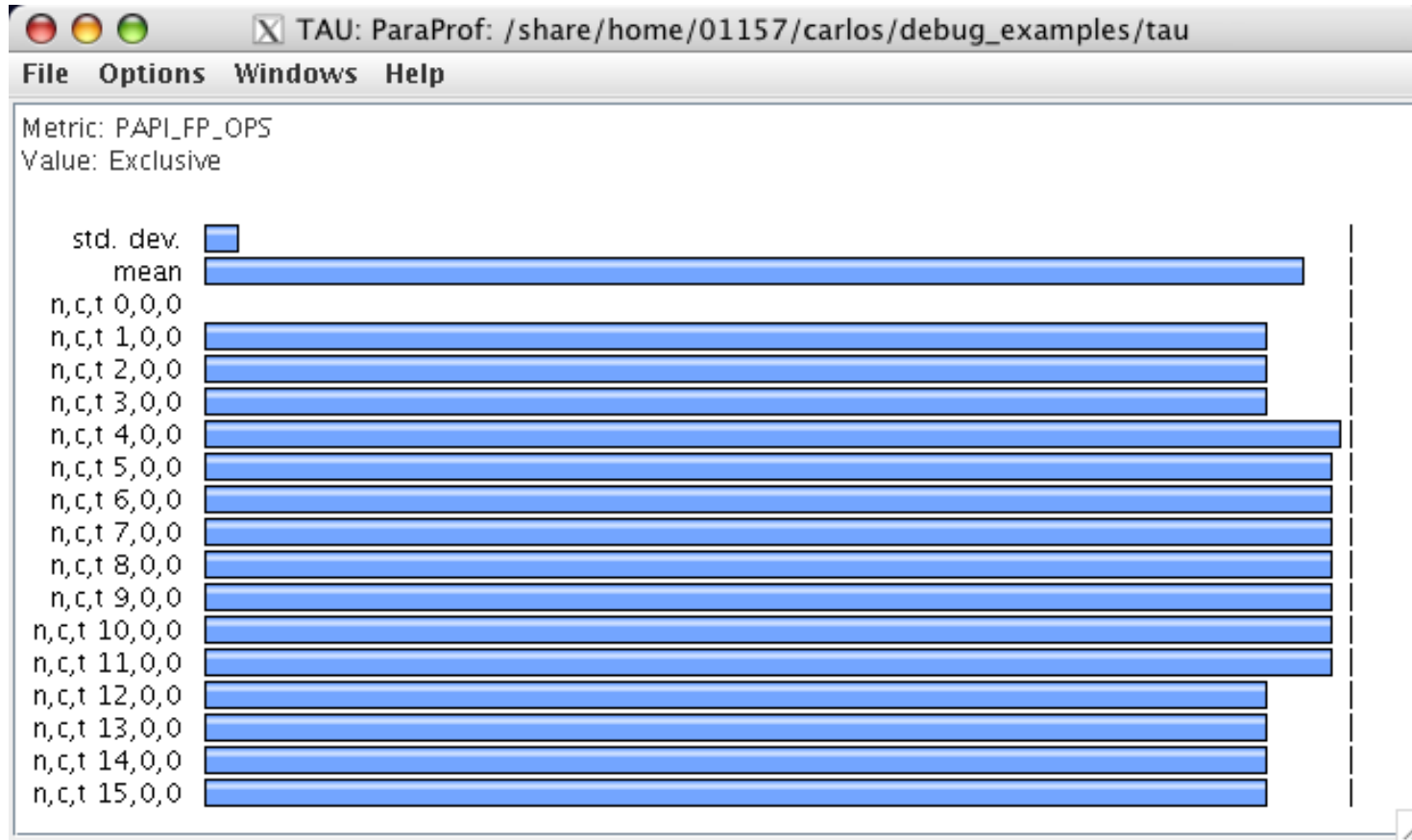
# Tau: Function Data Window

Click on any of the bars corresponding to function `multiply_matrices`. This opens the Function Data Window, which gives a closer look at a single function.

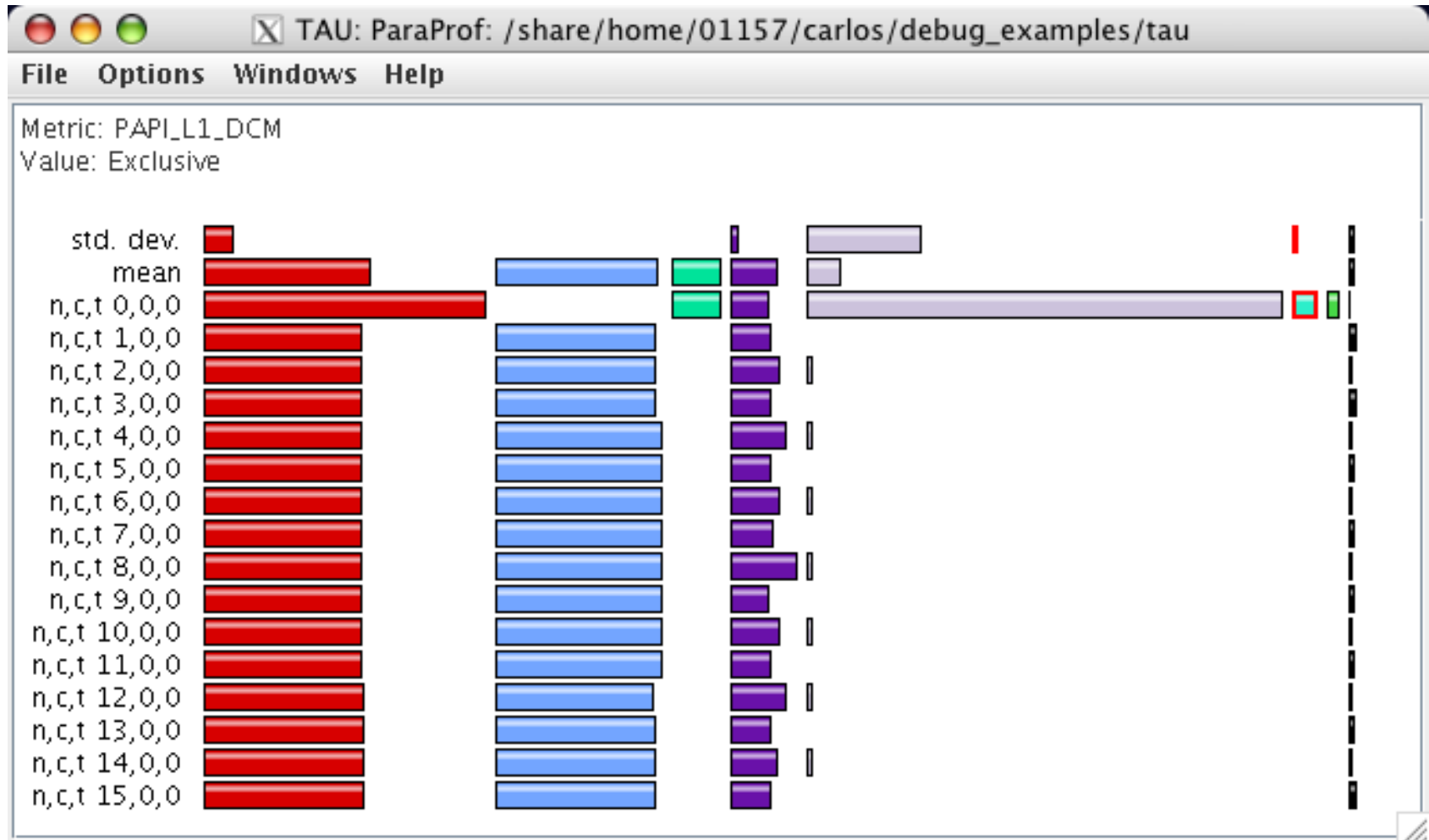


# Tau: Float Point OPS

In the ParaProf Metric Window go to Options -> Select Metric -> Exclusive

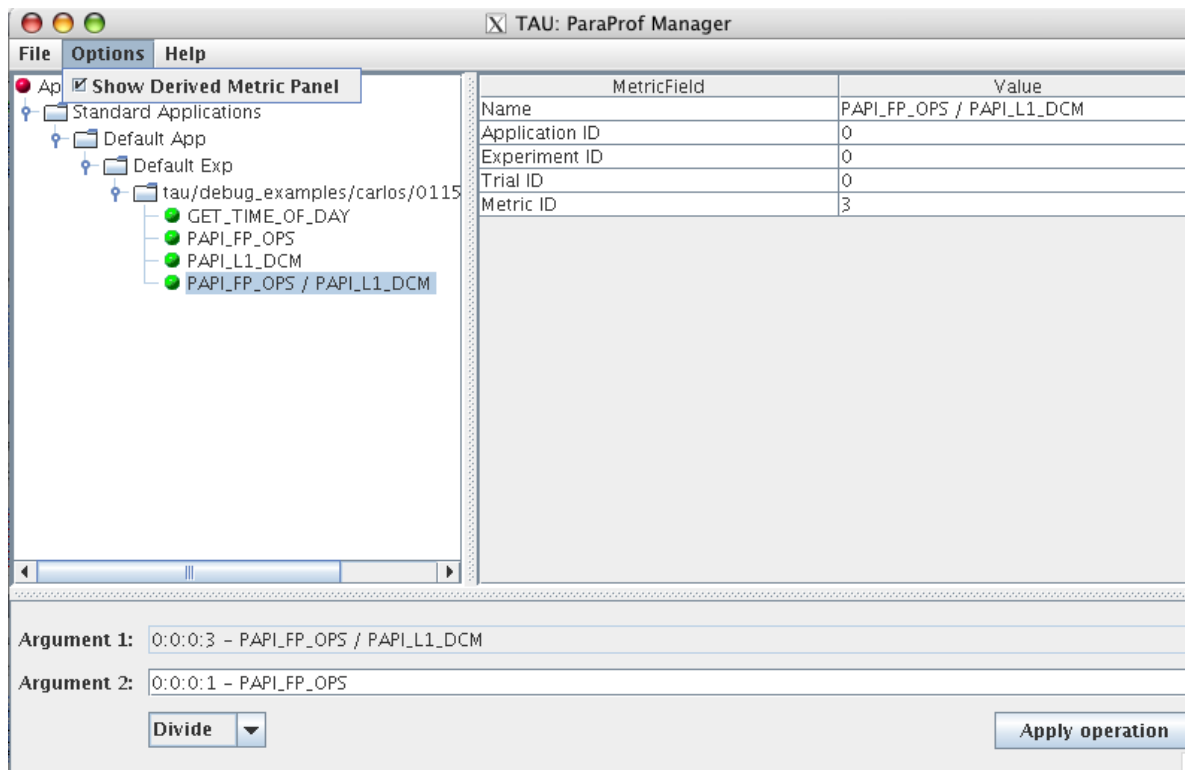


# Tau: L1 Cache Data Misses



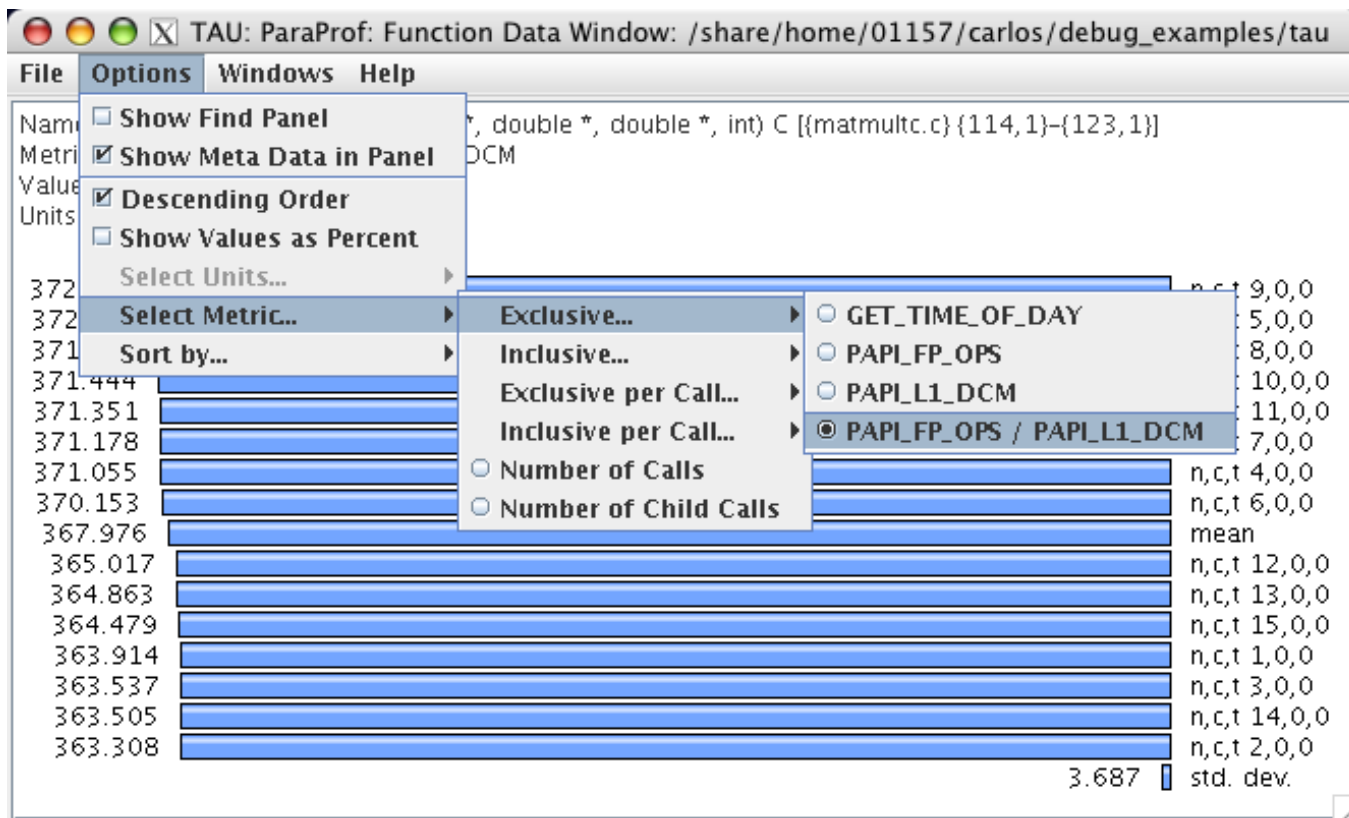
# Derived Metrics

- ParaProf Manager Window -> Options -> Show Derived Metrics Panel
- Select Argument 1 (PAPI\_L1\_DCM) and Argument 2 (PAPI\_FP\_OPS)
- Select Operation (Division) & Apply



# Derived Metrics (Cont.)

- Select a Function
- Function Data Window -> Options -> Select Metric -> Exclusive -> ...



# Callgraph

To find out about function calls within the program, follow the same process but using the following **TAU\_MAKEFILE**:

**Makefile.tau-callpath-mpi-pdt-pgi**

In the Metric View Window two new options will be available under:

Windows → Thread → Call Graph

Windows → Thread → Call Path Relations

# Profiling dos and don'ts

## DO

- Test every change you make
- Profile typical cases
- Compile with optimization flags
- Test for scalability (coming up next)

## DO NOT

- Assume a change will be an improvement
- Profile atypical cases
- Profile *ad infinitum*
  - Set yourself a goal or
  - Set yourself a time limit

# Other tools

- Valgrind\* [valgrind.org](http://valgrind.org)
  - Powerful instrumentation framework, often used for debugging memory problems
- MPIP [mpip.sourceforge.net](http://mpip.sourceforge.net)
  - Lightweight, scalable MPI profiling tool
- Tau [www.cs.uoregon.edu/research/tau](http://www.cs.uoregon.edu/research/tau)
  - Suite of **T**uning and **A**nalysis **U**tilities
- Scalasca [www.fz-juelich.de/jsc/scalasca](http://www.fz-juelich.de/jsc/scalasca)
  - Similar to Tau, complete suit of tuning and analysis tools.
- HPCToolkit [www.hpctoolkit.org](http://www.hpctoolkit.org)
  - Interesting tool with a lot of promise