Software Profiling with TAU

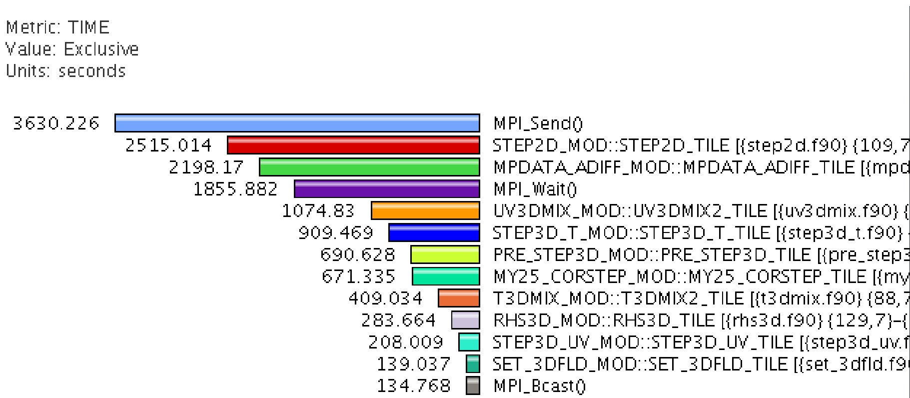
Prasad Maddumage
mhemantha@fsu.edu
Research Computing Center
Florida State University

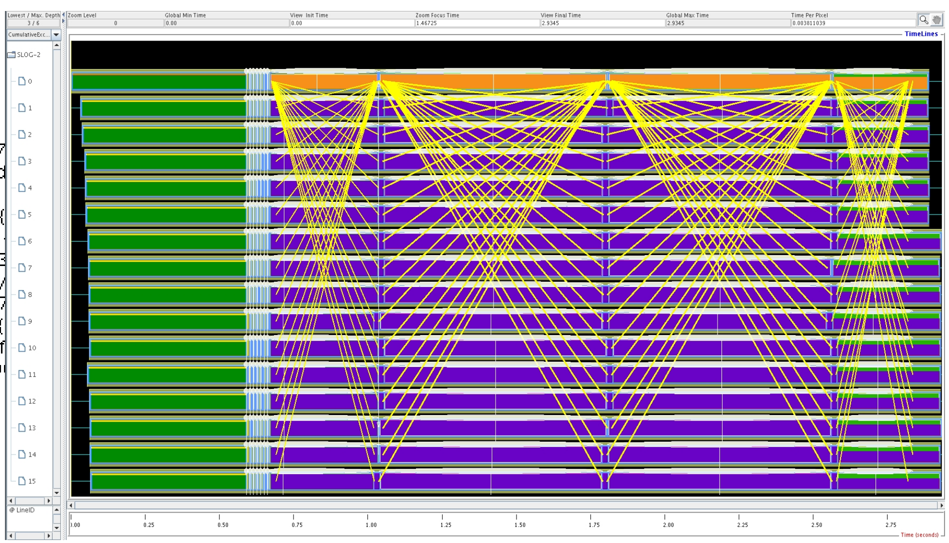
Software Profiling

- Dynamic program analysis using various measures related to code execution
  - CPU/memory utilization, frequency of function calls, I/O, MPI library usage, hardware counters, etc.
- Profilers *instrument* source or binary to obtain such measures during runtime
  - Instrumenting is inserting probes and replacing or wrapping function calls (eg: MPI calls, I/O) with modified calls of a source code
- Analyzing the results will help programmers/ scientists to improve code performance

Profile vs Trace

- Profile: statistical summary of all metrics measured
  - Shows how much total time/resources each call utilized
- Trace: timeline of runtime events took place
  - Shows when each event happened and where

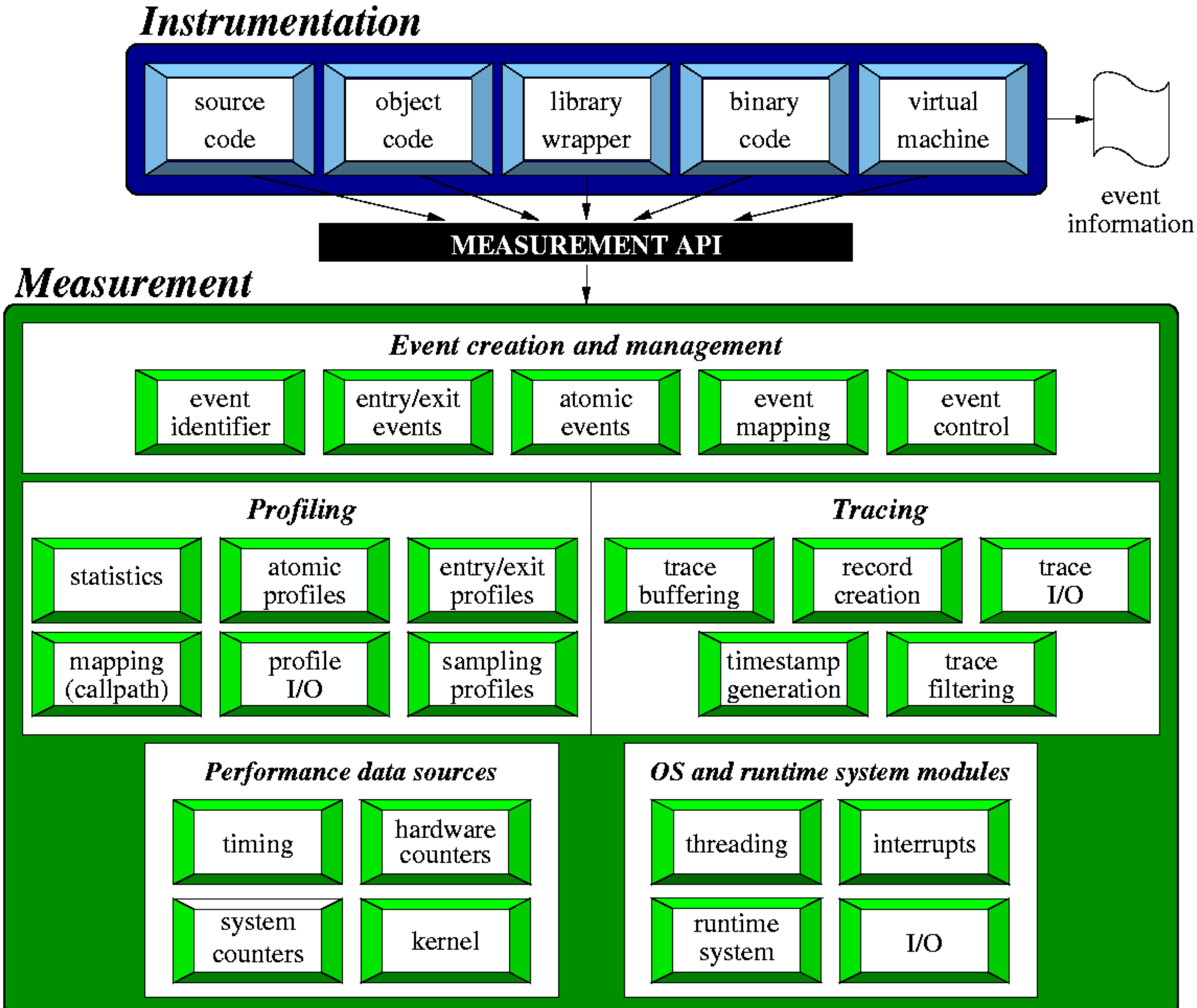




Why use TAU?

- Tuning and Analysis Utilities (20+ year project)
  - Actively developed by Univ. of Oregon, ANL, LANL, Julich
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Performance data management and data mining
  - Open source
- Easy to integrate in application frameworks
- Well documented

How does TAU work?



How does TAU work?

- Instrumentation
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable
- Measurement
  - Direct: interval events, Indirect: collect samples to profile statement execution
  - Per-thread storage of performance data
  - Throttling and runtime control of low-level events

How does TAU work?

- Analysis
  - TAU creates one profile file per node in a single location
  - Profile file names look like,

    `profile.0.0.0, profile.1.0.0, ...`
  - 2D and 3D visualization of profile data using pprof and **`paraprof`**
  - Trace conversion & display in external visualizers such as **`Jumpshot`**

TAU Event Types

- Interval: start-stop events (eg: function call)
- Atomic: trigger at a single point with data (eg: memory allocation)
  - Measures total, samples, min/max/mean/std. deviation statistics
- Context: atomic events with executing context
  - Measures total, samples, min/max/mean/std. deviation statistics

TAU event types

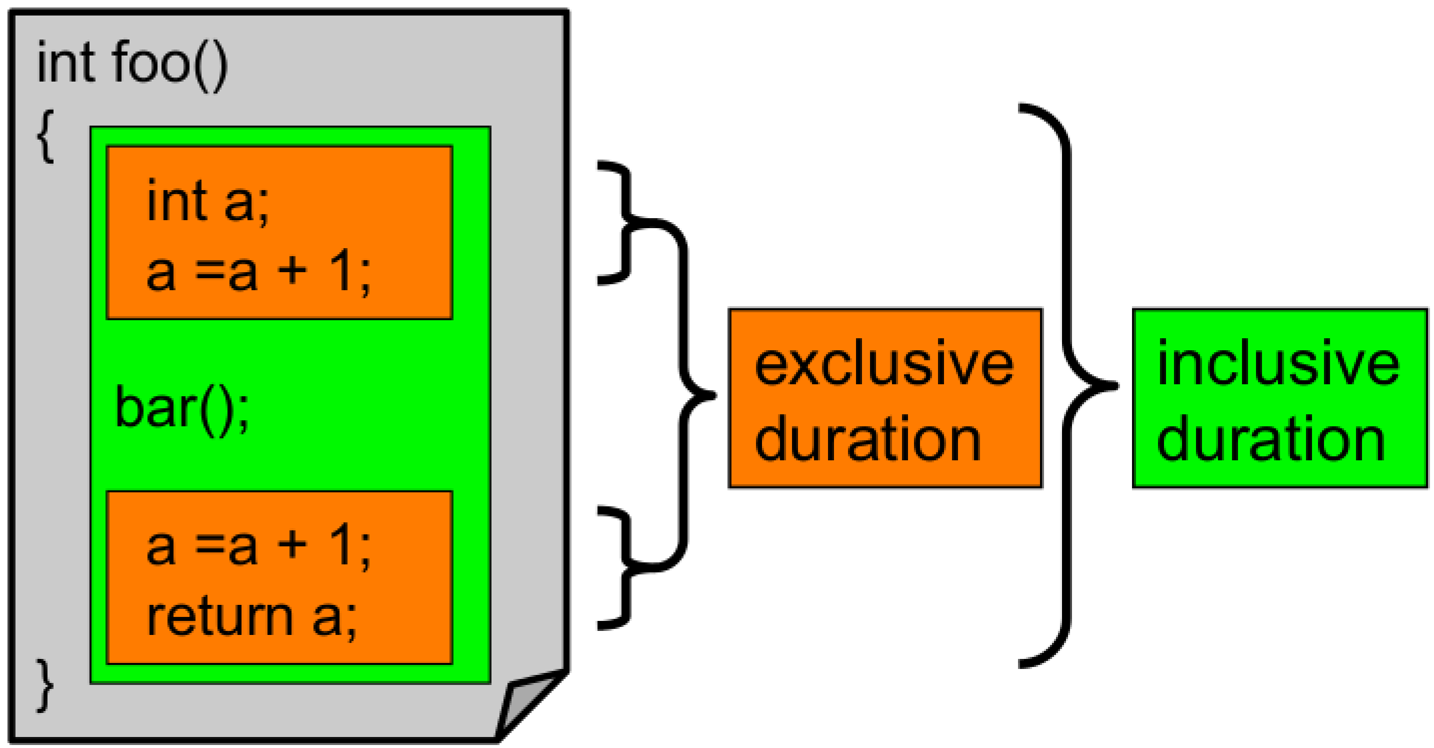
```
profile.0.0.0
---------------------------------------------------------------------
%Time Exclusive   Inclusive #Call #Subrs  Inclusive     Name
           msec  total msec                usec/call
---------------------------------------------------------------------
100.0  1:18.355  1:18.561       1    1818   78561006   .TAU application
  0.3       202       202    1814       0        112      read()
  0.0         3         3       2       0       1607      open()
  0.0     0.004     0.004       2       0          2      lseek()
---------------------------------------------------------------------

USER EVENTS: profile.0.0.0
---------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------
1812    8192    2174     8186    179.6  Bytes Read
1812    8192    2174     8186    179.6  Bytes Read : .TAU application => read()
906     8192    2174     8185    199.8  Bytes Read <file=data1.dat> :
                                               .TAU application => read()
906     8192    3467     8187    156.9  Bytes Read <file=data2.dat> :
                                               .TAU application => read()
1812    1170   0.113    913.9    124.7  Read Bandwidth (MB/s)
```

Interval events

Atomic event

Context events

Exclusive vs Inclusive time



TAU at RCC

- Currently available on HPC for all six compilers
  - GNU, Intel, PGI - OpenMPI and MVAPICH2
- To use serial version

```
module load tau-serial
```

- To use parallel version

```
module load tau
```

- Documentation: https://rcc.fsu.edu/software/tau

TAU Instrumentation

- Library interposition (dynamic instrumentation)
  - No need to recompile your code
  - `mpirun -np 4 tau_exec <options> <your binary>`
  - Can profile MPI (default), memory use, I/O, …
  - Cannot track user functions

Still buggy!

```
$ gfortran -o gauss gauss.f90
$ module load tau-serial
$ tau_exec -T serial -io ./gauss
$ pprof -s
Reading Profile files in profile.0.0.0.*
FUNCTION SUMMARY (total):
-------------------------------------------------------------
%Time Exclusive  Inclusive   #Call #Subrs Inclusive     Name
           Msec  total msec                usec/call
-------------------------------------------------------------
100.0  1:16.607  1:16.627        1   1818  76627147   .TAU application
  0.0        19        19     1814       0        11  read()
  0.0     0.026     0.026        2      0        13   open()
  0.0     0.001     0.001        2      0         0   lseek()
```

TAU Instrumentation

- Scripted Compilation
  - Use tau_f90.sh, tau_cc.sh, and tau_cxx.sh to instrument and compile Fortran, C, and C++ programs
  - Compiler Based Instrumentation
    - Use the compiler itself for instrumenting
    - Provides more detailed profiles than dynamic approach
    - Cannot profile user functions
    - Needs recompilation of the code

```
$ tau_cc.sh –tau_options=-optCompInst samplecprogram.c
```

TAU Instrumentation

- Source based instrumentation
  - Uses PDT (Program Database Toolkit) to fully instrument the source code
  - Able to generate complete profiles by measuring low level events (loops, hardware counters, etc.)
  - Needs recompilation of the code (Simply switch **`CC`** or **`FC`** with **`tau_cc.sh`** or **`tau_f90.sh`**)

```
$ module load gnu-openmpi
$ module load tau
$ tau_f90.sh -o mat_mul_par mat_mul_par.f90
$ msub mat_mul_par.sh
$ pprof
```

pprof

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time   Exclusive Inclusive #Call #Subrs  Inclusive  Name
             msec total msec                usec/call
---------------------------------------------------------------------------------------
100.0    0.139      55,725    1       1    55725516 .TAU application
100.0   26,947      55,725    1       7    55725377 MAT_MUL_PAR
 49.5   27,590      27,590    1       0    27590687 MPI_Gather()
  1.0      541         541    1       0      541913 MPI_Init()
  0.9      488         488    1       0      488823 MPI_Bcast()
  0.2       94          94    1       0       94155 MPI_Scatter()
  0.1       62          62    1       0       62281 MPI_Finalize()
  0.0    0.001       0.001    1       0           1 MPI_Comm_size()
  0.0        0           0    1       0           0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
1        6.4E+07      6.4E+07    6.4E+07          0  Message size for broadcast
1          4E+06        4E+06      4E+06          0  Message size for gather
1          4E+06        4E+06      4E+06          0  Message size for scatter
---------------------------------------------------------------------------------------
```

paraprof

Poor
load balancing!

How much time
spent on
each operation

Only MPI calls
are profiled

Source based Instrumentation

- There is more...
  - The TAU module picks a “Makefile” for you, depending on the compiler you are using
    - It is stored in the variable **`TAU_MAKEFILE`**
    - Eg: Default for gnu-openmpi is **`Makefile.tau-papi-mpi-pdt`**
    - Makefiles can be changed by user depending on the purpose

```
Makefile.tau-communicators-papi-mpi-pdt
Makefile.tau-headroom-papi-mpi-pdt
Makefile.tau-memory-papi-mpi-pdt
Makefile.tau-papi-mpi-pdt
Makefile.tau-papi-mpi-pdt-trace
Makefile.tau-phase-papi-mpi-pdt
```

Using Different Makefiles

```
---------------------------------------------------------------------------------------
%Time   Exclusive   Inclusive   #Call   #Subrs   Inclusive Name
             msec total msec                      usec/call
---------------------------------------------------------------------------------------
100.0       0.012    1:06.231       1        1    66231389 .TAU application
100.0       2,021    1:06.231       1        1    66231377 GAUSS
 96.9      19,828    1:04.209       1     6000    64209853 GAUSSJ
 45.8      30,348      30,348    4000        0        7587 OUTERPROD
 21.2      14,023      14,023    1000        0       14023 OUTERAND
  0.0           9           9    1000        0          10 SWAP
---------------------------------------------------------------------------------------
USER EVENTS: profile.-1.0.0
---------------------------------------------------------------------------------------
NumSamples   MaxValue MinValue  MeanValue Std. Dev.  Event Name
---------------------------------------------------------------------------------------
1       0         0         0         0  .TAU application - Heap Memory Used (KB)
1       0         0         0         0  GAUSS - Heap Memory Used (KB)
1       0         0         0         0  GAUSSJ - Heap Memory Used (KB)
1000    0         0         0         0  OUTERAND - Heap Memory Used (KB)
4000    0         0         0         0  OUTERPROD - Heap Memory Used (KB)
1000    0         0         0         0  SWAP - Heap Memory Used (KB)
---------------------------------------------------------------------------------------
```

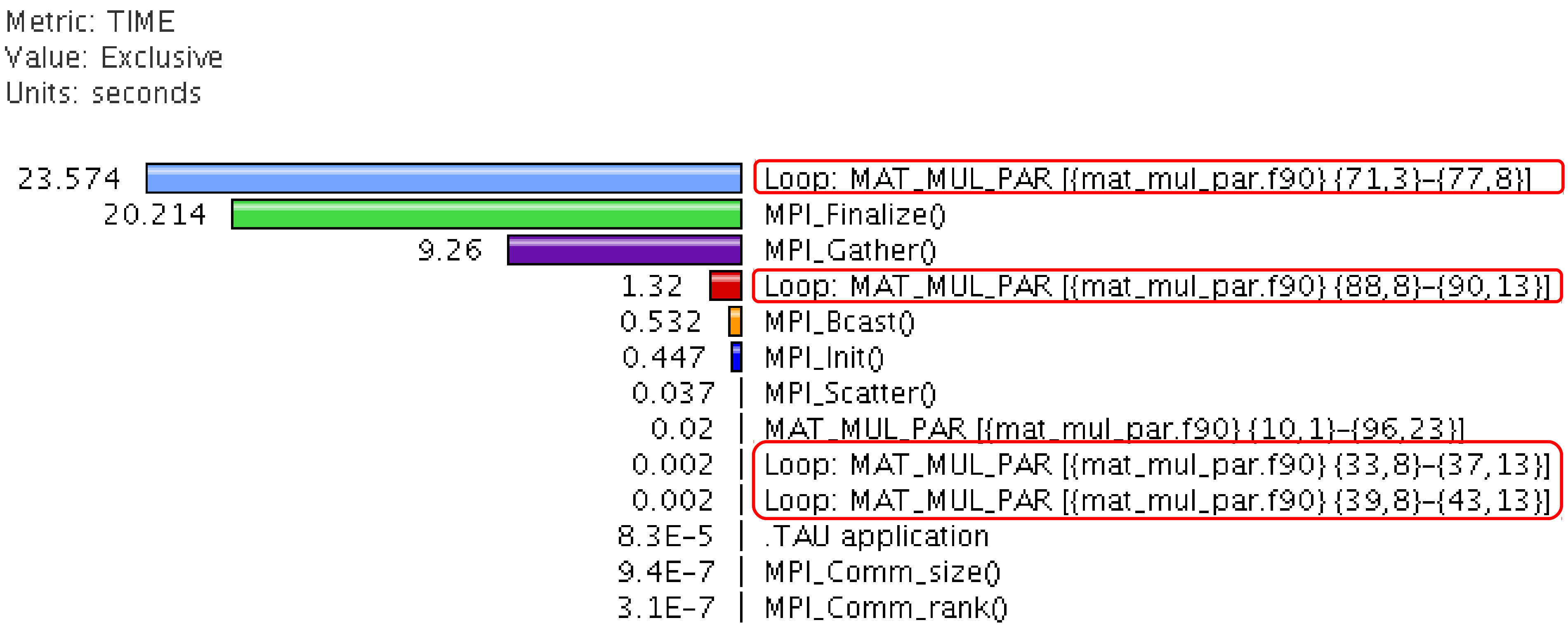
Selective Instrumentation

- Not all functions need to be profiled in large applications

```
export TAU_OPTIONS="-optTauSelectFile=select.tau"

cat select.tau
BEGIN_INSTRUMENT_SECTION
loops file="mat_mul_par.f90" routine="#"
END_INSTRUMENT_SECTION
```

Only need to profile outer loops of the given file

Selective Instrumentation



Selective Instrumentation

```
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
# The next line excludes all functions beginning with "sort_"
# and having arguments "int *"
void sort_#(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST
#Exclude these files from profiling
BEGIN_FILE_EXCLUDE_LIST
*.so
END_FILE_EXCLUDE_LIST
BEGIN_INSTRUMENT_SECTION
# instrument all the outer loops in this routine
loops file="loop_test.cpp" routine="multiply"
# tracks memory allocations/deallocations as well as
# potential leaks
memory file="foo.f90" routine="INIT"
# tracks the size of read, write and print statements in
# this routine
io file="foo.f90" routine="RINB"
```

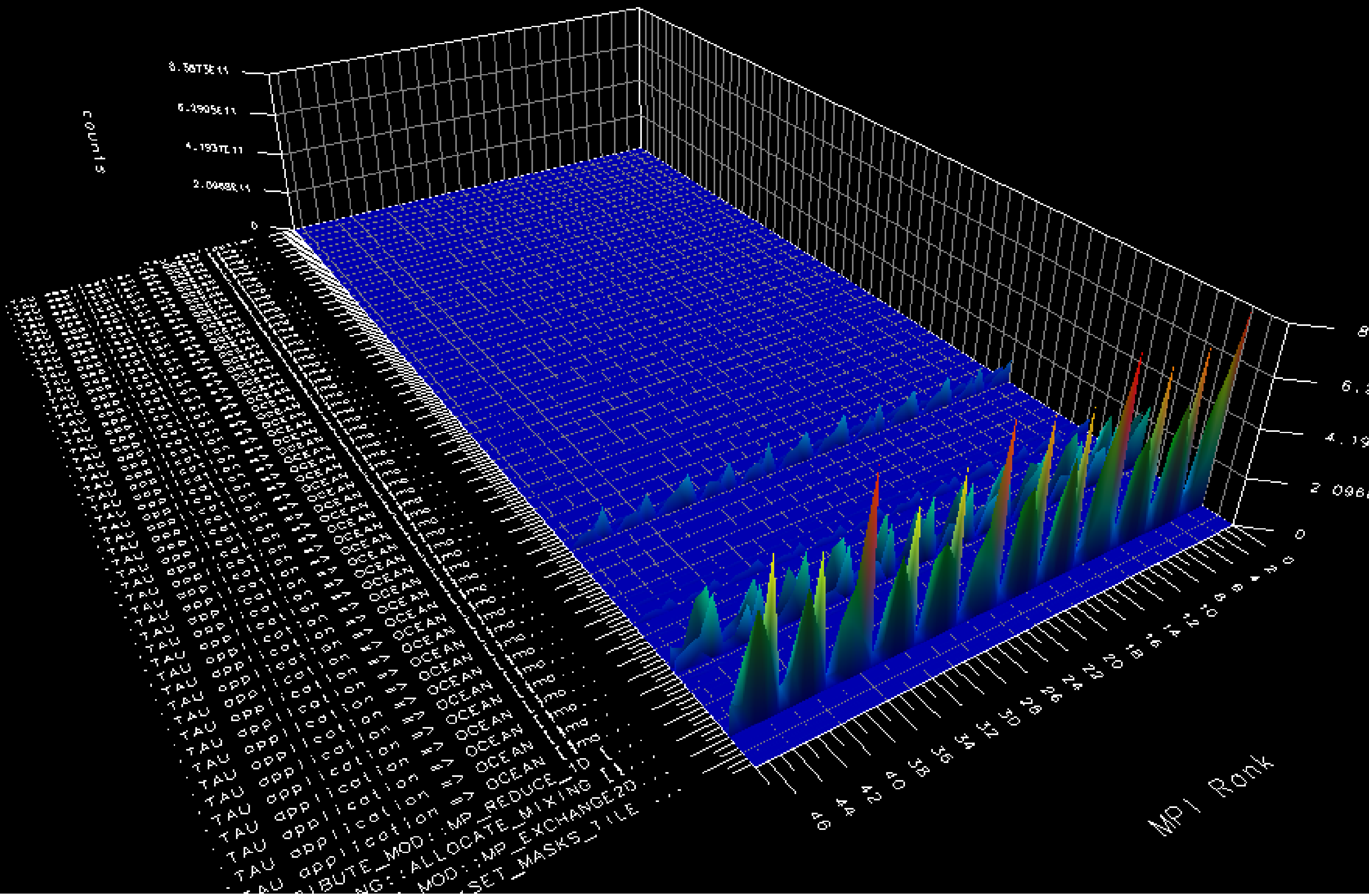
Using Optional TAU Compiler Options

- By setting **`TAU_OPTIONS`** variable or directly using TAU compiler options while compiling will change its behavior
  - **`-optTrackIO`** will profile I/O operations
  - **`-optHeaderInst`** will enable instrumentation of headers
  - For a full list, use **`tau_compiler.sh -help`** command

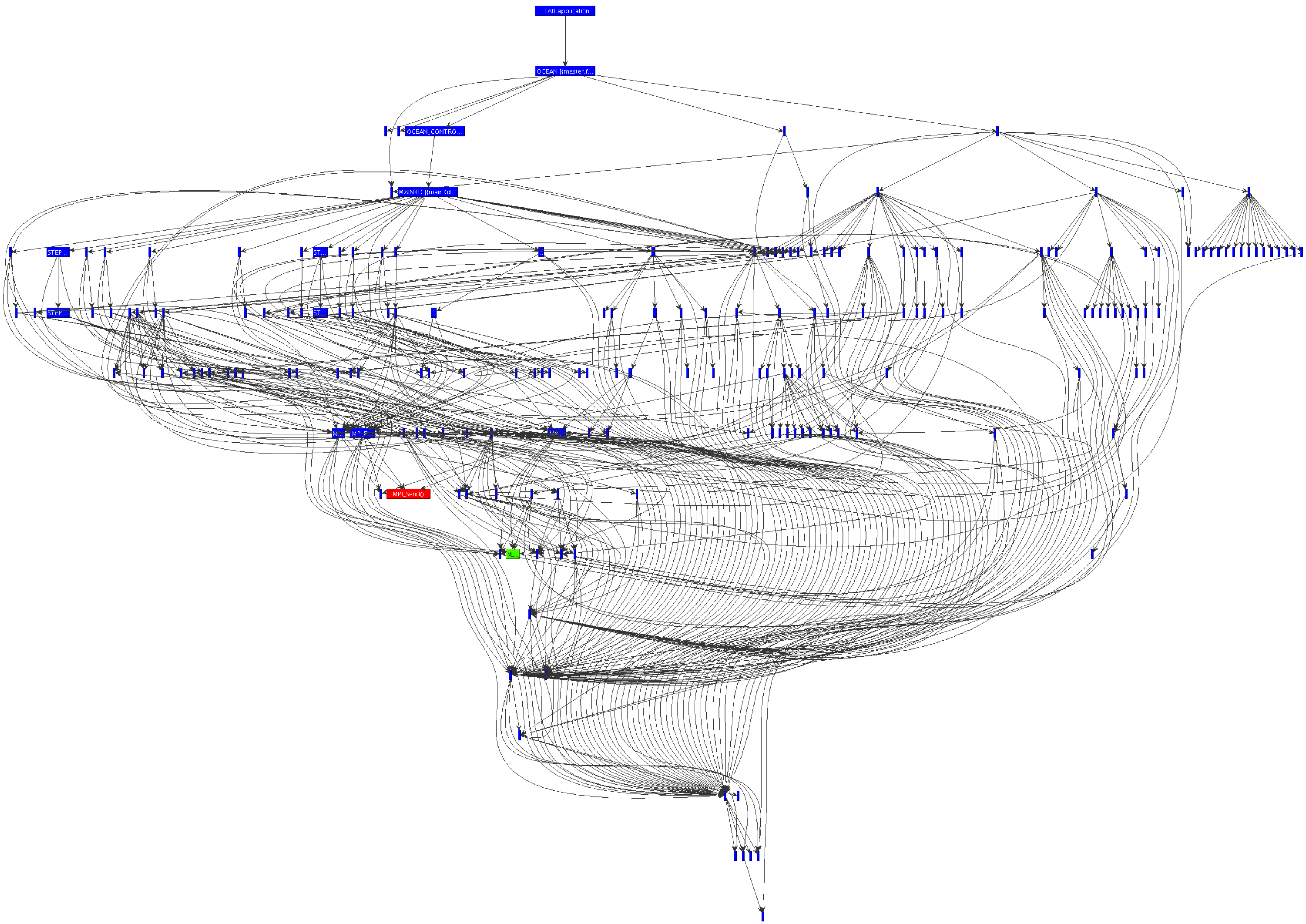
Runtime Environment Variables

Environment Variable	Default	Description
TAU_PROFILE	1	Set 0 to stop profiling (eg: for tracing)
PROFILEDIR	.	Location for profile files
TAU_TRACE	0	Set 1 for tracing
TAU_TRACK_HEAP TAU_TRACK_HEADROOM	0	Set 1 to track heap memory or headroom available
TAU_CALLPATH	0	Set 1 to start callpath profiling
TAU_COMM_MATRIX	0	Set 1 to generate communication matrix data
TAU_COMPENSATE	0	Set 1 to compensate instrumentation overhead
TAU_THROTTLE	1	Skip instrumenting functions called frequently
TRACEDIR	.	Location for tracing data

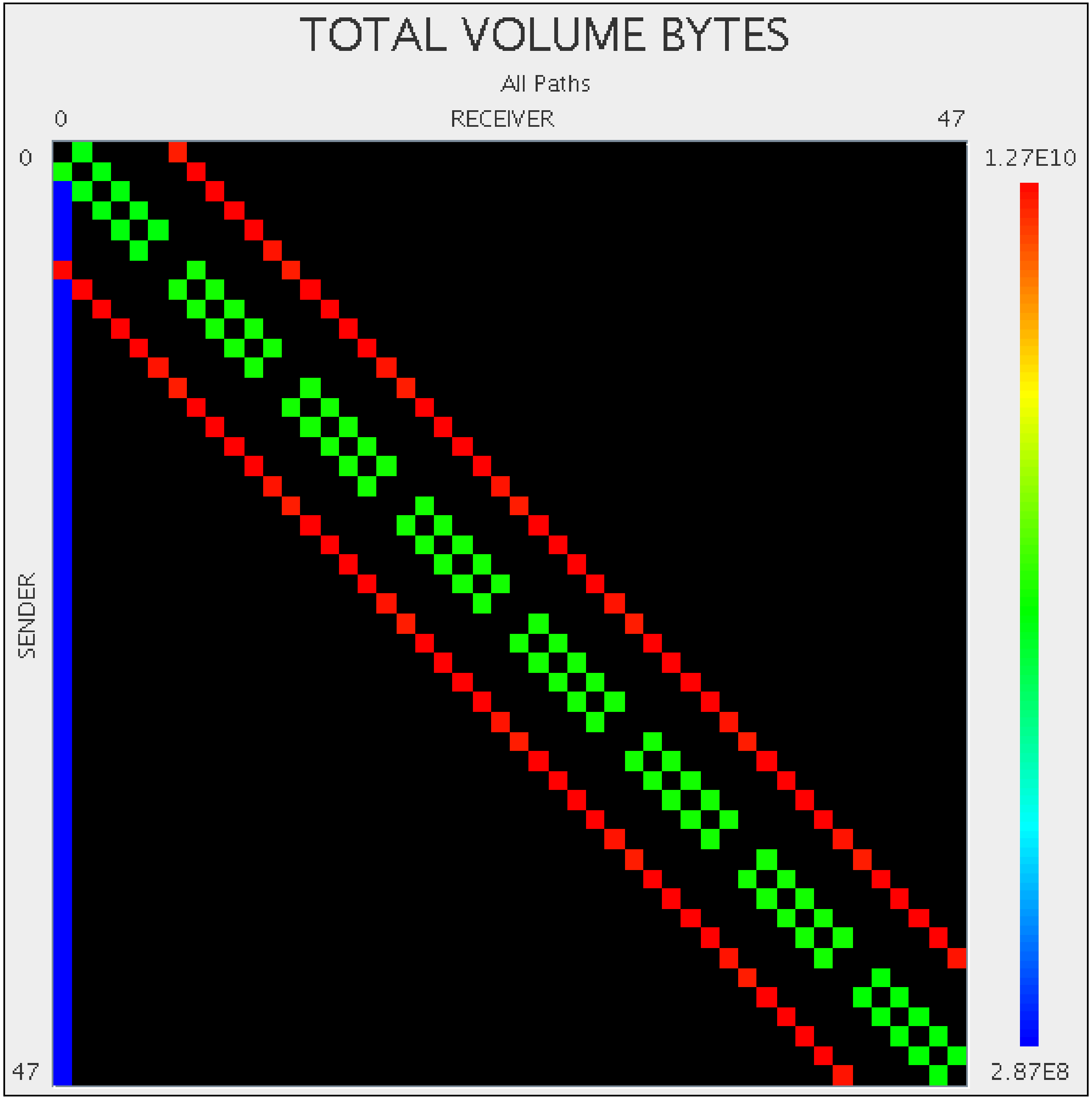
Real World Examples



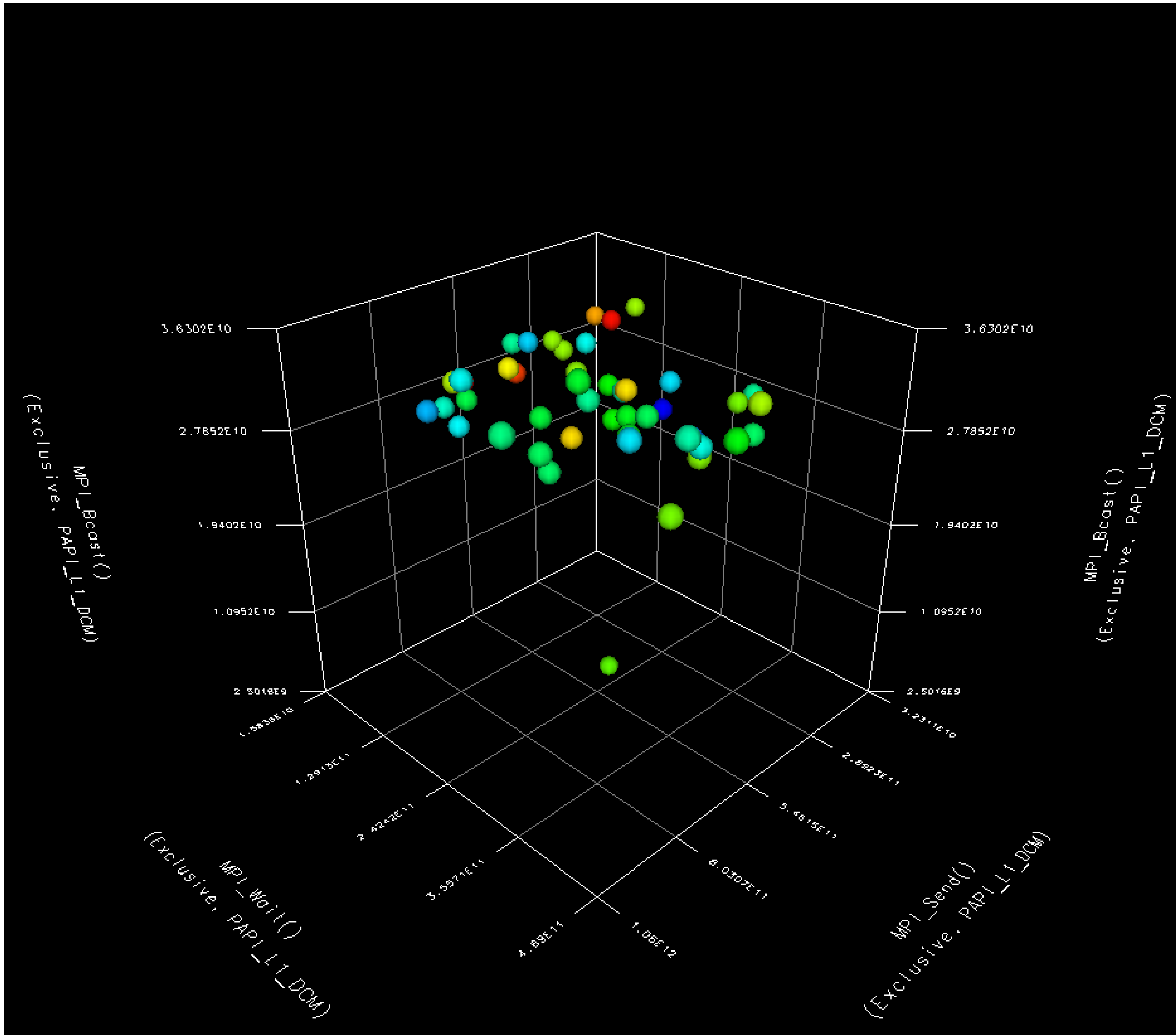
Call Path Graph



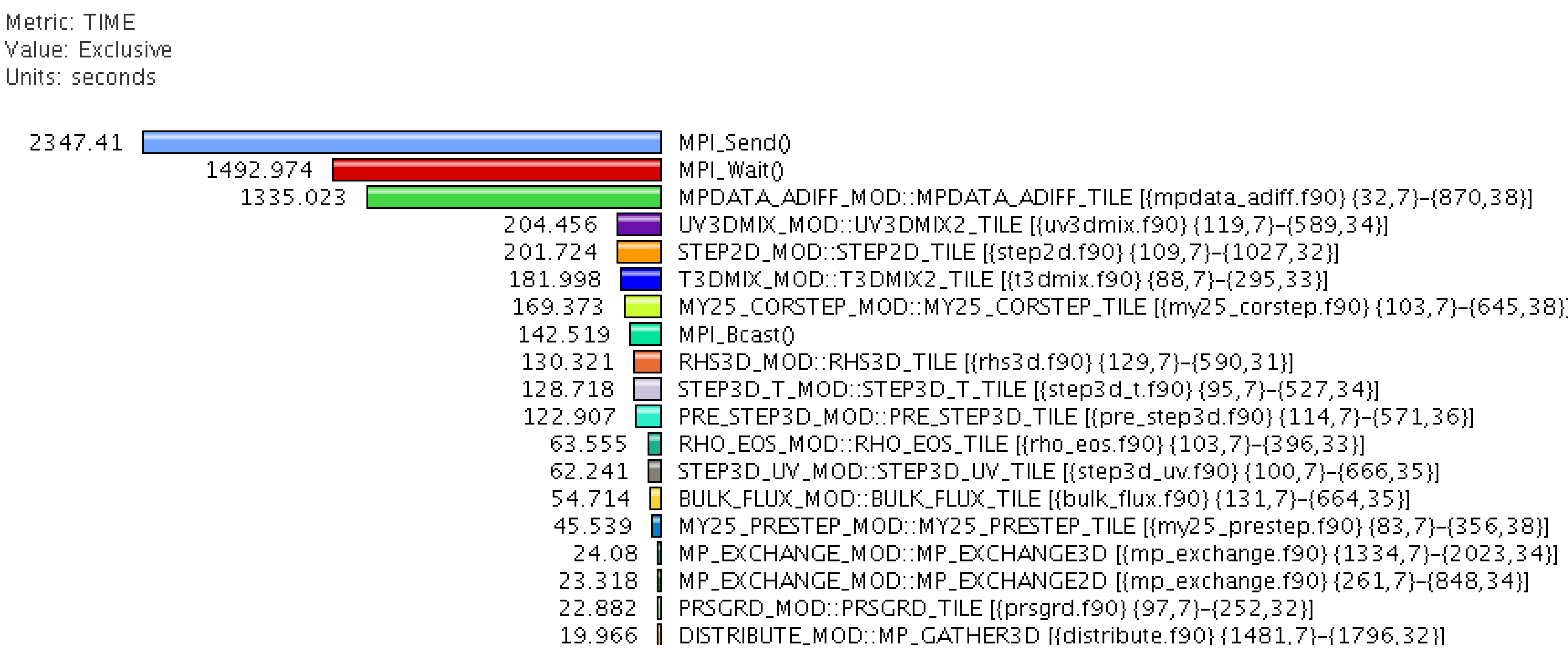
Communication Matrix



How and what each node is doing?



Flat profile of a real world case



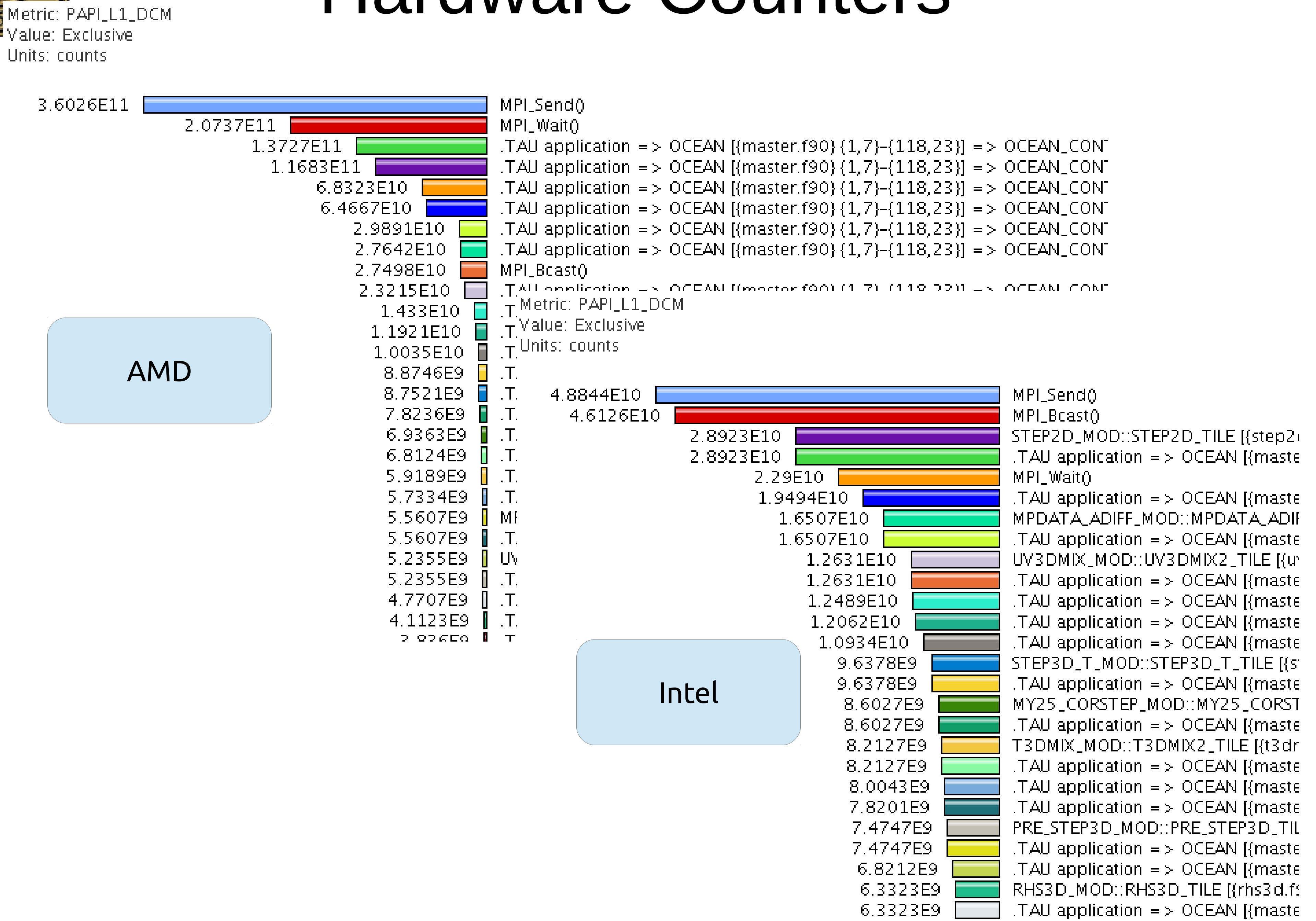
MPI_Send and MPI_Wait seems to be the culprit for slowdown
But why?
How do we find?

Hardware Counters

- TAU allows integration with other tools such as PAPI (Performance API)
- PAPI is installed on every HPC node and can be used to instrument a code using hardware counters as the metric
- **`papi_avail`** command will give you a complete list of available hardware counters on a specific node

```
export COUNTER1=GET_TIME_OF_DAY #To measure runtime
export COUNTER2=PAPI_L1_DCM #To find level 1 cache miss
export COUNTER3=PAPI_L2_DCM #To find level 2 cache miss
export COUNTER4=PAPI_FLOPS #To measure FLOPS
```

Hardware Counters



Measuring FLOPS

```
MULTI__PAPI_FLOPS/profile.1.0.0
---------------------------------------------------------------------
%Time Exclusive Inclusive     #Call #Subrs Count/Call Name
          Count total counts
---------------------------------------------------------------------
100.0 7.366E+06 5.984E+07        1      2   59837472 MATMUL_CACHE
 43.8      7739 2.624E+07        1      1   26235629 CACHE_MISS
 43.8      8066 2.624E+07        1      1   26235546 CACHE_NO_MISS
 43.8 2.623E+07 2.623E+07        1      0   26227890 Loop: CACHE_MISS
 43.8 2.623E+07 2.623E+07        1      0   26227480 Loop: CACHE_NO_MISS
MULTI__GET_TIME_OF_DAY/profile.-1.0.0
---------------------------------------------------------------------
%Time Exclusive  Inclusive   #Call #Subrs  Inclusive Name
           msec total msec                 usec/call
---------------------------------------------------------------------
100.0         2    79             1     2      79073 MATMUL_CACHE
 88.6     0.046    70             1     1      70078 CACHE_MISS
 88.6        70    70             1     0      70032 Loop: CACHE_MISS
  7.9      0.02     6             1     1       6230 CACHE_NO_MISS
  7.9         6     6             1     0       6210 Loop: CACHE_NO_MISS

CACHE_MISS 375 MFLOPS
CACHE_NO_MISS 4.4 GFLOPS
```

```fortran
program matmul_cache
  !Just calling the following functions here
end program matmul_cache

real function cache_miss(a, b, n)
  integer :: i, j
  real :: a(1024,1024), b(1024,1024)
  do i = 1, n
      do j = 1, n
          a(i,j) = b(i,j)
      enddo
  enddo
end function cache_miss

real function cache_no_miss(a, b, n)
  integer :: i, j
  real :: a(1024,1024), b(1024,1024)
  do j = 1, n
      do i = 1, n
          a(i,j) = b(i,j)
      enddo
  enddo
end function cache_no_miss
```

Hardware Counters

- To measure more counters at the same time,

  `export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM`

- Each counter will generate one profile in separate subdirectories that look like,

  `MULTI__GET_TIME_OF_DAY, MULTI__PAPI_L1_DCM`

- Intel and AMD nodes have different counters
- Up to 25 counters/events can then be recorded at a time

Tracing

- What happens in my code at a given time? When?
- Use **`Jumpshot`** to visualize results
- Significant overhead. Turn off profiling!

```
$ export TAU_MAKEFILE=/panfs/storage.local/\
> opt/hpc/gnu/openmpi/tau/x86_64/lib/\
> Makefile.tau-papi-mpi-pdt-trace
And compile your code, run
After job is finished, cd to TRACEDIR
$ tau_treemerge.pl
$ tau2slog2 tau.trc tau.edf -o tau.slog2
$ jumpshot tau.slog2
```

Tracing – **Jumpshot** view