Tau Commander User's Guide
Part II

D. Mackay, ParaTools, Inc.

Contents

Introduction:

This Guide provides more information to help software developers, scientists, and engineers using TAU Commander understand more of its features and the options that will allow to more effectively incorporate TAU Commander into development projects far more complex than those examples described in the introductory Guide. By this time the readers have been through the Introductory Guide and learned the basics of TAU Commander and have tested it on some simple projects. Alternatively, the readers have been using TAU Commander on their own project and have hurdles to overcome or want to learn more about TAU Commander features so they can use it more efficiently and do more with it. If one of the above cases does not describe your position, it is highly recommended that you work through the Introductory User's Guide Part I before continuing. This Guide provides more details about TAU Commander so the reader will be able to select the appropriate options and settings wisely. This Guide seeks to provide you a path to more efficiently incorporate TAU Commander into a productive development cycle to improve software performance.

Profiling:

Readers will recall profiling is the least data consuming method to collect and report software performance. The data produced by different analysis types is illustrated again below in Figure 1. It is fairly simple to understand that collecting detail information about events and when each event occurred and what threads were involved in the event will consume more data than simply measuring where in the code most of the time was spent. When more information is desired like loop region or callpath data to include information on which path caused a routine to be executed the collection becomes more data intensive than simply "where was execution time spent." Part I showed simple profiles collection. This guide covers the different methods of collecting profile data. This may be useful for those working on complex projects. There are three profile data collection methods for C/C++/Fortran code:sampling,source instrumentation and compiler instrumentation.

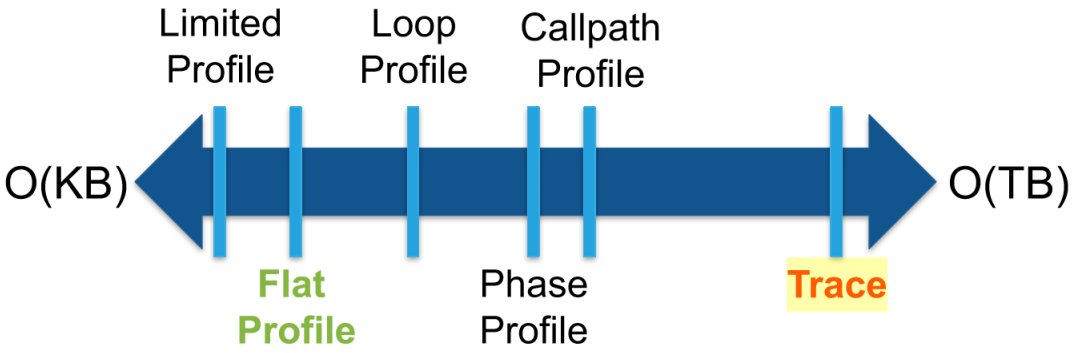

Figure 1 Relative quantity of data collected for different schemes

The first (and typically the default method) for collecting profile information is sampling. In sampling the software binary is periodically interrupted and the running state of the program is examined (e.g. instruction pointer). The sample information is aggregated to from the reports showing where in the code the processor spent of its time executing code. Sampling is easy to use – it does not require the readers to rebuild their software or make any changes. It also potentially avoids some unnecessary overhead. If a routine has a very short runtime but is called millions of times becoming one of the most time consuming parts of the code statistical sampling will identify this without the overhead of instrumenting every entry and exit from this routine. There can be some other limitations though. The reports are only as good as the information the sampling code can extract from the binaries. If the code is not built with -g data will still be collected and the execution profile will be shown for each routine as expected. However when -g is used to compile the code the compilers includes very useful and functional information that allows TAU Commander to point to specific lines of code in the source files. The -g option can be used with compiler optimization it is not restricted to no optimization. However, when the compiler optimizes code instructions are moved around so much that it is not possible to always map back to the exact line number so data may be off by a line or two sometimes. If the compiler inlines code, this can also alter the way data is shown. If a particular short routine is inlined in several different locations, the data may not be aggregated to show this one routine is a hotspot because it is replicated in several locations. Regardless sampling is very important and works even when source is not available – it will show the readers which routines and which shared objects or dynamically loaded dll in the .obj files which will allow TAU Commander to associate time at finer grain details to source code than simply a subroutine. Sampling does not require a rebuild or many other inconvenient operations and is easy to use.

The second instrumentation method is source instrumentation. This is done by the Program Database Toolkit (PDT). TAU Commander installs this automatically during installation and execution. PDT will parse the code prior to sending the code to the compiler. PDT will find function (and sometimes outer loop boundaries), insert TAU times and then send the code through the compiler as normal.

The third instrumentation method is compiler instrumentation. In compiler instrumentation the compiler inserts TAU calls directly during compilation. The compiler instrumentation process invokes the most overhead, and so it is not the preferred or default option.

Usually sampling will be all a developer needs. There are times that it doesn't work as desired these are the principle reasons it is so important to have the option to fall back to source instrumentation or compiler instrumentation. Another important reason is to allow the developer to explicitly define the instrumentation manually. When a TAU commander user selects source instrumentation `manual`, it means TAU Commander will not insert any instrumentation at all. It will link in the TAU libraries to fulfill the TAU APIs inserted by the developer of the code being profiled. By using the TAU APIs developers can define regions of the code in ways natural to them based on their understanding of the code. The TAU APIs are found in the TAU documentation at: https://www.cs.uoregon.edu/research/tau/docs/tauapis/index.html.

Instrumentation method is an element of the TAU Commander Measurement property. The example below show data collection using sampling, source instrumentation and compiler instrumentation for the matmult-omp found in the examples directly. After running tau init four measurements were setup by default on a linux system. The default was sampling. So for the first data collection there was no need select sampling. After generating data by sampling, the profile measurement was selected which relies on source instrumentation. Last profile was copied as comp-inst and that specification was changed to compiler-inst always to demonstrate compiler instrumentation. This code example uses both MPI and OpenMP so these were configured in as part of the initialization. OMP_NUM_THREADS was set to 2 and 4 MPI ranks were selected for the run. This oversubscribed the number of cores on the system so some imbalance in waiting for send and receives is expected. The following sequence of commands were executed on a linux system to generate Figures 2-5 below (please note tau accepts meas as shorthand for measurement. You may type out measurement in full if preferred):

```
cd ./examples/matmult_omp
tau init --mpi T --openmp T
tau meas edit sample --openmp ompt
make
tau mpirun -np 4 ./matmult
tau show
tau select profile
tau meas edit profile --openmp ompt
make clean
make
tau mpirun -np 4 ./matmult
tau show
tau meas copy profile comp-inst
tau meas edit comp-inst --compiler-inst always
tau select comp-inst
make clean
make
tau mpirun -np 4 ./matmult
tau show
```

The basic paraprof chart formed by each of the three tau show commands are below in Figures 2-4

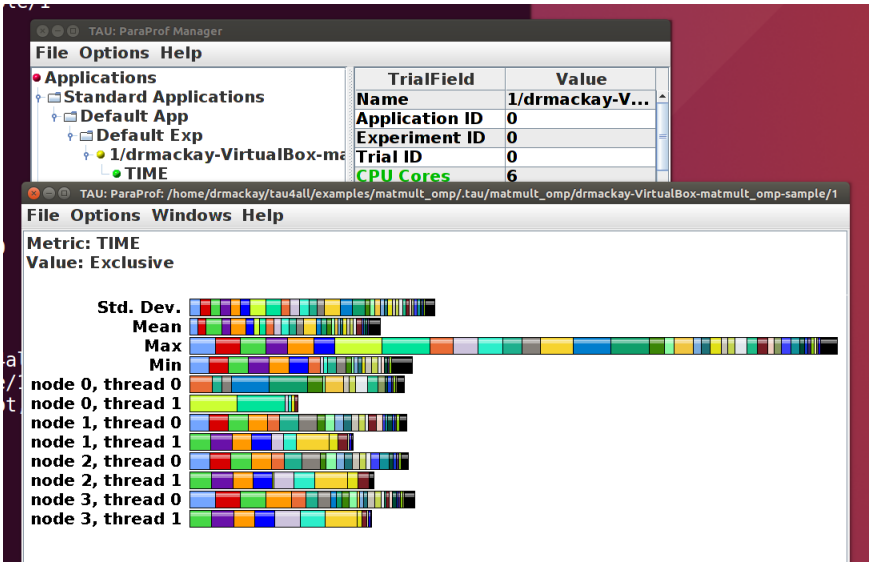

Figure 2 Matmult data with sampling

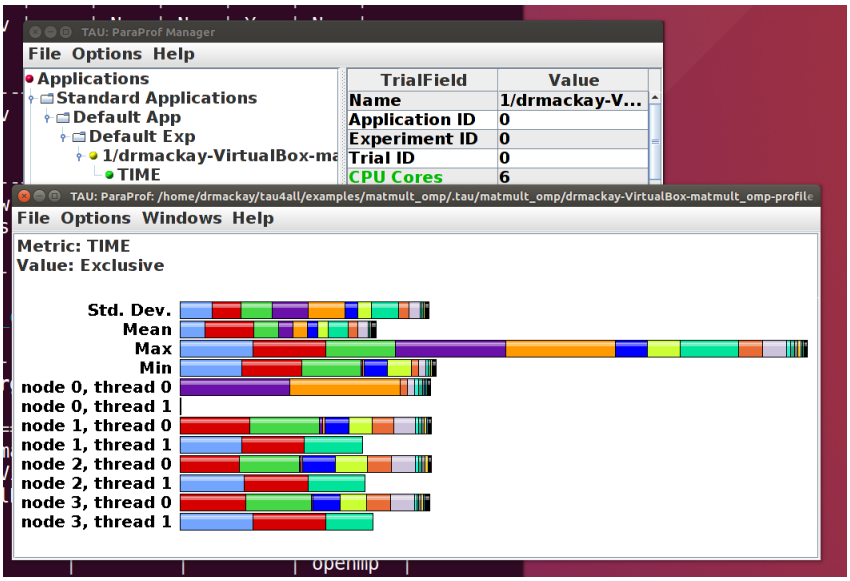

Figure 3 matmult data with source instrumentation

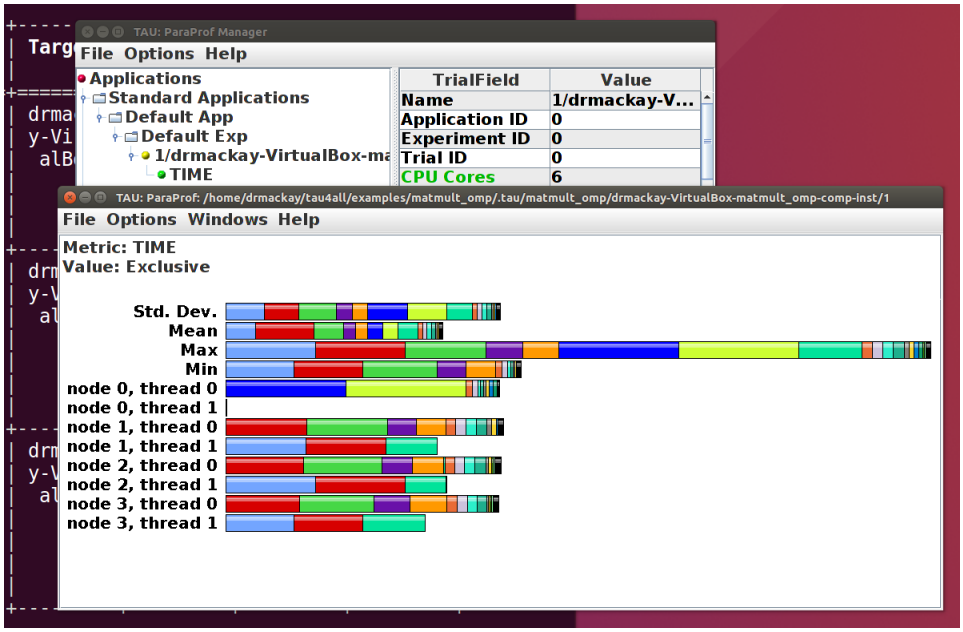

Figure 4 Matmult data with compiler instrumentation

The thread view of the mean average of all threads and processes for each of the three cases are assembled together below in Figure 5. Notice that the time spent in the main task is fairly consistent. Compiler instrumentation does have a bit more overhead than the other methods.

Figure 5 Mean Average of all threads for each data collection method (sample, source instrumentation and compiler instrumentation).

Throttling:

TAU Commander implements the elements of TAU such as throttling. The base documentation is here (https://www.cs.uoregon.edu/research/tau/docs/tutorial/ch01s05.html). Similar information is found in this section for the convenience of the reader. Instrumentation can cause overhead. Especially for functions with very short execution time that are called millions of times. THROTTLE is designed to reduce the computational overhead associated with instrumenting a program with TAU. This usually takes the form of selectively instrumented some functions but not others. Selecting the proper thresholds for throttling can sometimes best by done by beginning with data from an unthrottled measurement collection.

For example if you already had profile data showing a function is called on the order of 20,000,000 times that may be a significant source of overhead especially for an application that runs a short time. When a function is entered and exited a small amount of tauinstrument code is executed. When a function is called millions of times even that small amount of code can cause a slow down in execute

time. The settings shown below will remove the computational overhead of instrumenting a function that is called 20 millions times. Throttling is an element that is part of the TAU Commander Measurement property so these settings would be something like this:

```
tau measurement edit <measurement_name> --throttle T

tau measurement edit <measurement_name> --throttle-num-calls 400000

tau measurement edit <measurement_name> --throttle-per-call 3000
```

This will tell TAU Commander not to profile any functions which are called more than 400000 times and their inclusive time per call is less than 3 seconds.

By default TAU Commander may already be set up throttling active. This example illustrates the effect of throttling by showing a run with and without throttling. This illustration uses the pi example in ./examples/pi. If that is missing on your system the source code serialPi.cpp is appended to the end of this guide for your reference. First go to the directory pi. On a Linux system enter the following commands:

```
tau init
tau select profile
make
tau ./serialPi
tau show
```

The resulting display should be similar to what is shown below in Figure 6.

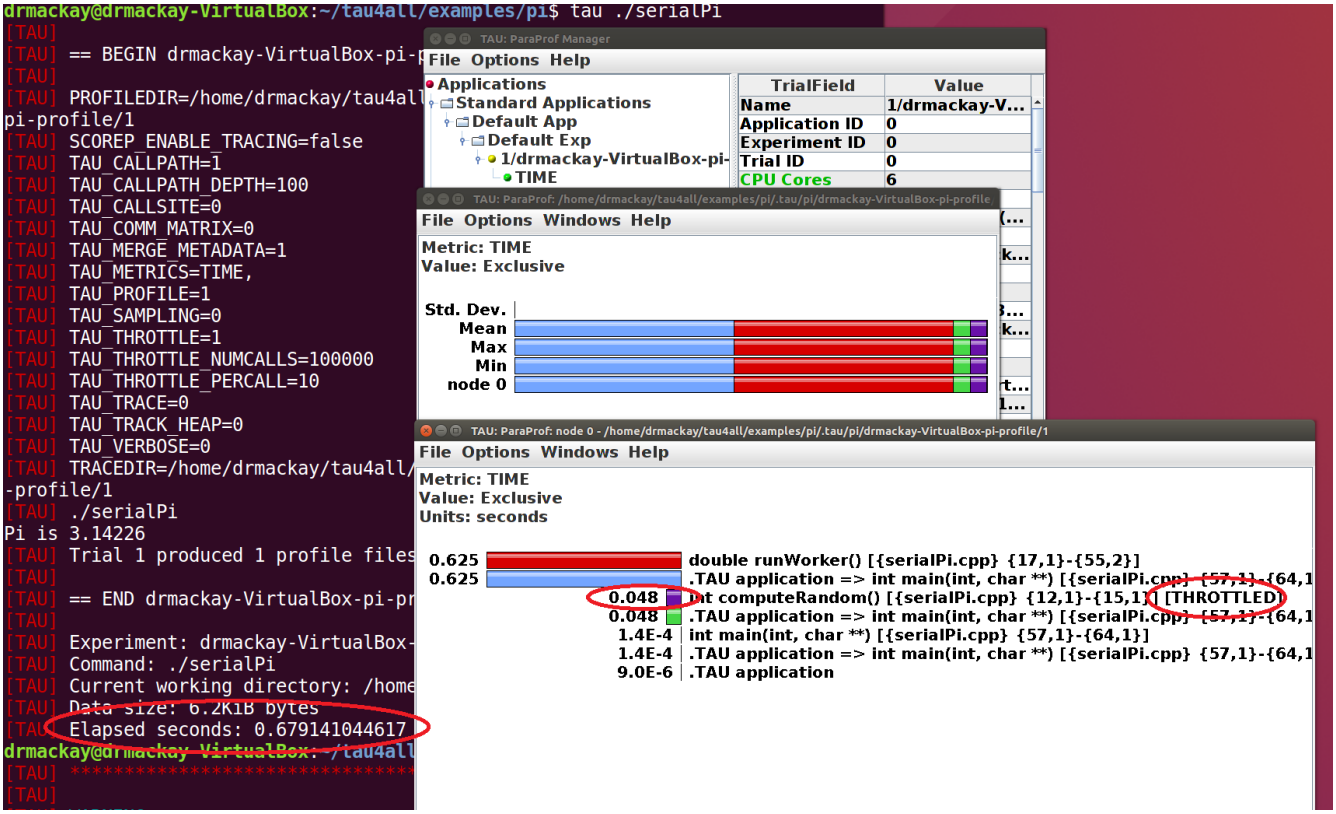

Figure 6 SerialPi captured with default Throttling

Now setup the same code without throttling. Once again on linux system this might be done with these commands:

```
tau meas copy profile profile-nothrot
tau meas edit profile-nothrot -throttle F
tau select profile-nothrot
make clean
make
tau ./serialPi
tau show
```

The resulting display should be similar to Figure 7. Notice that in Figure 6 next to function int computeRandom the text box '[THROTTLED]' appears. This is an indication that TAU Commander applied throttling. Look also at the terminal view and see that the elapsed time was only 0.679 seconds. Now contrast this to Figure 7. The first thing to notice is that int computeRandom occupies far more time than reported in the previous example. More important is that immense time dilation that occurred. More time is reported in ComputeRandom than the entire application took to run in the first case. This is also highlighted by the reported elapsed time in each case: 0.679 seconds vs. 8.798 seconds for the later (the elapsed time is circled by a red ellipse to make it easy to point out. If you now open up the statistical table for the thread as shown below in Figure 8 you will notice that computeRandom is called 20,000,000. The overhead of recording 20,000,000 entries and exits and call path consumers excessive time. Throttling captures the time without recording the number of calls and unwinding the call stack. You may explicity control the parameters for throttling: number of calls and time per call or accept the default settings.

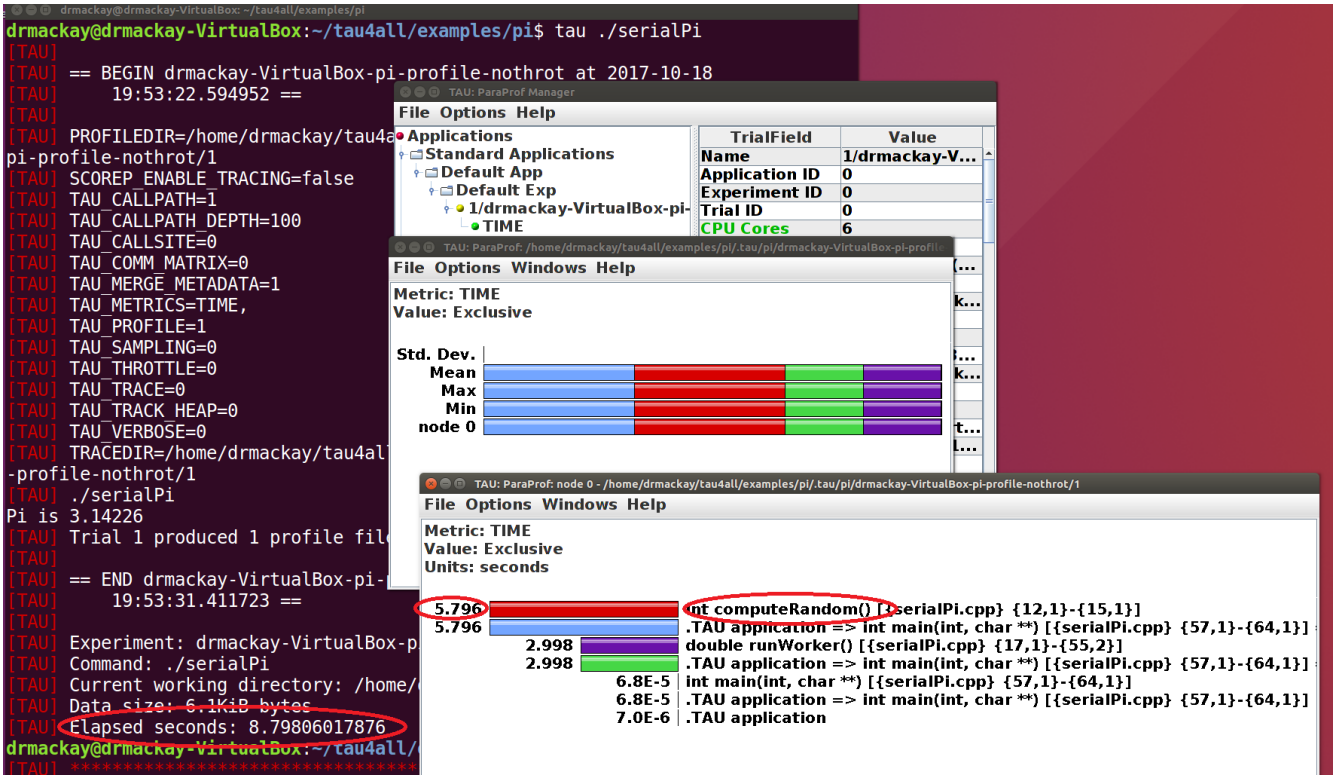

Figure 7 SerialPi profile without throttling.

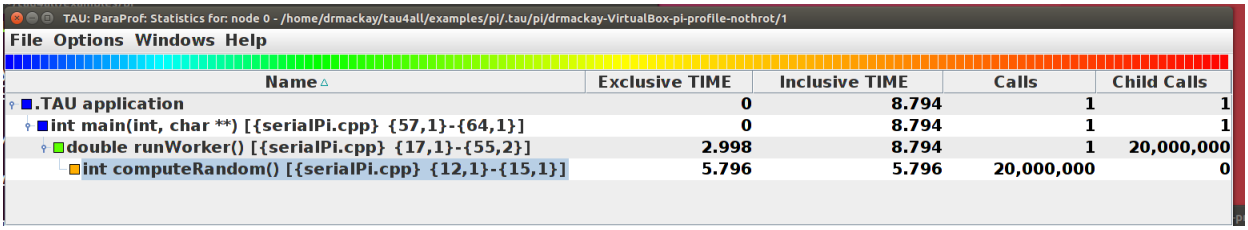

Figure 8 Paraprof display of Statistics Table for serialPi

Selective Instrumentation:

Selective instrumentation allows developers to choose to code sections to include or exclude from source instrumentation. Selective instrumentation is only available with source instrumentation – it does not work with sampling or compiler instrumentation. Selective instrumentation is an element of the TAU Commander application property. When a developer wants to use selective instrumentation he will enter: `` `tau application edit application_name --select_file <file_path>` ``. The TAU documentation for selective profiling is located here: https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01s03.html and most of it is reproduced here for the reader's convenience.

To specify a selective instrumentation file, create a text file and use the following guide to fill it in:

- Wildcards for routine names are specified with the # mark (because * symbols show up in routine signatures.) The # mark is unfortunately the comment character as well, so to specify a leading wildcard, place the entry in quotes.
- Wildcards for file names are specified with * symbols.

Here is a example file:

```
#Tell tau to not profile these functions
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

# A dynamic phase will break up the profile into phase where
# each events is recorded according to what phase of the application
# in which it occured.
dynamic phase name="foo1_bar" file="foo.c" line=26 to line=27

# 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"

END_INSTRUMENT_SECTION
```

Selective instrumentation files can be created automatically from `ParaProf` by right clicking on a trial and selecting the `Create Selective Instrumentation File` menu item.

An example of selective instrumentation is now run with the serialPi program used in the section above. In this example computeRandom is excluded rather than throttled. In the same directory create a file called ex.file with this text content:

```
#Tell tau to not profile these functions
BEGIN_EXCLUDE_LIST
int computeRandom()
END_EXCLUDE_LIST
```

Now since application pi is already defined and used without selective instrumentation in a previous example it is necessary to create a new application. Then edit the properties of the new application to accept the selective instrumentation file and then select the new application and measurement method, build and run. If you apply a selective instrumentation file prior to creating any trials it is not necessary to copy or create a new application definition. This set of instructions uses the measurements defined in the previous sections.

```
tau application copy pi pi-exclude
tau application edit pi-exclude --select-file ./ex.file
tau select pi-exclude prfile-nothrot
make clean
make
tau ./serialPi
tau show
```

(note you could also use the full path to the file /home/username/taucmdr/examples/pi/ex.file). The results are shown below in Figure 9. Notice that computeRandom disappears entirely from the view. Selective instrumentation is not a replacement for throttling. Throttling preserves time spent in the routine while filtering out collection of entries/exists and paths. In contrast selective instrumentation does not collect nor display the time spent in that region. It is all simply applied to the parent function. If a specific area of code really is not of interest – then it can be excluded in this manner. If a particular function is problematic (e.g. causing instrumented build to fail) it can be excluded and work can contine. There are other situations that may also apply, just make sure to understand the implications.

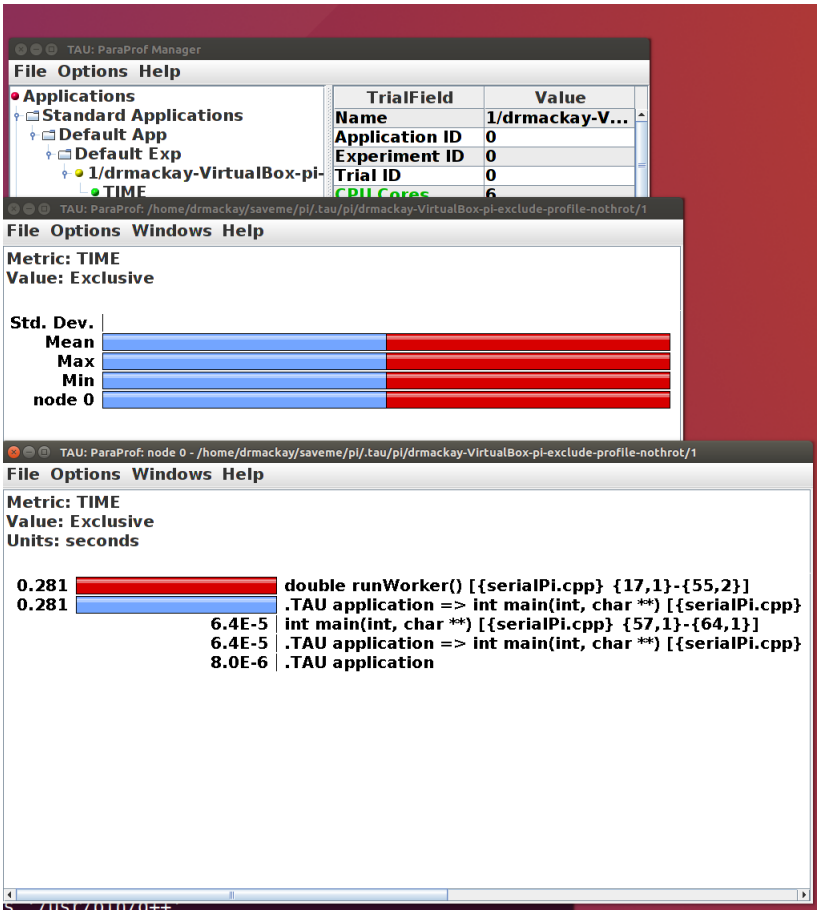

Figure 9 serialPi with selective instrumentation

See:
TAU Commander Quick reference card
TAU Commander Introductory Guide