Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Hands on TAU: Tuning and Analysis Utilities

L. Antonelli

RT-ICAR-NA-2011-01 **Dicembre 2011**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Napoli, Via P. Castellino 111, I-80131 Napoli, Tel: +39-0816139508, Fax: +39-0816139531, e-mail: napoli@icar.cnr.it, URL: *www.na.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Hands on TAU: Tuning and Analysis Utilities

L.Antonelli[1]

Rapporto Tecnico N.:01
RT-ICAR-NA-2011-01

Data:
Dicembre 2011

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Napoli, via P. Castellino, 111, 80131 Napoli

Hands On
TAU: Tuning and Analysis Utilities

L. Antonelli

Abstract

This report will describe the use of TAU (Tuning and Analysis Utilities) for software profiling. It was started within activities having as final aim performance analysis of the reliable and efficient software for multicore architectures([1]). The report aims to collect experiences and helpful advice for TAU using, but it is not exhaustive about profiler matter nor about TAU for a specific application.

1 Introduction to software profiling

Software optimization doesn't begin where coding ends: it is ongoing process that starts at design stage and continues all the way through development.

A profiler tool traces and analyzes program performances. Profiling highlights where the program spent its time and wich functions are the hotspots ([2]), and might be candidates for rewriting to make the program execution faster. Generally, profiling process is carried out at three stages:

1. Compiling and Linking program with profiling enabled (*instrumentation*);
2. Executing program to generate the file of profile data;
3. Using profiler tool to analyze the data profile (i.e. producing tables and/or graphs).

2 Overview on TAU

TAU (Tuning and Analysis Utilities) Performance System [1] is the product of eighteen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement,

[1]PILTP project: Parallel Inverse Laplace Transform Package. The project is in collaboration with Applied Science Department and with Statistic and Mathematic Department for Economic Research at University of Naples *Parthenope*

[2]hotspots are the area in the program that take accounting for a long time.

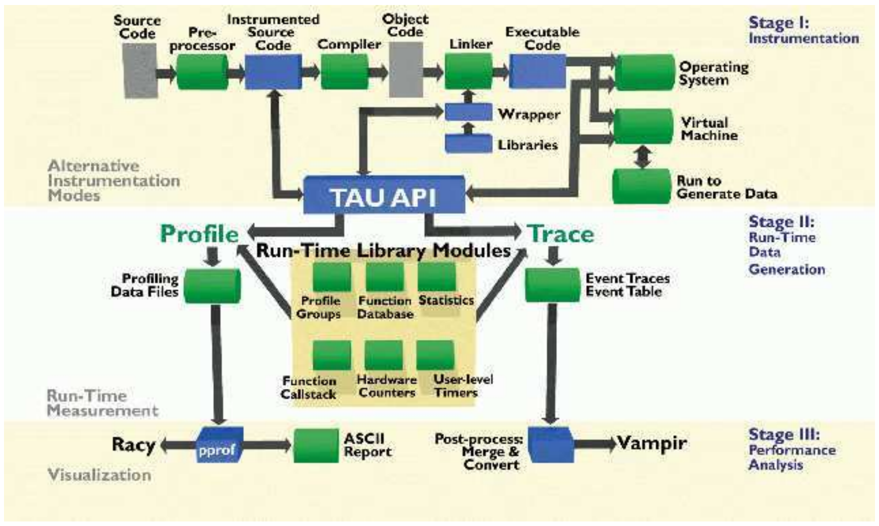


Figure 1: Architecture of TAU Performance System: Instrumentation and Measurement

analysis and visualization of large-scale parallel computer system and application. TAU project represents the combined efforts of researchers at University of Oregon, at the Research Centre Juelich and Los Alamos National Laboratory. TAU runs on all HPC platforms and it is free (BSD style license).
TAU provides profiling and tracing toolkit for performance analysis of parallel (and serial) programs written in Fortran, C, C++, Java, and Python:

- *profiling* shows how much (total) time was spent in each routine;
- *tracing* shows when the events take place in each process along a timeline.

The goal of the TAU project is developing program and performance analysis technology that meets both the challenges of evolving scalable parallel computing systems and the needs of programming methodologies used for next-generation scientific applications. The technology should be able to target the diversity of computing paradigms and machines while offering a framework of portable and reconfigurable measurement and analysis components that can be optimized and extended. While the tools and techniques implemented may address specific needs of a language or execution environment, they should be coherent, based on a unified analysis model and able to interoperate with other framework components. The skeleton of TAU system is shown in Figure 1, where you can see the three main components: instrumentation, measurement and performance analysis.

3 How to use TAU for software profiling

Since TAU system can be customized and configured for several performance experiment by composing specific modules and setting runtime control, you have to set TAU scope before profiling your application. Then, you can instrument your code and run it. Finally, you can analyze performance of your application by means TAU visualization tools.

3.1 Setting TAU scope

TAU analysis tools can be customized and configured for each performance experiment, for instance, MPI parallel program, using OpenMP directives, etc. This is the TAU scope.
TAU configuration generates several Makefile stubs as well as several libraries to set TAU scope. A Makefile stub name is like `Makefile.tau-options` as well as a library name is like `libtau-options-a`. Each Makefile stub defines a suitable TAU scope to performance analysis and/or measurement of a selected application.
To configure TAU for profiling experiments you have to follow these basic steps:

1. setting of TAU bin directory:

   ```
   user@home:> export PATH=$PATH:/taurootdir/[arch]/bin/
   ```

   (you can modify PATH variable in your bashrc or cshrc file);

2. choosing an appropriate Makefile stub in `taurootdir`:

   ```
   user@home:> ls /taurootdir/[arch]/lib:
   Makefile.tau-mpi-pdt
   Makefile.tau-mpi-openmp-pdt
   ...
   ```

3. setting of TAU scope by means Makefile stub:

   ```
   user@home:> export TAU_MAKEFILE=/taurootdir/[arch]/lib/Makefile.tau-mpi-pdt
   ```

4. choosing a TAU compilers for your apllication: `tau_f90.sh`, `tau_cxx.sh` or o`tau_cc.sh` for programs written in F90, C++ or C rispectively.

5. setting of TAU compiler options:

   ```
   user@home:> export TAU_OPTIONS=-optVerbose ...
   ```

   A list of options for TAU compiler scripts can be found by tiping:

   ```
   user@home:> tau_compiler.sh -h
   ```

If your application has an own Makefile, you can modify it suitably inserting the instructions described from 3 to 5 (see appendix A).

3.2 TAU instrumentation

In order to observe performance, additional instructions or probes are typically inserted into a source code. This process is called *instrumentation*. Instrumentation can be introduced in a program at several levels of the transforming process to generate executable code.
TAU provides three methods to instrument your application for performance analysis:

i *dynamic instrumentation*: library interposition using `tau_exec`;

ii *compiler based instrumentation*: compiler directives;

iii *source based instrumentation*: source transformation using PDT[3] [5, 6].

The requirements for each method increase from i to iii.

3.2.1 Dynamic instrumentation: `tau_exec`

Dynamic instrumentation (at runtime) is achieved through library preloading. The library choosen for preloading determine the scope of instrumentation (substituting I/O, MPI and memory allocation/deallocation routines with instrumented calls). Several options are enabled by help command-line of `tau_exec`, for instance:

```
user@home:> tau_exec


Usage:  tau_exec [options] [--] <exe> <exe options>

Options:
        -v                      verbose mode
        -qsub                   use qsub mode
        -io                     track I/O
        -memory                 track memory
        -T                      <DISABLE,ICPC,MPI,PDT,PROFILE,PTHREAD,PYTHON,SERIAL>:
                                specify TAU option
        -XrunTAUsh-<options>:   specify TAU library directly
```

[3]Program Database Toolkit (PDT) is a framework for analyzing source code written in several programming languages and for making rich program knowledge accessible to developers of static and dynamic analysis tools. PDT implements a standard program representation, the program database (PDB), that can be accessed in a uniform way through a class library supporting common PDB operations.
http://www.cs.uoregon.edu/Research/pdt/home.php

Since `tau_exec` can be used on both uninstrumented code and strumented code, different layers of instrumentation can be combined. In order to use `tau_exec` you have to insert this command before the executable program (and after the usual command for execution). For instance, I/O instrumentation is required for serial code:

```
user@home:> tau_exec -io a1.out
```

and for parallel MPI code:

```
user@home:> mpirun -np 4 tau_exec -io a2.out
```

3.2.2 Compiler based instrumentation

TAU provides these scripts: `tau_f90.sh`, `tau_cc.sh` and `tau_cxx.sh` to compile and instrument Fortran, C and C++ programs rispectively. Before usiningg these scripts TAU makefile and/or options have to be set (see section 3.1). For instance, C program compilation is shown:

```
user@home:> tau_cc.sh -tau_makefile=[pathTOmakefile]
                      -tau_options= -optCompInst sampleCprogram.c
```

`pathTOmakefile` is the TAU distribution directory like `/taurootdir/[arch]/lib`, where you can find several Makefile stubs for setting TAU scope. To use compiler based instrumentation you can also set TAU_OPTIONS variable with `-optCompInst` as described in step 5 of section 3.1, furthermore you can suitably modify the Makefile of your application (see appendix A).
If you use C preprocessor directives (e.g. `#include, #ifdef, #endif`,...), you will have to add also `-optPreProcess` to TAU options.

N.B. Before of the compilation process, it is possible to insert TAU instructions into the source code for performance measure (this is the *the manual* instrumentation of source code). Since TAU measurement API are written in C++, the (instrumented) source code have to be compiled with a C compiler script (`tau_cc.sh`) and linked with a C++ linker script (`tau_cxx.sh`). Moreover, at the beginning of each instrumented source file, you have to include the following header:

```
#include <TAU.h>
```

3.2.3 Source based instrumentation

The TAU framework for automatic source instrumentation uses the Program Database Toolkit (PDT). PDT consists of the DUCTAPE (C++ program Database

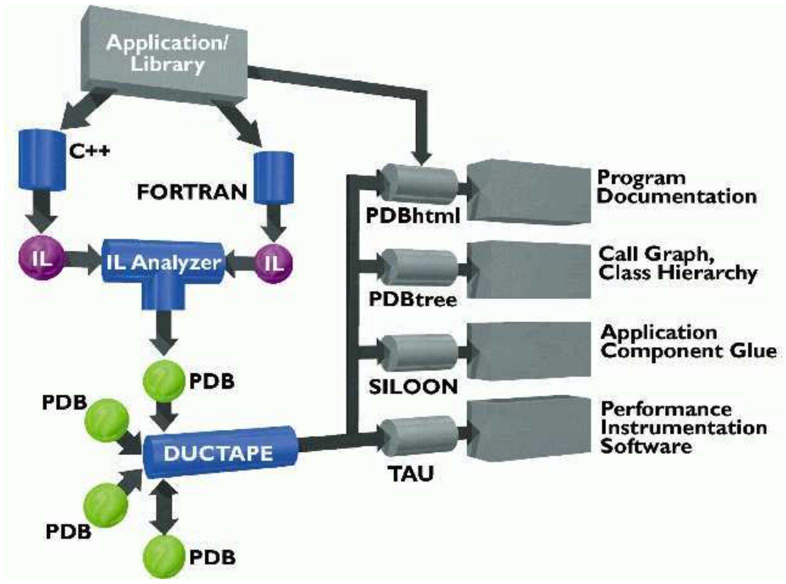


Figure 2: PDT System

Utilities and Conversion Tools APplication Environment) [8] and the IL (Intermediate Language) Analyzer [7]. The PDT system is shown in Figure 2. The compiling process based on source instrumentation follows these steps:

1. *Parsing Source Code*: IL Analyzer parses the source and produces an intermediate language file. The IL Analyzer processes this IL file, and creates a "program database" (PDB) file consisting of the high-level interface of the original source (i.e. the first step produces from `file.c --> file.pdb`). The use of the DUCTAPE library then makes the contents of the PDB file accessible to TAU applications.

2. *Instrumenting Source Code*: based on the analysis of PDB file, TAU instrumentor produces an instrumented source (i.e. the second step produces from `file.pdb --> file.inst.c`).

3. *Creating Object(s)*: a suitable script compiler of TAU produces the object code (i.e. the third step produces `file.inst.c --> file.o`) and then the executable program.

In source based instrumentation, the TAU makefile with "pdt" in the name (e.g. `Makefile.tau-mpi-openmp-pdt`) have to be used. Moreover, it is useful to set `TAU_OPTIONS` both `-optKeepFiles` for preserving instrumented source file(s) (see section 6.2 and `-optVerbose` for visualizing overall for visualizing overall of the instrumentation and compilation processes.

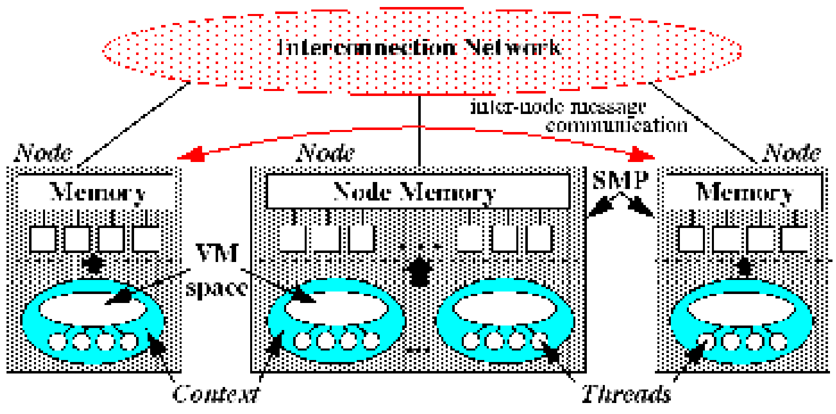


Figure 3: TAU Execution Model

You can see in appendix A an example of the source instrumentation via Makefile.

4 Running (instrumented) application

After compiling process, you have to continue with (usual) execution phase of your application. The run of your instrumented application will produce the usual output as well as data profile file(s). The name of data profile file(s) follows the HPC++ convention([4]) [1], so it will be like: `profile.<node>.<context>.<thread>` (e.g. `profile.0.0.0` for sequential run). A `node` is a shared-memory multiprocessor (SMP), having a coherent shared-address space that can be read and modified by any of its processors. A `context` refers to a virtual address space on a node that may be accessible to several different threads of control; for example, a Unix process is a context. It is possible to have more than one context per node. Multiple *threads*, both system and user level, may exist within a context; threads within a context share the same virtual address space. Threads in different contexts on the same node can interact via interprocess communication facilities, while threads in contexts on different nodes communicate using message passing libraries (e.g. MPI) or network IPC (see Figure 3).

5 TAU profiling

Once profile data are produced, TAU provides several analysis and visualization tools to process them. It provides simple command line reporting tools as well

[4]The library's interface evolved from the High Performance C++ Consortium, a collection of research groups from university, industry and government laboratories

as GUI analysis tools and a performance data management system.

5.1 Flat profiling

The most basic and standard type of profiling is called *flat profiling*. It shows tabular output of the wall clock time and the fraction of total time. The command for processing profile data in order to produce the flat profile is `pprof`. For example, we carried out an automatic source instrumentation via Makefile (see appendix A) on the example code described in appendix B. The flat profiling obtained is the following:

```
user@bladehpzmaster:$ pprof profile.0.0.0

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        1,870        1,873           1          10    1873573 int main() C
  0.1            2            2           5           0        520 void matvet() C
  0.0        0.019        0.019           5           0          4 void maxvet() C
```

The profile above are generated using the default profiling library. The functions are ordered in according to tree call of the program. Function name are shown in the last column of table, `Name`, furthermore:

`%Time`	- the percentage of the "total" time spent in this function and its children.
`Exclusive msec`	- the cumulative, exclusive time (milliseconds) summed over all the invocations of the functions. Exclusive time refers to the total time spent in the function minus any time spent in other profiled functions called from it.
`Inclusive msec`	- the time spent from the entry point to the exit, summed over all the function invocations.
`#Call`	- number of function invocations.
`#Subrs`	- number of invocated functions (multiple invocations are accounted).
`Inclusive usec/call`	- the inclusive time for call (microseconds).

5.2 Callpath profiling

Flat profiling keeps track of time spent in each function, but does not keep track of the relationship between functions. For instance, a flat profile will be able to tell you that your program spent 25.2 seconds executing function A, but will not be able to tell you that A ran for 10.5 seconds when called from B and 14.7 seconds when called from C. In other words, a flat profile records time spent with respect to functions rather than to function callstacks. Generating a *path profile* (or *callpath profile*) involves another method of collecting profile data in which statistical information is kept with respect to function callstacks. *Callpath profile* tracks time spent in function paths rather than time spent in each function([5]).
Note that TAU uses a slightly different definition of the *callpath profile* that is the distribution of performance along the dynamic routine calling path of an application. The *depth* of a callpath is the maximum recorded number of invocated functions. A *callpath* of *depth* 1 is a flat profile.
You can enable *callpath profiling* by setting the environment variable TAU_CALLPATH([6]):

```
user@home:> export TAU_CALLPATH=1
user@home:> export TAU_CALLPATH_DEPTH=2
```

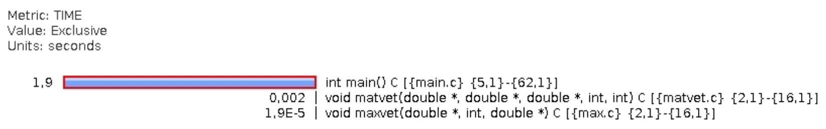
In this mode, TAU will record each event callpath to the depth set by the TAU_CALLPATH_DEPTH environment variable (default is 2).
For example, you can see the different tabular output obtained profiling the same example C code (see appendix B) when the *callpath* is enabled.
In the `Name` column, "a() => b()" describes the time spent (exclusive/inclusive) in routine "b()" when it is called by routine "a()".

```
user@bladehpzmaster:$ pprof profile.0.0.0

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        3,291        3,294         1         10    3294210 int main() C
  0.1            2            2         5          0        528 int main() C => void matvet
  0.1            2            2         5          0        528 void matvet() C
  0.0        0.053        0.053         5          0         11 int main() C => void maxvet
  0.0        0.053        0.053         5          0         11 void maxvet() C
```

[5]It is important to point out that a flat profile can be constructed from a path profile, but not vice versa.
[6]Default value is 0, *callpath profiling* is disabled.



```
Metric: TIME
Value: Exclusive
Units: seconds

    1,9                       int main() C [{main.c} {5,1}-{62,1}]
                       0,002 | void matvet(double *, double *, double *, int, int) C [{matvet.c} {2,1}-{16,1}]
                      1,9E-5 | void maxvet(double *, int, double *) C [{max.c} {2,1}-{16,1}]
```

Figure 4: Snapshot of ParaProf

Because instrumentation overhead will increase with the depth of the callpath, you should use the shortest call path that is sufficient.

N.B. Note that in order to use *callpath profiling* feature, you have to configure TAU with the -PROFILECALLPATH option.

5.3 Parallel profiling

The TAU performance measurement system is capable of producing parallel profiles for thousands of processes consisting of hundreds of events by means `ParaProf` profile analysis tool. In order to get an example of analysis that `ParaProf` displays, Figure 4 shows the snapshot of flat profile for the example code proposed in appendix B.

6 TAU Custom Profiling

6.1 Selectively profiling

TAU allows you to customize the instrumentation of a program by using a selective instrumentation file. This facility is only available when using source-level instrumentation (PDT). The instrumentation file is used to manually control which parts of the application are profiled and how they are profiled. If you are using one of the TAU compiler wrapper scripts to instrument your application you can use the `-tau_options=-optTauSelectFile=<file>` option to enable selective instrumentation.

6.2 Manual and automatic instrumentation

In this section, we discuss about TAU instrumentation comparing the main two methods: compiler instrumentation and source instrumentation. The first one is named also *manual instrumentation*, because it is used when the TAU measurements are inserted directly in the code by user, while the second one is also named *automatic instrumentation*, because TAU measurement are inserted by means PDT. Both methods produce an instrumented object code as we described in section 3.2.2 and 3.2.3, but they use different TAU levels because they operate on different compiling phases. The *manual* method instruments

the source code *during* the compilation, producing an instrumented object code (files .o), while *automatic* method operates *before* the compilation producing the instrumented source code (i.e. files .inst.c) and then produces the instrumented object code (files .o). Both methods can produce the same profile data.
At user level, the *automatic* instrumentation could be more helpful, expecially when the source code is long and complicated. The option `-optKeepFiles` allows to save the instrumented source files, so you can see and/or modify them as you prefer (i.e. moving TAU measurements towards critical sections). If you have modified the instrumented source files, you can recompile them by means *compiler* instrumentation. (In this way, you use for compiling before *automatic* instrumentation and after the *manual*). This artifice allows to avoid a (time consuming) manual instrumentation of your source code as well as to modify it. So, you have realized a "personal" and quick custom profiling.

6.3 Reducing overhead with TAU_THROTTLE

To reduce overhead and increase accuracy, it is recommended that some functions not be intrumented, those that are short running functions and are called many times. A good rule of thumb is to exclude those functions that have less than 10 microseconds per call inclusive time, and are called more than 100,000 times (`numcalls > 100,000 & usecs/call < 10`). By default, the TAU measurement system will throtthle these functions at runtime (and these functions will appear in a flat profile with "THROTTLED" marker). A throttled function will still have hooks into the measurement system, but those hooks will be disabled, reducing, but not eliminating the overhead. Throttling may be turned off by setting:

```
user@home:> TAU_THROTTLE 0
```

To change the values of `numcalls` and `usecs/call` you can set the following environment variables:

```
user@home:> TAU_THROTTLE_NUMCALLS 200,000
user@home:> TAU_THROTTLE_PERCALL 5
```

7 Helpful Advice for TAU

7.1 Bug on Generating data profile

TAU can be built in many ways to precisely suit your application and runtime system used. If your application don't make any MPI calls (`MPI_Init()/MPI_Finalize()`) and it linked in the MPI libraries then profiling execution causes the mismatch and the `profile.-1.0.0` file. In general, if your application makes no MPI calls, you will have to choose a stub makefile that does not have a -mpi in its name.

A fix for this bug is described in:
`http://www.nic.uoregon.edu/pipermail/tau-users/2007-June/000106.html`
Otherwise, you can rename the file `profile.-1.0.0` in `profile.0.0.0` before profiling it.

7.2 Helpful Links

- http://www.ruyk.com/tech/?p=49
- http://www.psc.edu/general/software/packages/tau/
- http://acts.nersc.gov/tau/documents/usingTAU.html
- http://www.cs.uoregon.edu/Research/tau/docs/newguide/bk05rn01.html
- https://modelingguru.nasa.gov/docs/DOC-1646
- http://www.cs.uoregon.edu/Research/paraducks/proj/doe2000/pi00_report.html
- http://www.cs.uoregon.edu/Research/paracomp/tau/tautools/docs/instr.html

Appendix

A Source instrumentation via Makefile

If the application that you have to profile has an its ownn Makefile, you can use it changing suitably the instructions. For example, we compile a C program described in appendix B composed by three functions written in three files: `main.c`, `matvet.c` and `max.c` (for details see section B.1 in appendix B).

```
#Begin Makefile

OUTDIR=.
INTDIR=.

# --------------- TAU directory and options --------------------
TAUROOTDIR      = /usr/local/tau-2.20.2/
TAU_OPTIONS = -optPreProcess -optKeepFiles -optVerbose
LIBS        = $(TAU_LIBS) -lmpich -lmpl -L/usr/local/lib

# --------------- Compile and link commands --------------------
CFLAGS = $(TAU_INCLUDE) $(TAU_DEFS)
CC = tau_cc.sh -tau_makefile=./TAU.Makefile -tau_options=$(TAU_OPTIONS)

# ------------------- Program name -----------------------------
PROG = myprog_tau.exe
.SUFFIXES: .c.o

# --------------- object list  ---------------------------------
OBJS= \
$(INTDIR)/main.o \
$(INTDIR)/matvet.o \
$(INTDIR)/max.o

# --------------- Object files creation rules -------------------
main.o : main.c
        $(CC) $(CFLAGS) -c  main.c
matvet.o : matvet.c
        $(CC) $(CFLAGS) -c  matvet.c
max.o : max.c
        $(CC) $(CFLAGS)-c  max.c

# ----------------"all" rule------------------------------------
all: $(PROG)

$(OUTDIR)/$(PROG): $(OBJS)
```

```
        $(CC) $(CFLAGS) -o $(OUTDIR)/$(PROG) $(OBJS) $(LIBS) -lm

clean:
        /bin/rm -f  ./$(PROG)
        /bin/rm -f  ./*.o ./*.inst.c ./*.pdb

#End Makefile
```

B The example C code

B.1 Original source

In this appendix is shown the source code used in this report as example (see section 5.1). It is C program composed by three functions saved in three different files named: main.c, matvet.c and max.c rispectively. The program carries out five matrix-vector product and the corresponding maximum value of vector product.

```
============================== main.c ========================
int main()
{
double *A,*x,*y,max;
int i,j,c,m,n,calls;

m = 100;
n=500;
calls = 5;
A = (double*)calloc(m*n,sizeof(double));
x = (double*)calloc(n,sizeof(double));
y = (double*)calloc(m,sizeof(double));

for (c=1; c<=calls; c++)
{    for (i=0; i<m; i++)
     {
          for (j=0; j<n; j++)
          {
              *(A+j+i*n) = (double) c*((j+1)*n);
          }
     }
     for (j=0; j<n; j++)
          *(x+j) = (double)c*j;
     matvet(A,x,y,m,n);
     printf(" *** --------------------------------------------------------- *** \n");
     printf("MATRIX VECTOR PRODUCT n. %d \n", c);
     printf("INPUT DATA: \n");
     printf("\n");
     for (i=0; i<m; i++)
     {
          for (j=0; j<n; j++)
          {
               printf("A(%d,%d)=%5.2f\t",i,j, *(A+j+i*n) );
          }
          printf("\n");
     }
     printf("\n");
     for (j=0; j<n; j++)
          printf("x(%d)=%5.2f\t",j,*(x+j));
     printf("OUTPUT DATA: \n");
     for (j=0; j<n; j++)
     {
          printf("y(%d)=%5.2f\t",j, *(y+j) );
     }
     printf("\n");
     maxvet(y,n,&max);
     printf("\n MAXIMUM VALUE %5.2f \n",max);
     printf(" *** ----------------------------------------------------------- *** \n");
}
return 0;
}

======================== matvet.c ========================

void matvet(double *A, double *x, double *y, int m, int n)
{
int i,j;

for (i=0; i<m; i++)
{
     *(y+i)=0.;
     for (j=0; j<n; j++)
     {
          *(y+i) += *(A+j+i*n)* *(x+j);

     }
}
return;
}

======================== max.c ========================

void maxvet(double *vet, int n, double *max)
{
int i;

*(max) = *(vet);

for (i=1; i<n; i++)
{
     if (*(vet+i) > *(max))
     {
         *(max) = *(vet+i);
     }
}
return;
}
```

B.2 Instrumented source

In this section is shown the instrumented version of the example code in appendix B produced by means the Makefile reported in appendix A. In order to preserve instrumented source file you have to set TAU options with `-optKeepFiles`.

```
======================= main.inst.c ========================

#include <Profile/Profiler.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main()
{
     TAU_PROFILE_TIMER(tautimer, "int main() C [{main.c} {5,1}-{62,1}]", " ", TAU_DEFAULT);
#ifndef TAU_MPI
#ifndef TAU_SHMEM
  TAU_PROFILE_SET_NODE(0);
#endif /* TAU_SHMEM */
#endif /* TAU_MPI */
        TAU_PROFILE_START(tautimer);
{
double *A,*x,*y,max;
int i,j,c,m,n,calls;

m = 100;
n=500;
calls = 5;
A = (double*)calloc(m*n,sizeof(double));
x = (double*)calloc(n,sizeof(double));
y = (double*)calloc(m,sizeof(double));

for (c=1; c<=calls; c++)
{    for (i=0; i<m; i++)
     {
           for (j=0; j<n; j++)
           {
               *(A+j+i*n) = (double) c*((j+1)*n);
           }
     }
     for (j=0; j<n; j++)
          *(x+j) = (double)c*j;
     matvet(A,x,y,m,n);
     printf(" *** ---------------------------------------------------------- *** \n");
     printf("MATRIX VECTOR PRODUCT n. %d \n", c);
     printf("INPUT DATA: \n");
     printf("\n");
     for (i=0; i<m; i++)
     {
          for (j=0; j<n; j++)
          {
               printf("A(%d,%d)=%5.2f\t",i,j, *(A+j+i*n) );
          }
          printf("\n");
     }
     printf("OUTPUT DATA: \n");
     for (j=0; j<n; j++)
          printf("x(%d)=%5.2f\t",j,*(x+j));

     printf("OUTPUT: \n");
     for (j=0; j<n; j++)
     {
          printf("y(%d)=%5.2f\t",j, *(y+j) );
     }
     printf("\n");
     maxvet(y,n,&max);
     printf("\n MAXIMUM VALUE %5.2f \n",max);
     printf(" *** ----------------------------------------------------------- *** \n");
}
{ int tau_ret_val =  0;  TAU_PROFILE_STOP(tautimer); return (tau_ret_val); }
}
        TAU_PROFILE_STOP(tautimer);
}

======================== matvet.inst.c ========================

#include <Profile/Profiler.h>
void matvet(double *A, double *x, double *y, int m, int n)
{
        TAU_PROFILE_TIMER(tautimer, "void matvet(double *, double *, double *, int, int) C [{matve
        TAU_PROFILE_START(tautimer);
{
int i,j;

for (i=0; i<m; i++)
{
     *(y+i)=0.;
     for (j=0; j<n; j++)
     {
          *(y+i) += *(A+j+i*n)* *(x+j);

     }
}
{  TAU_PROFILE_STOP(tautimer); return; }
}
        TAU_PROFILE_STOP(tautimer);
}
```

```
======================== max.inst.c ========================

#include <Profile/Profiler.h>
void maxvet(double *vet, int n, double *max)
{
        TAU_PROFILE_TIMER(tautimer, "void maxvet(double *, int, double *) C [{max.c} {2,1}-{16,1}]
        TAU_PROFILE_START(tautimer);
{
int i;

*(max) = *(vet);

for (i=1; i<n; i++)
{
     if (*(vet+i) > *(max))
     {
         *(max) = *(vet+i);
     }
}
{  TAU_PROFILE_STOP(tautimer); return; }
}
        TAU_PROFILE_STOP(tautimer);
}
```

References

[1] S.S. Shende, A.D. Malony: *The TAU parallel performance system*, The International Journal of High Performance Computing Applications, Vol. 20, n. 2, pp. 287-311, 2006

[2] *TAU Homepage* `URL: http://www.cs.uoregon.edu/Research/tau/home.php`

[3] S. Shende A.D. Malony, J. Cuny, K. Lindlan, P. Beckman, S. Karmesin: *Portable Profiling and Tracing for Parallel, Scientific Application using C++*, Proceeding of SPDT 98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, 1998

[4] S. Shende: *Hands-on Practical Parallel Application Performance Engineering using PAPI, PerfSuite, Scalasca, Vampir and TAU (slides)* Supercomputing 10 Conference, New Orleans, Louisiana, USA, November 15, 2010

[5] *PDT (Program Database Toolkit) Homepage*, `URL:http://www.acl.lanl.gov/pdtoolkit`

[6] Advanced Computing Laboratory, Los Alamos National Laboratory: *PDT: Program Database Toolkit*, Supercomputing '99 flyer, Los Alamos National Laboratory Publication LALP-99-204, November 1999.

[7] K. Lindlan, J. Cuny, A. D. Malony, S. Shende, P. Beckman: *An IL Converter and Program Database for Analysis Tools*, Proceeding of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, p.153, 1998

[8] B. Mohr: *DUCTAPE. Poster*, International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98), December 1998