TAU Performance System training course

Practical Sheet 3: Instrumenting and Running a test programme: Advanced Topics

This practical sheet expands on Sheet 2. It covers topics that are not well-documented in the TAU documentation (https://www.cs.uoregon.edu/research/tau/docs.php). This sheet will cover the following

1. Runtime selective instrumentation
2. Compile-time selective instrumentation
3. Snapshot profiling

3.1) Runtime selective instrumentation

Return to the example from Section 2.2 of the practical sheet. Open the results from the 'default' Source-based instrumentation profile. Left click on the results for node 0, and scroll through the list. Notice how there are hundreds of functions in the list, most of which run for a negligible amount of time, or even no time at all.

The collection of timing results from these unimportant functions incurs a cost to the overall execution of the AMG2013 code compared to a 'pristine' version without calls to the TAU profiling API. The user can prevent the collection of results from unimportant functions by using a runtime selective instrumentation file. Recall from Practical Sheet 1 the syntax of a TAU Selective Instrumentation File for source files:

```
BEGIN_INCLUDE_LIST
<Function name>
END_INCLUDE_LIST
```

A similar syntax exists for `BEGIN/END_FILE_EXCLUDE_LIST`. To use this approach at runtime, the environment variable `TAU_SELECT_FILE` must be set to point to a file with the above syntax, listing the functions to include and/or exclude from profiling. Note that a complete list of *all* functions between the `INCLUDE` and `EXCLUDE` list is not required. If only a list of functions to include is given, all other function will be excluded, and vice versa.

Creating a list of, for example, 20 functions to include in results sounds the best approach therefore. Function names, however, must contain, in the case of a C-like function a *full* function name (i.e. `<return type> <function name> (<function arugments>`). For C functions, a trailing ' C' after the `(<function arugments>`) is also reuired

Create a file called `select_include.tau`, and start the file with `BEGIN_INCLUDE_LIST`. Return to the node 0 profile by left-clicking on 'node 0' in the stacked bar chart. Paste the names of the top-consuming functions in Table 1 into `select_include.tau`.

Function Name	Source File
int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix *, int *, hypre_ParCSRMatrix *, int *, int, int *, int, double, int, int, int *, hypre_ParCSRMatrix **)	par_multi_interp.c
MPI_Waitall()	N/A
int hypre_BoomerAMGBuildCoarseOperator(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix **)	par_rap.c
int hypre_CSRMatrixMatvec(double, hypre_CSRMatrix *, hypre_Vector *, double, hypre_Vector *)	csr_matvec.c
hypre_ParCSRRelax_L1	par_relax_more.c
int hypre_BoomerAMGCoarsenRuge(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int, int **)	par_coarsen.c
double hypre_SeqVectorInnerProd(hypre_Vector *, hypre_Vector *)	vector.c
char *hypre_CAlloc(int, int)	hypre_memory.c
Int hypre_CSRMatrixCopy(hypre_CSRMatrix *, hypre_CSRMatrix *, int)	csr_matrix.c
MPI_Allreduce()	N/A

Table 1: Top 10 consuming function names and source file location

Now run the same command as for Section 2.2, but add an environment variables to point to the selective instrumentation file

```
tau_defaults
export TAU_SELECT_FILE=select_include.tau
mpirun -n 4 amg2013 -27pt -n 100 100 100 -P 2 2 1 >
log.amg2013.27pt.100 &
```

Opening the results in paraprof shows that only the 10 functions listed in Table 1 are included in the list of results. Removing the collection of timing data from 'unimportant' functions makes the results obtained more straightforward to view in paraprof. It is also straightforward to implement, requiring no recompilation of the code and can be easily used to run multiple simulations with different functions included in results.

Paraprof contains an automated method to generate runtime selective instrumentation files. From the stacked bar window, click on File → Create Selective Instrumentation File. Something like Figure 1 will appear. This tool will create an `EXCLUDE` list of everything that is to be removed from profiling. The option to remove Throttled routines is self-explanatory (see Practical Sheet 2). Excluding lightweight routines using the criteria below works in the same way as the `TAU_THROTTLE_PERCALL` and `TAU_THROTTLE_NUMCALLS` environment variables.

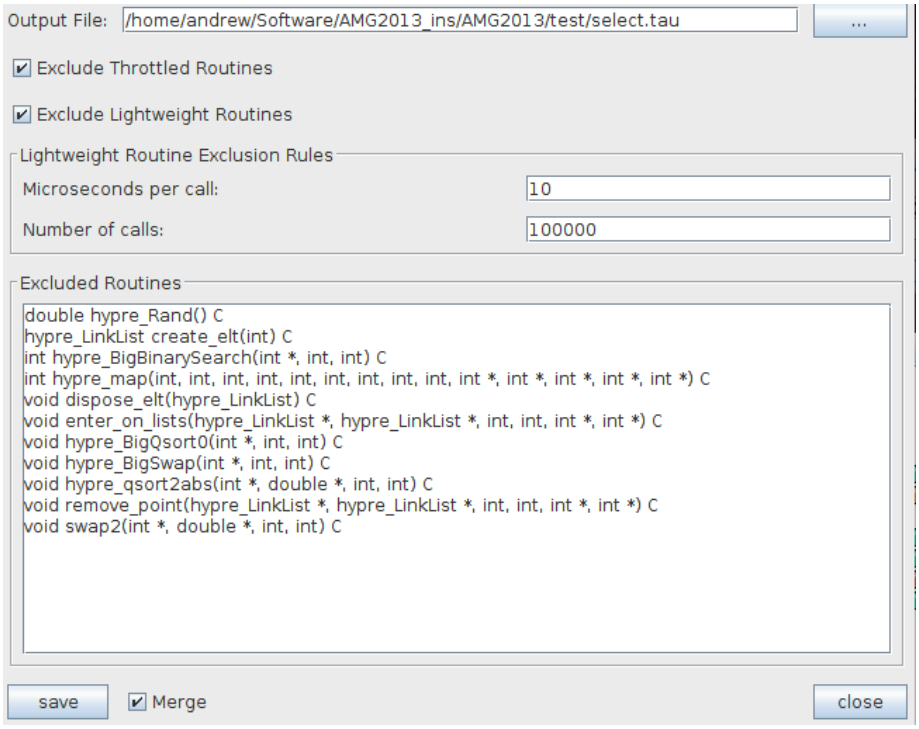

Figure 1: Example output from Selective Instrumentation File Generator

Note this method does not put calls to MPI library functions in this list, unless they have been throttled.

Runtime selective instrumentation has its limits. Firstly, the extra TAU Instrumentation API calls remain in the compiled code, even if data is not collected from these calls. Therefore, this approach cannot be used, for example, to exclude all functions from profiling and thus generate a 'pristine' experiment.

Secondly, enabling callpath profiling for the above by typing

```
export TAU_CALLPATH=1
export TAU_CALLPATH_DEPTH=15
mpirun -n 4 amg2013 -27pt -n 100 100 100 -P 2 2 1 >
log.amg2013.27pt.100 &
```

gives results for callpaths but *only those that link functions in the include list*. Therefore, incomplete callpaths are shown, because, for example, the `main` function is not in the include list.

Thirdly, runtime selective instrumentation can only take a list of functions to include and/or exclude. It cannot be used, as described in Practical Sheet 1, to control source file or loop inclusion. This requires a compile-time selective instrumentation file, which will be looked at in the next section

3.2) Compile-time selective instrumentation

Returning to Practical Sheet 1, recall that the syntax for compile-time selective instrumentation consists of

```
BEGIN_FILE_INCLUDE_LIST
<Sourcefile name>
END_FILE_INCLUDE_LIST

BEGIN_INCLUDE_LIST
<full function name>
END_INCLUDE_LIST

BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION
```

To begin this section, create a new copy of AMG2013. Return to the root directory (`cd $HOME`) and create a new folder called `AMG2013_select`. Copy amg2013.tar.gz to this directory, and unpack it from this new directory. Follow the instructions in Section 2.2 to edit the `Makefile.include` to change the compiler and flags. Create a new file called `select_compile.tau` in the root AMG2013 directory. Edit the environment variable `TAU_OPTIONS` to tell TAU to use this file at compile time

```
export TAU_OPTIONS="-optPDTInst -optRevert –optVerbose –
optTauSelectFile=/home/tautraining/AMG2013_select/AMG2013
/select_compile.tau"
```

Open the selective instrumentation file and type the following with the source file names from Table 1

```
BEGIN_FILE_INCLUDE_LIST
par_multi_interp.c
par_rap.c
csr_matvec.c
par_relax_more.c
par_coarsen.c
vector.c
hypre_memory.c
csr_matrix.c
END_FILE_INCLUDE_LIST
```

Then type `make` to compile. To show how compilation now works with the selective file, save the output of `make` using

```
make &> output &
```

Then, search for the name a source file that is included in the output using

```
egrep –A1 –B1 ‘par_multi_interp.c’ output
```

to see that `par_multi_interp.c` has been compiled using the process seen in Section 2.2. Search for a file that is not included, such as `amg2013.c` using the above command, and TAU will show

```
Debug: File excluded from instrumentation. Copying
original file as instrumented file.
Executing> cp amg2013.c amg2013.inst.c
```

demonstrating that TAU has skipped all the steps from Section 2.2 and will be compiling `amg2013.c` without any instrumentation. Note in the list of functions in Table 1, that `hypre_ParCSRRelax_L1` does not have a return type before or argument list after it. This difference has occurred because the source file, `par_relax_more.c` has not been compiled with PDT and therefore TAU has reverted to Compiler-based instrumentation. This is confirmed in the compilation log. Typing

```
egrep –A20 –B20 ‘cparse4101.*par_relax_more.c’ output
```

shows the lines from `output` relating to the compilation of `par_relax_more.c`. The output contains the line

```
PDT: Trying edgcpfe4101 without CSTOPT=--c99
```

that shows initial PDT instrumentation with the C99 standard fails, so this flag is removed. However, this also fails with

```
“par_relax_more.c”, line 1586: error: expected an
expression
     if (fabs(l1_norm[i] < DBL_EPSILON)
                           ^

“par_relax_more.c”, line 1857: error: expected an
expression
     if (fabs(l1_norm[i] < DBL_EPSILON)
                           ^
```

Scrolling down shows that TAU catches the error:

```
Error: Reverting to a Regular Make
Debug: PDT failed, switching to compiler-based
instrumentation
Debug: Using compiler-based instrumentation
Debug: Compiling with Instrumented Code
Executing> gcc –c –O2 par_relax_more.c <Includes> <TAU
headers>
```

The last line is the ‘compiler-based instrumentation’ part. This error does not occur during regular compilation with `mpicc`. The Edison Design Group front-end compiler in PDT has stricter error criteria, so often encounters errors which do not appear with non-TAU

compilation. The error above relates to the macro `DBL_EPSILON` defined in the standard C library header, `float.h`. The value of `DBL_EPSILON` is defined as `1e-15`, so to correct this error the macro can be replaced with its value.

Open `parcsr_ls/par_relax_more.c` using `gedit`

```
gedit parcsr_ls/par_relax_more.c &
```

and then use Ctrl+H to open up Find and Replace, and replace the two occurrences of `DBL_EPSILON` with `1e-15`. Save the file and return to the Terminal. Type

```
make clean
make &> output_fixed &
```

and then run the same `egrep` command on the new make output file. Notice that no errors occur and that PDT runs successfully.

Now, from the `test` directory run

```
tau_defaults
mpirun -n 4 amg2013 -27pt -n 100 100 100 -P 2 2 1 >
log.amg2013.27pt.100 &
```

Packing and viewing the results of the above will show that more functions have been instrumented than in Section 3.1. However, no functions that did not appear in, say, Section 2.2 with full instrumentation appear. To get a more detailed breakdown of where time is spent in each function, it is worth adding `BEGIN_INSTRUMENT_SECTION` part to `select_compile.tau`. At this point, no information is known about which parts of each function in each source file take longest to complete. Thus, it is best to instrument the outermost `for` and `while` loops in each function in each of the source files in the File Include list by adding

```
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION
```

Name	Exclusive TIME	Inclusive TIME ▽
TAU application	2.514	122.216
int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix *, int *, hypre_ParCSRMatrix *, int *, int, int *, int, double, int, int, int *, hypre_ParCSRMatrix **) C [{par_multi_interp.c} {22,1}-{1907,1}]	0.004	73.608
Loop: int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix *, int *, hypre_ParCSRMatrix *, int *, int, int *, int, double, int, int, int *, hypre_ParCSRMatrix **) C [{par_multi_interp.c} {1512,7}-{1737,7}]	47.891	52.23
Loop: int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix *, int *, hypre_ParCSRMatrix *, int *, int, int *, int, double, int, int, int *, hypre_ParCSRMatrix **) C [{par_multi_interp.c} {1411,5}-{1498,7}]	18.616	18.618
int hypre_BoomerAMGBuildCoarseOperator(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix **) C [{par_rap.c} {205,1}-{1845,1}]	0.098	11.01
int hypre_CSRMatrixMatvec(double, hypre_CSRMatrix *, hypre_Vector *, double, hypre_Vector *) C [{csr_matvec.c} {27,1}-{208,1}]	0.009	8.7
Loop: int hypre_CSRMatrixMatvec(double, hypre_CSRMatrix *, hypre_Vector *, double, hypre_Vector *) C [{csr_matvec.c} {171,7}-{190,7}]	8.308	8.308
int hypre_BoomerAMGCoarsenHMIS(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int **) C [{par_coarsen.c} {1954,1}-{1974,1}]	0	8.109
int hypre_BoomerAMGCoarsenRuge(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int, int **) C [{par_coarsen.c} {867,1}-{1929,1}]	0.076	7.276
MPI_Waitall()	6.918	6.918
int hypre_ParCSRRelax_L1(hypre_ParCSRMatrix *, hypre_ParVector *, double, double, double *, hypre_ParVector *, hypre_ParVector *, hypre_ParVector *) C [{par_relax_more.c} {1876,1}-{2157,1}]	0.007	6.819
Loop: int hypre_ParCSRRelax_L1(hypre_ParCSRMatrix *, hypre_ParVector *, double, double, double *, hypre_ParVector *, hypre_ParVector *, hypre_ParVector *) C [{par_relax_more.c} {1985,7}-{2055,7}]	6.359	6.359
Loop: int hypre_BoomerAMGBuildCoarseOperator(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix **) C [{par_rap.c} {1397,4}-{1718,4}]	5.498	5.498
double hypre_SeqVectorInnerProd(hypre_Vector *, hypre_Vector *) C [{vector.c} {417,1}-{437,1}]	0	4.027
Loop: double hypre_SeqVectorInnerProd(hypre_Vector *, hypre_Vector *) C [{vector.c} {433,4}-{434,38}]	4.027	4.027
Loop: int hypre_BoomerAMGCoarsenRuge(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int, int **) C [{par_coarsen.c} {1016,4}-{1024,4}]	3.875	3.875
MPI_Allgather()	3.065	3.065
char *hypre_CAlloc(int, int) C [{hypre_memory.c} {107,1}-{137,1}]	2.967	2.967
Loop: int hypre_BoomerAMGBuildCoarseOperator(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix **) C [{par_rap.c} {1121,4}-{1350,4}]	2.636	2.637

Figure 2: Example output from compile-time selective instrumentation of loops

Type `make clean` to start a new build. and then run `make`. Rerun the program and load the results in paraprof. Now, looking at Figure 2, instead of individual functions, individual loops of the function are identified. For example, `int hypre BoomerAMGBuildMultipass` is broken down into two main loops; one at line numbers 1512-1737 in `par_multi_interp.c`, and one at line numbers 1411-1498 in `par_multi_interp.c`. In this example, breaking down the Inclusive time shows that of 73 s spent in `int hypre_BoomerAMGBuildMultipass`, 68 s is spent exclusively in these two outer loops. Further inspection of the loop between lines 1512-1737 would therefore yield information about a loop that consumes 48 of the 122 s runtime (the Inclusive value for .TAU application')

To inspect the code further, phase-based profiling must be used. This breaks chunks of code into 'phases' in which the time spent in the phase of code is measured. The syntax for adding this to the compile-time selective instrumentation file consists of

```
BEGIN_INSTRUMENT_SECTION
static phase name="<name>" file="<source file>"
line=<start> to line=<end>
END_INSTRUMENT_SECTION
```

where the keyword `static` means that all instances of this phase are registered as the same 'event' and this timing is collated. The alternative is to make a phase `dynamic`, which means that each instance of the phase is a separate 'event' and timing of each event is therefore separated. Dynamic phases occur an additional overhead, but could be useful to determine if a particular iteration of, for example, a loop, takes longer than others.

Line numbers 1512-1737 in `par_multi_interp.c` contains the following loop

```
1512            for (pass=2; pass < num_passes; pass++)
1513            {
1514-1736       <body of loop>
1737            }
```

Automatic loop instrumentation using TAU can only instrument the outermost loops in a particular function. This above loop contains several sub-loops. To break down the time spent in these sub-loops, they need to be labelled as phases. The two loops that are likely to use the most time are those with significant computation. Creating the following phases captures two such loops:

```
static phase name="pass_array_to_diag_par_multi_interp"
file="par_multi_interp.c" line=1619 to line=1655
```

which contains the partial loop

```
for (i=ns; i < ne; i++)
{
 i1 = pass_array[i];
<code>
```

```
            tmp_marker_offd[j1] = i1;
        }
```

and

```
static phase name="a_diag_par_multi_interp"
file="par_multi_interp.c" line=1656 to line=1685
```

which contains

```
        for (j=A_diag_i[i1]+1; j < A_diag_i[i1+1]; j++)
        {
            j1= A_diag_j[j];
        <code>
                sum_N += A_diag_data[j];
            }
        }
```

Note that phases cannot overlap with other instrument sections. For example, `loops routine="#"` overlaps with the two above sections, as the outer loop from lines 1512-1737 contains the two phases named above. Recompile the code (remembering to `make clean`) and run again. Open the results in paraprof and bring up the bar chart for node 0. The results should look something like Figure 3.

Name	Exclusive ...	Inclusive T...
.TAU application	2.073	145.712
int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix *, int *, hypre_ParCSRMatrix *, int *, int, int *, int, double, int, int, int *, hypre_ParCSRMatrix **) C [{par_multi_interp.c} {22,1}-{1907,1}]	28.77	101.119
a_diag_par_multi_interp	39.574	39.574
pass_array_to_diag_par_multi_interp	21.637	21.637
int hypre_BoomerAMGCoarsenHMIS(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int **) C [{par_coarsen.c} {1954,1}-{1974,1}]	0	10.803
int hypre_BoomerAMGBuildCoarseOperator(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, hypre_ParCSRMatrix **) C [{par_rap.c} {205,1}-{1845,1}]	6.204	9.73
int hypre_BoomerAMGCoarsenRuge(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int, int **) C [{par_coarsen.c} {867,1}-{1929,1}]	8.494	9.031
MPI_Waitall()	8.831	8.831
int hypre_CSRMatrixMatvec(double, hypre_CSRMatrix *, hypre_Vector *, double, hypre_Vector *) C [{csr_matvec.c} {27,1}-{208,1}]	7.854	7.854
MPI_Allgather()	6.902	6.902
int hypre_ParCSRRelax_L1(hypre_ParCSRMatrix *, hypre_ParVector *, double, double, double *, hypre_ParVector *, hypre_ParVector *, hypre_ParVector *) C [{par_relax_more.c} {1876,1}-{2157,1	6.393	6.497
double hypre_SeqVectorInnerProd(hypre_Vector *, hypre_Vector *) C [{vector.c} {417,1}-{437,1}]	3.014	3.014
char *hypre_CAlloc(int, int) C [{hypre_memory.c} {107,1}-{137,1}]	2.362	2.362
int hypre_BoomerAMGCoarsenPMIS(hypre_ParCSRMatrix *, hypre_ParCSRMatrix *, int, int, int **) C [{par_coarsen.c} {1992,1}-{2645,1}]	0.191	1.772
MPI_Allreduce()	1.275	1.275
int hypre_CSRMatrixCopy(hypre_CSRMatrix *, hypre_CSRMatrix *, int) C [{csr_matrix.c} {298,1}-{334,1}]	0.393	0.393
MPI_Finalize()	0.355	0.355
int hypre_SeqVectorCopy(hypre_Vector *, hypre_Vector *) C [{vector.c} {292,1}-{312,1}]	0.328	0.328
int hypre_CSRMatrixMatvecT(double, hypre_CSRMatrix *, hypre_Vector *, double, hypre_Vector *) C [{csr_matvec.c} {220,1}-{420,1}]	0.237	0.237
int hypre_SeqVectorAxpy(double, hypre_Vector *, hypre_Vector *) C [{vector.c} {389,1}-{411,1}]	0.225	0.225

Figure 3: Example of phase profiling – Inclusive Time

In Figure 3, `a_diag_par_multi_interp` uses the most exclusive time, with 39.6 s of the 145.7 s runtime spent in that phase. The other phase, `pass_array_to_diag_par_multi_interp`, uses 21.6 s of exclusive time. These two phases account exclusively for 61 s of the 101.1 s inclusive time spent in `int hypre BoomerAMGBuildMultipass` and its child functions. In Figure 2, the loop between lines 1512-1737 uses a similar fraction of the functions inclusive time: 47.9 s of 73.6 s, or 65 %. Thus, the two phases identified represent the majority of the time spent in the loop.

Note also that total runtime with only 2 phases instrumented and no loops instrumented is longer at 145.7 s than that for the experiment with all outer loops instrumented (122 s), demonstrating that phase profiling does incur significant overheads.

3.3) Snapshot profiling

All of the previous profiling experiments have given a value to the total time spent in a particular function *at the end of execution of the program*. This section will focus on an approach that allows the user to examine the cumulative time spent in functions *at specific points in the code*. That is, the user can get a 'snapshot' of the total time spent in each instrumented function, loop or phase between the start of execution and the point in the code at which the snapshot is taken. By taking multiple, regular snapshots of a code at the same point (for example, in an iterative loop), the *differential time* spent between snapshots can be calculated and thus the user can investigate *when* (during execution) increases in execution time occur, not just *where* they occur.

Snapshot profiling is hidden away in TAU's Instrumentation API. To use it, the user must insert the relevant API calls into a program's source code at any point and then compile the program. There are two API calls that write snapshots. For C/C++, the first is

```
TAU_PROFILE_SNAPSHOT(name);
```

where name is of type `char*`. This call writes a snapshot profile with label name. Multiple snapshots from the same node are merged together by default to form a `snapshot.*` file. The second call is

```
TAU_PROFILE_SNAPSHOT_1L(name, number);
```

which works as for the first call, with `number` of type `int` appended to the snapshot label. This form works well within loops to provide regular snapshots, where `number` could be the loop index or a value derived from it. The above calls require that

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

is placed at the top of the source file so that the API calls are defined and thus able to be used.

The above snapshot API calls need to be combined with the TAU instrumentation workflow in order for the instrumented functions to be captured in snapshots. To start, create a new instance of `AMG2013` for this section

```
mkdir ~/AMG2013_snapshot
cp ~/amg2013.tgz AMG2013_snapshot
cd AMG2013_snapshot
tar xvzf amg2013.tgz
cd AMG2013
```

Follow the instructions in Section 2.2 to edit the `Makefile.include` to change the compiler and flags. Now copy the `select_compile.tau` file from the `AMG2013_select/AMG2013` directory that was used in the Section 3.2, and comment

out (# before every line) the lines between BEGIN/END_INSTRUMENT_SECTION but keep the source file names in the BEGIN/END_INCLUDE_FILE list.

Recall Section 2.2, which details the commands that tau_cc.sh uses to perform PDT source-based instrumentation. Snapshot calls need to be inserted into a *.inst.c files once it is generated (example below)

```
Debug: Instrumenting with TAU
Executing> <path to TAU install>/tau_instrumentor
amg_linklist.pdb amg_linklist.c -o amg_linklist.inst.c
```

By default, TAU_OPTIONS does not keep the *.inst.c files (or any other intermediate files). To keep hold of these files during PDT instrumentation, TAU_OPTIONS needs to be adjusted. Enter the following commands

```
tau_defaults
export TAU_OPTIONS=
"-optPDTInst -optRevert –optVerbose –optKeepFiles -
optTauSelectFile=/home/tautraining/AMG2013_snapshot/AMG20
13/select_compile.tau"
```

and then compile the code as normal using

```
make &> output &
```

Now, recall from Section 3.2 that par_multi_interp.c contained two loops that contributed to the majority of execution time. Navigate to the directory this source file is located in, parcsr_ls. Locate the *.inst.c file, par_multi_interp.inst.c and open it using gedit. The first change to make to the file is to add

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

below the first line of the file, #include <Profile/Profiler.h>. Now, locate the beginning of the function declaration, int hypre_BoomerAMGBuildMultipass(hypre_ParCSRMatrix ...). Go through the list of variables until the line

```
int       *tmp_array, *tmp_array_offd;
```

and add the following variables beneath

```
char*          snapshot_nloop1 = "Snapshot_nloop1";
char*          snapshot_nloop2 = "Snapshot_nloop2";
int            snapshot_nloop1_iter;
int            snapshot_nloop2_iter;
```

These variables will be used to generate two TAU_PROFILE_SNAPSHOT_1L calls in the two biggest-consuming loops identified in Section 3.2. The first of these loops begins around

line 1411 (N.B. the line numbers have changed from the original source file, due to added lines from TAU instrumentation) with

```
   else /* no distinction between positive and negative offdiagonal element */
   {
   /* determine P for points of pass 1, i.e. neighbors of coarse points */
    pass_length = pass_pointer[2]-pass_pointer[1];
#define HYPRE_SMP_PRIVATE <OpenMP variables>
#include "../utilities/hypre_smp_forloop.h"
    for (k = 0; k < num_threads; k++)
    {
       <loop body>
```

Since OpenMP is not being used, the statement `for (k = 0; k < num_threads; k++)` has no effect. In this case, the main body of the loop is located further down after, reading

```
for (i=ns; i < ne; i++)
{
```

Creating a snapshot profile for every iteration of this loop would be foolish, as too many snapshots would be created, generating output files too large to analyse. Instead, generate a snapshot every 20000 iterations using

```
snapshot_nloop1_iter = i;
if ( ( i - ns) % 20000 == 0)
       TAU_PROFILE_SNAPSHOT_1L(snapshot_nloop1, snapshot_nloop1_iter)
```

The second loop to create snapshots for starts around line 1512 with

```
for (pass = 2; pass < num_passes; pass++)
{

   if (num_procs > 1)
   {
```

Again, the main body of the loop begins similarly to the first one, with

```
    for (i=0; i < num_cols_offd; i++)
         tmp_marker_offd[i] = -1;

     for (i=ns; i < ne; i++)
     {
    /*for (i=pass_pointer[pass]; i < pass_pointer[pass+1]; i++)
    {*/
       i1 = pass_array[i];
       sum_C = 0;
       sum_N = 0;
```

beginning at around line 1615. Add the call to the snapshot at the beginning of the loop with the same interval (20000) as before

```
snapshot_nloop2_iter = i;
if ( ( i - ns) % 20000 == 0)
       TAU_PROFILE_SNAPSHOT_1L(snapshot_nloop2, snapshot_nloop2_iter)
```

Save the file. The next step is to recreate the compilation process that `tau_cc.sh` uses to compile and link `par_multi_interp.inst.c`. To do this, search for the compilation steps in `output` using

```
egrep –A20 'par_multi_interp.inst.c' ../output
```

The following lines should appear

```
Executing> <TAU install dir>/bin/tau_instrumentor par_multi_interp.pdb
par_multi_interp.c –o par_mulit_interp.inst.c –c –f
/home/tautraining/AMG2013_snapshot/AMG2013/select_compile.tau

Debug: Compiling with Instrumented Code
Executing> gcc –c –O2 par_multi_interp.inst.c <Includes> <Options etc> -o
par_multi_interp.o
...
```

The command of interest here is the one that generates the object file `par_multi_interp.o`. To repeat this command, copy the command from '`gcc ...`' to '`… –o par_multi_interp.o`' and execute it again. The updated object file then needs to be updated in the library `libparcsr_ls.a`. Search for the command to create this library using

```
egrep –A20 'libparcsr_ls.a' ../output
```

and a set of commands like

```
Building libparcsr_ls.a …
Ar –rcu libparcsr_ls.a <list of object including par_multi_interp.o>
Ar: `u' modifier ignored since `D' is the default (see `U')
Ranlib libparcsr_ls.a
```

should appear. Re-execute these commands to update the library. The final step is to recompile the executable, `amg2013`, in the `test` directory. The command for this occurs at the end of `make`. Therefore, show the last 15 lines of `output` using

```
tail –n15 ../output
```

which should bring up

```
Linking amg2013
tau_cc.sh –o amg2013 amg2013.o <List of libraries>
Debug: Moving these libraries to the end of the link line:
```

Re-execute the second of these lines from the `test` directory. This should bring up

```
Debug: Moving these libraries to the end of the link line:
Debug: Linking with TAU Options
Executing> gcc –o amg2013 amg2013 <list of options>
```

and create a new version of the executable `amg2013`.

From the `test` directory, run the example problem again

```
mpirun -n 4 amg2013 -27pt -n 100 100 100 -P 2 2 1 > log.amg2013.27pt.100 &
```

As well as the usual `profile.*` files, a separate set of `snapshot.*` files are now written. Open one of these `snapshot.*` files using paraprof. The first difference to note is that a new window appears, the Snapshot Controller (see Figure 4 for an example) appears.

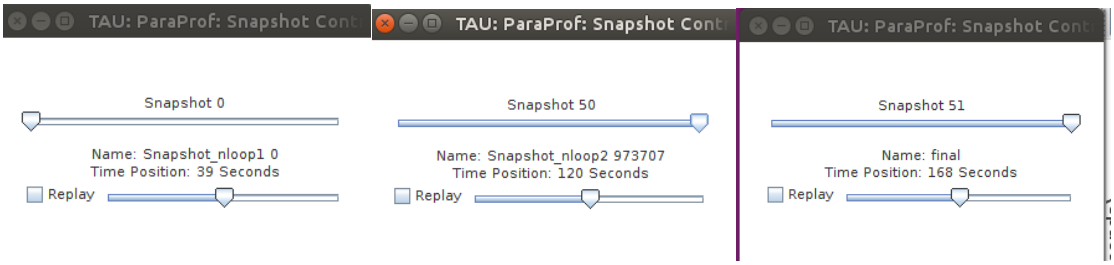

Figure 4: Paraprof Snapshot Controller

Moving the slider at the top will bring up a different snapshot and then update the view of the results window(s) that are currently open – including the Text Window, Table and all other types result window(s). The windows are updated with the new cumulative timings until the final snapshot is reached, which gives results identical to those in the `profile.*` file. Clicking 'Replay' will automatically cycle through all the snapshots.

Note that, in this example, the names of the snapshots follow their order in the source file – for example, the snapshot_nloop1 snapshots precede the snapshot_nloop2 snapshots. The numbers after each snapshot are also 20000 apart. Figure 5 shows, from the value of 'Time Position', that it took 39 s to reach Snapshot 0, the first, and 120 s to reach Snapshot 50, the penultimate snapshot, and that total execution time was 168 s.

Right clicking on 'node 0' (for example), will bring up a new option: 'Show Snapshots for Thread'. This will bring up an entirely new view. Figure 5 gives an example. To chart that comes up shows Execution Time on the x-axis. Each vertical stacked bar represents one snapshot, with the actual point of the snapshot in time located to the right side of the bar (i.e. snapshot 0 is located at t = 39 s; the first bar therefore has a domain of t = 0 to t = 39 s). The colours represent the functions named in the legend below the chart, and show the difference in execution time between the previous snapshot and the current snapshot.

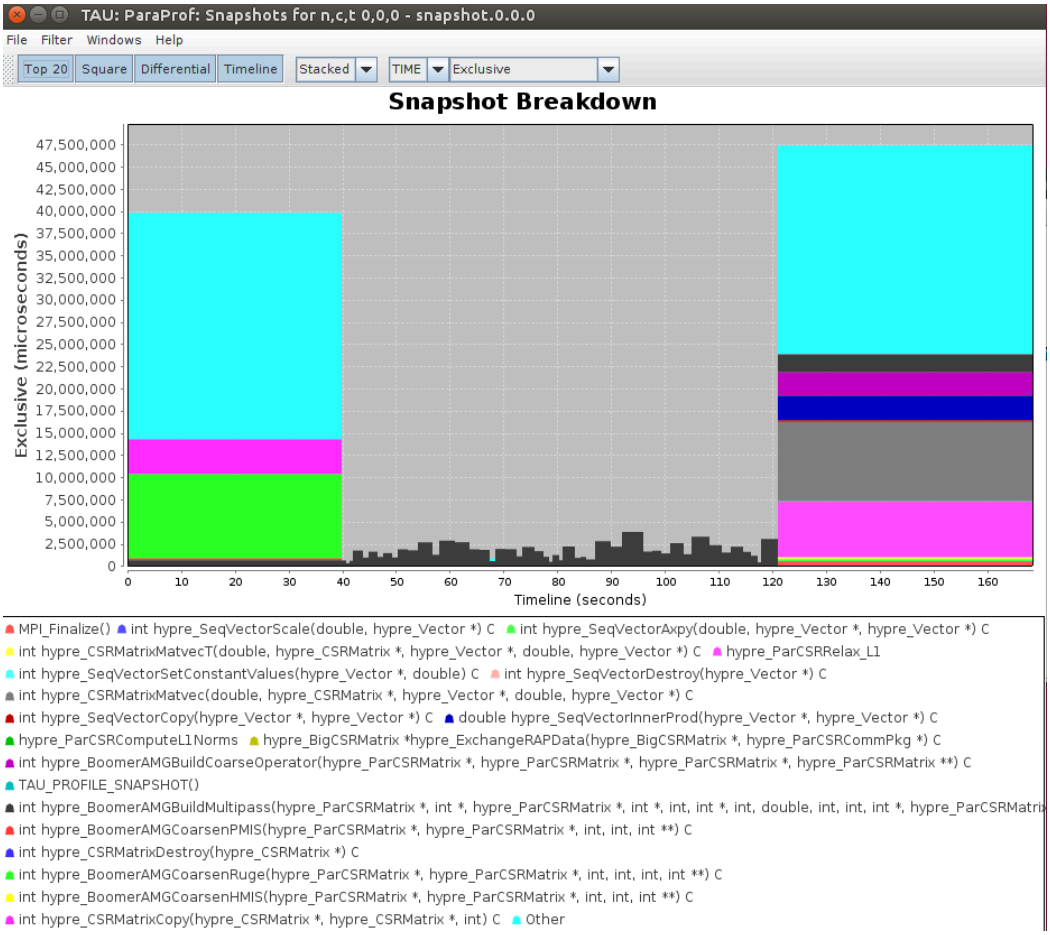

Figure 5: Paraprof Snapshot Breakdown

The above explanation is why Figure 5 has two large bars with multiple colours from t = 0 - 40 s and t = 120 - 170 s, and a section in the middle with small black bars. Recall that no snapshot was recorded until t = 39 s, and none from t = 120 s to t = 168 s, the difference between start and 39 s is 40 s, the height of the first bar. Similarly, the height of the last bars is 48 s, the difference in time between the last two snapshots. The bars in the middle represent the small increases in exclusive time between successive snapshots confined only to the calling of the BoomerAMGBuildMultiPass function.

To generate a more conventional chart that tracks total time (rather than difference), click on 'Top 20' to show all functions (rather than the Top 20), 'Differential' to show total time instead. Something like Figure 6 should appear. This shows more effectively that between each snapshot taken within BoomerAMGBuildMultiPass, exclusive time increases steadily. Hovering the cursor over a part of the chart will show the value at that point, which is useful when the legend shows many functions with similar colours. For example, placing the cursor over the dark green bar at the top right in this chart shows the bar represents MPI_Allgather(), and that it uses 16 s of exclusive time between t = 120 and t = 168 s.

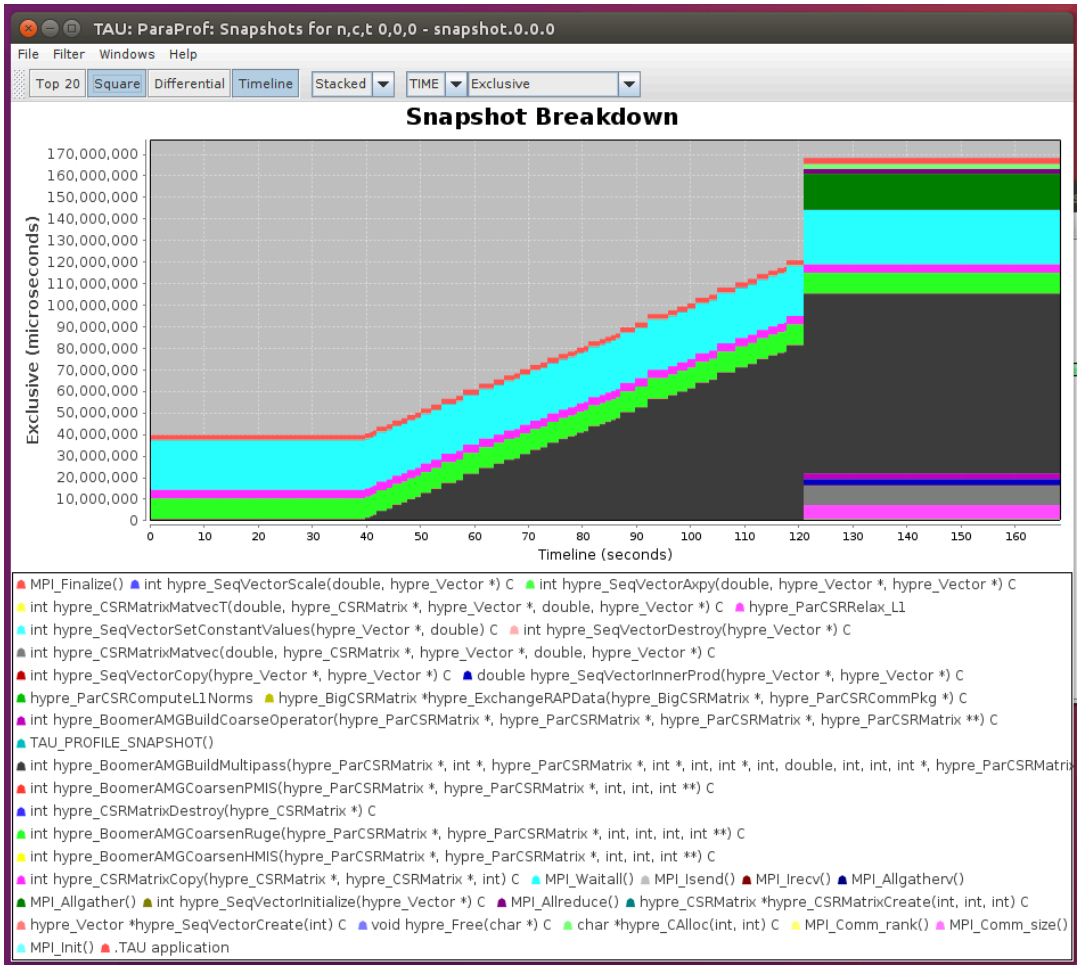

Figure 6: Paraprof Snapshot Breakdown – adjusted

Note that if a large amount of functions are profiled the list at the bottom of the Snapshot Breakdown may become unworkably large. Thus, Snapshot Breadowns are best viewed when only a small number of functions are profiled. Snapshots can also be combined with the Loops and Phases used in Section 3.2