TAU Performance System training course

Practical Sheet 1: Installing and Configuring TAU on a Linux system

1.1) Installing TAU

To begin, download and unpack the latest version of TAU to the home directory:

```
wget http://tau.uoregon.edu/tau.tgz
tar xvzf tau.tgz
```

Then go into the base TAU directory and download and unpack the Program Database Toolkit (PDT) tar

```
cd tau-2.27
wget http://tau.uoregon.edu/pdt.tgz
tar xvzf pdt.tgz
```

From the resulting PDT directory, configure and make PDT

```
cd pdtoolkit-3.25
./configure
make && make install
```

This creates a set of binaries, libraries and include files in the x86_64 directory. These form the PDT, which TAU uses to instrument source code.

Next, return to the base TAU directory. To configure TAU's installation, run the configure script

```
./configure -pdt=/home/tautraining/tau-2.27/pdtoolkit-
3.25 -bfd=download –mpi
```

The above will configure TAU with PDT and MPI support, and download and install the binutils (BFD) library. This is a basic configuration. Additional configuration options are available such as:

`-openmp` for OpenMP threads
`-papi=<path to papi install>` for including Performance API library
`-shmem` for the TAU SHMEM library wrapper
`-cuda=<path to cuda dir>` for OpenCL and CUDA profiling

For the purpose of the tutorial and the following practical sheets, the above PDT, binutilis and MPI configuration will be used. Once the configuration is complete, type

```
make install
```

This will place the installed TAU objects in the `x86_64` directory. Importantly, this will include a file called `Makefile.tau-gnu-mpi-pdt`. This file needs to be set as an environment variable, `TAU_MAKEFILE`, so that TAU chooses this configuration of TAU. There are also some other environment variables related to TAU that need to be set. This is best achieved by putting the appropriate commands into the `~/.bashrc` file:

```
export TAU_LIB=/home/tautraining/tau-2.27/x86_64/lib
export TAU_DIR=/home/tautraining/tau-2.27
export TAU=/home/tautraining/tau-2.27/x86_64/lib
export TAU_BIN=/home/tautraining/tau-2.27/x86_64/bin
export TAU_ARCH=x86_64
export TAU_MAKEFILE="/home/tautraining/tau-
2.27/x86_64/lib/Makefile.tau-gnu-mpi-pdt"
export PATH=/home/tautraining/tau-2.27/x86_64/bin:$PATH
export PATH=/home/tautraining/tau-2.27/x86_64/lib:$PATH
```

These commands will put TAU's compiler wrappers into the path, so that you can use them as you would standard compilers. For example, `tau_cc.sh` launches the C compiler wrapper, and `tau_f90.sh` launches the F90 compiler wrapper. Also available as commands include `paraprof`, TAU's results viewer, and `tau_exec`, which is used to execute programs (more on these commands later).

1.2) Configuring TAU

Instrumenting with TAU takes 3 different forms:

1. library interposition with `tau_exec`
2. compiler directives
3. source transformation with PDT.

The table below summarises their features.

Table 1.1. Different methods of instrumenting applications

Method	Requires recompiling	Requires PDT	Shows MPI events	Routine-level event
Interposition			Yes	
Compiler	Yes		Yes	Yes
Source	Yes	Yes	Yes	Yes

The requirements for each method increases as we move down the table: tau_exec only requires a system with shared library support. Compiler based inst
requires PDT. For this reason we often recommend that users start with Library interposition and move down the table if more features are needed.

event	Low level events (loops, phases, etc...)	Throttling to reduce overhead	Ability to exclude file from instrumentation	Ability to exclude other regions of code
		Yes		
		Yes	Yes	
	Yes	Yes	Yes	Yes

r based instrumentation requires re-compiling that target application and Source instrumentation aditionally
ded.

Method 1 is utilised by simply calling

```
tau_exec executable
```

or

```
mpirun –n x tau_exec mpi_executable
```

Methods 2 and 3 require the use of alternative C/C++/Fortran compilers built with TAU. These are `tau_cc.sh`, `tau_cxx.sh`, and `tau_f90/tau_f77.sh`:

```
tau_cc.sh my_code –o my_program.x
```

then

```
./my_program.x
```

or

```
mpirun –n x my_program.x
```

or

```
mpirun –n x tau_exec my_program.x
```

As you can see, methods 2 and 3 use the same compiler commands. Thus, further customisation of TAU is required to distinguish between the two. This is achieved by setting a further environment variable, `TAU_OPTIONS`

```
export TAU_OPTIONS=”-optPDTInst -optRevert -optVerbose”
```

In which `–optPDTInst` instructs TAU to use Source-based PDT instrumentation, `-optRevert` instructs TAU to revert to compiler-based instrumentation if PDT fails to parse a source code file, `-optVerbose` gives more verbose output, which is useful for debugging.

1.2.1) Further configuration options

There is a flag to set the C++ parser in PDT, ‘`-optDefaultParser`’. This is important because the default C++ parser, `cxxparse`, does not support the C++11 standard. Setting the C++ parser to `cxxparse4101` should set the parser to v4.10.1 with full C++11 support, but the flag is overwritten in the compiler wrapper `tau_cxx.sh`:

```
TAUCOMPILER_OPTIONS="-optDefaultParser=cxxparse
$TAUCOMPILER_OPTIONS"
```

Editing the above line in `/home/tautraining/tau2.27/x86_64/bin/tau_cxx.sh` to read

```
TAUCOMPILER_OPTIONS="-optDefaultParser=cxxparse4101
$TAUCOMPILER_OPTIONS"
```

Sets the default parser to v4.10.1. It is also useful to edit `cxxparse4101` so that it is more verbose, and therefore prints full commands for its individual parts. To edit this, change the flag in `cxxparse4101` in

```
/home/tautraining/tau-2.27/pdtoolkit-3.25/x86_64/bin
```

to read `VERBOSE=on` to `VERBOSE=off`.

The above set of changes can also be made to the `tau_cc.sh` C compiler wrapper, replacing `cparse` with `cparse4101`.
`TAU_OPTIONS` can also include `-optTauSelectFile=<filename>`. `<filename>` has a list of source files, loops, phases etc. to include, or exclude when using instrumentation. Note, if no excludes are present, only things 'included' are instrumented. For example:

```
BEGIN_FILE_INCLUDE_LIST
<Sourcefile name>
END_FILE_INCLUDE_LIST
```

Instruments only the source files listed in between the begin and end statements. A similar syntax exists with `EXCLUDE` instead of `INCLUDE`. The next level of selective instrumentation is:

```
BEGIN_INCLUDE_LIST
<full function name>
END_INCLUDE_LIST
```

Where the full function name includes the return type and arguments to the function, not just the name. Going further still, we have

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

In which `#` is the wildcard character, which in this case, tells PDT to instrument all `for` and `while` loops in all source files. Note - `#` cannot appear at the beginning of line as this is a comment. # at beginning of line must therefore be quoted. See https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01s03.html for more details on these selective instrumentation options. See Practical Sheet 3 for a look at how to use selective instrumentation.