Using CrayPAT and Apprentice2: A Step-by-step guide

Cray Centre of Excellence for HECToR

June 2013

Abstract

This tutorial introduces Cray XE6 users to the Cray Performance Analysis Tool and its Graphical User Interface, Apprentice2. The examples are based on the code supplied in the Image Sharpening Tutorial, however, the techniques can easily be applied to any application that is compiled and executed on a Cray supercomputer.

Introduction

The Cray Performance Analysis Tool (CrayPAT) is a powerful framework for analysing a parallel application's performance on Cray supercomputers. It can provide very detailed information on the timing and performance of individual application procedures, directly incorporating information from the raw hardware performance counters available on AMD Opteron processors.

Sampling vs. Tracing

CrayPAT has two modes of operation, Sampling and Tracing. Sampling takes regular snapshots of the application, recording which routine the application was in. This can provide a good overview of the important routines in an application without interfering with the run time, however it has the potential to miss smaller functions and cannot provide the more detailed information like MPI messaging statistics or information from hardware performance counters.

Tracing involves instrumenting each subroutine with additional instructions that can record this extra information when they enter and exit. This approach ensures full capture of information, but can result in high overheads, especially where individual functions and subroutines are very small (as is typical in Objected Oriented languages like C++), it can also generate very large amounts of data which become difficult to process and visualise.

CrayPAT's Automatic Program Analysis aims to capture the most important performance information without distorting the results by over instrumentation or generating large volumes of data. APA uses two steps, the first uses sampling to identify important functions in the application, it then uses this data, along with information about the size and number of calls to generate a modified binary with tracing included. This approach aims to cover the vast majority of application runtime with the minimum of overhead and provides a quick and straightforward method of analysing an application's performance on Cray supercomputers.

A step-by-step guide to using APA

This step-by-step guide demonstrates how to profile an application using CrayPAT's Automatic Program Analysis.

First, after logging on to the main system, users should load the `perftools` module, this loads the module `xt-craypat` and `apprentice2` modules.

```
module load perftools
```

The `perftools` module has to be loaded while all source files are compiled and linked. The Image Sharpening Tutorial can be built with a simple call to:

```
Table 1:  Profile by Function
 Samp%  | Samp  | Imb.  |  Imb.  |Group
        |       | Samp  | Samp%  | Function
        |       |       |        |  PE=HIDE
 100.0% |  38.9 |    -- |     -- |Total
|--------------------------------------------
|  51.0% |   19.8 |    -- |     -- |USER
||-------------------------------------------
||  23.9% |   9.3 |   3.7 |  29.5% |filter
||  14.1% |   5.5 |   3.5 |  40.5% |dosharpen
||  12.5% |   4.8 |   2.2 |  31.8% |_EXP_15
||===========================================
|  48.0% |  18.7 |    -- |     -- |MPI
||-------------------------------------------
||  23.4% |   9.1 |   0.9 |   9.4% |MPI_Barrier
||  19.4% |   7.5 |   0.5 |   6.0% |MPI_Bcast
||   5.2% |   2.0 |   1.0 |  33.3% |MPI_Reduce
||===========================================
|   1.0% |   0.4 |    -- |     -- |ETC
|============================================
```

Table 1: The tabular output for the C-MPI sharpen benchmark.

```
make
```

To instrument then the binary, run the pat_build command with the `-O apa` option. This will generate a new binary with +pat appended to the end.

```
pat_build -O apa sharpen
```

You should now run the new binary on the backend using the `batch.pbs` script. With the image sharpening example you should edit the `PBS` submission script, `submit.sh` and change the name of the executable to sharpen+pat. You should then submit this executable to run on the Cray XE6 backend.

```
qsub submit.sh
```

Once this has run, you will see that the run has generated an extra file, `sharpen+pat+<number>sdot.xf`. This file contains the raw sampling data from the run and needs to be post processed to produce useful results. This is done using the pat_report tool which converts all the raw data into a summarised and readable form.

```
pat_report sharpen+pat+25165-2sdot.xf
```

This tool can generate a large amount of data, so you may wish to capture the data in an output file, either using a shell redirect like >, or adding the `-o <file>` option to the command.

Table 1 shows the results from sampling the application. Program functions are separated out into different types, USER functions are those defined by the application, `MPI` functions contains the time spent in MPI library functions, `ETC` functions are generally library or miscellaneous functions included. `ETC` function can include a variety of external functions, from mathematical functions called in by the library (as is this case) to system calls.

The raw number of samples for each code section is show in the second column and the number as an absolute percentage of the total samples in the first. The third column is a measure of the imbalance between individual processors being sampled in this routine and is calculated as the difference between the average number of samples over all processors and the maximum samples an individual processor was in this routine.

This report will generate two more files, one with the extension .ap2 which holds the same data as the .xf but in the post processed form. The other file has a .apa extension and is a text file with a suggested configuration for generating a traced experiment. You are welcome and encouraged to review this file and modify its contents in subsequent iterations, however in this first case we will continue with the defaults.

```
Suggested trace options file:   sharpen+pat+25165-2sdot.apa
  Trace option suggestions have been generated into a separate file
  from the data in the next table.  You can examine the file, edit it
  if desired, and use it to reinstrument the program like this:
            pat_build -O sharpen+pat+25165-2sdot.apa
```

This file acts as the input to the pat_build command and is supplied as the argument to the -O flag.

```
pat_build -O sharpen+pat+25165-2sdot.apa
```

This will produce a third binary with extension +apa. This binary should once again be run on the back end, so the input submit.sh script should be modified and the name of the executable changed to sharpen+apa. The script is then submitted to the backend.

```
qsub submit.sh
```

Again, a new .xf file will be generated by the application, which should be processed by the pat_report tool. As this is now a tracing experiment it will provide more information than before

```
pat_report sharpen+apa+25218-2tdot.xf
```

Table 2 is the more detailed version of Table 1 that is the result of tracing data. In this case, true timing information is available (averages per processor) and the number of times each function is called. Table 3 shows the information available for individual functions. Timings are more accurate and features like the number of calls are available. Information from the Opteron's hardware performance counters is also available, specifically in this case details relating to the number of floating point operations, cache references and TLB buffer. There are a large number of performance counters available from the Opteron however only 4 may be run concurrently.

Additional document ion is available for CrayPAT and can be access either through the man pages for individual commands or through the interactive CrayPAT command (requires perftools to be loaded):

```
pat_help
```

Or though man pages:

```
man intro_pat
man pat_build
man pat_report
```

Apprentice2

Apprentice2 is the Graphic User Interface and visualisation suite for CrayPAT's performance data. It reads the .ap2 files generated by pat_report's processing of .xf files. It is launched from the command line with:

```
app2 <file>.ap2
```

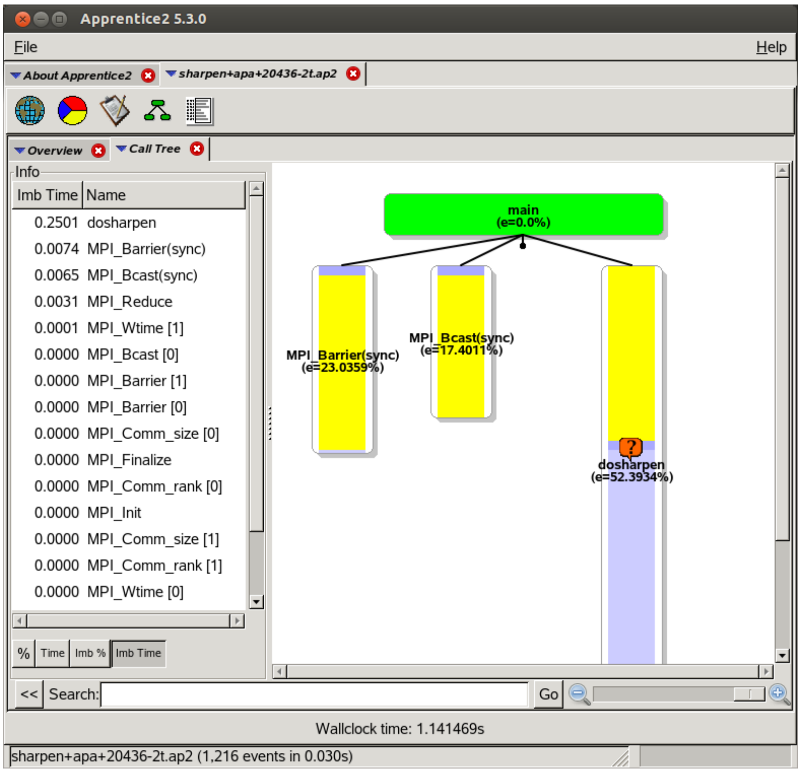
Figure 1 shows a screen shot of the call tree information available from CrayPAT. It shows how time is spent along the call tree, inclusive time corresponds to the width of boxes, excluding time to the height. Yellow represents the load imbalance time between processors. Extra information is provided by holding the mouse over areas of the screen, the "?" box will provide hints on how to interpret the information displayed.

```
Table 1:  Profile by Function Group and Function
 Time%  |    Time  |    Imb.  |  Imb.   | Calls  |Group
        |          |    Time  | Time%   |        | Function
        |          |          |         |        |  PE=HIDE
 100.0% | 0.639638 |       -- |      -- |   36.0 |Total
|-----------------------------------------------------------------
|  51.7% | 0.330767 |       -- |      -- |    3.0 |USER
||----------------------------------------------------------------
|  51.7% | 0.330409 | 0.250140 |  44.5% |    1.0 | dosharpen
||================================================================
|  41.2% | 0.263766 |       -- |      -- |   11.0 |MPI_SYNC
||----------------------------------------------------------------
||  23.0% | 0.146806 | 0.141102 |  96.1% |    5.0 |MPI_Barrier(sync)
||  18.3% | 0.116941 | 0.116862 |  99.9% |    5.0 |MPI_Bcast(sync)
||================================================================
|   7.1% | 0.045105 |       -- |      -- |   22.0 |MPI
||----------------------------------------------------------------
||   5.8% | 0.037132 | 0.003068 |   7.9% |    1.0 |MPI_Reduce
||   1.2% | 0.007741 | 0.000044 |   0.6% |    5.0 |MPI_Bcast
|=================================================================
```

Table 2: A more detailed function profiling

```
========================================================================
USER / dosharpen
------------------------------------------------------------------------
  Time%                                             63.4%
  Time                                           0.499493 secs
  Imb.Time                                       0.182487 secs
  Imb.Time%                                         29.1%
  Calls                         2.0 /sec              1.0 calls
  PAPI_L1_DCM                 1.622M/sec           809965 misses
  PAPI_TLB_DM                 0.033M/sec            16296 misses
  PAPI_L1_DCA               847.673M/sec        423413396 refs
  PAPI_FP_OPS               345.699M/sec        172677072 ops
  User time (approx)            0.500 secs     1048953736 cycles 100.0%Time
  Average Time per Call                          0.499493 secs
  CrayPat Overhead : Time        0.0%
  HW FP Ops / User time     345.699M/sec        172677072 ops   4.1%peak(DP)
  HW FP Ops / WCT           345.699M/sec
  Computational intensity        0.16 ops/cycle      0.41 ops/ref
  MFLOPS (aggregate)        8296.78M/sec
  TLB utilization          25983.32 refs/miss      50.749 avg uses
  D1 cache hit,miss ratios     99.8% hits            0.2% misses
  D1 cache utilization (misses)  522.76 refs/miss  65.344 avg hits
========================================================================
```

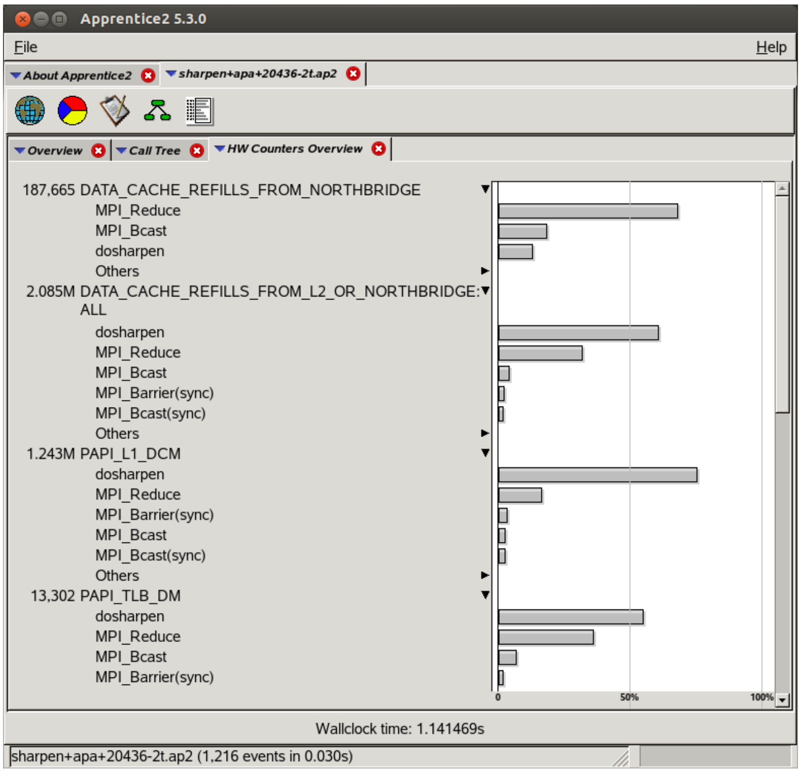
Table 3: Hardware Performance Counter information from a traced CrayPAT APA experiment



Imb Time	Name
0.2501	dosharpen
0.0074	MPI_Barrier(sync)
0.0065	MPI_Bcast(sync)
0.0031	MPI_Reduce
0.0001	MPI_Wtime [1]
0.0000	MPI_Bcast [0]
0.0000	MPI_Barrier [1]
0.0000	MPI_Barrier [0]
0.0000	MPI_Comm_size [0]
0.0000	MPI_Finalize
0.0000	MPI_Comm_rank [0]
0.0000	MPI_Init
0.0000	MPI_Comm_size [1]
0.0000	MPI_Comm_rank [1]
0.0000	MPI_Wtime [0]

```
Wallclock time: 1.141469s
sharpen+apa+20436-2t.ap2 (1,216 events in 0.030s)
```

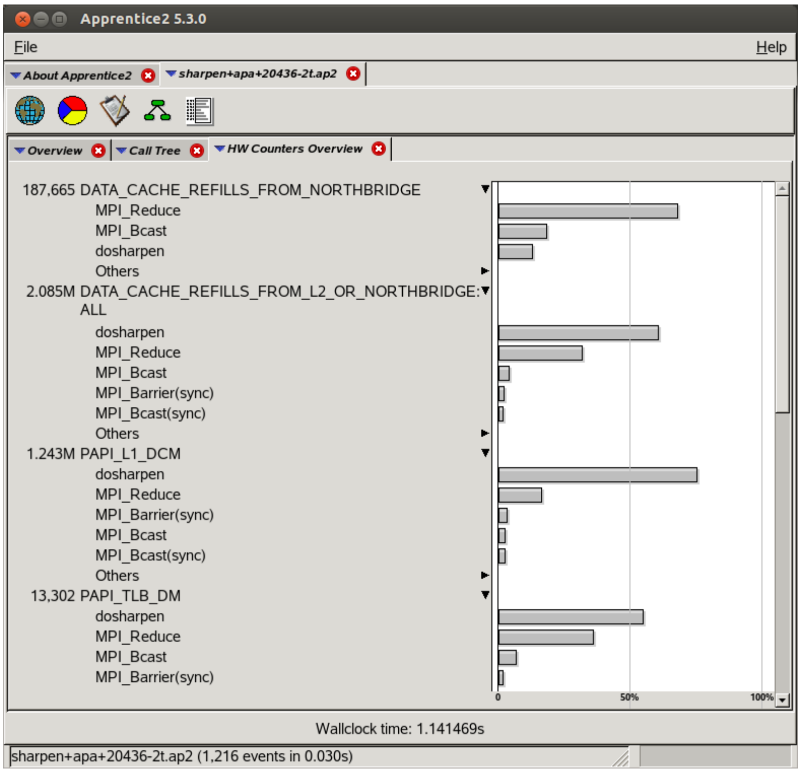
Figure 1: Apprentice2: Application call tree



```
187,665 DATA_CACHE_REFILLS_FROM_NORTHBRIDGE
        MPI_Reduce
        MPI_Bcast
        dosharpen
        Others
2.085M DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE:
    ALL
        dosharpen
        MPI_Reduce
        MPI_Bcast
        MPI_Barrier(sync)
        MPI_Bcast(sync)
        Others
1.243M PAPI_L1_DCM
        dosharpen
        MPI_Reduce
        MPI_Barrier(sync)
        MPI_Bcast
        MPI_Bcast(sync)
        Others
13,302 PAPI_TLB_DM
        dosharpen
        MPI_Reduce
        MPI_Bcast
        MPI_Barrier(sync)

Wallclock time: 1.141469s
sharpen+apa+20436-2t.ap2 (1,216 events in 0.030s)
```

Figure 2: Apprentice2: Summarised Hardware Performance information



```
187,665 DATA_CACHE_REFILLS_FROM_NORTHBRIDGE
        MPI_Reduce
        MPI_Bcast
        dosharpen
        Others
2.085M DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE:
       ALL
        dosharpen
        MPI_Reduce
        MPI_Bcast
        MPI_Barrier(sync)
        MPI_Bcast(sync)
        Others
1.243M PAPI_L1_DCM
        dosharpen
        MPI_Reduce
        MPI_Barrier(sync)
        MPI_Bcast
        MPI_Bcast(sync)
        Others
13,302 PAPI_TLB_DM
        dosharpen
        MPI_Reduce
        MPI_Bcast
        MPI_Barrier(sync)
```

Wallclock time: 1.141469s

sharpen+apa+20436-2t.ap2 (1,216 events in 0.030s)

Figure 3: Apprentice2: Application's runtime environment

Accessing Temporal Information

Tracing an application can potentially generate very large amounts of data, to reduce this volume the CrayPAT will, by default, summarise the data over the entire application run. To see more detailed information about the timing of individual events (like the sequencing of MPI messages between processors or the number of hardware counter events in a time interval) CrayPAT has to be instructed to store all data from throughout the run. This is controlled by the `PAT_RT_SUMMARY` environment variable, setting it to `0` in `batch.pbs` will prevent summarising and allow access to even more data.

```
export PAT_RT_SUMMARY=0
```

Warning! Running tracing experiment on a large number of processors for a long period of time will generate VERY large files! Most tracing experiments should be conducted on a small number of processors (≤ 256) and over a short wall clock time period (< 5 minute).

Figures 3,4,5 and 6 show captures of the additional screens available in Apprentice2 when analysing non-summarised data.

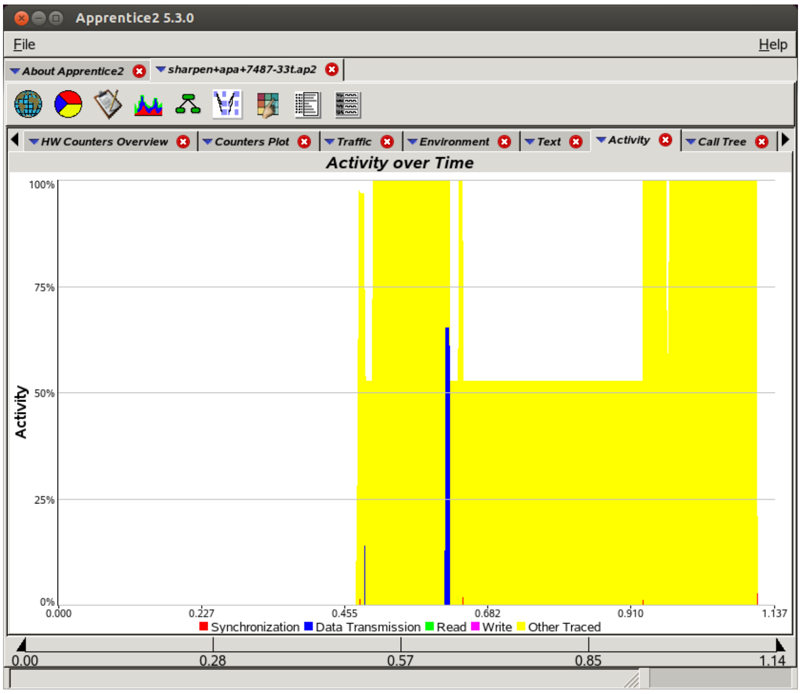


Figure 4: Apprentice2: Application activity information over time

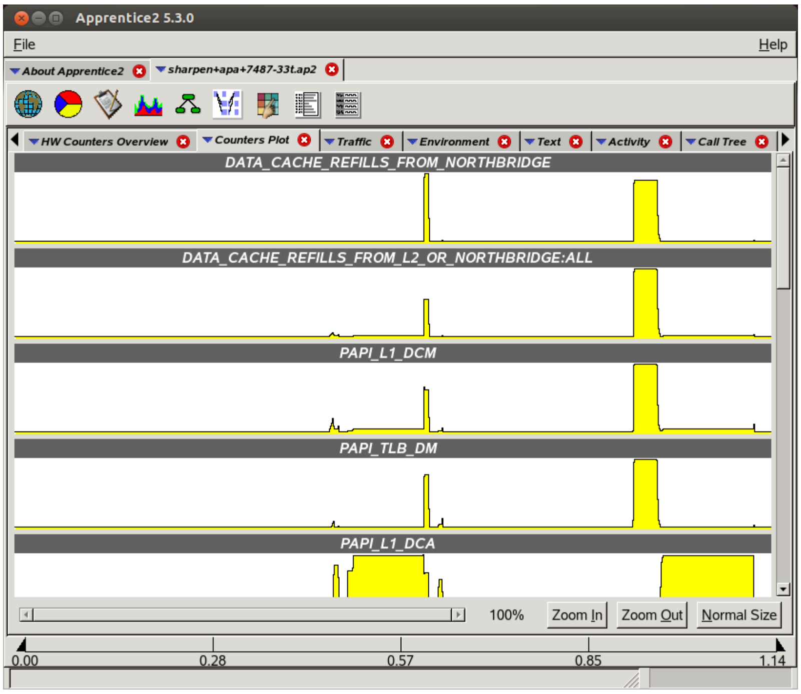


Figure 5: Apprentice2: Hardware counter values over time

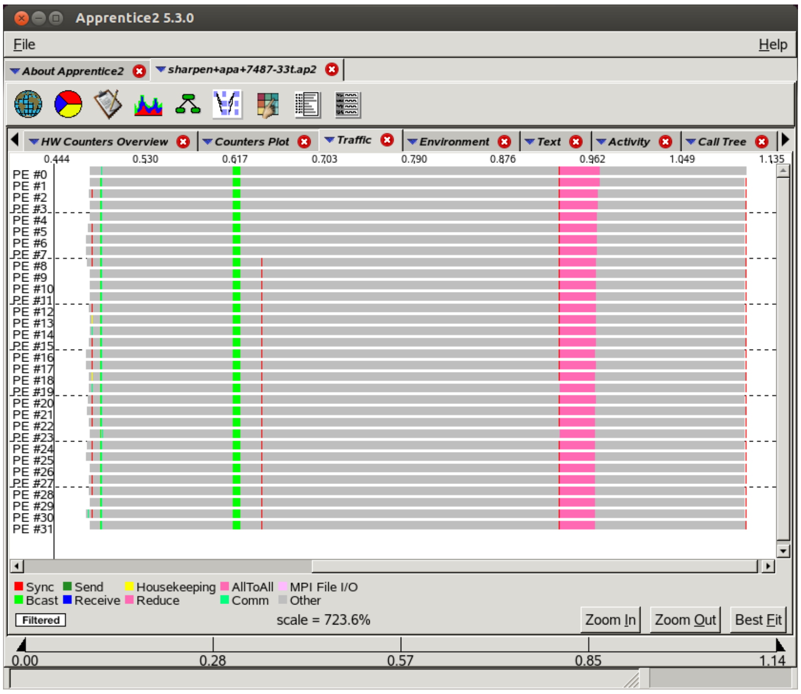


Figure 6: Apprentice2: Message traffic information panel