Using CrayPat and Apprentice^2: A Step-by-step guide

Abstract

This tutorial introduces Cray XC30 users to the Cray Performance Analysis Tool and its Graphical User Interface, Apprentice^2. The examples are based on the code supplied in the downloaded packages, 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 Intel Xeon processors. In the next section we introduce the fastest way to get a profile (using craypat-lite) and in subsequent sections expand on the types of experiments that CrayPat can run and explain the more advanced step-by-step process.

A quick profile

First, after logging on to the main system, users should load the perftools-lite module. You should also load the netcdf module to get access to the netcdf libraries.

```
module load cdt/15.11
module load perftools-base
module load perftools-lite
module load cray-netcdf4/4.3.2
```

The perftools module has to be loaded while all source files are compiled and linked. The VH1 supplied CFD application can be built with a simple call to:

```
cd src; make
```

The perftools module adds instrumentation (via the compilation wrappers) as the application is built.

You should now run the new instrumented binary on the backend using the appropriate run_XXX.pbs script in the run directory. On some systems you may need to choose a specific queue or account. You should then submit this job script ...

```
qsub run_XXX.pbs
```

Note that the job script places the output in a directory. The job output includes a performance report on the run which includes basic information and performance tables for time spent in various functions, time spent in MPI and some I/O statistics. In addition note that the run generated extra files like the following

```
vh1-mpi-cray+10516-224s.ap2
```

```
vh1-mpi-cray+10516-224s.rpt
```

As you will see in the subsequent sections the ap2 file contains performance information for the run and the rpt file is a copy of the text report you saw in the job output.

You can use the pat_report command to produce new reports from the ap2 file. The following sections explain the operation of CrayPat in more detail using the more advanced step-by-step features.

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 executing at the time. 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 detailed information from hardware performance counters.

Tracing involves instrumenting each subroutine to record extra information on entry 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 time-consuming functions, 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.

As before load the relevant modules but use the perftools module this time (not perftools-lite)…

```
module unload perftools-lite
module load perftools
cd src; make
```

This time you need to instrument then the binary yourself, so run the pat_build command with the -O apa option. This will generate a new binary with +pat appended to the end.

```
cd ../bin
pat_build -O apa vh1-mpi-cray
```

You should now run the new binary on the backend using the run_XXX.pbs script in the run directory. In this example you should edit the batch script change the name of the executable to vh1-mpi-cray+pat. You should then submit this executable to run on the Cray XC30 backend.

```
qsub run.pbs
```

Once this has run, you will see that the run has generated an extra file, vh1-mpi-cray+pat+<number>sdot.xf. This file contains the raw sampling data from the run and needs to be post-processed to produce human-readable results. This is done using the pat_report tool which converts all the raw data into a summarised and readable form.

```
pat_report vh1-mpi-cray+pat+2681227-198s.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:  Profile by Function

 Samp%  |  Samp  | Imb.  |  Imb.  |Group
        |        | Samp  | Samp%  | Function
        |        |       |        |  PE=HIDE

 100.0% | 1663.1 |    -- |     -- |Total
|-----------------------------------------------
|  75.1% | 1248.7 |    -- |     -- |USER
||----------------------------------------------
||  24.5% |  406.9 |  30.1 |   7.2% |parabola_
||  16.0% |  266.9 |  46.1 |  15.4% |riemann_
||   8.2% |  136.1 |  12.9 |   9.0% |sweepz_
||   7.1% |  118.7 |  23.3 |  17.1% |remap_
||   4.6% |   75.9 |  19.1 |  21.0% |paraset_
||   4.5% |   74.2 |  13.8 |  16.3% |sweepy_
||   2.8% |   46.3 |  12.7 |  22.5% |evolve_
||   2.6% |   43.4 |  11.6 |  22.0% |states_
||   1.8% |   29.9 |   9.1 |  24.4% |flatten_
||   1.3% |   21.5 |   9.5 |  31.8% |sweepx1_
||   1.1% |   18.7 |   8.3 |  32.0% |sweepx2_
||==============================================
|  18.7% |  310.6 |    -- |     -- |MPI
||----------------------------------------------
||  16.0% |  266.2 |  35.8 |  12.4% |mpi_alltoall
||   1.2% |   19.6 |  10.4 |  36.2% |MPI_ALLREDUCE
||   1.0% |   17.0 |   4.0 |  19.9% |mpi_finalize
||==============================================
|   5.5% |   91.8 |    -- |     -- |ETC
||----------------------------------------------
||   3.3% |   55.4 |  22.6 |  30.2% |__cray_sset_SNB
||   2.0% |   33.6 |  10.4 |  24.7% |__cray_scopy_SNB
|==============================================
```

Table 1 - User functions profiled by samples

Table 1 above 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 the .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.

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

```
pat_build -O vh1-mpi-cray+pat+2681227-198s.apa
```

This will produce a third binary with extension +apa. Remember to move this into the bin directory. This binary should once again be run on the back end, so the input run.pbs script should be modified and the name of the executable changed to vh1-mpi-cray+apa. The script is then submitted to the backend.

```
qsub run_XXX.pbs
```

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 vh1-mpi-cray+apa+2681298-198t.xf
Table 1:  Profile by Function Group and Function

 Time%  |     Time  |    Imb.  |  Imb.  |    Calls  |Group
        |           |    Time  | Time%  |           | Function
        |           |          |        |           |  PE=HIDE
 100.0% | 43.070384 |       -- |     -- | 7373984.5 |Total
|-------------------------------------------------------------------------
|  82.3% | 35.442827 |       -- |     -- | 7373051.0 |USER
||------------------------------------------------------------------------
||  24.8% | 10.688417 | 1.364842 |  11.8% |  460800.0 |remap_
||  11.6% |  4.985583 | 1.556333 |  24.8% | 4147200.0 |parabola_
||  11.2% |  4.818683 | 0.540746 |  10.5% |      50.0 |sweepz_
||  10.6% |  4.571071 | 0.530418 |  10.8% |     100.0 |sweepy_
||   6.4% |  2.755588 | 0.539899 |  17.1% |  460800.0 |riemann_
||   5.0% |  2.155727 | 0.265681 |  11.4% |      50.0 |sweepx1_
||   4.9% |  2.110443 | 0.254360 |  11.2% |      50.0 |sweepx2_
||   2.3% |  0.982799 | 0.342153 |  26.9% |  921600.0 |paraset_
||   2.1% |  0.901174 | 0.158770 |  15.6% |  460800.0 |evolve_
||   1.4% |  0.593570 | 0.170956 |  23.3% |  460800.0 |flatten_
||   1.3% |  0.576879 | 0.174349 |  24.2% |  460800.0 |states_
||========================================================================
|  14.6% |  6.288273 |       -- |     -- |     361.2 |MPI_SYNC
||------------------------------------------------------------------------
||  11.6% |  5.002999 | 4.661168 |  93.2% |     300.0 |mpi_alltoall_(sync)
||   2.9% |  1.260776 | 1.258619 |  99.8% |      51.0 |mpi_allreduce_(sync)
||========================================================================
|   3.1% |  1.339015 |       -- |     -- |     371.3 |MPI
||------------------------------------------------------------------------
|   2.6% |  1.101674 | 0.059202 |   5.3% |     300.0 | mpi_alltoall
|=========================================================================
```

Table 2 – User functions profiled using tracing

The updated table above (Table 2) is the version generated from tracing data instead of the previous sampling data table (Table 1). This version makes true timing information 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 Xeon hardware performance counters is also available, specifically in this case details relating to the number of cache references and TLB buffer. There are a large number of performance counters available from the Xeon however only a subset can be collected concurrently.

```
==============================================================================
  USER / remap_
------------------------------------------------------------------------------
 Time%                                                  24.8%
 Time                                               10.688417 secs
 Imb. Time                                           1.364842 secs
 Imb. Time%                                             11.8%
 Calls                              0.039M/sec       460800.0 calls
 PERF_COUNT_HW_CACHE_L1D:ACCESS                    16640767864
 PERF_COUNT_HW_CACHE_L1D:PREFETCH                   1090541046
 PERF_COUNT_HW_CACHE_L1D:MISS                       3868292423
 CPU_CLK_UNHALTED:THREAD_P                         89548181241
 CPU_CLK_UNHALTED:REF_P                             3067017219
 DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                  11536258
 DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                 23360841
 L2_RQSTS:ALL_DEMAND_DATA_RD                        2466573233
 L2_RQSTS:DEMAND_DATA_RD_HIT                        2247462515
 User time (approx)               11.683 secs    31555692812 cycles  100.0% Time
 CPU_CLK                           2.920GHz
 TLB utilization                  618.95 refs/miss     1.209 avg uses
 D1 cache hit,miss ratios          82.1% hits          17.9% misses
 D1 cache utilization (misses)      5.58 refs/miss     0.698 avg hits
 D2 cache hit,miss ratio           94.3% hits           5.7% misses
 D1+D2 cache hit,miss ratio        99.0% hits           1.0% misses
 D1+D2 cache utilization           98.58 refs/miss    12.322 avg hits
 D2 to D1 bandwidth            12886.083MB/sec  157860686896 bytes
 Average Time per Call                              0.000023 secs
 CrayPat Overhead : Time           10.7%
```

Table 3 – Per function hardware performance counter information

Additional documentaion 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
```

Apprentice^2

Apprentice^2 is the Graphical 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
```

Accessing Temporal Information

Tracing an application can potentially generate very large amounts of data, to reduce this volume 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 run.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 minutes). It is possible to use environment variables and/or the CrayPat API to appropriately reduce the amount of data collected but we will not explore that in this exercise

For the VH1 application the number of cycles can be reduced by editing the input file (indat).