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

Abstract

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

```
module load perftools
```

The perftools module has to be loaded while all source files are compiled and linked. Next, load the NetCDF and HDF5 module (required by VH1):

```
module load cray-hdf5
module load cray-netcdf
```

The VH1 can be built with a simple call to:

```
cd src; 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.

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

You should now run the new binary on the backend using the **`run.sh`** 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 backend.

sbatch run.sh

(Replace the XXXX with the reservation name given to you by the course organisers). Once this has run, you will see that the run has generated an extra directory, **vh1-mpi-cray+pat+<number>s** in a directory <jobid_number>. 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 vh1-mpi-cray+pat+2681227-198s

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% | 2,359.3 |   -- |    -- |Total
|-----------------------------------------------
|  57.7% | 1,361.6 |   -- |    -- |USER
||----------------------------------------------
||  20.0% |   472.2 | 38.8 |  7.9% |parabola_
||  12.7% |   298.7 | 43.3 | 13.2% |riemann_
||   5.9% |   140.2 | 20.8 | 13.5% |sweepz_
||   5.6% |   133.2 | 33.8 | 21.1% |remap_
||   3.2% |    76.3 |  9.7 | 11.8% |sweepy_
||   3.0% |    71.5 | 14.5 | 17.6% |paraset_
||   1.9% |    45.8 | 14.2 | 24.8% |evolve_
||   1.9% |    44.2 | 14.8 | 26.2% |states_
||   1.1% |    27.1 |  7.9 | 23.5% |flatten_
||   1.0% |    22.8 | 11.2 | 34.4% |sweepx1_
||==============================================
|  37.6% |   887.8 |   -- |    -- |MPI
||----------------------------------------------
||  36.5% |   861.8 | 83.2 |  9.2% |mpi_alltoall
||==============================================
|   4.2% |   100.0 |   -- |    -- |ETC
||----------------------------------------------
||   2.6% |    60.3 | 13.7 | 19.3% |__cray_sset_SNB
||   1.5% |    34.4 |  8.6 | 20.9% |__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.

The profile also generated two other files that are useful in the profiling directory, 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.

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 build_options.apa
```

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

```
sbatch run.sh
```

(Replace the XXXX with the reservation name given to you by the course organisers). Again, a new profile directory 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+2681298-198s
```

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE
100.0%	67.740903	--	--	7,373,686.5	Total
76.4%	51.733412	--	--	7,372,951.0	USER
24.8%	16.776085	2.516269	13.6%	460,800.0	remap_
11.1%	7.529205	3.421455	32.6%	1.0	vhone_
11.0%	7.447873	0.953479	11.8%	50.0	sweepz_
10.6%	7.186668	0.967835	12.4%	100.0	sweepy_
9.3%	6.289532	2.906604	33.0%	4,147,200.0	parabola_
4.5%	3.048572	0.517124	15.1%	460,800.0	riemann_
1.6%	1.104668	0.641691	38.3%	921,600.0	paraset_
1.4%	0.964630	0.366043	28.7%	460,800.0	evolve_
1.1%	0.727949	0.346536	33.7%	460,800.0	flatten_
1.0%	0.658229	0.320822	34.2%	460,800.0	states_
21.9%	14.863376	--	--	363.2	MPI_SYNC
17.5%	11.826659	9.977789	84.4%	300.0	mpi_alltoall_(sync)
3.9%	2.608508	2.592821	99.4%	51.0	mpi_allreduce_(sync)
1.7%	1.144092	--	--	371.3	MPI
1.6%	1.086442	0.023866	2.2%	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 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 Rome hardware performance counters is also available.

```
==============================================================================
  USER / remap_
------------------------------------------------------------------------------
  Time%                                              24.8%
  Time                                           16.776085 secs
  Imb. Time                                       2.516269 secs
  Imb. Time%                                         13.6%
  Calls                      0.025M/sec          460,800.0 calls
  CPU_CLK_UNHALTED:THREAD_P                     87,504,487,183
  CPU_CLK_UNHALTED:REF_P                        2,979,085,085
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK              20,952,547
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK              7,282,943
  L1D:REPLACEMENT                               1,714,128,948
  L2_RQSTS:ALL_DEMAND_DATA_RD                   1,861,959,568
  L2_RQSTS:DEMAND_DATA_RD_HIT                   1,747,562,343
  FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE                 1,642
  FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE            772,302,817
  FP_COMP_OPS_EXE:X87                           845,393,483
  FP_COMP_OPS_EXE:SSE_PACKED_SINGLE             127,020,303
  SIMD_FP_256:PACKED_SINGLE                     5,099,855,700
  User time (approx)           18.663 secs     50,407,589,746 cycles  100.0% Time
  CPU_CLK                      2.94GHz
  HW FP Ops / User time        2,300.039M/sec     42,924,624,751 ops   10.6%peak(DP)
  Total SP ops                2,254.740M/sec     42,079,229,626 ops
  Total DP ops                   45.299M/sec        845,395,124 ops
  MFLOPS (aggregate)             55,200.93M/sec
  D2 cache hit,miss ratio         93.3% hits             6.7% misses
  D2 to D1 bandwidth           6,089.462MiB/sec  119,165,412,368 bytes
  Average Time per Call                                0.000036 secs

  CrayPat Overhead : Time          11.7%
```

Table 3 – Per function hardware performance counter information

Additional documentation is available for CrayPAT and can be accessed 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 from the profile output directory generated by pat_report's the profile files. It is launched from the command line with:

app2 <profile_directory>

Figure 1 shows a screenshot 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.

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 run.sh 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).

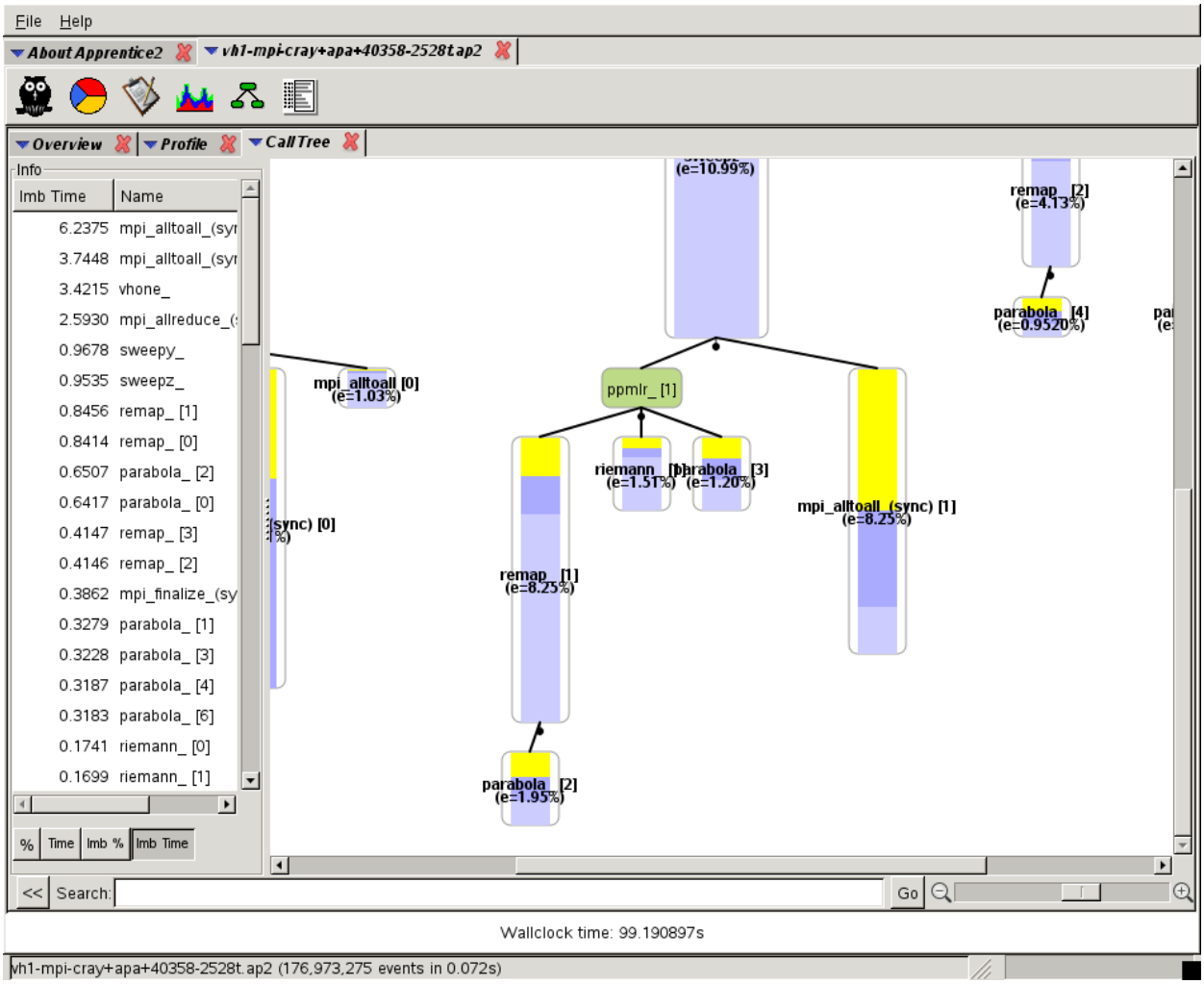

Figure 1 – A screenshot of Apprentice 2.