Using CrayPAT, Apprentice2, and Reveal

Abstract

This tutorial introduces Cray XE/XC users to the Cray Performance Analysis Tool and its Graphical User Interface, Apprentice2. The examples are based on the code supplied in the Himeno 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. In the present examples the Intel Haswell counters are shown but other architectures are also supported.

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 that 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 information from hardware performance counters.

Tracing requires instrumenting each subroutine with additional instructions that can record this extra information whenever they enter and exit. This approach ensures full capture of information, but can result in large 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 combines sampling and tracing and 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. **However, it is not mandatory to run the APA for each application.** The user can also do a sampling or tracing separately. For small applications it is more convenient for instance to run directly a tracing experiment.

This tutorial contains a step-by-step guide to APA followed by instructions how do run only a sampling or tracing experiment. In addition, the loop profiling experiment with CrayPat will be illustrated.

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. Prior to analysis copy all the files of interest to a subdirectory of the lustre file system. The behaviour of CrayPAT can differ, i.e. the way how raw data is stored, depending on the file system. First, users should load the perftools module, which provides all the CrayPAT tools as well as Apprentice2 (and Reveal)

```
> module load perftools
```

The perftools module has to be loaded while all source files are compiled and linked. For example, the Himeno example can be built[1] with a simple call to:

```
> make
```

To instrument the binary, just run the pat_build command[2]. This will generate a new binary with +pat appended to the end.

```
> pat_build himeno.exe
```

You should now run the new binary on the back end using the job.pbs script. For the Himeno example edit the submission script job.pbs and change the name of the executable to `himeno.exe+pat`. You should then submit this executable to run on the Cray backend.

```
> qsub job.apa.samp.pbs
```

Once finished, you will see that the run has generated an extra file, himeno.exe+pat+<number>.xf (the precise name of this file is given in the standard output). 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 –o myrep.txt himeno.exe+pat+<number>.xf
```

This tool can generate a large amount of data, so the –o <file> option is used to capture the information in a file (in this case myrep.txt). Conversely, you can use a shell redirect like >.

Table 1 shows the results from sampling the application. Program functions are separated into different groups, USER functions are those defined by the application, MPI functions contain 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, Cray specific calls (as is this case) to system calls. Note that the MPI group has been pruned in this table due to thresholds.

[1] The Makefile for Himeno preserves the *.o file which is needed by CrayPat. In case that a program is linked and compiled in one step, the *.o file is usually deleted. In this case CrayPat stores the *.o file in a temporary directory showed in a warning. To prevent this behavior you can use the –hkeepfiles option to CCE.

[2] In the meanwhile the APA is the default experiment and the -O apa option is no longer needed.

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

 100.0% | 3915.0 |   -- |    -- |Total
|-----------------------------------------------
|  89.7% | 3510.8 |   -- |    -- |USER
||----------------------------------------------
||  86.3% | 3379.4 | 14.6 |  0.5% |jacobi
||   3.4% |  131.4 |  4.6 |  3.9% |initmt
||==============================================
|   9.5% |  371.6 |   -- |    -- |ETC
||----------------------------------------------
||   8.0% |  311.4 |  5.6 |  2.0% |__cray_scopy_HSW
||   1.5% |   60.2 |  2.8 |  5.0% |__cray_sset_HSW
|==============================================
```

Table 1: User functions profiled by samples. Data obtained on the Hornet system at HLRS featuring Intel Haswell processors. Numbers may vary.

The absolute percentage of the total samples for each code section is shown in the first column and the raw number of samples in the second column . 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 MPI portion is very small, since the ratio between computation and communication is very high and it is very likely that for some of your runs this group does not appear in the report at all. The `pat_report` utility prunes the groups which contribute percentages of total time less than a given threshold (0.5% in this case). If you want to see the full report you have to add the -T option or increase the communication to computation ratio. This will automatically happen in a strong scaling analysis.

Independent on the -T option, the pat_report will generate two more files with the same stem as the .xf file, one with extension .ap2, which holds the same data as the .xf but in the post processed form. The other file has an .apa extension and is a text file with a suggested configuration for generating a tracing experiment. You are welcome and encouraged to review this file and modify its contents in subsequent steps, however in this first case we will continue with the defaults.

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

```
> pat_build -O himeno.exe+<number>.apa
```

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

```
> qsub job.apa.trace.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 himeno.exe+<number2>.xf
```

```
Time% |      Time |     Imb. |  Imb. | Calls |Group
      |           |     Time | Time% |       | Function
      |           |          |       |       |  PE=HIDE

 100.0% | 39.330470 |       -- |    -- | 935.0 |Total
|-----------------------------------------------------------
|  99.2% | 39.017306 |       -- |    -- |   5.0 |USER
||----------------------------------------------------------
||  94.3% | 37.107192 | 0.092340 |  0.3% |   2.0 |jacobi
||   4.9% |  1.909905 | 0.004930 |  0.3% |   1.0 |initmt
|===========================================================
```

Table 2: User functions profiled using tracing. Data obtained on the Hornet system at HLRS featuring Intel Haswell processors. Numbers may vary.

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 also available. Information from the CPU's hardware performance counters is also available, in this case details relating to the number of floating point operations, cache references and TLB buffer. The collection of the hardware counters has been enabled by the -Drtenv=PAT_RT_PERFCTR=default option in the .apa file. There are a large number of performance counters available (run `papi_avail` on the back end) for the Haswell CPU, however only a few of them may be run concurrently.

```
==============================================================================
  USER / jacobi
------------------------------------------------------------------------------
  Time%                                              94.3%
  Time                                           37.107192 secs
  Imb. Time                                       0.092340 secs
  Imb. Time%                                          0.3%
  Calls                              0.054 /sec        2.0 calls
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                3691869744
  CPU_CLK_THREAD_UNHALTED:THREAD_P              104182792444
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK              29018416
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK              6669417
  L1D:REPLACEMENT                                2854645539
  L2_RQSTS:ALL_DEMAND_DATA_RD                    1971612771
  L2_RQSTS:DEMAND_DATA_RD_HIT                     599543135
  MEM_UOPS_RETIRED:ALL_LOADS                     7680113595
  User time (approx)              37.107 secs     92803909494 cycles  100.0% Time
  CPU_CLK                          2.822GHz
  TLB utilization                215.20 refs/miss      0.420 avg uses
  D1 cache hit,miss ratios        62.8% hits           37.2% misses
  D1 cache utilization (misses)    2.69 refs/miss      0.336 avg hits
  D2 cache hit,miss ratio         51.9% hits           48.1% misses
  D1+D2 cache hit,miss ratio      82.1% hits           17.9% misses
  D1+D2 cache utilization          5.60 refs/miss      0.700 avg hits
  D2 to D1 bandwidth            3243.016MiB/sec  126183217368 bytes
  Average Time per Call                             18.553596 secs
  CrayPat Overhead : Time            0.0%
==============================================================================
```

Table 3: Per function hardware performance counter information. Data obtained on the Hornet system at HLRS featuring Intel Haswell processors. Numbers may vary.

Additional documention is available for CrayPAT and can be accessed either through the man pages for individual commands `intro_craypat, pat_build, and pat_report` OR through the interactive CrayPAT command (requires perftools to be loaded): `pat_help`

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
```

One can visualize 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, exclusive 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 batch.pbs will prevent summarising and allow access to even more data.

```
export PAT_RT_SUMMARY=0
```

Warning! Running tracing experiment with this option 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).

Bypassing the APA: Direct Tracing or Sampling

The APA procedure described above is well suited to get familiar with large applications that are approached for the first time. If tracing overhead does not represent a problem one can directly instrument the application for the tracing experiment with

```
> pat_build -u -g <groups> himeno.exe
```

or

```
> pat_build -w -g <groups> himeno.exe
```

The -w option enables the tracing experiment, where all functions in the <groups> list are traced, and with -u all the user defined functions are traced in addition. Tracing small, frequently called functions can result in a notable overhead. The pat_build utility allows to set different thresholds.
After instrumentation the new application himeno.exe+pat is run on the back end (use job.trace.pbs) and the resulting .xf files can be processed with pat_report as explained above. The resulting reports will contain tables with timings as in `Table 2`. Similarly, the user can perform only a sampling experiment. This is useful to get a rough overview of your application but without instructions for further tracing as in the APA.

The pure sampling experiment requires the -S option to `pat_build`

```
> pat_build -S himeno.exe
```

After instrumentation run the new application himeno.exe+pat on the back end (use job.samp.pbs) and post-process the resulting .xf files with pat_report as explained above. The resulting reports will contain tables with samples as in `Table 4`. You can still see an *.apa file containing information on how to proceed with a tracing experiment, but the current sampling experiment differs from the sampling within the automatic performance analysis explained at the beginning.

Getting Loop Profile information with CrayPat.

As mentioned during the course there is a further experiment related to CrayPat which aims at profiling especially the loops in the considered code. Identifying the time expensive loops in an application is a good basis for the introduction or improvement of OpenMP. And this is also a key ingredient for the reveal tool which will also be presented during this workshop. The loop profiling with CrayPat is only supported within the Cray compilation environment (PrgEnv-cray) and requires an additional compiler flag -h profile_generate. You also have to turn off OpenMP with -h noomp and general optimizations[3]. After a successful build of the Himeno code the procedure is the same as in the tracing experiment, i.e.

```
pat_build -w himeno.exe
```

and when the execution of himeno.exe+pat has finished the *.xf files can be processed with pat_report. The resulting text profile will contain tables as shown below highlighting the time spent in the loops. Am more complex example of loop profiling is given in the first part of the Reveal tutorial.

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE
100.0%	34.798622	--	--	10.0	Total
100.0%	34.798569	--	--	6.0	USER
94.7%	32.939101	0.000002	0.0%	2.0	jacobi
5.3%	1.844985	0.014246	0.9%	1.0	initmt

Table 4: Profile by Function Group and Function

Loop Incl Time%	Loop Incl Time	Time (Loop Adj.)	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
24.2%	32.939088	0.338033	2	26.5	3	50	jacobi.LOOP.1.li.235
21.7%	29.469804	0.000944	53	255.0	255	255	jacobi.LOOP.2.li.239
21.7%	29.468860	0.131651	13515	255.0	255	255	jacobi.LOOP.3.li.240
21.6%	29.337209	29.337209	3446325	511.0	511	511	jacobi.LOOP.4.li.241
2.3%	3.131252	0.001347	53	255.0	255	255	jacobi.LOOP.5.li.262
2.3%	3.129905	0.279740	13515	255.0	255	255	jacobi.LOOP.6.li.263
2.1%	2.850165	2.850165	3446325	511.0	511	511	jacobi.LOOP.7.li.264

Table 5: Loop Stats by Function (from -hprofile_generate)

[3] The environment variable PAT_RT_SUMMARY=0 is not allowed with this experiment.

Reveal

The purpose of Reveal is to assist the user with parallelizing more complicated loops such as those that contain calls to functions. It does not remove dependencies to make loops parallel, but rather calls out issues with parallelization and automates tedious and error-prone tasks for the user. The user will most likely need to perform code restructuring before such loops can successfully run with multiple threads, but reveal provides the first steps for introducing multiple levels of parallelism to a program.

The utilization of Reveal is illustrated by means of the Program VH1 which is contained in the example directory. This exercise[4] will show how Reveal can assist the user in creating a hybrid program when adding OpenMP to a pure MPI program. It explains how to identify loop candidates for parallelization, navigate to these relevant loop candidates, view compiler optimization information, scope variables, view dependency information, and create example OpenMP directives which can then be inserted in the code.

Steps

1) Identify loops that are potential candidates for parallelization because of their trip counts and an estimate the amount of work in the loops.

a) Load modules needed for application, performance data collection and reveal

```
> module load cray-netcdf
> module load perftools
```

b) Collect loop work estimates using perftools software

```
> make -f makefile.hprof clean
> make -f makefile.hprof
> qsub vh1.pbs
```

NOTE: In `makefile.hprof` you can see the usage of the `-h profile_generate` flag

```
ftn -h profile_generate -h keepfiles -h noomp [files].f
```

which is necessary for loop profiling. The -h noomp is necessary to turn off OpenMP and no optimization flags are used. The makefile does the instrumentation for you, you can see the command

```
> pat_build -w ./vhone
```

which yields the file vhone+pat. The batch script vh1.hornet.pbs runs the program on 16 MPI ranks, with the following command. `aprun -n 16 ./vhone+pat > my_output 2>&1`

One can monitor the progress of the job with `qstat` and by looking at the ouput file `my_output`. Once the simulation has successfully finished one can process the performance data:

```
> pat_report vhone_loops.xf > vhone_loops.rpt
```

The output file `vhone_loops.rpt` contains the loop statistics which is shown in`Table 6`.

[4] This exercise is derived from a live demonstration which is a courtesy of Heidi Poxon (Cray US).

Loop Incl Time Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
2.400214	100	25	0	25	sweepy_.LOOP.1.li.32
2.400103	2500	25	0	25	sweepy_.LOOP.2.li.33
2.331944	50	25	0	25	sweepz_.LOOP.05.li.48
2.331883	1250	25	0	25	sweepz_.LOOP.06.li.49
1.266740	187500	107	0	107	riemann_.LOOP.2.li.63
1.172848	50	25	0	25	sweepx2_.LOOP.1.li.28
1.172771	1250	25	0	25	sweepx2_.LOOP.2.li.29
1.164904	50	25	0	25	sweepx1_.LOOP.1.li.28
1.164849	1250	25	0	25	sweepx1_.LOOP.2.li.29
0.388353	20062500	12	0	12	riemann_.LOOP.3.li.64
0.250488	1687500	104	0	108	parabola_.LOOP.6.li.67

Table 6: Loop Stats by Function (from -hprofile_generate)

2) Reveal improves the visibility of compiler optimization feedback by summarizing existing information from loopmark, decompiled code, compiler messages, etc.

a) Rebuild the program with optimization and create program library[5] `vhone.pl`

```
> make -f makefile.pl clean
> make -f makefile.pl
```

b) The `makefile.pl` includes the `-rm` compiler flag which generates listing (.lst) files and decompilation (*.cg and *.opt) files. Please have a look at these files to see readability improvement with reveal later on over searching in large text files. For instance in `riemann.lst` one can see a loopmark legend at the top of the file as well as loopmarks at various loops such as the one at source code line 63. The plus sign points to an additional compiler messages given at bottom of the function. Note that this loop has no OMP directive in front.

c) The `makefile.pl` also includes the `-rd` compiler flag (summarized in `-rmd`) which generates decompilation (*.cg and *.opt) files. The file `init.opt` contains inlined code in `init` from the subroutine `grid`. Even though this subroutine is given in init.opt one cannot easily see where it's called from. Reveal shows grid call site and expanded code from inlining and makes it much easier to see how inlined code fits into source code.

d) Launch reveal with program library

```
> reveal vhone.pl &
```

In order to review loopmark information and get explanations for a compiler message you can go to `loop@63` in subroutine `RIEMANN` in `riemann.f90`. With a right-click on compiler message you get explanations to '*not vectorized for unspecified reason*' directly without the need to look at the end of the routine in another window or shell output. In addition, one can directly see decompiled code and loopmark information and can benefit from improved search capability. Click on `init.f90` in navigation panel and then in source panel. With `Ctrl-F` one can search for grid (see occurrence in orange) until reaching an inlined call. There you can see loopmark information and pseudo code for grid. The compiler messages are categorized in green and red representing positive and negative feedback, respectively.

[5] The perftools module is need for launching Reveal but not necessarily for the generation of a program library.

3) The loop statistics from step 1) shown in `Table 6` can be loaded into Reveal in order to highlight the most time consuming loops. Under **`File->Attach Perf Data`** you can choose vhone_loops.ap2 and then click ok. Note that you can also add the performance statistics when first launching Reveal. From the shell you can do a `> reveal vhone.pl vhone_loops.ap2`. Now you Change navigation view to "Loop Performance"

4) General information on scoping loops can be found in **`Help->Getting Started->Scoping Loop Selection`**. Right click over a loop in navigation panel to bring up the "Scope Loop" option. Select "Scope Loop" for the riemann: loop@63 in the navigation panel. This will open the "Reveal OpenMP Scoping" window. You can choose more loops to scope like loop@28 in sweepx1.f90 in the same way. They are all listed in the "Scope Loops" tab of the "Reveal OpenMP Scoping" window. Now click on "Start Scoping" and let Reveal come back with the appropriate information.

5) The scoping information can now be reviewed. Note that loops with scoping information are colored red or green in the navigation panel. After clicking on a scoped loop in the navigation panel the scoping information appears in the "Reveal OpenMP Scoping" window under the "Scoping Results" tab. The "Scoping Loops" tab should be empty by now.

a) On riemann, loop@63 you can click on variables in the "Scoping Results" tab and see them highlighted in the source panel. You can change the scope of a variable by clicking for instance on 'l' and change scope to "unresolved". Several variables in loop@28 in sweepx1.f90 are unresolved. You can review OpenMP tips under **`Help->OpenMP Tips`** and try to understand why the variables are preventing Reveal to scope them.

b) In the "Scoping Results" tab you have the choice of "Insert Directive" or "Show Directive". If you chose to insert the directive you can see it in the source panel afterwards. Note that the directive will not be saved to your original source unless you click the "Save" button in the upper right corner of the source panel or if you choose to save your work when you exit Reveal. Insert all directives and save them.

e) Note that modifying source is optional and directives inserted by Reveal are intended to be used as placeholders as they may not be complete. The intent is that the user can then move to their favourite editor for code restructuring work with scoping information available from Reveal. Click on **File->Quit**. Select files you wish to update in the "Save Changes" window and then click OK.