THE UNIVERSITY of EDINBURGH

Parallel Performance Tools

Profiling the traffic model using craypat

epcc

Contents

1 Introduction and Aims

So far we have studied the performance of parallel programs by measuring the execution time and inferring where the time was spent using simple models. For example, we attributed the poor scalability of the traffic model for small roads to a large amount of time spent in communications as opposed to calculation (i.e. using Gustfason's law).

However, tools exist which monitor a programme during its execution and measure the time spent in various routines or lines of code. These profiling tools operate in two main ways:

a. Statistical sampling, where the program is interrupted regularly (e.g. every 10 milliseconds) and a record made of where it is in the source code. If there are 10,000 samples and the program was in function "X" 4,000 times, it is assumed that "X" takes up 40% of the run time.
b. Tracing, where the code is modified so that timers are inserted at the start and end of function calls to explicitly measure how long they took.

Although tracing is in principle more accurate, it can be very invasive and have a large effect on a program's performance, e.g. if it makes millions of calls to a very small function then the overhead of tracing that function may become significant.

Portable tools exist, but the Cray Performance Analysis Tool CrayPAT is the easiest to use on ARCHER as it is specifically designed for the Cray architecture. Its ease-of-use makes it ideal for this exercise, but you should be aware that it is only available on Cray platforms.

The aim of this exercise is therefore to use the CrayPAT analysis tool to investigate the performance and scalability of the traffic model. A useful reference is the ARCHER virtual tutorial (slides + video) "Performance analysis on ARCHER using CrayPAT" by Gordon Gibb from 10th May 2017, available at http://www.archer.ac.uk/training/virtual/.

2 Sampling

The stripped-down "lite" version of CrayPAT is the easiest way to get an initial idea of the execution profile of a program on ARCHER. There are four steps:

1. load the appropriate modules;
2. recompile the code;
3. run the code;
4. examine the performance report.
5.

To access the modules use:

```
module load perftools-base
module load perftools-lite
```

You will need to do `make clean` then `make` to ensure the code is completely rebuilt. It can be submitted with your standard PBS script. You can use any configuration you want, but the parameters I found most illustrative were using `NCELL` = 10^5 and 10^8, and running on 24, 48, 96 and 192 cores. Increasing the default value of `maxiter` by a factor of 1000 gives reasonable runtimes for these settings.

2.1 Example results

You will get a text report in the PBS log file as well as in a separate '.rpt' file. There is a lot of information but the summary table is probably the most useful - here is an example where the code has ceased to scale well:

```
Table 1:  Profile by Function Group and Function (top 4 functions shown)

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

 100.0% | 2,323.7 |    -- |    -- |Total
|-----------------------------------------------
|  74.7% | 1,735.1 |    -- |    -- |MPI
||----------------------------------------------
||  60.3% | 1,402.2 | 131.8 |  8.8% |MPI_Allreduce
||  12.8% |   298.2 | 579.8 | 67.4% |MPI_Sendrecv
||   1.5% |    34.7 |  44.3 | 57.3% |MPI_Scatter
||==============================================
|  25.2% |   585.0 |    -- |    -- |USER
||----------------------------------------------
||  19.1% |   443.5 |  48.5 | 10.1% |updateroad
||   6.1% |   141.5 |  67.5 | 33.0% |main
|===============================================
```

`Samp%` gives the fraction of time spent in each routine. You can immediately see that the time is dominated by MPI (75%) over user code (25%). Not surprisingly, `updateroad` takes the majority of non-MPI time.

`Imb Samp%` gives a measure of the load imbalance, i.e. how much the measured time varies across processes.

2.2 Investigation

Look at the CratPAT reports for a number of different runs and compare to the overall run times. Do the results agree with what you would generally expect in terms of Amdahl's and Gustafson's laws?

3 Tracing

In a real application with many hundreds of function calls it can be difficult to decide which functions to trace. Tracing them all can introduce unnecessary overheads – there is little point in collecting statistics on a function that takes less than 1% of the overall runtime. However, the performance profile can change significantly based on, e.g., the input dataset and the number of processors, so it may not be obvious which functions can be ignored.

To address this, CrayPAT supports Automatic Program Analysis (APA) where information on what functions to trace is taken from a previous sampling experiment. There are 8 stages to doing an APA run:

1. load the full perftools module;
2. build the executable;
3. create a new executable to provide the APA information;
4. perform a sampling run;
5. produce a ".apa" file from the ".xf" output which describes what to trace;
6. build a new program with tracing inserted based on the ".apa" file;
7. run the instrumented program;
8. produce a tracing report from the new ".xf" file.

The sampling run at stage 4 will also produce performance information which is more detailed than obtained from the "lite" version used in Section 2. However, here we are not interested in its contents – we are just using the sampling run to help build an instrumented code with tracing enabled.

The commands are:

1. `module unload perftools-lite; module load perftools;`
2. `make clean; make;`
3. `pat_build -o traffic+samp traffic;`
4. edit PBS script to run the new executable `traffic+samp` and submit;
5. `pat_report traffic+samp+9434-13s.xf` (substitute the .xf file from the sampling run you want to use);
6. `pat_build -O traffic+samp+9434-13s.apa;`
7. edit PBS script to run the new executable `traffic+apa` and submit;
8. `pat_report traffic+apa+7508-20t.xf > tracereport.txt` (substitute the .xf file from the tracing run you want to use).

The file `tracereport.txt` contains a lot of information, but again there is a single summary table that gives a general overview. When you run `pat_report`, it produces a ".ap2" file which can be viewed with the Apprentice performance GUI "app2". However, that is beyond the scope of this exercise.

3.1 Example results

Below I show output from a tracing run of the same traffic program we previously analysed with sampling. If you look at the runtimes (or, equivalently, the MCOP figures) you will see that the tracing version runs substantially slower than the sampling version. This unfortunately can have the effect of altering the performance profile – below, the program spent almost 50% of its time in user code compared to 25% previously.

```
Table 1:  Profile by Function Group and Function

  Time% |       Time |      Imb. |  Imb. |        Calls |Group
        |            |      Time | Time% |              | Function
```

```
        |           |           |       |              |  PE=HIDE

 100.0% | 71.065950 |       -- |    -- | 8,000,012.0 |Total
|-----------------------------------------------------------------------
|  48.0% | 34.099764 |       -- |    -- |         2.0 |USER
||----------------------------------------------------------------------
||  48.0% | 34.099730 | 4.577386 | 12.1% |         1.0 |main
||======================================================================
|  27.1% | 19.252680 |       -- |    -- | 2,000,003.0 |MPI_SYNC
||----------------------------------------------------------------------
||  26.4% | 18.755673 | 9.627330 | 51.3% | 2,000,000.0 |MPI_Allreduce(sync)
||======================================================================
|  24.9% | 17.713506 |       -- |    -- | 6,000,007.0 |MPI
||----------------------------------------------------------------------
||  16.9% | 11.993398 | 0.306735 |  2.5% | 2,000,000.0 |MPI_Allreduce
||   8.0% |  5.714678 | 7.367837 | 57.5% | 4,000,000.0 |MPI_Sendrecv
|======================================================================
```

One of the most interesting features is that the time taken by MPI routines simply wating to synchronise (`MPI_SYNC`) is reported separately from their execution time, which helps to diagnose load balance issues. If a program is poorly load balanced, for example, then different processes may call a collective such as `MPI_Allreduce` at different times. Some processes will have to wait for the others to enter the routine which shows up as `MPI_Allreduce(sync)`.

3.2 Investigations

Run your own tracing experiments and compare the results with those from sampling. Questions to answer include:

- Does the amount of time spent in MPI routines behave as you would expect as the number of processes and/or the problem size is varied?
- Do the number of function calls reported by CrayPAT match what you would expect?
- How much is the tracing overhead affected by the problem size?