Iowa High Performance Computing Summer School 2011

Profiling for Code Optimization

This document covers how to use profiling tools on Moffett using the parallel hydrodynamics code `HYDRO` as an example. Details for the use of profiling tools on Moffett are given in Chapter 5 of the SiCortex Programming Guide (a link to this guide is on the IHPC 2011 website).

1. To do profiling, the sample run need not be very long (but should be at least a few seconds to minimize statistical noise, a few minutes is probably ideal).

2. First, you want to recompile the source code for `HYDRO` with the `-g` compiler flag on for profiling. To do this first erase the current executable using `make clean`. Then, edit the Makefile, changing the system option `PROFILE=true`. Then compile the code using `make`.

3. The `HYDRO` input file `profile1a.in` has been provided for your use in these profiling exercises.

1 Profiling with PAPI

PAPI (Performance Application Programming Interface) profiling tool provides a general overview of the code's performance.

1. Run `papiex` on the input file `profile1a.in` for `HYDRO` using 14 processors with the command
   `srun -p scx-comp -n 14 papiex -a ./hydro.e profile1a.in`

2. When this run is complete, it will create a directory in the current directory with the painfully long name
   `<executable>.papiex.<size>.<host>.<proc-id>.<instance>`
   which, in this case, will look something like
   `hydro.e.papiex.14.<host>.<proc-id>.<instance>`
   where the last three for the case I run looked like `scx-m32n13.22891.1`.

3. Go into that directory and look at the file `job_summary.txt`. This file contains a great number of Derived Metrics that tell you a wide range of information about the computational efficiency of your code. Detailed information about all of these metrics is given in Chapter 5 of the SiCortex Programming Guide. I want to point out a few of the more important lines from this file:

   (a) `MFLOPS Aggregate (wallclock) ..... 332.93`
   This is the total floating-point computational speed of your parallel computation in 10^6 FLOPS (FLoating point OPerations per Second). This is the only line that give the sum of the value from each MPI task—all of the other metrics described below are averages over all of the MPI tasks.

   (b) `MFLOPS ..... 24.16`
   Average floating-point computational speed per processor.

   (c) `Non-FP Instructions % ..... 76.87`
   `FP Instructions % ..... 23.13`
   These two lines give the percent of the total executed instructions in your code that were either non-floating point instructions (including integer loads, stores, conditionals, moves, synchronizations, and all forms of integer arithmetic) and the percent that were floating point instructions. In general, you want a higher percentage of FP instructions than Non-FP instructions, as it is the FP instructions that get you closer to computing your answer.

(d) `FP Arith. Instructions % ..... 7.79`
gives the percent of the total executed instructions in your code that actually compute a result (get you closer to your answer).
`FMA Instructions % ..... 1.87`
gives the percentage of all instructions that are variants of the highly efficient, fused multiply/add (madd) instruction.

(e) `Flops per Load/Store ..... 0.23`
gives the number of FLOPS per load or store instruction, and is also known as the *computational intensity*.
`Flops per L1 D-cache Miss ..... 3.70`
is the number of FLOPS for every miss of the L1 data cache. For both of these values, higher numbers are better.

(f) All of the lines involving Cache Hits and Misses pertain to the efficiency of memory access. There is a lot of detailed information here, but the bottom line is
`Total Est. Memory Stall % ..... 36.32`
which gives the percentage of time the application stalled on various levels of the cache hierarchy and main memory. This represents the potential for improved performance with better memory management.

(g) `Total Measured Stall % ..... 12.64`
represents the total amount of time spent in stalls for which papiex could actually count (not estimate), including TLB misses, branch mispredictions, and dependency stalls.
`Total Underestimated Stall % ..... 40.05`
gives a lower bound on the estimate of the total time lost due to estimated stalls on memory, TLB, and branch mispredictions.
`Total Overestimated Stall % ..... 48.96`
gives an upper bound on the estimate the total time lost due to estimated memory stalls and on dependency stalls.

(h) `Ideal MFLOPS (max. dual) ..... 62.14`
`Ideal MFLOPS (cur. dual) ..... 64.67`
give two estimates of the maximum floating-point computational speed per processor if this application ran free of all stalls.

(i) `MPI cycles % ..... 2.08`
gives the total percent of time the code spent inside of the MPI calls (which is the *parallel overhead* due to communication).

4. In addition to the `job_summary.txt` file, each MPI task produces its own file `task_*.txt` with the derived metrics for that processor. Also included in these `task_*.txt` files are memory usage statistics, which can be very valuable in evaluating a parallel code. We can pull out just the single important line from each of these files using
   `grep 'resident peak' task_*.txt`
   which gives the result:
   ```
   task_0.txt:Mem.  resident peak KB ......................  62464
   task_1.txt:Mem.  resident peak KB ......................  61696
   task_10.txt:Mem.  resident peak KB .....................  61888
   task_11.txt:Mem.  resident peak KB .....................  61824
   task_12.txt:Mem.  resident peak KB .....................  61824
   task_13.txt:Mem.  resident peak KB .....................  59136
   task_2.txt:Mem.  resident peak KB ......................  61696
   task_3.txt:Mem.  resident peak KB ......................  61888
   task_4.txt:Mem.  resident peak KB ......................  61696
   task_5.txt:Mem.  resident peak KB ......................  61824
   task_6.txt:Mem.  resident peak KB ......................  61824
   task_7.txt:Mem.  resident peak KB ......................  61824
   task_8.txt:Mem.  resident peak KB ......................  61824
   task_9.txt:Mem.  resident peak KB ......................  61824
   ```
   We can see from this output that the master node (task 0) uses a little more memory than the rest of the nodes (this is pretty typical). Also, task 13 uses less memory than the other tasks; this is due to the fact the the input file specifies a simulation with `nx=1024` gridpoints in the x-direction, but $nproc = 14$ does not divide evenly into 1024, so task 13 gets a slightly smaller subdomain (this is apparent if you look at the how the domain decomposition is accomplished in `hydro_grid.f90`).

5. Note that the poor performance metrics for `HYDRO` above are due in part to the fact that `papiex` profiles the entire run of the code, include the initialization. As with most profiling tools, one can insert into the source code the calipers `papiex_start()` and `papiex_stop()` to focus on specific regions of the code, particularly the main timestep loop.

2 Profiling with MpiP

We can also profile the time the application spends communicating using MPI with the profiling tool `MpiP`. Using this tool it is easy to examine the *load balance*.

1. Run `mpipex` on the input file `profile1a.in` for `HYDRO` using 14 processors with the command
   `srun -p scx-comp -n 14 mpipex ./hydro.e profile1a.in`

2. When this run is complete, it will create a file with the painfully long name
   `<executable>.mpipex.<size>.<host>.<proc-id>.<instance>.txt`

3. Looking at the file `hydro.e.mpiP.14.scx-m32n13.25160.1.txt`, we see

```
@--- MPI Time (seconds) ------------------------------------------------------
--------------------------------------------------------------------------------
 Task   AppTime   MPITime   MPI%
   0      96.9     0.411    0.42
   1      96.8      3.09    3.19
   2      96.8     0.513    0.53
   3      96.8     0.287    0.30
   4      96.8     0.707    0.73
   5      96.8     0.378    0.39
   6      96.8     0.442    0.46
   7      96.8      3.09    3.19
   8      96.8     0.454    0.47
   9      96.8      2.93    3.03
  10      96.8      1.69    1.74
  11      96.8      1.58    1.63
  12      96.8      1.67    1.72
  13      96.8      16.4   16.90
```

   Here we see that task 13 appears to spend much more time in the MPI routines. This is because, since this processor has fewer computational gridpoints to update, it finishes first and has to sit idle waiting for the other processors to catch up.