HPC Workshops

Exercise 1

2019

Christian Arnold

OpenMP and code profiling

1) Blurring an image

The directory `blur_image_openmp` contains a program which can be used to blur an image. The code can run both with and without OpenMP. It can be compiled by typing `make` (this runs the Makefile). You will need to load the appropriate compiler module; ideally an intel compiler (on Hamilton: `module load intel/xe_2018.2`, on Cosma `module load intel_comp`). Make sure you always remove the old binaries by typing `make clean` before recompiling the code with a different setting. You can run the program calling
`./mycode input_file.ppm output_file.ppm`,
where the input and output files should be replaced with the appropriate path (you find a couple of feasible input pictures in the `assets` subdirectory).

1. Compile the code with the default settings and run it 3 times on an input picture of your choice. The code will tell you how much time it needs to blur the image. Take a look at the input and output images.
2. Now add the appropriate compiler flags to allow code profiling with `gprof`. Run the code again with these flags enabled and have a look at the output using `gprof` to identify the hotspots. Which function takes the bulk of the runtime?
   Add the `-g -pg` flags to the `CFLAGS` line in your makefile, recompile and run and use `gprof ./mycode` to have a look at the output. The bulk of the runtime is consumed in `blur_mean_automatic`.
3. Have a look at the OpenMP implementation for this function.
4. Now add the appropriate compiler flags to allow the program to run in parallel, recompile and run the code on 2, 4, 6, 8, 16, 32, and 64 threads. Plot the runtime against the number of threads (including your result for the serial run) and also add a graph showing the expected ideal speedup to the same plot. Include this plot in your solution. *Add the `-fopenmp` flag to the `CFLAGS` line in your makefile. The code should scale relatively well (but of course not perfectly) to the size of your node (probably 16 cores). Beyond the node size, no further improvement should be observed.*
5. Rerun the code on 4 threads using intel Vtune/Amplifier:
   - Load the correct module for Vtune.
   - Run the code performing a HPC analysis:
     `amplxe-cl -collect hpc-performance ./mycode input.ppm output.ppm`
   - Take a look at the generated report using the graphical interface: `amplxe-gui result_directory`

- What fraction of the runtime does the code spend in the serial/parallel parts? What are the main hotspots in the serial/parallel parts?

2) Performing expensive calculations

The directory `add_numbers_serial` contains a code which reads random numbers from a file (`numbers.dat`), performs some (no particularly useful but computationally expensive) calculation for each number in the file and adds all results. The actual sum suffers from floating point roundoff errors, so don't be surprised if the result differs depending how you add the numbers. The code you will find in the source files is not parallelised yet. You can compile it in the same way as for problem 1).

1. Run the code in serial and identify the hotspots using `gprof` or Vtune. Which functions take the bulk of the runtime? Which part of the code would you parallelise to speed it up?
   The bulk of the runtime is spend in the main loop in the `add_numbers` function. It would thus make sense to parallelise this function first.

2. Now parallelise the part you have identified as a hotspot using OpenMP. By how much does the runtime for the calculations decrease if you run on 4 threads?
   See `add_numbers_omp` for code solution. The code will run approximately 2.5 *times faster on 4 cores compared to 1. The scaling is suboptimal because of the reduction which can not be done in parallel.*

3. Run your code on 4 threads using Vtune/Amplifier again. Which functions are the main hotspots in the parallel/serial parts?