332 Advanced Computer Architecture

Exercise (ASSESSED): The "Smooth" Challenge

This is the second of two equally-weighted assessed coursework exercises. **You may work in groups of two or three if you wish, but your report must include an explicit statement of who did what.** Submit your work in a pdf file electronically via CATE.[1] The CATE system will also indicate the deadline for this exercise.

Background

This exercise concerns an computational kernel extracted from a research project at Imperial on dynamic mesh adaptation (https://launchpad.net/pragmatic), led by Gerard Gorman (http://www3.imperial.ac.uk/people/g.gorman) and George Rokos (http://www.doc.ic.ac.uk/~gr409), a PhD student in the Software Performance Optimisation group.

The application context is computational fluid dynamics, using a unstructured mesh as the discretisation of space. Unstructured meshes (usually triangles in 2D, tetrahedra in 3D) are often preferred over simpler (grid-like) structured meshes, as they allow more flexibility in capturing complex geometry and focusing computational effort where it is needed. Mesh adaptivity is used in applications like modelling deep ocean convection:

`http://amcg.ese.ic.ac.uk/index.php?title=Open_ocean_deep_convection`

Computational simulation is essential for understanding this key aspect of the global energy and carbon transport cycle, which is geographically isolated and hard to observe. Fine mesh resolution is needed to resolve the convective structure but is not required across the entire ocean.

The full mesh adaptation process includes operations such as mesh refinement and coarsening, which try to adapt the mesh, so that it is better suited for the solution of a particular partial differential equation. For this exercise, we focus on anisotropic mesh smoothing, an algorithm which tries to improve mesh quality by moving mesh vertices using a metric tensor field. This is a discretised (vertex-wise) field which encodes for each mesh vertex the desired length and orientation of an edge containing that vertex. The objective is for the mesh to adequately represent the state of the fluid. Apart from the average quality, the key overall quality metric is the minimum of local quality measured for each individual element - that is, what we care about is the quality of the *worst* triangle.

Running the benchmark

Getting the benchmark code

Copy the benchmark code to your own directory, e.g.

[1] `https://cate.doc.ic.ac.uk/`

```
prompt> mkdir /homes/yourid/ACA13 (you will probably have already done this)
prompt> cd /homes/yourid/ACA13
prompt> cp -r /homes/phjk/ToyPrograms/ACA13/Smooth ./
```

(The `./` above is the destination of the copy – your current working directory).

List the contents of the benchmark directory:

```
prompt> cd Smooth
prompt> ls
ACA2-2013.cpp  Makefile    Mesh.cpp  small.vtu  Smooth.hpp  SVD2x2.hpp
large.vtu      medium.vtu  Mesh.hpp  Smooth.cpp  SVD2x2.cpp
```

To compile and run the code:

Compile it:

```
prompt> make
```

Run the program:

```
./ACA2-2013 small.vtu
```

This should take 5-15 seconds. You may wish to try with the "medium" and "large" meshes provided; these obviously take much longer to run. All three meshes have been refined already, so their mean quality is quite high. What is low is the quality of the worst element, which is what vertex smoothing tries to fix.

All-out performance

Basically, your job is to figure out how to run this program as fast as you possibly can, and to write a brief report explaining how you did it.

Rules

1. You can choose any hardware platform you wish. You are encouraged to find interesting and diverse machines to experiment with. The goal is high performance on your chosen platform, so it is OK to choose an interesting machine even if it's not the fastest available. On Linux, type `cat /proc/cpuinfo`.

   You are encouraged to look at graphics processors, mobile devices, etc. If in doubt please ask. If you are interested in using research facilities please ask.

2. Make sure that the results are correct *every* time you run the program. The program checks that the mean and minimum quality metrics are sufficiently improved - this is the correctness criterion (that is, it is not strictly necessary to get *exactly* the same answer, as long as the mesh quality is OK).

3. Make sure the machine is quiescent before doing timing experiments. Always repeat experiments for statistical significance.

4. Always script your experiments, so that it is easy to check how each data point was generated, and it's easy to rerun your experiments when you change something.

5. Choose a mesh size which suits the performance of the machine you choose - the runtime must be large enough for any improvements to be evident.

6. Take care to report precisely and fully the details of the hardware (CPU part number, clock rate) and software (compiler version numbers, OS etc).

7. If you use a laptop, make sure it's plugged into mains power, as some mobile platforms can adapt and mess up your experiments.

8. You can achieve full marks even if you do not achieve the maximum performance.

9. Marks are awarded for

    - Systematic analysis of the application's behaviour
    - Systematic evaluation of performance improvement hypotheses
    - Drawing conclusions from your experience
    - A professional, well-presented report detailing the results of your work.

10. You should produce a report in the style of an academic paper for presentation at an international conference such as Supercomputing.[2] The report should be not more than seven pages in length.

Changing the rules

If you want to bend any of these rules just ask.

Hints, tools and techniques

Building on other platforms: Building the code requires installation of VTK5.10. This is installed for you in the CSG machines (in homes/gr409/public_html/vtk). For other platforms you should be able to install a binary package; you will need to modify the Makefile accordingly.

Performance analysis tools: You may find it useful to find out about:

- cachegrind and cg_annotate
- kcachegrind - graphical interface to cachegrind: http://kcachegrind.sourceforge.net
- oprofile: http://oprofile.sourceforge.net - performance profiling by sampling hardware performance counters. Should install with apt-get on Ubuntu; works nicely with kcachegrind.
- Likwid-perfctr (http://code.google.com/p/likwid/wiki/LikwidPerfCtr) - handy performance counter tools that you can easily install on your own (non-virtual) Linux system.
- Intel's VTune - tool (Windows, Linux, MacOS) for understanding CPU performance issues and mapping them back to source code (free trial). http://www.intel.com/software/products/vtune.

[2] http://sc12.supercomputing.org/

- AMD's CodeAnalyst (http://developer.amd.com/tools/codeanalyst/pages/default.aspx)
- Apple's XCode "Instruments" performance tools
- OpenSpeedshop for Linux: http://www.openspeedshop.org/wp/

Compilers You could investigate the potential benefits of using more sophisticated compilers:

- Intel's compilers: these are not installed on CSG linux systems but you should be able to download student and evaluation copies for your own machines from http://www.intel.com/software/products/compilers
- AMD's APP SDK for OpenCL on x86 CPUs and AMD GPUs: http://developer.amd.com/gpu/AMDAPPSDK
- NVIDIA's CUDA and OpenCL compilers: http://developer.nvidia.com/object/gpucomputing.html

Source code modifications

You are strongly invited to modify the source code to investigate performance optimisation opportunities.

How to finish The main criterion for assessment is this: you should have a reasonably sensible hypothesis for how to improve performance, and you should evaluate your hypothesis in a systematic way, using experiments together, if possible, with analysis.

What to hand in Hand in a concise report which

- Explains what hardware and software you used,
- What hypothesis (or hypotheses) you investigated,
- How you evaluated what the potential advantage could be,
- How you explored the effectiveness of the approach experimentally
- What conclusions can you draw from your work
- If you worked in a group, indicate who was responsible for what.

Please do not write more than seven pages.

George Rokos, Paul H.J. Kelly, Gerard Gorman, Doru Bercea, Imperial College London, 2013