Performance Programming Practical

Overview

This practical can be done in either Fortran or C. The aim is to take a poorly written code, and make it go faster. This code is a simple Molecular Dynamics program: it uses classical (Newtonian) dynamics to move some atoms around according to the forces acting on them. It does this by integrating an 'equation of motion' so given a set of initial positions, it can calculate all future positions. The output is written to a log file called '`simple_md.log`'. This important – you want to make sure that as you modify the original code, you do not change any of the results, so you must keep a reference copy of the log file to check against!

Compiling

Download the file '`simple_md.tar`' and then unpack using

```
tar -xf simple_md.tar
```

and then build using either

Fortran:

```
make simple_md
```

C:

```
make simple_md_C
```

Reference Run

First you should run the once to get a benchmark answer and baseline timing:

```
Fortran:    /usr/bin/time ./simple_md
C:          /usr/bin/time ./simple_md_C
```

And then when the run has finished make a backup of the output log file

```
cp simple_md.log simple_md.log.ref
```

so you can use this to check that (apart from minor changes due to rounding) nothing fundamental has changed with the answers!

Compiler-based Optimization

To see how well the compiler can improve the code, we need to tell it to do so!

```
Fortran:    make simple_md_opt
C:          make simple_md_C_opt
```

This now does full optimization, using some of the tricks that were discussed in the lectures. Now re-run the code,

```
Fortran:    /usr/bin/time ./simple_md_opt
C:          /usr/bin/time ./simple_md_C_opt
```

and compare the timings. The compiler should have made the code significantly faster with very little effort from you! You should always check that the results have not changed:

```
            diff simple_md.log simple_md.log.ref
```

However, you can do a better job than the compiler! So you need to know where in the code it spends its time – which means using the 'gprof' and 'gcov' programs as in the lectures.

Profiling

You need to add the '-pg' flag to the compiler – and the Makefile has been set up to make this very easy:

```
Fortran:    make simple_md_pg
C:          make simple_md_C_pg
```

Now execute the code and then use the profiler:

```
Fortran:    gprof ./simple_md_pg > gprof.out
C:          gprof ./simple_md_C_pg > gprof.out
```

and then use 'more' or an editor to view the output file

```
            more gprof.out
```

to see which functions/subroutines are the bottlenecks.

To get a more detailed line-by-line picture, you should use the coverage analysis tool:

```
Fortran:    gcov ./simple_md.f90
C:          gcov ./simple_md_C_pg.gcno ./md_stuff_C_pg.gcno
```

[NB the C build uses multiple source files, so need to tell gcov about each gcno file generated to see the profile of each source file at a time.]

and then view the resulting output file

```
Fortran:    more ./simple_md.f90.gcov
```

C:
```
more ./simple_md.c.gcov ./md_stuff.c.gcov
```

Manual Optimizations

Now that you know where the bottlenecks are, you can try to eliminate them. First, make a backup of the source files before you make any edits, and then see what improvements you can make. You can use the Makefile to re-build the code, and then use the combination of `time/gprof/gcov` as appropriate. Always `diff` your output logfile against the reference logfile to make sure that your optimizations have not broken anything!

Hints:

1) Floating point optimizations
   Look at the most often executed lines. Is there any way of simplifying the maths? Some floating point operations are much more expensive than others …
2) Algorithmic optimizations
   Can you improve /change the algorithms that are being used? For example, look at the force calculation for particles '`i`' and '`j`' – are there any symmetries that can be used? This might require a bit of extra code to be added to get the answers the same but it can make a big difference!
3) Compiler flags
   Look at the flags in the Makefile, and read the manpage for the compiler you are using. Experiment with different flags to turn on different optimizations, or more advanced machine code instruction sets, etc.
4) Loop optimizations
   Are there any calculations that are loop invariant and can therefore be moved? Can you unroll or re-order any loops? Or eliminate any if-branches etc?

With a bit of effort, it should be possible to get over x100 speedup between the original code compiled at `-O0` and your optimized code with the best compiler flags. See how fast you can get it! If the code starts to become too quick to accurately time, then you should make it do more MD steps by changing the value of '`nsteps`' from 100 to 1000 etc.

Have fun!

Matt Probert v2.0 18/09/2019