Loop parallelism

Nick Brown

8 March 2017

1 Introduction

In this practical we are going to go back to the pollution in a pipe problem, where the implementation strategy pattern that you were working with was Single Program Multiple Data, with the same code solving different parts of the domain. Instead in this practical we are going to look at implementing the decomposition using loop parallelism. OpenMP is a more suitable technology to use for loop parallelism than MPI, therefore in this practical we will go back to the serial version of the code (in ***serial.c*** or ***serial.F90***) and parallelise this using OpenMP. It is an interesting exercise as it also illustrates the differences between using these two technologies. From a code perspective you will probably find the OpenMP version simpler than the MPI version, however the MPI version is far more scalable as it can be run across multiple processes.

2 OpenMP

If you took our threaded programming module, then you will have covered OpenMP already. If not, or if you are a bit rusty on OpenMP, then a good online resource can be found at https://computing.llnl.gov/tutorials/openMP/ which should help you with the implementation of this practical.

Remember, in order to enable OpenMP directives in the compiler then you will need to provide the additional ***-fopenmp*** argument for GCC (and **-openmp** for Intel.)

3 Loop parallelism

You should do this practical on the CP Lab machine as Cirrus will be unavailable

Starting with the original serial version of the pollution code (in ***serial.c*** or ***serial.F90***) :

- Initially just concentrate on parallelising the initial residual calculation, the current solution residual calculation and the Jacobi solver. It is entirely up to you how you do this, I would suggest putting a parallel region around the entirety of your intended parallelism, and then add in the appropriate directives into here, but you can do it whichever way you prefer.
- The residual calculations will need to accumulate so you probably want some sort of ***reduction*** clause in these ***do/for*** loop directives.

- Consider carefully which variables will need to be private (I would suggest you loop counters as a minimum)
- You will also probably need to use the ***single*** and/or ***master*** directives (for instance when swapping the array pointers around.) Remember that, by default, all threads will block on ***single*** whereas ***master*** is non-blocking.
- You can set the number of threads by exporting the bash environment variable ***OMP_NUM_THREADS*** to be the desired number before running your code.