OpenMP Exercises

Erik Schnetter
(Dated: June 6-8, 2012, IHPC 2012, Iowa City)

There are three exercises, ordered by increasing level difficulty. In each exercise, a serial program is given, and your task is to parallelise this program with OpenMP. The parallelised program should run faster than the serial one (since this is the reason one wants to parallelise), and must still yield the same result (otherwise there is an error).

The first and second exercises require Fortran programming. Fortran is easy to learn, and in summer school settings, Fortran code is much easier to get correct than C or C++. Do not try to rewrite the programs in C or C++ – even if you prefer these languages, you will learn more by solving these exercises in Fortran.

Apply the methods outlined in the lecture to parallelise these codes. Use the Cheat Sheet for instructions on how to compile and run OpenMP codes on Helium. There is a makefile for each example program; use `make` to build the serial version of the program. Modify the makefile as necessary.

The first and the second exercises are similar, except that the second is more complex. If you find the first exercise too easy, skip to the second. The third exercise is open-ended and also requires MPI programming. You may skip this exercise.

Each exercise comes with a brief outline of the numerical methods that are used in the respective codes. You do not need to understand these methods to complete the exercise. It suffices to examine the codes and understand their data dependencies (which variables are used where, and in what way).

Exercise 1: Calculate π

We want to calculate an approximation to the number $\pi = 3.14\ldots$. While there exist many very efficient methods to do this, we intentionally use an algorithm that is simple and requires much computing power (aka is "stupid). This will allow us to apply an OpenMP parallelisation more easily.

We approximate π by calculating the area of the unit circle, which is π. We do this by integrating over the first quadrant of the circle, i.e. we evaluate $\int_0^1 \sqrt{1-x^2}\,dx = \pi/4$. We discretise the integral into n vertical boxes, each with a width $\Delta x = 1/n$, and with a height $y_i = \sqrt{1-x_i^2}$ where $x_i = i\,\Delta x$, and where $i = 1\ldots n$.

The example code is called `calcpi.f90`, and is a fully functional serial code.

Note that the code uses a non-default integer kind that provides at least 15 digits. This is necessary because the default integer kind can only hold up to (about) 10 digits, and we will want to use more than 10^{10} integration steps. (This corresponds roughly to using `long long` in C.)

Your tasks are:

1. Run the serical code as-is. Modify the parameter n, which defines the number of integration steps, and see how this influences the accuracy and the run-time.

2. Parallelise the code. Run the code with varying numbers of OpenMP threads. Ensure that the result is the same as for the serial code, and is independent of the number of OpenMP threads you are using. Remember to use the `-openmp` compiler flag, and to set `OMP_NUM_THREADS` when running! See the Cheat Sheet.

3. Choose a large number of integration steps that runs for about a minute with the serial code (maybe $n = 10^{13}$?). Compare performance for different numbers of OpenMP threads N_{OpenMP}. Run on one of the 12-core nodes of Helium, and create a table comparing run times for $N_{\mathrm{OpenMP}} \in [1, 2, 3, 6, 12, 24]$. Create a log-plot with the results: x axis shows number of OpenMP threads with logarithmic scale, y axis shows wall time with a linear scale beginning at 0.

4. Explain the behaviour of this graph.

Exercise 2: Poisson Equation

Many systems in science and engineering are described by PDEs (Partial Differential Equations). A simple example for a PDE is the *Poisson Equation*, describing e.g. the gravitational potential or the electric potential, if the mass or charge distribution is given. For simplicity, we consider here a system with two spatial dimensions and Cartesian coordinates x and y.

Background

Given a mass distribution $\rho(x, y)$, the resulting gravitational potential $U(x, y)$ is given by

$$\Delta U(x, y) \;=\; 4\pi\rho(x, y) \quad , \tag{1}$$

where Δ is the Laplace operator div $\cdot$ grad, i.e. $\partial_x^2 + \partial_y^2$ in our case.

A particularly simple way of calculating $U(x, y)$ is the *Jacobi Method*, which is defined as follows:

1. Choose any initial guess for the potential, e.g. $U(x, y) := 0$
2. Evaluate the residual: $r(x, y) := \Delta U(x, y) - 4\pi\rho(x, y)$
3. Calculate the L_2 norm of the residual: $L_2[r] := \left(\int |r(x, y)|^2 \, dx \, dy/V\right)^{1/2}$, where V is the volume of the domain
4. If the L_2 norm of the residual is small enough, we are done
5. Otherwise, add a small multiple of the residual to the potential: $U(x, y) \to U(x, y) + \alpha r(x, y)$
6. Repeat from step 2

It is important to choose a good value for $\alpha > 0$. If α is too large, the Jacobi method is unstable, and the residual will grow without bounds. If α is small enough, the residual will converge to zero.

> *Note:* This algorithm is both simple and spectacularly inefficient. We use it here only because it leads to a simple code. Do not use this algorithm to solve a real-world problem. Two much better classes of algorithms are *multi-grid methods* and *Krylov subspace methods*. There exist efficient, generic, ready-to-use libraries for these methods, such as e.g. PETSc.

We discretise the Poisson equation by employing *finite differences*. We represent the domain by two-dimensional arrays $[0, n] \times [0, n]$, where $n + 1$ is the number of grid points, and $h = 1/n$ is the spatial resolution. We discretise the Laplace operator via second-order centred finite differences:

$$\left(\partial_x^2 + \partial_y^2\right) U(x, y) \;:=\; \frac{U_{i-1,j} - 2\,U_{i,j} + U_{i+1,j}}{h^2} + \frac{U_{i,j-1} - 2\,U_{i,j} + U_{i,j+1}}{h^2} \quad . \tag{2}$$

Implementation

The example code is called `potential.f90`, and is a fully functional serial code. It consists of three parts: Some declarations in the beginning, a set of subroutines for various tasks, and the main program driving the calculation.

There are two parameters that can be modified. The parameter n defines the spatial resolution, and the parameter niters specifies the number of iterations in the Jacobi method. Larger values for these parameters will increase both the accuracy and the run time. The default settings of these parameters is tuned for quick test runs; you will need to increase them for production simulations.

Tasks

Your tasks are:

1. Run the serical code as-is. Modify the parameters n and niters, and see how this influences the accuracy and the run-time.

2. Examine the code. Which parts of the code can be parallelised? Where is most of the time spent? Which parts should therefore be parallelised?

3. Parallelise the code. Run the code with varying numbers of OpenMP threads. Ensure that the result is the same as for the serial code, and is independent of the number of OpenMP threads you are using.

4. Set $n = 200$, and choose niters so that the code runs for about a minute (maybe niters $= 10^4$). Compare performance for different numbers of OpenMP threads N_{OpenMP}. Run on one of the 12-core nodes of Helium, and create a table comparing run times for $N_{\mathrm{OpenMP}} \in [1, 2, 3, 6, 12]$. Create a log-plot with the results: x axis shows number of OpenMP threads with logarithmic scale, y axis shows wall time with a linear scale beginning at 0.

5. Choose a large value for n, e.g. $n = 10^4$. Reduce niters correspondingly, e.g niters $= 100$. Compare run times for different values of N_{OpenMP}. What do you observe? Why?

Exercise 3: Hybrid Parallelism

This is an open-ended exercise for the adventurous. Its aim is to demonstrate that the methods shown in this summer school do not stand in isolation, but may need to be combined in real-world programs to achieve good performance.

Take one of the MPI examples from yesterday's exercises, and add an OpenMP parallelisation. That is, the resulting code should be parallelised via both MPI and OpenMP at the same time. The basic idea is that each node of an HPC system will run one (or a few) MPI processes, and each MPI process will in turn run on multiple cores via OpenMP multi-threading.

Note:

1. To keep things simple, ensure that all MPI calls remain in serial regions of the code.

2. What determines how many MPI processes should be running on a single compute node? What determines how many OpenMP threads should be used on a compute node?

3. You will need to use both `mpirun` and `OMP_NUM_THREADS`. You will need to look at the system's documentation to find out how to do this – this is usually not trivial.

4. Compare the performance of this code with various combinations of N_{MPI} (number of MPI processes) and N_{OpenMP} (number of OpenMP threads). Examine in particular the cases where $N_{\mathrm{MPI}} = 1$ or $N_{\mathrm{OpenMP}} = 1$.