An Introduction to HPC and Scientific Computing – How to Multi-task on CPUs Using OpenMP

– Practical Sessions –

Ian Bush ***ian.bush@oerc.ox.ac.uk***

Contents

1 Introduction

In these exercises we will look at programming with OpenMP. When answering them firstly remember to always check that any programs you have written are correct, it doesn't matter how fast it runs or how well it scales if the answers are wrong! Secondly in some of the questions you are asked to look at the performance of the code. To do this you really must do this with the batch system (we provide the batch scripts) as running on the front end will not give reliable timings due to the interaction with other users.

The practicals will be performed on htc-login. Log on using the username and password that have been provided, and the files are under the directory `lecture6`

Remember, to log in use:
`ssh -CX [username]@htc-login.arc.ox.ac.uk`

For getting all the practicals do not forget to use command `git pull` in the previous cloned directory or download the repository again with the command:
`git clone https://github.com/wesarmour/CWM-in-HPC-and-Scientific-Computing-2023.git`

Also, this time all executions will go through the SLURM scheduler system. When submitting the sbatch file will ask for a free node, where the job will run and a new file will appear in the directory with the output. How to run the sbatch is described in each tasks.

2 Hello World

About

This exercise introduces the basic functionality of OpenMP with one of the simplest programs possible, the "hello world" code. The source files are found in the directory `hello`:

- `hello_omp.c` – the OpenMP "hello world" program

Tasks

1. Before we start each question we need to set up the software environment. First issue a `module purge` command to ensure that you are stating from a clean sheet. Now we need to load the module that makes the compiler available. Throughout these exercises we shall use gnu compiler. To load this issue `module load gcc`. If you have time at the end you might to try the practicals with the intel compiler. If so instead use `module load intel-compilers`. You won't need to change any of your programs if you do this, but the makefiles will need changing as the intel compiler accepts different flags.
2. Edit the "hello world" program `hello_omp.c` and try to understand it
3. Edit the file `makefile` and try to figure out what it does. Compile the OpenMP "hello world" programs by running the command `make`.

4. Run the executable. To do this, submit the provided job script `run_hello_omp.sh` with the command `sbatch`. Observe the output, which will be in a file called like `slurmXXXXX.out` created by the batch environment in the directory that you submitted the job, and try to correlate that with the source code.

5. Run the executable several times. Does the order in which each thread outputs a message change?

6. Repeat the exercise but using just 4 threads. How do you achieve this?

3 Trapezoidal Integration

About

This exercise is based on a more "real-life" application than the previous: the computation of a function integral using the trapezoidal rule.

The source files are found in the directory `integral`:

- `integral.c` – a serial implementation,
- `integral_omp.c` – the OpenMP implementation

You are asked to complete the programming of the OpenMP source code, starting from the existing files. The OpenMP code uses parallel region construct and the OpenMP reduction operation.

Notes

- The serial implementations use the function `trapInt`, defined in file `integral`. With two real variables (`a` and `b`) and an integer (`N`) as arguments, this function computes the integral of the function `f()` (also defined in `integral`) between `a` and `b`. `trapInt` uses the simple trapezoidal rule on `N` trapeziums to compute the integral of `f()`.
- All code is programmed to read the parameters `a`, `b` and `N` from standard input. The standard input can be redirected from a file, such that the exectables can be run in a non-interactive manner. The input file is provided; to run the serial implementation, for example, the command is `integral < integral.inp`.
- All source code uses real numbers in double precision.
- Time is measured in the serial code `integral` using the function `wall_clock_time` (a wrapper around `gettimeofday`). In the OpenMP code, it is replaced with the OpenMP function `omp_get_wtime`, although `wall_clock_time` would work just as well.

Tasks

1. It's good practice to make sure that you have a clean software environment at the start of each question. So again issue `module purge` and `module load gcc`. Are you sure you understand what these commands are doing?

2. Compile the serial code (`make integral`) and run it with the command `./integral < integral.inp`, using the input file provided `integral.inp`. Using equal values for `a` and `b` but opposite in sign and `N=1024`, the integral value is approximately pi. This is correct as the integral that is being evaluated is `f(x)=2.0*sqrt(1.0-x*x)`, and the bounds are -1 to 1; the analytic answer is exactly pi.

3. Edit `integral_omp` and program the threaded computation of the integral. To achieve this simply transform the original serial `trapInt` function by threading the `for` loop and use the threaded function to compute the integral. Remember that the OpenMP directive must include a reduction on the integral value variable `v`. Compile the OpenMP code with the command `make integral_omp`, run it using the script `run_integral_omp.sh` provided and verify that it produces the correct answer: `sbatch run_integral_omp.sh`.

4. Look at how the time to solution for your OpenMP program varies with the number of intervals being used (the last number in the input file), and also the number of threads. Examine 10000, 100000 and 1000000 intervals, and 1, 2, 4, 8 and 16 threads. Plot a speed up curve for your results - Speed up is how much faster a calculation runs on P threads compared to the same calculation on 1 thread, i.e. `S=time(1)/time(P)`. Can you make sense of these results?

4 The 1D Heat Equation

About

The code in this exercise solves the one-dimensional heat equation using finite differences and OpenMP programming. The finite-difference method provides a realistic yet simple algorithmic framework to exercise some of the important basic aspects of parallel programming: communication, synchronisation and reduction.

The source files are found in the directory `heat`. The source files are:

- `heat.c` – a serial implementation
- `heat_omp.c` – the corresponding OpenMP implementation
- `heat_omp_easy.c` – the "easy" OpenMP implementation

Notes

Without going deep into the numerical analysis aspects, here are the key concepts to help you go though this exercise.

The 1D heat equation is a second order partial differential equation, whose solution describes the time-evolution of temperature along a heat-conducting rod, starting from an initial prescribed temperature distribution. In this exercise, it is considered that the temperatures at both ends (the boundary conditions) of the rod are constant (Dirichlet type) and equal to zero. Mathematically, we seek the solution to

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial t^2} \tag{1}$$

$$u(x,0) = u_0(x) \tag{2}$$

$$u(0,t) = u(1,t) = 0 \tag{3}$$

The numerical solution to this problem consists of an array of unknown temperature values corresponding to equally spaced points along the rod at any given time. The numerical algorithm chosen for this exercise updates this array within a discrete time-loop, with all the temperature values at one time-level depending on the temperature at the previous time-level (explicit scheme). The outcome of the algorithm is thus the temperature distribution at the final simulated time.

Assuming the rod extends from 0 to 1, and `n` is the number of points along the rod (corresponding to `n-1` equal-length intervals), the coordinate of the points are `x[j] = j*dx` with `dx = 1/(n-1)` and `j` going from `0` to `n-1`. Also, denoting by `n_time_steps` the number of time levels for which the solution is computed and by `u` the array of temperature values, the numerical algorithm is the following time for-loop in C

```
// time loop
for (t=0; t<n_time_steps; t++) {

  // store the old solution
  for (j=1; j<n-1; j++) {
    uo[j] = u[j];
  }

  // finite difference scheme
  for (j=1; j<n-1; j++) {
    u[j] = uo[j] + nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
  }
}
```

The above code snippet represent the core of the serial source code `heat`.

Note how at each time step `n` and before the update, the current temperature values in `u` are copied to an additional array `uo`, from which they are used within the update. (A pointer swap is obviously more efficient for achieving this but less clear in an exercise.)

The key part of this algorithm is the C line

```
u[j] = uo[j] + nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
```

in which `nu` is a constant that has to be less than or equal to 0.5 in order to preserve numerical stability of the scheme. The array values `u` are updated based on the values of the previous time-step, stored in `uo`. At each point `j` along the rod, the update is carried out in a stencil involving three points along the rod (`j-1`, `j` itself and `j+1` as well as two points along the time-line (again `j` but for time steps `t` and `t-1`).

Tasks

1. Continue practicing the good habit of making sure you have exactly the software environment you need by again issuing `module purge` and `module load gcc`.

2. Edit the serial code `heat.c` and try to understand what it is doing.

   - Notice how the data in the finite difference stencil is used, how the maximum change at each time step is evaluated and how the error of the numerical solution is computed at the end of the code. (Starting from a sinusoidal initial temperature distribution and imposing constant zero temperature at the ends of the rod at all times, the exact analytic solution of the heat equation is known and the error can be measured exactly against this.)
   - Compile the code with the command `make heat`. Run the executable using the provided input file (`./heat < heat.inp`) and using the submission script `run_heat.sh`. Also, you can edit `heat.inp` in order to change the parameters of the simulation (number of points along the rod and number of time-steps).

3. Edit the OpenMP code `heat_omp.c` and understand how it differs from the serial version in terms of loop threading.

   - Compile the code with the command `make heat_omp` and execute the code redirecting the standard input (*i.e.* using `< heat.inp`) to provide the run parameters and using the provided job script `run_heat_omp.sh`.

     Check the code is running correctly by comparing the output with the serial code.
   - Identify the loop that initialises the values of the array `u` and parallelise it using an OpenMP parallel region.
   - Identify the loop computing the root mean square error between the computed and analytic solution and parallelise it using an OpenMP parallel region. Is the error value computed close to zero as expected? Make sure a reduce statement on the variable `rms` exists. (The parts in the source where you are asked to insert code is marked with ellipsis.)

4. Measure the performance of the OpenMP code relative to that of the serial code. In order to do so, use a large number of points ($\sim$ 100k) and a large number of time steps ($\sim$ 10k). Pay particular attention to the scaling, using 1, 2, 4, 8 and 16 processes/threads. How do the parallel codes run on 1 thread compare to the serial one? Does doubling the number of processes/threads double the performance? To do this you will need to add the timers we used in the integral code to the heat code - when making these additions make sure you are measuring what you want to measure. It will probably also be useful to modify the code to print out how many threads are being used. Be careful when doing this so that you don't get this being reported multiple times.