Exercise sheet

"Shared Memory Programming With OpenMP"

March 2021

General remarks: parallel performance assessment

In many exercises we ask for performance assessment. For that you need to put timers into the code, as shown in the lectures. However it is often not easy to get stable and reproducible results.

A few general remarks:

- Avoid hardware shared between a number of users
  - In particular front-end nodes of the SNIC clusters are used simultaneously by a number of users and some users abuse them. In particular for testing parallel performance results obtained on the front-end show large variations and are typically worthless. Most SNIC centres actually regard testing of parallel codes on front-end nodes as abuse. Parallel tests use all the cores of the front end system and impede on other users.
  - The cores of a compute node are not truly independent - this will be discussed in detail throughout the course. While shared nodes are ok for correctness checking and many production situations, they are problematic regarding performance testing. For the slurm scheduler the line

    ```
    #SBATCH --exclusive
    ```

    in the header of the submission script gives you exclusive access to a node. You should use this with consideration for the other course participants. This will charge you for all the cores assigexned to the job, whether or not you use them.
- To get stable and reproducible results you might need to place a loop inside the code that runs the timing section many times. This smoothes over a lot of the operational noise typical for a UNIX system.
- If you are timing code sections in the micro-second range, you should place your timers outside that loop. The starting timer directly before and the stopping timer directly after the loop.
- Use thread binding, that also reduces the effect operational noise has on your code. Be aware that the map of your binding will affect the timing, but this way you get at least results for a situation you understand instead of something that is all-over-the-place.
- Experiment with the problem size. OpenMP calls carry a significant overhead. You need enough computational work between the OpenMP directives to get a benefit from the parallelisation. **However:** Correctness tests and debugging is best performed on small problem sizes.

Important practicality:

Keep copies of all solutions, including serial codes developed prior to starting the parallelisation. You may need them as starting point for later exercises.

1 First Day 1

1.1 Exercise: Compiling and Running OpenMP in an HPC environment

text

1.2 Exercise: Serial and threaded "hello world" code

In this exercise we write a parallel version of the Worlds most famous application: "*Hello World*"

1.2.1 Serial code

Write a C or Fortran code that prints:

```
Hello world, I am a serial code!
```

Execute your codes using the job scheduler (batch system).

1.2.2 Parallel code

Now parallelise your code using OpenMP. You should place a print statement inside a parallel region. The thread number should be controlled with the appropriate environment variable. Each thread should print its thread number and the total number of threads utilized concurrently. For example thread number 2, when utilising a total of 4 threads should print

```
Hello world, I am thread 2 of 4 threads!
```

The parallel version of the code should still be able to compile serially and produce the same result as in exercise 1.2.1. To achieve this, you have to use C-preprocessor directives or the "`!$`"-sentinel.

2 Second Day

2.1 Exercise: Calculating Pi

It can be shown that:

$$\sum_{n=1}^{N} \frac{1}{n^2} \overset{N\to\infty}{\longrightarrow} \frac{\pi^2}{6} \tag{1}$$

2.1.1 Serial program

We provide serial template program in C and Fortran, which sum $1/n^2$ for a large value of N. Experiment with different values of N and make sure your result is correct.

2.1.2 Parallel program, manual work distribution

Make a copy of the serial template program (or write your own if you prefer) and parallelise it using the OpenMP construct *parallel*.

In this exercise you should write the code (C or Fortran) to distribute the work onto the threads. That is, starting with the value of N, using the number of threads and the thread-id number, each thread should determine the n-range it is working on. For this exercise it is ok, if you assume that N is divisible by the thread count. Also due to the complexity of this management code, it is ok if your parallel OpenMP version of the code would not compile in serial.

You will need to introduce additional variables. Think carefully, which variables need to be declared as `shared` and which ones need to be declared as `private`. We strongly encourage the use of a `default(none)` clause. Please do not use any other declarations here. In this exercise it is easy to create a *data race*. This has to be avoided while still maintaining high performance (minimise serialisation).

Once the code is finished you should check that

1. The parallel code is producing a correct results
2. The results are independent of the thread count
3. Introduce a timer into your code and experiment with different values of N. Rerun you code with 1, 2, 4, 8, 16 and 28 threads. For small values of N you should notice that the code will run slower for larger thread counts, while for large values of N it will run faster when more threads are used.
4. Do repeated runs of the same run. In particular when using all the threads per node, do you get the same times.

2.2 Exercise: Summation over triangular area

Using polar coordinates it can be shown

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \exp(-(x^2+y^2))dxdy = \pi \tag{2}$$

The double integral on the left hand side can, using the symmetries available be approximated by

$$\lim_{h\to 0}\lim_{N\to\infty}\left[8\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} h^2 \exp\left(-h^2(i^2+j^2)\right)\right] \tag{3}$$

Note that the second sum has its index starting at $i+1$ instead of 0.

2.2.1 Serial code

In a first step you should write a serial code to implement the above sum. Using $N = 40000$ and $h = 0.0002$ gives an estimate for π which is 4 digit accurate. Before proceeding, make sure you keep a copy of your serial code - we need it in a later exercise.

2.2.2 OpenMP parallelisation

Once you have a working serial code, you should parallelise it using OpenMP. This is best accomplished using the loop construct. To achieve good performance, each thread should sum its contribution into a private variable and only in the end do a protected update to a shared variable for the final result. You might want to clean up your code that it compiles in serial and parallel.

- Check that your results do not change when you use different thread numbers.
- Measure the performance. Are you achieving good parallel speed up when using multiple threads? Run your program using 1, 2, 4, 8 and 16 threads.

2.2.3 Schedule

If you haven't specified a `schedule` clause, you should have observed that the speed-up on multiple processors was poor. Specifying a schedule should improve the situation. When using 16 threads, investigate the performance for a static schedule for a chunk size of 1, 10, 100, 1250, 2500. Can you achieve better results using dynamic scheduling?

3 Third Day

4 Fourth Day