<u>**Exercise sheet**</u>

"Shared Memory Programming With OpenMP"

Authors: Joachim Hein, Pedro Ojeda May

2021, 2022, 2023

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 a cluster are simultaneously used by a number of users. This results in a variable background workload on the node, which has a strong influence on the performance of any application running on the nodes. In particular for testing parallel performance results obtained on the front-end show large variations and are typically worthless. Most HPC centres actually regard testing of parallel codes on front-end nodes as abuse. Parallel tests utilise many or 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 assigned 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 Exercise: Compiling and Running OpenMP in an HPC environment

Before compiling your code it is recommended to purge the module system:

```
ml purge
```

Now, we can compile our code with the compiler we choose:

1.1.1 GNU Compilers

```
ml purge
module load GCC/11.3.0
# Fortran
gfortran -O3 -march=native -o code_serial_exe code.f90     #Serial code
gfortran -O3 -march=native -fopenmp -o code_exe code.f90   #OpenMP aware code

# C
gcc -O3 -march=native -o code_serial_exe code.c -lm        #Serial code
gcc -O3 -march=native -fopenmp -o code_exe code.c -lm      #OpenMP aware code
```

1.1.2 Intel Compilers

```
ml purge
module load intel-compilers/2022.1.0

# Fortran
ifort -O3 -march=core-avx2  -o code_serial_exe  code.f90     #Serial code
ifort -qopenmp -O3 -march=core-avx2 -o code_exe  code.f90   #OpenMP aware code

# C
icc -O3 -march=core-avx2 -o code_serial_exe  code.c -lm        #Serial code
icc -qopenmp -O3 -march=core-avx2 -o code_exe  code.c -lm      #OpenMP aware code
```

Notice that we can compile the code with optimisation flags such as `-O3 -march=native` in case of a GCC compiler. On COSMOS the standard `-xHost` of the Intel compiler will not result in efficient code, since COSMOS features AMD processors. In this case it is typically advantageous to explicitly specify the instruction set of the deployed processor. In case of COSMOS this is accomplished by using the option `-march=core-avx2`. We can set the number of threads to execute our code with an environment variable for the OpenMP aware code:

```
export OMP_NUM_THREADS=4
./code_exe
```

On HPC systems running the previous line on the login node will cause interference with others' workflow. Thus, one should use the batch queue to run jobs. A typical batch job looks like this:

```
#!/bin/bash
#SBATCH -A lu2023-7-61                    # you have one for this course
#SBATCH -t 00:03:00                       # time the code will take
#SBATCH -c 2                              # number of cpus requested
#SBATCH -J data_process                   # name of job
#SBATCH -o process_omp_%j.out             # output file
#SBATCH -e process_omp_%j.err             # error messages
#SBATCH --reservation=lu2023-7-61_dayX  # reservation change X for the day

cat $0                                    # it will log this script

ml purge  > /dev/null 2>&1
module load GCC/11.3.0                        # choosing GNU compilers
#module load intel-compilers/2022.1.0         # choosing Intel compilers

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK # setting nr. threads

./code_exe
```

In the present exercise, located in the folder:

```
Templates/Day_1/Running_OpenMP/SampleScript
```

you can find the `code.f90` and `code.c` files that you can

1. compile with the compiler of your preference in serial and parallel (OpenMP) mode.
2. submit the serial and parallel executable codes to the SLURM queue by using the `job.sh` file which is also provided in the folder (modify it to set the project ID, the number of cores you want, the time, and the reservation queue).
3. submit the job to the queue with the command: `sbatch job.sh`, this will output a number which is the "job ID"

1.2 Exercise: Serial and threaded "hello world" code

In this exercise we write a parallel version of the World's 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.

1.3 Exercise: Calculating Pi

It can be shown that:

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

1.3.1 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.

1.3.2 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, 12 and 24 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 time?

2 Second Day

2.1 Exercise: Summation over triangular area

Using polar coordinates it can be shown

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

For sufficiently large N and sufficiently small h the double integral on the left hand side can be approximated as

$$8\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1}h^2\exp\left(-h^2(i^2+j^2)\right)\approx\pi \tag{3}$$

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

2.1.1 Serial code

We provide serial code in C and Fortran that implement the above sum. Using $N = 44800$ and $h = 0.0002$ gives an estimate for π which is 4 digit accurate. Before proceeding, make sure you keep a copy of the serial code.

2.1.2 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, 16 and 28 threads.

2.1.3 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 28 threads, investigate the performance for a static schedule for a chunk size of 1, 10, 100, 800, 1600. Can you achieve better results using dynamic scheduling?

2.2 Exercise: Reduction

2.2.1 Revised "Calculating Pi" code

Copy your parallel code from exercises 1.3. Upgrade the copies to use loop construct and a reduction operation. Make sure your code is neat and tidy. The code should compile serially and parallel.

2.2.2 Working for any number of iterations and any number of threads

Show that the code in exercise 2.2.1 give correct answers, within rounding, even even if the iteration count and the number of threads do not divide. You will not get this for the version in exercises 1.3. This is easiest when you use a moderate iteration count and compare for different numbers of threads.

2.2.3 Revised triangular sumation code

Copy your parallel code from the exercise 2.1. Upgrade the copies to use loop construct and a reduction operation. Make sure your code is neat and tidy. The code should compile serially and parallel.

2.3 Exercise: Nowait and Orphaning

As discussed in the lectures, orphaning and nowait clauses can be deployed to boost the performance of OpenMP codes. While many of the exercise of this course can be reasonably coded without the use of functions and subroutines, this is not an option for proper scientific applications. The aim of this exercise is to study how the efficiency of the parallelisation of a modularised code can be improved.

2.3.1 Modular serial code

For this exercise we need a simple code which uses two modules:

1. A subroutine (Fortran) or function of type `void` (C) which initialises a previously allocated vector v_i
$$v_i = i \tag{4}$$
2. A function which returns a `double precision` (Fortran) or `double` (C) variable. The function should implement the Euclidian norm
$$||v|| = \sqrt{\sum_i v_i v_i} \tag{5}$$

Use these modules to implement a program, which allocates/mallocs a long vector, initialises it and calculates the norm of the vector. The final result should be printed. The program is parallelised in three different ways and the performance to be compared.

2.3.2 Parallelisation inside the modules

A very simple way to implement a parallelisation is to start a `parallel do/for` inside each of the two modules and using a reduction for the norm. Implement and test it gives the same result as the serial code. Insert timers into your code and check that the code gets faster if you use more threads. You might need to experiment with the vector size, about 40000 seems a good starting point when using the Intel compiler with reasonably aggressive optimisation.

2.3.3 Orphan directives

The previous parallelisation is very easy to implement but has the downside of opening and closing two parallel sections. By using orphan directives we can keep a modular code while having only one parallel section for the entire code.

Start your parallel section before the call to the initialisation routine and close it after the function calculating the norm. Inside the two modules you place a `do/for` directive instead of `parallel do/for`. This should be straight forward for the initialisation routine, but for the norm matters are more complicated since

$$\sqrt{a+b} \neq \sqrt{a} + \sqrt{b} \tag{6}$$

To get the correct result it might be most elegant to change the second module to a function that calculates

$$||v||^2 = \sum_i v_i v_i \tag{7}$$

This way the summation can be easily parallelised. Take care to avoid any data race. The root can be taken in the calling program after the parallel region has been closed.

2.3.4 Nowait

By choosing an appropriate schedule for your loop constructs you should convince yourself that there is no need for a barrier at the end of the two loop constructs (in initialisation and calculation of the norm square). The implied barrier at the end of the parallel region should be enough to ensure the correctness of your code. Use `nowait`-clauses to remove the implied barriers from the loop constructs.

It is interesting to see how the code with the `nowait`-clauses produces the wrong results if e.g. a default dynamic schedule is used. However the code without the `nowait` will still be correct. To see this, you should run the timing section only once per program execution.

Compare the performance of the three parallelisations for different numbers of treads.

3 Third Day

3.1 Exercise: OpenMP tasks

3.1.1 Prepare your serial code

Take your serial code from exercise 2.1 and modify it:

> Define a block size B and transform the loop implementing the sum over i into two loops. The outer on counting the blocks, while the inner the individual iterations. The M denotes the number of blocks, we have $B \cdot M = N$:
>
> $$\lim_{h\to 0}\lim_{N\to\infty}\left[8\sum_{b=0}^{M-1}\left[\sum_{i=b\,B}^{(b+1)B-1}\sum_{j=i+1}^{N-1} h^2 \exp\left(-h^2(i^2+j^2)\right)\right]\right] \tag{8}$$

Confirm you are getting the same results as before.

3.1.2 Parallelisation using tasks and returning the result utilising a shared variable

We now want to parallelise the code using OpenMP tasks. For each b your code should create a task, which evaluates the two inner sums over i and j. When parallelising your code, consider that:

1. Each task is created exactly once.
2. A way to get a result back from the task is a shared variable.
   (a) Updates to shared variables need to be protected against data races (e.g. `atomic`, `critical`)
   (b) The constructs used for this protection consume quite a bit of time, use them only once per task.
   (c) Make sure shared variables do not go out of scope before the last task has finished.
3. Reductions for task constructs are part of OpenMP 5.0 and are discussed in exercise part 3.1.4.

3.1.3 Performance

Check that your code speeds up when using more threads. By changing the block size you can alter the number of tasks created. If you use large block sizes your load balance will become poor.

3.1.4 Using reduction constructs for tasks

Take a copy of your code for exercise 3.1.2 and use reduction constructs such as `task_reduction` and `in_reduction` to obtain your result. When doing so your will most likely need a `task_group` as well.

Which code performs better? The one using a shared variable or the one with the reduction construct?

Remark: OpenMP 4.5 introduced the `taskloop` construct, with which this exercise would be easier to parallelise.

3.2 Exercise: First Touch

3.2.1 Vector multiplication

Write a serial code which:

- Initialises three vectors $a_i = 2i$, $b_i = 3i$ and $c_i = i$
- In a separate loop sets a third vector $c_i = a_i\, b_i$
- In a third loop check for correctness, that is, whether $c_i = 6i^2$ for all i holds. At the end of the loop you should have a logical variable stating whether or not the code passed or didn't pass the test. Demand reasonable accuracy in your test (e.g. a relative uncertainty of about 14 digits for a double precision code seem ok).

  If you are doing this exercise in C, make sure not to use `calloc()`.

3.2.2 Parallel code without *first touch*

Parallelise the loop assigning $c_i = a_i\, b_i$ using the loop construct. You should then place timers around all the loops to measure their performance. To get stable timings, you should place an outer loop around the entire sequence of initialisation, calculation and verification. Run the sequence a number of times and average the results.

Modern multicores processors have very large shared caches (several 10 MB). Make sure your vectors are larger than the cache sizes.

3.2.3 Add *first touch*

Place a parallel loop construct (separate parallel region) around your initialisation loop. Confirm whether or not that has an effect on the performance of the loop you parallelised in part 3.2.2 of this exercise.

3.2.4 Parallel verification

Since we have all these processors, we might also use them to speed up the verification part of the code. Implementing this is easiest, when using a reduction operation for logical variables. Confirm that your test work, by artificially placing a wrong result in a specific location of the vector c.