# Final Programming Project

Parallel Searching using OpenMP and MPI

CSC4005 - High Performance Computing: Principles of Parallel Programming
Semester 1 (2020/21)

Contact: Dr Blesson Varghese
*b.varghese@qub.ac.uk*

**Submission Deadline: Tuesday, 08 December 2020, 18:00**

## Contents

# 1 Overview

The final programming project builds on your previous assignments. Given a collection of $p$ patterns and $t$ texts the aim is to detect and report the positions of certain patterns in certain texts. In some cases you will be required to detect whether the pattern occurs in the text, for others you will be required to detect every occurrence of the pattern in the text. There will be a separate control file to specify which patterns are to be sought in which texts.

You will design and implement two solutions, one using OpenMP (`project_OMP.c`) and one using MPI (`project_MPI.c`). The challenge is to design programs that will exploit parallel programming and high-performance computing techniques you have studied to solve the problem as quickly as possible. You have complete flexibility in the strategy you adopt. For example, you may opt for an embarrassingly parallel solution or a more sophisticated fine-grain solution. The faster your solution *correctly* solves the problem the more marks you will accumulate. (Note: you are not required to count comparisons.) However, your program must be based on the straightforward pattern matching algorithm described in Assignment 1. The pattern must be searched from left to right and each character tested separately (in particular you must not skip characters or use the `memcmp` function). Other techniques, such as hashing or advanced search algorithms may not be used.

You should assume that: (i) you have 4 cores for your OMP program on the same physical node, and (ii) 4 cores for your MPI program on different physical nodes of the cluster. Each core may execute only one thread or process.

# 2 Inputs

You will be provided with an example folder called `small-inputs` in which you will find

- $p$ patterns, `pattern0.txt`, `pattern1.txt`, ...
- $t$ texts, `text0.txt`, `text1.txt`, ... and
- a control file, `control.txt`.

Each line in the control file will contain three numbers and will correspond to a single search.

- The first number will be 0, meaning find whether the pattern occurs, or 1, meaning find every occurrence.
- The second number will indicate which text to use.
- The third number will indicate which pattern to use.

You will also be provided with a file `small-inputs.sorted` which contains the expected output for this input.

You may assume that no text or pattern file will exceed 20MB. However, a pattern file may exceed the size of a text file. You may also assume that $1 \leq p \leq 20$ and $1 \leq t \leq 20$.

# 3 Outputs

Your program must output its results to a single file, `result_OMP.txt` or `result_MPI.txt`, where each line will contain three integers indicating, respectively, the text id, the pattern id and a result from the search. If searching for any occurrence the result should be -2 to indicate that the pattern was found and -1 to indicate that it was not found. If searching for every occurrence the pattern positions should be reported, with -1 reported if the pattern is not found. Text positions should be numbered from 0.

In the example below are 2 texts and 2 patterns. Here `pattern0.txt` is found at positions 445 and 666 in `text0.txt` and is not detected in `text1.txt`; `pattern1.txt` is found at position 3334 in `text0.txt` and at the start of `text1.txt`. Note that the lines in the output file may appear in any order and that the horizontal lines shown in the output here are for clarity only. Use a single space (no tabs) to separate the

numbers on each line.

| Input | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

| Output | | |
|---|---|---|
| 0 | 0 | -2 |
| 0 | 0 | 455 |
| 0 | 0 | 666 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 0 | 1 | -2 |
| 0 | 1 | 3334 |
| 1 | 1 | -2 |
| 1 | 1 | 0 |

# 4 Test Data

Test data is available in the folder called `small_inputs` and the results file, `small_inputs_sorted.txt` in `HPCProject.tar.gz`. These should be used to check the logic, not the performance, of your programs. It is strongly recommended that you devise your own test data to test the performance of your programs. There are also script files `execute_OMP` and `execute_MPI` which must be used to compile and run your programs.

These script files take a single parameter which is the name of a folder containing data. This will be linked to the name `inputs` for your program to read from. Note that any existing folder called `inputs` will be deleted so you should not store data in a file or folder called `inputs`. You may find the Unix commands `sort` and `diff` helpful when checking the output of your programs, for example your batch job might contain:

```
./execute_OMP small_inputs
sort -k 1,1n -k 2,2n -k 3,3n result_OMP.txt > sorted_OMP.txt
diff sorted_OMP.txt small_inputs_sorted.txt
```

# 5 Submission Instructions

You should submit your final project to Canvas as a .zip file named ⟨`student-number`⟩`.zip`. This must only contain the following four items:

- The two programs named `project_OMP.c` and `project_MPI.c`, and
- The corresponding job scripts named `jobscript_OMP.sh` and `jobscript_MPI.sh`.

I will run the submitted programs using the job scripts you provide and the `execute_OMP` and `execute_MPI` scripts that were provided with the test data on the Kelvin cluster in the batch mode using an unseen dataset. You need to assume that the data will be in a local folder named `large-inputs`. Therefore, your job script must take this into account. For example your batch job may contain:

```
./execute_OMP large_inputs
sort -k 1,1n -k 2,2n -k 3,3n result_OMP.txt > sorted_OMP.txt
```

Note: The file names should be as specified. There will be penalties for not adhering to the constraints provided in the brief.

Please ensure that your code is well commented - proper and intuitive naming convention, a preamble for the code, inputs and outputs to functions, description of functions etc. In the code, you must highlight where strategies for optimising performance are used and must justify the strategies used in textual form as comments. There are marks allocated for highlighting and defending the strategies used.

# 6 Overview of Mark Distribution

This final project carries 50% of the module mark.

The following is an overview of the mark distribution.

**OpenMP Source Code and Execution: 25 marks**

- Error free execution and correct outputs on unseen input - 10 marks
- Performance of the program - 6 marks
- Strategies for optimisation - 6 marks
- Neatness of code - 3 marks

**MPI Source Code and Execution: 25 marks**

- Error free execution and correct outputs on unseen input - 10 marks
- Performance of the program - 6 marks
- Strategies for optimisation - 6 marks
- Neatness of code - 3 marks

Note: Please adhere to hard constraints provided in the brief to avoid any penalties. Your program must execute on the unseen inputs to be considered for marks against performance, optimisation and neatness of code.

If I am unable to execute your program on the unseen inputs, then you will be asked to re-submit a working program. In the event of a resubmission that executes properly, the mark for your program will be capped at 10/25.

**All the best!**