**CSC4005 – High Performance Computing: Principles of Parallel Programming**

*Assignment 2: Parallel Searching using MPI*

Adam Coyle

40178464

October – November 2020

**Table of Contents**

[Abstract 3](#_Toc56673450)

[Introduction 3](#_Toc56673451)

[1 - Test 0 4](#_Toc56673452)

[1.1 - Algorithm 4](#_Toc56673453)

[1.2 - Results 4](#_Toc56673454)

[2 - Test 1 5](#_Toc56673455)

[2.1 - Algorithm 5](#_Toc56673456)

[2.2 - Results 5](#_Toc56673457)

[3 - Test 2 8](#_Toc56673458)

[3.1 - Hypothesis 8](#_Toc56673459)

[3.2 - Results 9](#_Toc56673460)

[4 - Test 3 10](#_Toc56673461)

[4.1 - Algorithms 10](#_Toc56673462)

[4.2 - Results 11](#_Toc56673463)

[Conclusion 14](#_Toc56673464)

# Abstract

Parallel computing has become a common practice in recent times with the increase in the number of CPU cores on an available processing unit. Shared memory architectures have demonstrated that using multiple cores can have both a positive and a negative effect depending on the program being parallelised. Distributed memory architectures can also attain performance improvements; however, implementation is not always cut and dry, and is often left to the programmer to explicitly specify a suitable parallelized version of the program. In these sets of experiments, we will be investigating the performance difference (if any) obtained from parallelizing a straightforward pattern searching algorithm using Message Passing Interface (MPI). MPI enables programs to be run across processes on different physical computers. We will be testing two different MPI pattern searches, one using only the barebones MPI initialization, the embarrassingly parallel program. The other program adopts a master-slave model which is considerably more hands on. These programs will be compared to the sequential pattern searching program over a dataset consisting of 8 patterns and a single text. The results show that the embarrassingly parallel model will eventually outperform the sequential search, given enough processes, although it is not nearly as big an improvement as expected. The master-slave model shows real potential, but in its current state performs only slightly better than the embarrassingly parallel program. However, it does have some important optimizations which can be implemented to drastically improve performance.

# Introduction

Sequential computer programs can be parallelised using different methods which utilise shared memory architectures, distributed architectures, or a combination of both. The focus of this report will be on parallelisation of a naïve string searching algorithm (referred to as the sequential search) using the Message Passing Interface (MPI) standard. MPI is a shared memory architecture, where a program will use multiple CPUs (nodes) across a cluster to execute the program. Each node will have their own copy of the program, allowing for different values for variables per node, as well as enabling the engineer to design a specific set of instructions for each node. In these tests, we will be parallelising the sequential search, first using an embarrassingly parallel algorithm and then by using a fine-grained implementation which will utilise a master-slave programming model. The results of these tests will display any observed performance changes for differing process sizes (2,4 and 8). By comparing the results of the parallel programs to the sequential search, we will attempt to draw any conclusions regarding the performance implications of using MPI, and its overall feasibility for this program.

# 1 - Test 0

This test will observe the performance of the sequential search modified in such a way that it searches only one text for several patterns, in sequential order. The results of this test will act as a baseline to be compared against the MPI implementations in subsequent tests.

## 1.1 - Algorithm

Modified Sequential Search

1. readText()
2. for patternNumber to maxPatterns
3. readPattern(patternNumber)
4. getNanos() // time start of search
5. processData()
6. getNanos() // time end of search
7. Print pattern search time
8. Print program time

Taking the sequential searching program that was previously developed, several modifications were made to the program to meet the assignment requirements:

* The program no longer takes in arguments to determine which test to run
* The readData function was refactored into two functions: readText, which takes no arguments, and readPattern which takes in the pattern number (ranging from 1 to 8) as the only argument.
* The runTest function used previously was modified such that the readText function is called only once since there is only 1 text file. The function then searches for each pattern in the text sequentially.

## 1.2 - Results

Upon submitting the job to the kelvin cluster, the program took 117 seconds (3 s.f) to run. Some interesting observations were that odd numbered jobs (1,3,5,7) took around 29 seconds each to run, accounting for approximately 99.99% of the program time. In comparison, the longest search time obtained for even-numbered patterns took just over 4ms (4x10-3). The difference between these sets of patterns is the point in the text at which they occur; even-numbered patterns are found very early (latest at index 3000) in the text, while odd-numbered patterns are found much later (all found in the last 4000 characters).

# 2 - Test 1

Test 1 will involve re-engineering the sequential search used in test 0 into a perfectly parallel program, with each process searching for different patterns. The program will utilise a minimum of 2 processes up to 8 maximum. The results of these tests will be used to evaluate the parallel speedup (PS) and parallel efficiency of this basic MPI implementation.

## 2.1 - Algorithm

Very few changes were needed to morph the sequential search into a perfectly parallel program. Since all processes are expected to execute the same code, the only modifications required were the inclusion of MPI declarations (initialization, getting processes and finalizing) and instructing the processes which tests to perform, depending on their own rank (process ID or procId as it is referred to in code).

**Parallel Algorithm (searching\_MPI\_0)**

1. MPI\_Init(&argc, &argv)
2. MPI\_Comm\_size(&nProc)
3. MPI\_Comm\_rank(&procId)
4. readText()
5. testNumber = procId + 1
6. while readPattern(testNumber) do
7. getNanos() // time start of search
8. processData()
9. getNanos() // time end of search
10. Print pattern search time
11. testNumber += nProc
12. MPI\_Finalize()

Each process will run an identical program, with the only difference between them being the value of their process ID (procId). Using 2 processes, process 0 will take tests 1,3,5,7, and process 1 will take tests 2,4,6,8.

## 2.2 - Results

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Processes | Elapsed Time (s) | Parallel Speedup (PS) | Parallel Efficiency (PE) |
| 2 | 119.034 | 0.981862325 | 0.490931163 |
| 4 | 59.84 | 1.953125 | 0.48828125 |
| 8 | 32.977 | 3.544136823 | 0.443017103 |

Table : Results obtained by the embarrassingly parallel program.

The results showed that the program did experience a significant speedup once it used 4 processes or greater. This does not line up with the typical expectation of parallel programs, that is a rough improvement of Nx speedup for N processes.

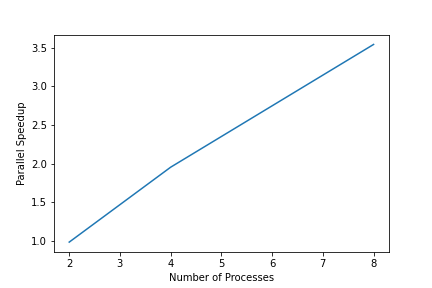


Figure : Parallel speedup of the perfectly parallel program.

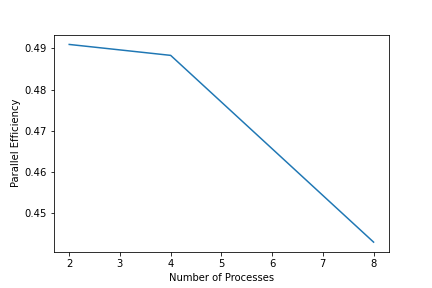


Figure : Parallel efficiency of the perfectly parallel program.

Interestingly when using 2 processes, the program performed slightly worse than the sequential program (2 seconds longer), which can be attributed to the extra overhead required by MPI. Additionally, there are two possible explanations for these disappointing results:

1. Each process must search for a pattern in the entire text. Therefore, the performance of program is limited by the worst test pattern (in this case, it was all the odd-numbered patterns).
2. The workload is not distributed equally between processes; when using only 2 processes, it happened that process 0 searched for odd-numbered patterns in the text. Recall in section 1 how these tests accounted for 99.99% of program time, hence why there was a negligible improvement when using 2 processes. This also occurs when using 4 and 8 processes and is a consequence of the input data.

To confirm these results, we can look at the individual search times for each pattern, and the process responsible for searching:

|  |  |  |
| --- | --- | --- |
| Process | Pattern | Search Time (s) |
| 0 | 1 | 29.635999420 |
| 1 | 2 | 0.001766797 |
| 0 | 3 | 29.651291962 |
| 1 | 4 | 0.004969915 |
| 0 | 5 | 29.643374608 |
| 1 | 6 | 0.003318355 |
| 0 | 7 | 29.528196764 |
| 1 | 8 | 0.000112662 |

Table : Individual pattern search times for the embarrassingly parallel program, using 2 processes.

In section 3, we will explore a potential workaround for these issues to improve the performance of the program.

# 3 - Test 2

For this test, we will be using the program developed in test 1, with the aim of improving performance by alteration of the input data. The results will demonstrate the difference in PS and PE, particularly for lower process counts, which appear to have limited PS relative to the sequential program.

## 3.1 - Hypothesis

The key issue for the program stems from the fact that the workloads are not evenly distributed.

One potential workaround to improve the PS for the program, would be to reorganize the input data such that the workload becomes better distributed between processes. This might improve the results when using 2 and 4 processes, but not for 8 since each process only searches for 1 pattern. The limitations of the first issue will remain unless the program is altered.

To test this hypothesis, the order of the input data was adjusted using a points system to assess imbalances between processes, with -1 used to denote the performance of a worst-case pattern, and +1 used to denote best-case pattern. The reorganisation for using 8 processes is omitted since each process only searches for one pattern.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process - 2 MPI processes | Process - 4 MPI processes | Pattern (original) | Score | Pattern (adjusted) | Score (adjusted) |
| 0 | 0 | 1 | -1 | 2 | +1 |
| 1 | 1 | 2 | +1 | 4 | +1 |
| 0 | 2 | 3 | -1 | 6 | +1 |
| 1 | 3 | 4 | +1 | 8 | +1 |
| 0 | 0 | 5 | -1 | 1 | -1 |
| 1 | 1 | 6 | +1 | 3 | -1 |
| 0 | 2 | 7 | -1 | 5 | -1 |
| 1 | 3 | 8 | +1 | 7 | -1 |

Table : Score distribution before and after adjusting the data

|  |  |  |
| --- | --- | --- |
| Process | Score (original) | Score (adjusted) |
| 0 | -4 | 0 |
| 1 | +4 | 0 |

Table : Total scores for each process (using 2 MPI processes)

|  |  |  |
| --- | --- | --- |
| Process | Score (original) | Score (adjusted) |
| 0 | -2 | 0 |
| 1 | +2 | 0 |
| 2 | -2 | 0 |
| 3 | +2 | 0 |

Table : Total scores for each process (using 4 MPI processes)

The tables above demonstrate there is a potential ordering of input data that improves performance for some processes, but reduces it for others, resulting in all processes having a balanced workload.

The experiment was conducted by duplicating and renaming the patterns in a separate folder, \_\_inputs. This folder was transferred to the kelvin cluster and renamed inputs (the original inputs were temporarily name \_inputs). Alternatively, the program could be modified to take in the name of the folder containing inputs, however this idea was dismissed since the assignment brief explicitly forbids program modification for this test.

## 3.2 - Results

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Processes | Elapsed Time (s) | Parallel Speedup (PS) | Parallel Efficiency (PE) |
| 2 | 61.21 | 1.909410227 | 0.954705114 |
| 4 | 30.373 | 3.847989991 | 0.961997498 |
| 8 | 29.967 | 3.900123469 | 0.487515434 |

Table : Results obtained by the parallel program, using reordered inputs.

The results demonstrate that there is indeed a marked improvement in PS and PE of the program obtained by simple reorganization of the input data. Surprisingly, there was a slight improvement when using even 8 processes, albeit quite small in comparison to the results obtained using 2 and 4 processes. Despite this result, the use of 8 processes confirms that the first issue (performance limited by worst-case search) remains unaddressed by this reorganization.

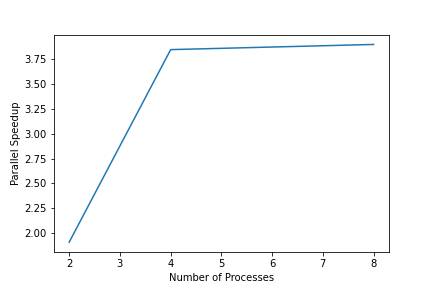


Figure : Parallel speedup of the perfectly parallel program, using reordered input data.

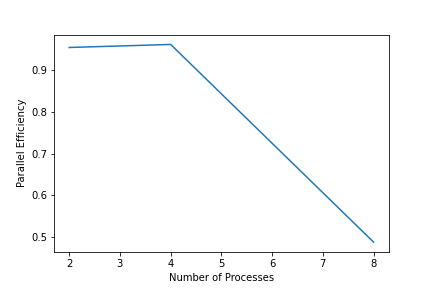


Figure : Parallel efficiency of the perfectly parallel program, using reordered input data.

# 4 - Test 3

In test 3, a fine-grain MPI program will be implemented such that all processes will search for 1 pattern by dividing the text data into (ideally) equal sections. We will use a master-slave model where the master (identified as process 0) will read the text data, divide it between processes; send the pattern data to each process and collect the results of their searches to provide a final, reduced result. The results of this program will be compared to both the sequential program and the perfectly parallel program implemented in test 1 to determine if the extra tuning of the program is worth the performance improvement.

## 4.1 - Algorithms

Utilization of the master-slave model requires that two distinct algorithms be implemented: one for the master and one for all other processes. The master algorithm handles division of text into segments for each process, it then sends the data to the slave processes and gets the final result by selecting the maximum of all results.

**Master Loop Algorithm (searching\_MPI\_1)**

1. Read text using text data arguments passed into function
2. Divide the workload for each process
3. Calculate text displacement for processes
4. Distribute workloads using MPI\_Scatter()
5. Distribute displacements using MPI\_Scatter()
6. Distribute text data of length workload starting from displacement, using MPI\_Scatterv()
7. Initialise finished flag to 0
8. Declare finalResult to store reduced result
9. While patterns are being read, do
10. Broadcast the finished flag (set to 0) to all slaves
11. Broadcast the pattern length to all slaves
12. Broadcast the pattern data of size pattern length to all slaves
13. Master process searches allocated text data for pattern
14. Get final result using MPI\_Reduce
15. Print pattern search time after getting all results back
16. Increment patternNumber
17. Broadcast the finished flag (set to 1) to all slaves
18. Free allocated arrays

**Slave Loop Algorithm (searching\_MPI\_1)**

1. Receive workload from master
2. Receive displacement from master
3. Receive text data from master of length workload
4. Initialize finished flag
5. While not finished, do
6. Receive broadcasted finished flag from master
7. If finished is set to 1 do
8. Break out of while loop
9. Receive broadcasted pattern length from master
10. Allocate memory to pattern data of length pattern
11. Receive broadcasted pattern data of length pattern length from master
12. Process searches received text data for pattern
13. Process reduces result of search to the master process
14. Free allocated arrays

Additional consideration was given to remove excessive for loops in the algorithms. For sending the text data, both the workload and displacement for each process needed to be calculated due to the fact that the workload distribution function takes into account the possibility of imbalanced workloads (where text length modulo processes is greater than 0). This might have been excessive given that the text provided was easily divisible using the process constraints but provides more robust functionality for reuse in the future. The distribution of remainders ensures that only slave processes receive the extra work, since the master is already responsible for calculating and sending data, as well as reducing the final result.

One feature that should have been implemented is allowing neighbouring processes to communicate with each other. This arises the possibility that a pattern might exist between two processes (the pattern begins in one process and finishes in the neighbour). However, the difficulty with this consideration is that the provided patterns do not meet this requirement, so it would be difficult to test without creating a new set of inputs. Regardless, it was considered for implementation and will be in future.

An alternative method of sending/receiving pattern data would have been to iterate through each process and send the data, calling receiving functions in the slave loops. The use of the broadcast functions encapsulates this design and is a suitable alternative since all slave processes are receiving the exact same pattern data.

The finished flag communicated between master and slaves serves to prevent the slave processes from attempting to receive pattern data after there are no patterns left. As such, it prevents the program from hanging since the value of the flag is checked by the slaves before receiving text data.

## 4.2 - Results

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Processes | Elapsed Time (s) | Parallel Speedup (PS) | Parallel Efficiency (PE) |
| 2 | 108.678 | 1.075424649 | 0.537712324 |
| 4 | 56.923 | 2.053212234 | 0.513303059 |
| 8 | 30.495 | 3.832595507 | 0.479074438 |

Table : Results obtained using the master-slave model for 2,4,8 processes.

The results obtained from by the program are like those results obtained by the test performed in section 2, by the embarrassingly parallel program. In terms of code structure, the fine-grain program developed for this test is not dissimilar from the original sequential program; it searches for each pattern sequentially and outputs the result.

Despite their similarities in performance, there are major differences in the individual search times for the programs. The embarrassingly parallel program showed that it was limited by the worst pattern search, since each process performed a full search of the text. The fine-grain program attained consistent search times for all patterns:

|  |  |  |
| --- | --- | --- |
| Process | Pattern | Search Time (s) |
| 0 | 1 | 14.028267779 |
| 1 | 2 | 12.966057154 |
| 0 | 3 | 13.975920999 |
| 1 | 4 | 12.961007101 |
| 0 | 5 | 14.025638322 |
| 1 | 6 | 12.982319464 |
| 0 | 7 | 14.057806434 |
| 1 | 8 | 12.972661323 |

Table : Individual pattern search times for the fine-grain program, using 2 processes.

The results demonstrate that the fine-grain program now takes considerably longer to search for patterns where both the sequential and embarrassingly parallel programs took a fraction of a second. In comparison, the fine-grain program is not limited by the worst pattern, unlike the embarrassingly parallel program. The issue is that there is no communication between processes regarding the existence of the pattern, and therefore each process will search regardless of whether the pattern has already been found. In the previous programs, the algorithm returns upon finding the pattern, and does not waste time searching afterwards.

One method of optimization that will be investigated in the future, is enabling processes to have awareness of the search progress of other processes. In this way, when a process finds the pattern early on (as in the case of some of the provided patterns), it will notify other processes of this result such that they can stop their search immediately.

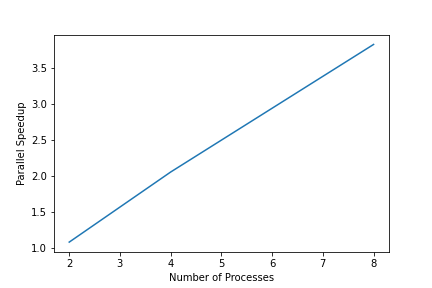


Figure : Parallel speedup of the fine-grain program using a master-slave model.

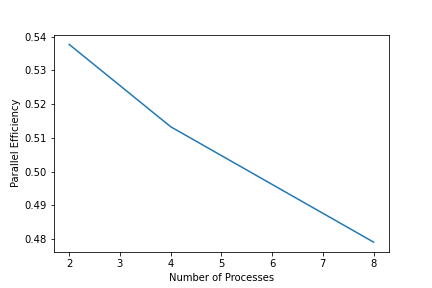


Figure : Parallel efficiency of the fine-grain program using a master-slave model.

# Conclusion

The sequential search has been shown to perform very well in the best-case, attaining sub-second search times for patterns found early in a text, while it performs poorly for worst-case searches. Since the provided inputs contained a balance of both, the sequential programs performance struck a balance between the two cases. The embarrassingly parallel program by comparison, needed to either use double the expected number of processes or to reorganize the inputs to attain an acceptable parallel speedup, and was still hindered by the worst-case pattern. The fine-grain model achieved only a slight improvement over the embarrassingly parallel program, performing twice as good in the worst-case searches while performing worse by several orders of magnitude for the best-case searches. The advantage of the fine-grain program over the embarrassingly parallel model is that it does not face the same limitations, since it performed worst-case searches faster than previous programs, and is open for future optimizations that will ideally attain near perfect parallel speedups.

To conclude, use of MPI to parallelise programs yields performance entirely dependent on the programmer; minimal effort as in the case of the embarrassingly parallel program yielded less-than-optimal results, while the effort put into the fine-grain program provides a good baseline algorithm for attaining better performance only with additional effort put into it.