University of Queensland

COSC3500 Assignment - Testing Report

Longest Common Subsequence

Author:
Nicholas Lambourne

Supervisors:
Dr. Joel Fenwick
Dr. Michael Bromley
Assoc. Prof. Marcus Gallagher

November 9, 2019

Contents

Chapter 1

Problem Description & Serial Implementation

1.1 Problem Description

Finding the longest common sub-sequence (LCS) between a set of two or more sequences (or strings) is a longstanding problem in computer science with many practical implications. It finds use, along with other sub-sequence problems, in applications where the differences between sequences are important, including linguistics, genomics and version-control software [1].

In order to present the problem unambiguously, it is important to define several core terms:

- Sequence: a contiguous series of symbols or characters. For example "$hello, world$" would be considered a single sequence.
- Sub-sequence: a collection of (not necessarily contiguous) characters from an existing sequence. The characters in the sub-sequence must occur in the same order as in the sequence it is derived from. For example "$howd$" is a sub-sequence of "$hello, world$", but "$hwod$" is not.

These distinctions are particularly vital in differentiating between the LCS problem and the similarly named, though very different Longest Common Sub-String problem, which requires the elements of the sub-sequence to occur contiguously in the original sequence.

The longest common sub-sequence is not limited to being performed between two sequences, and can easily be generalised to any number of strings, however this analysis focuses solely on LCS comparisons between pairs of sequences.

1.2 Implementation

A wide variety of solutions to the LCS problem exist and various heuristics have been developed but the research literature has centralised around what is referred to as the "traditional" or dynamic programming (DP) approach.

This approach involves the construction of an $(m+1)\cdot(n+1)$ matrix of integers initialised at zero, before progressively stepping through the matrix and populating it based on the indices of table mapping to the indices of the two sequences.

An example of the matrix for the sequences "$SIDUBCHISG$" and "$BEMDKGCQIK$" is demonstrated below. The matrix (in red) has been augmented with indices and the characters of each string:

			B	E	M	D	K	G	C	Q	I	K
		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
S	1	0	0	0	0	0	0	0	0	0	0	0
I	2	0	0	0	0	0	0	0	0	0	1	1
D	3	0	0	0	0	1	1	1	1	1	1	1
U	4	0	0	0	0	1	1	1	1	1	1	1
B	5	0	1	1	1	1	1	1	1	1	1	1
C	6	0	1	1	1	1	1	1	2	2	2	2
H	7	0	1	1	1	1	1	1	2	2	2	2
I	8	0	1	1	1	1	1	1	2	2	3	3
S	9	0	1	1	1	1	1	1	2	2	3	3
G	10	0	1	1	1	1	1	2	2	2	3	3

Figure 1.1: The LCS Table

The table is populated using the following DP-based algorithm (the comments on each node explain what is to be done at each cell):

Listing 1.1: LCS Algoritm

```
// Assumes the existence of an int table[len_x + 1][len_y + 1]
for (int i = 0; i <= len_x; i++) {
    for (int j = 0; j <= len_y; j++) {
        if (i == 0 || j == 0) {
            // All cells in both the first row and the first column contain zero.
            table[i][j] = 0;
        } else if (x[i - 1] == y[j - 1]) {
            // If symbols match, current cell = adjacent cell (up and left) + 1
            table[i][j] = table[i - 1][j - 1] + 1;
        } else {
            // Otherwise, current cell = max of adjacent left and up cells
            table[i][j] = max(table[i - 1][j], table[i][j - 1]); // otherwise
        }
    }
}
```

Once the table has been populated, the actual longest common subsequence can be ascertained by "backtracking" through the matrix from the bottom right cell (which contains the length of the LCS) following the same rules that it was constructed by (the comments on each node explain what is to be done at each cell):

Listing 1.2: Reconstruction Algorithm

```
// Assumes the existence of an int table[len_x + 1][len_y + 1] that has been prepopulated
int i = x.length();
int j = y.length();
while (i != 0 && j != 0) {
    if (x[i] == y[j]) {  // Characters match so add to LCS
        lcs.append(x, i, 1);
        // Now try and move diagonally, within space bounds
        i = max(i - 1, 0);
        j = max(j - 1, 0);
    } else if (i == 0) {  // Cannot go any further left, so go up
        j--;
    } else if (j == 0) {  // Cannot go any further up, so go left
        i--;
    } else if (table[i][j-1] > table[i-1][j]) {  // Left is larger so go left
        j--;
    } else { // Up is larger or they are the same, so go up
        i--;
    }
```

```
19 }
20 reverse_string(lcs);  // String is output in reverse order by algorithm, so reverse again
```

1.3 Verification

The program's efficacy was evaluated on two fronts: correctness, whether the program produced the correct output for the given sequence input, and; performance, how quickly the program was able to produce that output.

1.3.1 Correctness

The program's correctness was assessed using a suite of tests. Below, you'll find tests grouped by category or justification. Test cases were verified by hand.

Input String X	Input String Y	Expected Output	Justification
∅ (empty string)	∅	∅	Edge case: empty vs empty
∅	A	∅	Edge case: string vs empty
∅	A	∅	Edge case: empty vs string
A	A	A	Edge case: single character matching
A	B	∅	Edge case: single character not matching
AAAAAAA	AAAAAAA	AAAAAAA	Typical case: short, equivalent strings
AAAAAAA	BBBBBBB	∅	Typical case: short, no common sub-sequence
AAAAAAA	BBAAABB	AAA	Typical case: short, common sub-sequence
AAAAAAA	∅	∅	Edge case: short, short vs empty
(see medium-0.txt)	(see medium-0.txt)	dVTs	Typical case: medium (length: 25) vs medium, Edge Case: special characters
(see medium-1.txt)	ss	ss	Typical case: medium (length: 25) vs short,
(see long-0.txt)	(see long-0.txt)	ss	Typical case: long (length: 100) vs long
(see super-long.txt)	(see super-long.txt)	(see super-long.out)	Typical case: extra long (length: 10000) vs extra long

1.3.2 Performance

Performance was appraised both theoretically using asymptotic analysis and practically by running a series of automated tests.

Asymptotic Analysis

Time Complexity

It has been established that without narrowing the problem domain (in particular through reliance on heuristics), the time complexity of the traditional (dynamic programming) approach to LCS algorithms is $O(n^2)$ [2].

Space Complexity

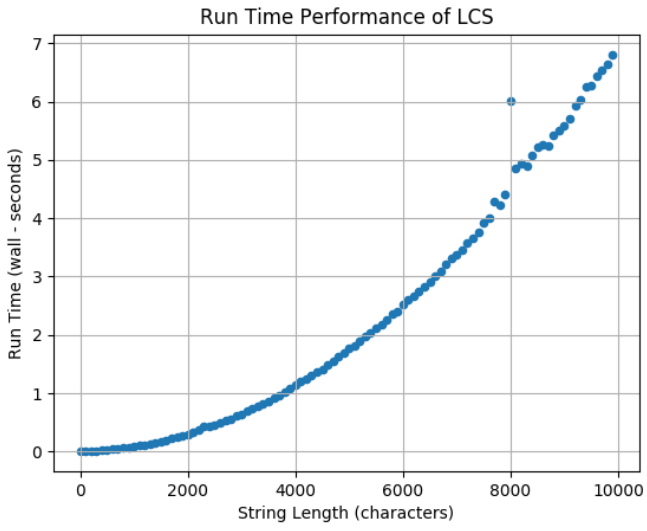
The traditional LCS algorithm utilised here requires the following non-constant space:

- $m + m$ space for storage of the two sequences of lengths n and m.
- $(m+1) \cdot (n+1)$ space to for the two-dimensional matrix used to store the values used as part of the dynamic programming approach.

Thus, due to the dominant nature of the $(m + 1) \cdot (n + 1)$ term, the total space complexity for the algorithm is $O(m \cdot n)$ (the constant factors being dominated by the quadratic component).

Testing

Performance testing was conducted by comparing two randomly generated strings of the same length (ensuring consistency). A Python script was created that generated random strings of lengths up to a given maximum and populated input files with them. The script then ran the C++ binary *lcs* a specified number of times. The figure below shows the results of 1000 tests run over strings varying in length from one to ten thousand characters. It is clear the "wall" time of the program subscribes roughly to the quadratic performance ascertained by the above asymptotic analysis and the research findings of Bergroth et al. for this type of approach [1].



The Python script can be run using the instructions in section B of the appendix.

Chapter 2

Parallelisation of the Problem

Given the typically $O(n^2)$ behaviour of the longest common sub-sequence problem, one way of accelerating the computation time is to break the problem up into distinct subsections and approach it in parallel. The following sections describe the approach taken to paralleise the problem using the OpenMP and Open MPI C++ libraries.

2.1 Parallelisation Strategy

2.1.1 Multi-Threading with OpenMP

Due to the fact that multiple threads can be run in a single process and that OpenMP utilises threads for its multi-processing, I decided early on that in order to maximise the performance of the program that the OpenMP optimisations would take place inside each MPI process.

Utilising OpenMP effectively proved challenging and required a complete overhaul of the serial code for populating the table described in Section 1.2 and visualised in Figure 1.1. Instead of populating the table iteratively by row and column, cells were populated on the anti-diagonal (see Figure 2.1.1). Because each cell only relies on the cells to its left and top, this allows every cell the entire anti-diagonal to be calculated simultaneously. In theory, if the same number of OpenMP threads were allocated to length of the diagonal, each cell in that diagonal could be calculated independently (and in parallel.

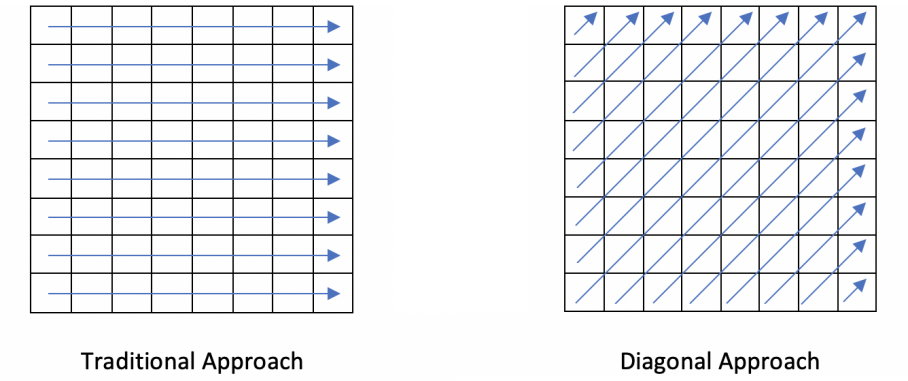


Figure 2.1: Population Techniques

This required the pre-computation of an order of execution more complicated than iterating over the length and height of the matrix. A C++ `vector<vector<int>>` object was used to store the co-ordinates of each cell making up each "diagonal." These diagonals were then processed iteratively, with each constituent cell being processed in parallel using an OpenMPI `parallel for` construct (see Listing 2.1).

Listing 2.1: OpenMP Utilisation

```
// The variable indices is vector<vector<int>> containing a list of diagonals
// which themselves contain (x, y) co-ordindate pairs.
for (int i = 0; i < (int) indices.size(); i++) {
    vector<vector<int>> diagonal = indices[i];
    #pragma omp parallel for shared(table, diagonal, a, b, top, left)
    for (int j = 0; j < (int) diagonal.size(); j++) {
        vector<int> cell = diagonal[j];
        int x = cell[0];
        int y = cell[1];
        int result = calculate_cell(table, x, y, a, b, top, left);
        table[y][x] = result;
    }
}
```

2.1.2 Multi-Processing with OpenMPI

Transforming the program so that it could make use of multiple processes was incredibly challenging. Splitting up the problem space into neat sub-sections was non-trivial, when accounting for the potential of irregularities around the "edges" of the parent matrix (i.e. "non-full" sections). The order in which the segments are processed is practically identical to that of the OpenMP calculations for each cell, but instead of individual cells being computed in parallel by threads, whole sections are computed in parallel by MPI processes (each internally using OpenMP for cell-level calculations).

The number of processes provided to the program is fetched dynamically and determines the number of segments. One process is reserved for the parent or "master," with remainder being used as child, or "worker" processes. The number of segments in the anti-diagonal is matched to the number of processes so that at peak parallelism (when the most cells are able to be processed in parallel), the program can take full advantage.

In figure 2.2 below, the program has been allocated five processes, four of which have been made worker processes. This has resulted in a diagonal size of four, and a total sub-section count of sixteen. Each set of sub-sections of the same colour represent "diagonals." Each sub-section in a diagonal can have its constituent cell values calculated in parallel with those of the other sub-sections in that diagonal. The white arrow indicates the direction that diagonals are processed. The red and violet sections (the first and last diagonals) are processed by the master process due to either being necessary for the first worker processes or requiring the output of all previous cells, respectively.

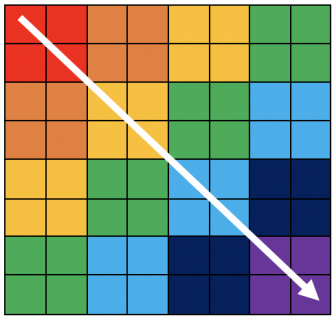


Figure 2.2: MPI Segmentation Technique

2.1.3 Information Sharing

OpenMP

Passing information to OpenMP was trivial due to the fact that each thread wrote to a discrete set of cells and read from cells that had already been populated (and would remain unchanged) in the calculation of the previous diagonal. In light of this, the master table was simply passed to each thread

MPI

Passing information was significantly more difficult between processes and required the use of the MPI message passing interface. For the computation of each sub-section additional data is required from the adjoining cells to the top and left. Figure 2.3 shows a single sub-segment (in orange) and the "top" (in blue) and "left" (in red) required to compute it. Each child, on inception, blocks and waits (using `MPI_Recv()`) for three batches of information: the co-ordinates of the corners of the section being computed, the "top" data (converted into a one-dimentional `int` array), and the "left" data (also converted), which are sent (using `MPI_Send()`) by the master process. After computation, the contents of the subsection (in one-dimenional format) are sent back to the master process and are used to populate the master table.

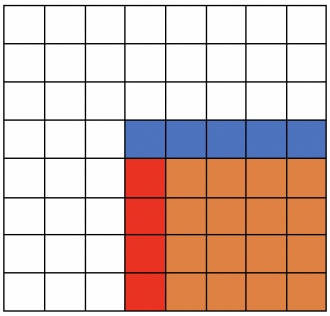


Figure 2.3: Top (Blue) and Left (Red) Sections

2.1.4 Challenges

There were numerous false starts when devising a hybrid solution to the LCS problem. One particular example was a method devised by [3]. However, upon implementation it became apparent that the algorithm provided had several errors. It proved easier implementing, from scratch, the logic for handling both the OpenMP and MPI portions of the program.

Resource access was also a point of difficulty, particularly when trying to run tests with more than one or two processes (let alone multiple nodes) in the week leading up to the submission of milestone two.

Edge cases surrounding the behaviour of the program when assigned less than three processes were also encountered, overcome by reverting the application to utilising only OpenMP in these cases.

It also proved infeasible to parallelise the reconstruction of the longest common subsequence from the table (back-tracking), as each step relied on the state acheived by the previous one. Many possibilities were considered but the nature of the problem procluded any obvious speedups from either OpenMP or MPI.

2.2 Verification

Due to the increased complexity of the code when deconstructing the problem to be performed over multiple threads and multiple processes it was essential to verify that the correctness of the program was maintained. The test suite from Section 1.3.1 was rerun under each of the following conditions:

- With a single MPI process.
- With three MPI processes (the lowest size that will trigger MPI multi-processing).
- With four MPI processes (typical usage).
- With sixteen MPI processes (high usage).

Each of these tests was run with an allocation of a single OpenMP thread (effectively serial), then again with with multiple threads.

2.3 Scalability Test Plan

2.3.1 Strong Scaling

Strong scaling measures the increase in performance for a fixed problem size. To test this form of scaling, two randomly generated strings of 100,000 each will be generated and run with the configurations in Table 2.1. These vary over two primary parameters: number of processes and number of threads. Due to the nature of the test clusters, when the number of processes exceeds the available number of CPU cores on a node, multiple nodes are used and the processes are split up evenly over multiple nodes.

Nodes	**Number of Tasks (processes)**	**Number of Tasks Per Node**	**CPUs Per Tasks (threads)**
1	1	1	1
1	2	2	2
1	4	4	4
1	8	8	8
1	16	16	8
1	16	16	16
1	16	16	32
2	32	16	8
2	32	16	16
2	32	16	32
4	64	16	32

Table 2.1: Strong Scaling Plan

These runs will be run on UQ's `goliath` cluster using `slurm` (`sbatch`) and a configuration shell script. An example script has been included in Appendix 3. Each run will be timed and compared to the serial solution and one another. The comparison with the serial solution will allow for the determination of how effective the solution is with respect to strong scaling or "speed up."

2.3.2 Weak Scaling

Weak scaling measures the effect of a program's parallelism as the size of the input changes. To test this type of scaling, I will run the same tests used for each of the following randomly generated string lengths:

Input Length
1
10
100
1000
10000
100000

Table 2.2: Weak Scaling

In order to offset any abnormalities caused by particular (randomly generated) input strings, each test will be run multiple times and averaged out. The `testing.py` script used for the first milestone will be modified to support this behaviour.

2.3.3 Test Plan Timeline

In order to avoid the trouble I faced with respect to node availability, I expect to front-load a lot of the work for the final milestone. The table below outlines my timeline:

Week	**Activity**	**Rationale**
Midsemester Break	Development of testing.py script for automating testing	Automating the testing will save the manual entry of parameters
Week 11	Strong Scaling	Strong scaling is the simple case and can be tackled first
Week 12	Weak Scaling	This should be just the addition of extra parameters (and iteration) to the strong scaling
Week 13	Buffer Time	I have left this space free in case testing takes longer than expected, but would prefer to avoid it as the clusters will be busy

Table 2.3: Timeline

Chapter 3

Testing Parallelisation Efficiency

N.B: My code has changed significantly since Milestone 2, it has been committed to the MS3 directory of the provided SVN repository.

3.1 Test Framework

To offset the tedious work of running the numerous tests outlined by the initial testing plan, the Python script developed for the basic testing carried out in Chapter 1 was extended to automate the process for both strong and weak testing. As such, the procedure for reproducing these results consists of only eleven commands, these can be found in Appendix D. This was accomplished by programmatically populating a `slurm` template (found in Appendix E) and submitting tasks using `sbatch`. The script was also used to collate and summarise the output of the tests, producing both the results charts found below, and CSV reports of the findings. Single jobs can also be run by populating the template and running `sbatch template.sh`.

3.2 Deviations from Initial Test Plan

Due to unforeseen limitations imposed by the available hardware on the `goliath` and `getafix` clusters where the tests were performed, both strong and weak scaling plans needed to be revised. Specifically, the fact that each node of the coursework partition contains only 64 cores proved a limiting factor. This was expected, but what was not expected was that the allocation of each thread to OpenMP would require the exclusive utilisation of a whole core.

3.2.1 Strong Scaling

The strong scaling test plan was the most severely affected by the limitations described above. Previous plans to have configurations containing up to 32 threads were scrapped, and more configuration options were added due to the relative ease of assigning them programmaticaly. This primarily consisted of ensuring that each task/node configuration was tested using all compatible thread configurations. The final set of configurations can be found in Appendix F. All tests were conducted on a problem size of 10,000 characters.

3.2.2 Weak Scaling

Due to a fundamental misunderstanding about how tests for weak scaling were to be performed, the test input sizes needed to be seriously curtailed as tests of 10,000 or 100,000 characters (per processing unit) would not be feasible due to the time limits imposed by the course. Test results for an input length of 1 character were discarded as they consisted entirely of noise. The final input length options are included in the table below. Each test of each configuration was conducted twenty times for each input size (problem size/processing unit).

Input Size (per processing unit)
10
100
1000

Table 3.1: Actual Weak Scaling Input Magnitudes

3.3 Test Results

3.3.1 Strong Scaling

Strong scaling tests were conducted on both the `goliath` and `getafix` clusters . A summary of mean results across the 20 samples collected for each configuration is included in Appendix G. Figure 3.1 below shows the aggregated performance of the program on a fixed problem size of 10,000 characters, with varying combinations of MPI processes and OpenMP threads.

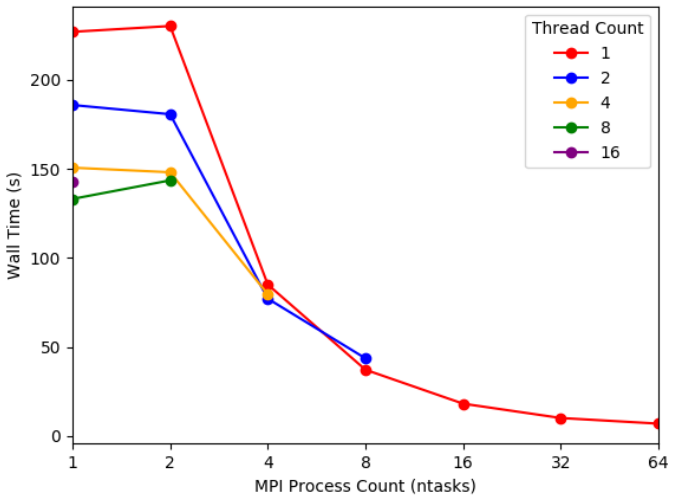


Figure 3.1: Strong Scaling Results

Multi-threading Performance

As can be seen in Figure 3.1, the addition of OpenMP threads has a distinct impact on overall run-time, particularly for configurations consisting of few MPI processes. However, the effect of additional threads has a diminishing rate of return, and can even be detrimental. This can be seen in the one-task, 16-thread configuration that has

demonstrably worse performance than the one-task, eight-thread configuration even when averaged over 20 samples. The positive impact also (at ≥ 8 threads) diminishes quite rapidly; for example, the first additional thread reduces compute time by 18%, but it takes an additional two processes to achieve an additional performance impact of the same magnitude. More threads may have a significantly more beneficial impact on larger problem sizes, where the diagonal length of sub-sections far outstrips the number of threads available.

Multi-processing Performance

The addition of processes also had a clearly positive effect, far more impactful than that of threads. There is one notable exception, at the configuration involving two-task, one thread, which performed worse than with one-task. This makes sense in the context of the segmentation heuristic responsible for allocating work to child processes. Due to the complex nature of splitting the problem, configurations with two processes will short-circuit to using a single processor. Given the overhead of setting up a process and not using it, the difference in performance at $ntasks = 2$ is to be expected. This problem is discussed in greater detail in section 3.5.3, below.

3.3.2 Weak Scaling

A summary of mean results across the 20 samples collected for each configuration of weak testing is included in Appendix H. Figure 3.2 below plots the weak scaling results for the three different problem sizes (per unit).

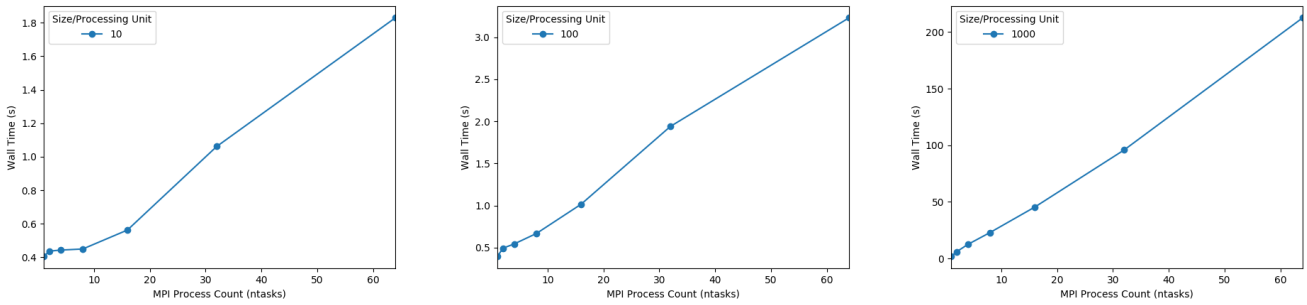


Figure 3.2: Weak Scaling Results

All three problem magnitudes appear to be growing extremely linearly with increased processing units and respective problem size, excepting a small amount of noise in the $n = 10$ test results. This is deceptive, as we shall see in the efficiency analysis, because the absolute extent of the gradient is hidden by the axes' limits being set relative to the data points.

3.4 Scaling Efficiency

3.4.1 Strong Scaling Efficiency

Despite the strong apparent showing of the results in the previous section, examining the results in terms of "speed-up" relative to a serial solution, showed several surprising results. The following formula was used to calculate the actual speedup:

$$\frac{runtime_{serial}}{runtime_{parallel}}$$

This was compared to the theoretical maximum speedup as proposed by Amdahl [4], calculated with the following formula:

$$\frac{1}{\frac{f(N)}{p} + (1 - f(N))}$$

Where N is the problem size, p is the number of processing units and $f(N)$ is the proportion of the program's run-time that can be parallelised. Both threads and processes, given their requirement for a single core per unit, were treated as coequal processing units in the graph below. Due to the overwhelming proportion of time in the parallelisable section, this figure was set at 0.95 (i.e. 95% of time was spent in parallelisable sections).

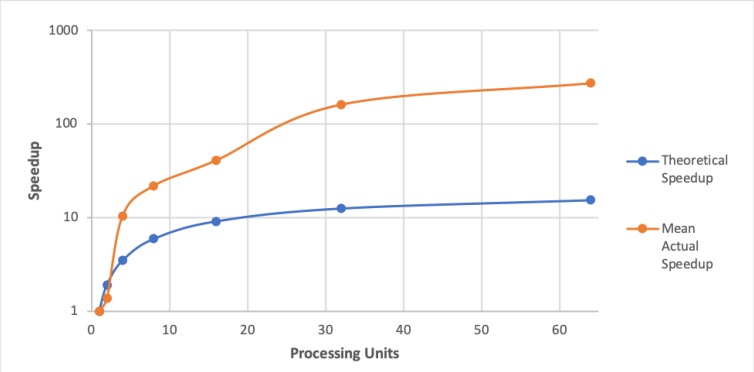


Figure 3.3: Strong Scaling Efficiency

The first thing that stands out is that the actual speedup almost immediately overshoots the theoretical maximum by a non-trivial amount (note the logarithmic scale of the vertical axis). While this should not theoretically be possible (and had to be verified to ensure it was not a mistake) there are numerous possible explanations for this, the most likely being that the serial solution has been made artificially slow by the modification of the code to allow for the possibility of parallelisation. Other possible reasons could include environmental factors, though this is less likely due to the fact that twenty iterations were conducted and the mean taken.

3.4.2 Weak Scaling Efficiency

Weak scaling efficiency was calculated using the same formula as strong scaling, with a correction for the increasing problem size:

$$\frac{runtime_{pserial}}{runtime_{parallel}/processing\,units}$$

The theoretical maximum was calculated in accordance with the rule proposed by Gustafson [5]:

$$p + (1 - p) \cdot T_{serial}$$

Where T_{serial} is the proportion of time spent on serial (or non-parallelisable) sections of code.

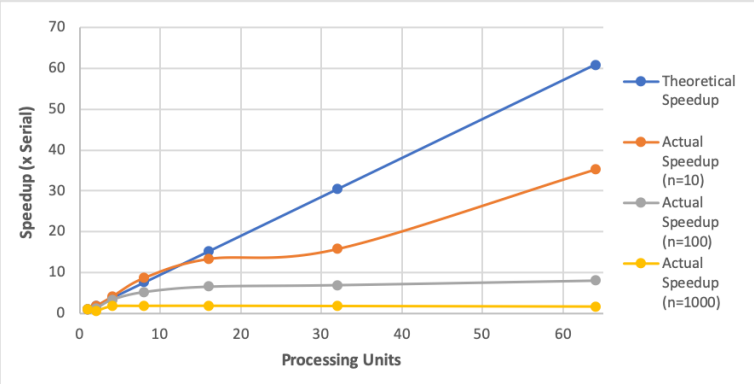


Figure 3.4: Weak Scaling Efficiency

These figures are far more inline with expected behaviour, though below eight processing units the speedup still exceeded the theoretical maximum. Again, this could be due to noise given the small problem size, but the excessive sensitivity to the serial time is more likely to blame.

The behaviour of the tests at the $n = 1000$ level was somewhat unexpected, as the larger problem size should have offset a lot of the communication and setup overhead, but on further inspection it became apparent that the communication load grows linearly with the problem size. However, this does not explain why the performance at this level is so much worse than linear. This could suggest a severe overestimation of the parallelisable proportion of the code, but such a hypothesis would not be consistent with the results of the strong scaling tests.

3.5 Findings

3.5.1 Impediments

There were numerous instances in which errors in programming or test design led to less than satisfactory outcomes during the course of development and testing of the parallel LCS solution. Several of the most impactful of these have been summarised below:

- Memory Allocation: initial phases of testing were conducted under simple conditions, and total input size never exceeded a character length of 10,000 (resulting in a table size of 100 million elements). However, on proceeding to larger input sizes, memory allocation for the table contained in the master process began to fail. In order to optimise on memory usage, the entire codebase was translated from using C++ `std::vector` objects to using raw integer arrays to store the master and sub-section tables. This proved insufficient, as it later emerged that the master table was being constructed (duplicated) in every process, despite being unused in worker processes. Removal of this bug allowed for normal operation until significantly larger problem sizes (>2.5 billion element master tables) where the problem re-emerged. This was solved by requesting larger memory allocations in the `slurm` shell scripts submitted to `sbatch`.
- Resource Binding: another problem encountered due to my relative naivete with respect `slurm` and programming for clusters was not realising that, by default, OpenMP threads allocated to the job will be bound to a single core. This was overcome by passing the `--bind-to none` flag to `mpirun`. The incredible impact of this change on performance can be seen in Figure 3.5, though these results were confounded by a change in cluster from `goliath` to `getafix`. Notice in particular the order of magnitude change in run time. This reduced run time also impacts the uniformity of the results, despite both test runs containing the same number of samples.

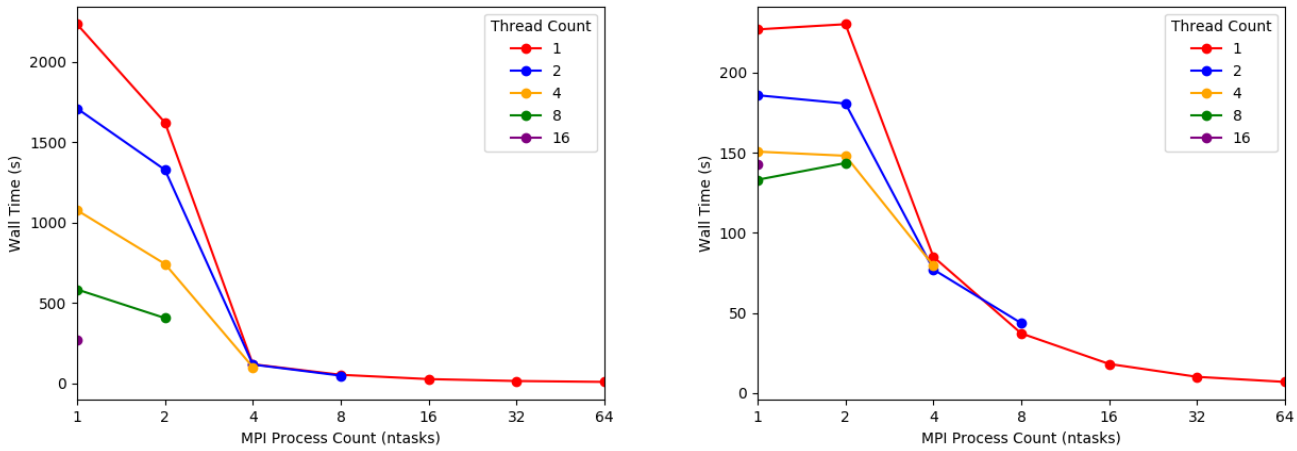


Figure 3.5: Pre-Resource Binding (left, goliath) and Post Resource Binding (right, getafix)

- Level of Threading Support: my initial plans for parallelising the distribution of work to worker processes involved using OpenMP to have each send and receipt completed by a separate thread, IO-heavy work being a perfect place to implement threading. Unfortunately this implementation would have required a level of threading support (`MPI_THREAD_MULTIPLE`) not supported on either `goliath` or `getafix` (both of which only support `MPI_THREAD_SERIALIZED`); the level being set at compile-time (both clusters being compiled). This would have allowed for OpenMP multi-threading in both the master and worker classes simultaneously. Though not implemented, alternative options for parallelising this section are discussed below.

3.5.2 Strengths

The primary strength of the approach taken is that it takes full advantage of the multi-processing capabilities of the clusters on which it was tested. By utilising the diagonal calculation methodology at both the thread and process level, the program was able to achieve speed-ups in excess of the theoretical maximum. Though this is likely due to poor serial performance, it does indicate that the parallelisation has been particularly successful.

3.5.3 Weaknesses & Possible Improvements

There are numerous areas in which the program, as written, lacks sophistication and could stand to be improved. Three of the most serious issues have been detailed below.

Problem Division

By far the greatest weakness of the parallel algorithm as it stands is the relatively naive way in which the problem space is divided into sub-problems, and the way these sub-problems are allocated to worker processes.

The diagonal calculation algorithm utilised in the program is most efficient when sub-sections are perfectly square. This enables the most threads to simultaneously calculate cells. The present heuristic divides the total problem into $n \cdot n$ subsections (where n is the number of processes), without considering the dimensions of the problem. A more advanced heuristic considering these factors, and the relative importance of prioritising the size optimality of the whole problem (where each sub-section is dealt with by processes) against that of the subsections (where each cell on a diagonal may be calculated by independent threads). Dynamic detection of the number of and threads (in addition to that of processes, which is already utilised) may assist in optimising performance in this area.

Allocation

In order to simplify the process of allocating and distributing process, the program short-circuits parallel operation for any less than three processors, due to difficulties imposed by the previously mentioned segmentation heuristic. Optimally, the actual segmentation would remain unchanged, but each sub-section would be allocated to the sole worker process in order. The master process could also be utilised for more actual calculations, making use of the downtime between sending and receiving large sub-sections from worker processes.

Distribution

Due to numerous, significant, and time-consuming problems in implementing asynchronous behaviour, all MPI send and receive operations are conducted synchronously. It is strongly recommended that the sending of sub-section data to worker processes and the receipt of completed sub-sections be conducted in parallel. The nature of the algorithm would necessitate that all sub-sections on a diagonal be finished before the next begins. Hence, an `MPI_Waitall` will be required between diagonals. There is no discernible benefit in asynchronously sending and receiving on the worker-process end, due to the linear nature of their operation.

3.5.4 Further Scaling

The range of testing configurations was severely limited by the resources allocated to the course, specifically the number of nodes available on the `getafix` *cosc* and `goliath` *coursework* partitions. Time was also a limiting factor as due to the shared nature of these resources.

In order to support truly massive input sizes, the program would need to be modified to offset limitations imposed by a lack of available hardware.

Memory proved to be the most pressing resource constraint in the small-scale testing conducted for this report, and there is no reason to expect this would not be the case for even larger input sizes. The sheer size of the master table (already in the billions of elements for conducted tests) will eventually result in a lack of sufficient RAM to store the table. This could be offset by keeping only those values necessary for computation in memory and storing already computed values on disk. Entire sections could be saved to files and reloaded on demand. This would introduce additional IO overhead, but may be unavoidable and would certainly be offset by sufficient processing gains. This would also have the advantage of providing progress "checkpoints" that may allow for the resumption of processing in the event of interruption.

Processing power will also (eventually) hit physical limitations. It may be worth modifying the division of the problem such that the number of segments on the major anti-diagonal outnumbers the process count, such that each each segment is computable in a reasonable amount of time with the (potentially) limited resources of each node. This would introduce additional logical complexity in the distribution and receipt of segments, but may produce additional performance gains in the event of a cluster consisting of a high number of low resource nodes.

Bibliography

[1] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pp. 39–48, Sep. 2000.

[2] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, pp. 350–353, May 1977.

[3] Z. Li, A. Goyal, and H. Kimm, "Parallel longest common sequence algorithm on multicore systems using OpenACC, OpenMP and OpenMPI," in *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, IEEE, Sept. 2017.

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[5] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, pp. 532–533, May 1988.

Appendices

Appendix A

Running the Program (Deprecated)

```
svn co https://source.eait.uq.edu.au/svn/cosc3500-s4261833/trunk/MS1
cd MS1
make
./bin/lcs test_input.txt test_output.txt
```

Listing A.1: Usage Instructions

Appendix B

Running the Performance Tests (Deprecated)

```
svn co https://source.eait.uq.edu.au/svn/cosc3500-s4261833/trunk/MS1
cd MS1
make
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python -m testing <num_tests> <max_length>
```

Listing B.1: Usage Instructions

Appendix C

Slurm Script

```
#!/bin/bash
#SBATCH    partition=coursework
#SBATCH     job  name=:sad_parrot:
#SBATCH     nodes=1
#SBATCH     ntasks=4
#SBATCH     ntasks  per  node=4
#SBATCH     cpus  per  task=4

export OMP_NUM_THREADS=4
export SLURM_TASKS_PER_NODE=4
export SLURM_NPROCS=4

DATE=$(date +"%Y%m%d%H%M")
echo "time started "$DATE
echo "This is job  $SLURM_JOB_NAME  (id: $SLURM_JOB_ID) running on the following nodes:"
echo $SLURM_NODELIST
echo "running with OMP_NUM_THREADS= $OMP_NUM_THREADS "
echo "running with SLURM_TASKS_PER_NODE= $SLURM_TASKS_PER_NODE "
echo "running with SLURM_NPROCS= $SLURM_NPROCS "
echo "Now we start the show:"
export TIMEFORMAT="%E sec"

module load mpi/openmpi-x86_64
time mpirun -n ${SLURM_NPROCS} ./bin/lcs-hybrid test_input.txt test_output.txt

DATE=$(date +"%Y%m%d%H%M")
```

```
echo "time finished "$DATE
# echo "we just ran with the following SLURM environment variables" # env | grep SLURM
```

Listing C.1: Slurm Script (go.sh)

Appendix D

Testing Framework Operation

```
svn co https://source.eait.uq.edu.au/svn/cosc3500-s4261833/trunk/MS3

make

python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

python -m testing run strong 10000 20
# Wait for the resulting slurm jobs to finish
mv ./tests ./tests-strong
python -m testing report strong ./tests-strong

python -m testing run weak 20
# Wait for the resulting slurm jobs to finish
mv ./tests ./tests-weak
python -m testing report weak ./tests-weak
```

Listing D.1: Testing Framework Operation

Appendix E

Slurm Template

```
#!/bin/bash
#SBATCH --partition=coursework
#SBATCH --job-name={{name}}
#SBATCH --nodes={{nnodes}}
#SBATCH --ntasks={{ntasks}}
#SBATCH --ntasks-per-node={{ntasks_per_node}}
#SBATCH --cpus-per-task={{cpus_per_task}}
#SBATCH --time=1:00:00

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export TIMEFORMAT="%E sec"

echo "{{name}} - {{input}}"

module load mpi/openmpi-x86_64
time mpirun -n ${SLURM_NPROCS} {{binary}} {{input}} {{output}}

DATE=$(date +"%Y%m%d%H%M")
echo "time finished "$DATE
```

Listing E.1: Slurm Template (template.sh)

Appendix F

Actual Strong Testing Configuration

Nodes	Tasks	Tasks/Node	CPUs/Task
1	1	1	1
1	1	1	2
1	1	1	4
1	1	1	8
1	1	1	16
1	2	2	1
1	2	2	2
1	2	2	4
1	2	2	8
1	4	4	1
1	4	4	2
1	4	4	4
1	8	8	1
1	8	8	2
1	16	16	1
2	2	1	1
2	2	1	2
2	2	1	4
2	2	1	8
2	4	2	1
2	4	2	2
2	4	2	4
2	8	4	1
2	8	4	2
2	16	8	1
2	32	16	1
4	4	1	1
4	4	1	2
4	4	1	4
4	8	2	1
4	8	2	2
4	16	4	1
4	32	8	1
4	64	16	1

Table F.1: Strong Scaling Test Configuration

Appendix G

Full Strong Scaling Results

Nodes	Tasks	Tasks/ Node	CPUs/ Task	Problem Size	Problem Size/ Task	Run Time	Processing Units	Theoretical Speedup	Actual Speedup
1	1	1	1	10000	10000	226.8824	1	1.00	1.00
1	1	1	2	10000	10000	185.8092	2	1.90	1.22
1	1	1	4	10000	10000	150.6327	4	3.48	1.51
1	1	1	8	10000	10000	133.09	8	5.93	1.70
1	1	1	16	10000	10000	142.4498	16	9.14	1.59
1	2	2	1	10000	5000	228.7812	2	1.90	0.99
1	2	2	2	10000	5000	178.9482	4	3.48	1.27
1	2	2	4	10000	5000	148.2034	8	5.93	1.53
1	2	2	8	10000	5000	146.5848	16	9.14	1.55
1	4	4	1	10000	2500	84.7677	4	3.48	2.68
1	4	4	2	10000	2500	77.944	8	5.93	2.91
1	4	4	4	10000	2500	82.579	16	9.14	2.75
1	8	8	1	10000	1250	36.5175	8	5.93	6.21
1	8	8	2	10000	1250	60.5753	16	9.14	3.75
1	16	16	1	10000	625	18.1023	16	9.14	12.53
2	2	1	1	10000	5000	231.4666	2	1.90	0.98
2	2	1	2	10000	5000	182.3175	4	3.48	1.24
2	2	1	4	10000	5000	147.9168	8	5.93	1.53
2	2	1	8	10000	5000	140.4557	16	9.14	1.62
2	4	2	1	10000	2500	85.2426	4	3.48	2.66
2	4	2	2	10000	2500	78.4404	8	5.93	2.89
2	4	2	4	10000	2500	83.6625	16	9.14	2.71
2	8	4	1	10000	1250	37.9601	8	5.93	5.98
2	8	4	2	10000	1250	35.2048	16	9.14	6.44
2	16	8	1	10000	625	18.3757	16	9.14	12.35
2	32	16	1	10000	312	10.2094	32	12.55	22.22
4	4	1	1	10000	2500	85.1312	4	3.48	2.67
4	4	1	2	10000	2500	74.9924	8	5.93	3.03
4	4	1	4	10000	2500	72.5175	16	9.14	3.13
4	8	2	1	10000	1250	37.2422	8	5.93	6.09
4	8	2	2	10000	1250	35.1404	16	9.14	6.46
4	16	4	1	10000	625	18.0858	16	9.14	12.54
4	32	8	1	10000	312	10.0879	32	12.55	22.49
4	64	16	1	10000	156	6.9848	64	15.42	32.48

Table G.1: Full Strong Scaling Results (getafix, n=20)

Appendix H

Weak Scaling Results

Nodes	Tasks	Tasks/ Node	CPUs/ Task	Problem Size	Problem Size/Task	Run Time	Processing Units	Theoretical Speedup	Actual Speedup
1	1	1	1	10	10	0.434	1	1	1.00
1	2	2	1	20	10	0.465	2	1.95	1.86
1	4	4	1	40	10	0.414	4	3.85	4.19
1	8	8	1	80	10	0.395	8	7.65	8.78
1	16	16	1	160	10	0.518	16	15.25	13.40
2	32	16	1	320	10	0.877	32	30.45	15.83
4	64	16	1	640	10	0.787	64	60.85	35.25
1	1	1	1	100	100	0.462	1	1	1.00
1	2	2	1	200	100	0.749	2	1.95	1.23
1	4	4	1	400	100	0.551	4	3.85	3.36
1	8	8	1	800	100	0.700	8	7.65	5.28
1	16	16	1	1600	100	1.119	16	15.25	6.61
2	32	16	1	3200	100	2.114	32	30.45	6.99
4	64	16	1	6400	100	3.644	64	60.85	8.11
1	1	1	1	1000	1000	8.036	1	1	1.00
1	2	2	1	2000	1000	24.439	2	1.95	0.66
1	4	4	1	4000	1000	17.651	4	3.85	1.82
1	8	8	1	8000	1000	33.397	8	7.65	1.92
1	16	16	1	16000	1000	66.100	16	15.25	1.95
2	32	16	1	32000	1000	136.411	32	30.45	1.89
4	64	16	1	64000	1000	292.969	64	60.85	1.76

Table H.1: Full Weak Scaling Results