Parallelisation of the Smith-Waterman Algorithm

Au Liang Jun (A0173593W), Hu Yiqing (A0167053H), Yan Hong Yao Alvin (A0159960R)
CS3211 Parallel & Concurrent Programming AY18/19 Semester 2

1 Introduction

The Smith-Waterman algorithm[1] is a dynamic programming algorithm used for local sequence alignment of 2 strings, commonly used for nucleic acid sequences or protein sequences. In this report, we detail our implementation of sequential and parallel versions of the algorithm in x10[2], a Java-like programming language built for parallel computing. We then run experiments to draw insights, including the speedup obtained.

2 Problem

The algorithm performs local sequence alignment between two strings, i.e. aligning the characters in the most optimal way according to a specified scoring system, a set of rules and parameters. In Smith-Waterman, there is no negative scoring, and a traceback across computed scores takes place from the highest scoring matrix cell until a cell with the score 0 is encountered. More details are available on Wikipedia[2]. In our report, we will be utilising terms related to the algorithm extensively. We define them here:

- *string1, string2*: The first and second strings the algorithm receives as input
- *m, n:* The length of the longer and shorter string respectively. Without loss of generality, we consider *string1* to be the longer string and *string2* be the shorter string.
- *affine gap*: The *affine gap* is the gap-scoring scheme utilised in our implementation. A gap-scoring scheme is a method of scoring alignments that allow the introduction of gaps at a specified cost. The *affine gap* takes the form $f(x) = ux + v$, where x is the length of the gap - 1, u is the *gap extension cost* and v is the *gap opening cost*.
- *substitution*: A constant size matrix specifying the score rewarded or penalty incurred for successful and unsuccessful matches respectively. Together with the gap-scoring scheme, the substitution matrix makes up the scoring system. In this report, we experimented with the `BLOSUM62`, a popular substitution matrix used in bioinformatics[3].
- *score*: A $(m + 1) \times (n + 1)$ matrix containing the scores calculated. The left and top borders are padded with *0s*, and the value at the *ith* and *jth* index correspond to the score attained by aligning the *i-1th* character in one string and *j-1th* character of the other.
- *directions*: A $m \times n$ matrix containing the directions taken by each cell (i.e. $\rightarrow$, $\downarrow$ or $\searrow$). The value at the *ith* and *jth* index corresponds to the direction taken to reach the cell.

---

[1] "Smith–Waterman algorithm - Wikipedia." https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm. Accessed 21 Apr. 2019.
[2] "The X10 Programming Language." http://x10-lang.org/. Accessed 21 Apr. 2019.
[3] "BLOSUM - an overview | ScienceDirect Topics." https://www.sciencedirect.com/topics/medicine-and-dentistry/blosum. Accessed 21 Apr. 2019.

3 Documentation

3.1 Sequential Algorithm

Our implementation of the algorithm comprised 2 iterations. The first is the original Smith-Waterman algorithm as described which has an asymptotic runtime of $O(m^2n)$, while the second is an algorithm proposed by Gotoh over the original Smith-Waterman algorithm which gives a faster runtime of $O(mn)$.

Naive Implementation

This implementation of the Smith-Waterman algorithm does not store intermediate results and repeatedly recalculates the best index to open a gap from.

We make use of the `Array_2` data structure, 2 dimensional arrays provided by x10 to represent the matrices inside the problem. The following 2 arrays are used in our implementation:

`score`: Represents the *score* matrix. The highest score in the matrix and its coordinates are tracked with variables `globalMax:Long` and `maxCoordinates:Pair[Long, Long]`.

`directions`: Represents the *directions* matrix. In our implementation, we represent → as -1, ↓ as 1 and ↘ as 0.

Algorithm

1. We traverse the `score` array in a row-major order to perform the dynamic programming step to compute the maximum score for each cell. For each cell:

    1.1. Create `max:Long` and `dir:Long` variables to represent the score and direction of the best possible step to have taken to arrive at this cell.

    1.2. Check the score for aligning the respective characters in each string (↘ movement). Update `max` and `dir`.

    1.3. Get the score for creating a gap in *string1* (↓ movement) with the `checkUpwards()` method. `checkUpwards()` considers all possible lengths of gaps to open in *string1* to determine the maximum score. Update `max` and `dir` if score is greater than present `max`.

    1.4. Check the score for creating a gap in *string2* (→ movement) with the `checkLeftwards()` method. As in 1.3, the algorithm checks and all possible lengths of gaps to open in *string2*. Update `max` and `dir` if score is greater than present `max`.

    1.5. Update the `globalMax` and `maxCoordinates` if applicable, and set `matrix(i, j)` to `max` and `directions(i, j)` to `dir`.

2. Finally we perform the traceback step by traversing backwards, beginning from the `maxCoordinates` and following the reverse of the directions in `directions`. This will result in the sequence of steps that provide the maximum score as stored in our array.

Asymptotic Analysis

Step 1 of the algorithm iterates over an m by n matrix. Step 1.3 performs the `checkUpwards()` method, which has a runtime of $O(m)$ (iterating the entire length of *string1*). Similarly, Step 1.4's `checkLeftwards()` runs in $O(n)$. The other substeps of Step 1 are constant time operations. As a result, Step 1 has an overall runtime of $O(mn \times (m + n)) = O(m^2n + mn^2)$.

Step 2, the traceback algorithm, has a runtime of $O(m + n)$, as the longest length of the resultant string (and the order of operations of the traversal takes) is the length of *string1* and *string2*.

The overall runtime of the algorithm is thus $O(m^2n + mn^2 + m + n) = O(mn^2)$.

As two data structures `score` or `directions` of size $\sim m \times n$ are created, the space complexity is $O(2mn) = O(mn)$.

Time Complexity: $O(m^2n)$

Space Complexity: $O(mn)$

Memoized Implementation

To improve runtime, we implemented a second algorithm similar to Gotoh's version of the Smith-Waterman. Gotoh's algorithm is largely similar to the original Smith-Waterman, however, the dynamic programming step is sped up by trading in memory.[4]

Instead of a single `score` matrix, scores are tracked in three separate `Array_2` matrices. For any *i, j*, the following matrices store the score of best alignment up until the *(i - 1)th* character of *string1* and the *(j - 1)th* character of *string2* subject to:

`bestLeftwards`: Ending with a gap in *string2*.

`bestUpwards`: Ending with a gap in *string1*.

`bestDiagonal`: Ending with a character match or mismatch.

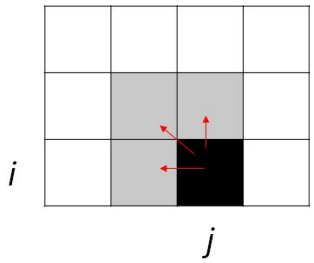


Figure 3.1.1 Obtaining the Score and Direction Taken for some *i, j*

In order to obtain the score and direction taken to reach *i,j*, the values at the *i,j* index of `bestLeftwards`, `bestUpwards` and `bestDiagonal` scores are compared to find the maximum value. This is representative of the act of finding the best possible way to reach an alignment at the *ith, jth* index.

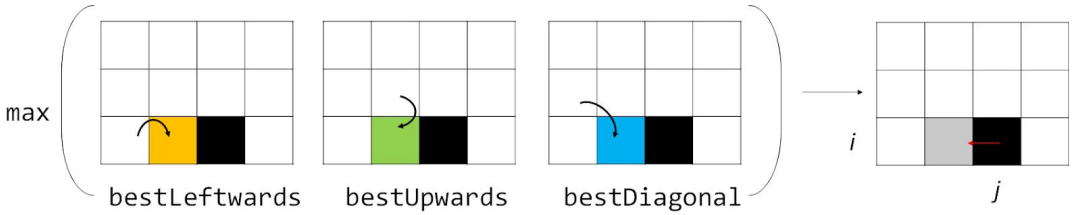


Figure 3.1.2 Calculation of → Movement for *(i, j)*

[4] (n.d.). General & Affine Gap Penalties. Retrieved April 18, 2019, from https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/gaps.pdf

To illustrate how the *ith,jth* value of the score matrices are calculated, consider the calculation of the `bestLeftwards` value (Figure 3.1.2). This corresponds to a → movement from the previous cell (*i, j-1)*, and as such the best possible score of (*i, j-1)* has to be obtained first, and this is performed by considering all the possibilities of reaching it. This includes extending an existing gap in *string2* (`bestLeftwards`), closing a gap in *string1 and opening* a new gap in *string2* (`bestUpwards`), or opening a new gap in *string2* following a match/mismatch (`bestDiagonal`). This corresponds to the following x10 code block:

```
bestLeftwards(i, j) = max(
      gapOpening + bestDiagonal(i, j - 1),
      gapExtension + bestLeftwards(i, j - 1),
      gapOpening + bestUpwards(i, j - 1));
```

Algorithm

1. As in the naive implementation, we traverse the `bestLeftwards`, `bestUpwards` and `bestDiagonal` arrays in row-major order to perform the dynamic programming step to compute the maximum score for each cell. For each *i, j*:

    1.1. Calculate the ith, jth value of `bestDiagonal` (↘ movement).

    1.2. Calculate the ith, jth valueof `bestUpwards` (↓ movement)

    1.3. Calculate the ith, jth value of `bestLeftwards`

    1.4. Compare the ith, jth values of `bestDiagonal`, `bestLeftwards` and `bestUpwards` to obtain the `max` value. Set value in `directions` matrix based on the source of `max` and update `globalMax` and `maxCoordinates` if applicable.

2. Perform traceback as we do in the naive implementation.

Asymptotic Analysis

Memoization removes the extra linear time search performed within the substeps of Step 1, allowing each cell's score to be computed in constant time *O(1)*. Step 2, as before, has an asymptotic runtime of *O(n + m)*. Overall time complexity is thus *O(mn + m + n) = O(mn)*.

As scores are tracked with 3 matrices of size *m x n* instead of 1 matrix of the same size, taking in account the fixed m x n directions matrix, the memory space required by the program is effectively doubled. Space complexity is *O(4mn) = O(mn)*.

Time Complexity: *O(mn)*

Space Complexity: *O(mn)*

As this implementation is significantly more performant than the non-memoized version, we have decided to utilise it for experimentation and parallelisation. For the rest of the report, we will only be considering the memoized version of the Smith-Waterman algorithm. The source code can be found in the `match()` method of `MemoizedSW.x10`.

3.2 Parallel Algorithm

To explore how the algorithm could be parallelizable, we sought to find out how the problem could be decomposed into tasks that could be executed concurrently.

We first noticed that in the score matrices, the value in each cell *(i, j)* depended on the scores in cells *(i - 1, j)*, *(i, j - 1)* and *(i - 1, j - 1)*, i.e. the cell directly above, to the left and diagonally upwards and leftwards. Cells on the edges also shared the same dependencies where they exist. In Figure 3.3.1, the dependencies of the cells in the score matrices are illustrated with red arrows, with cells in black depending on the values of the cells in grey.

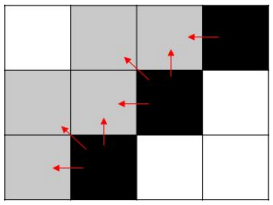


Figure 3.2.1 Dependencies of Cells in a Diagonal

This meant that cells in the same diagonal do not depend on each other, instead depending on the immediate 2 diagonals that were calculated before it. Figure 3.2.1 illustrates this with an arbitrary diagonal (denoted by the black squares) in a 3 x 4 matrix and its dependencies (grey squares). We thus came up with the idea to perform diagonal striding in the manner shown in Figure 3.2.1.

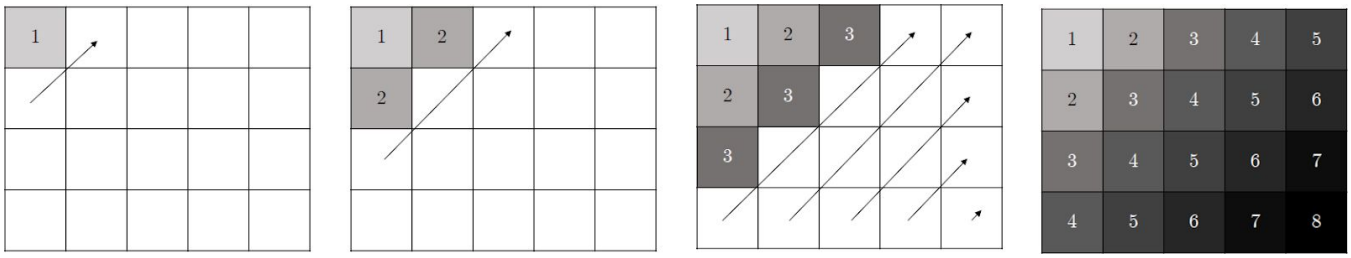


Figure 3.2.2 Diagonal Striding of a *4* x *5* Matrix

In the figure, the number *i* in each cell represents that the cell value is calculated in the *i*th diagonal stride in the array.

Furthermore, within each diagonal stride, the cells calculated are independent of each other, not requiring any other cell in the stride to complete before it can carry out the calculation.

We thus achieved data decomposition of the problem. More specifically, we achieved output data decomposition. Each diagonal stride can be parallelised by having each cell be computed by a different thread. The end of each stride is asynchronous and waits for all the threads to be completed. An illustration of the sequential and diagonal components of the program is shown in Figure 3.2.2.

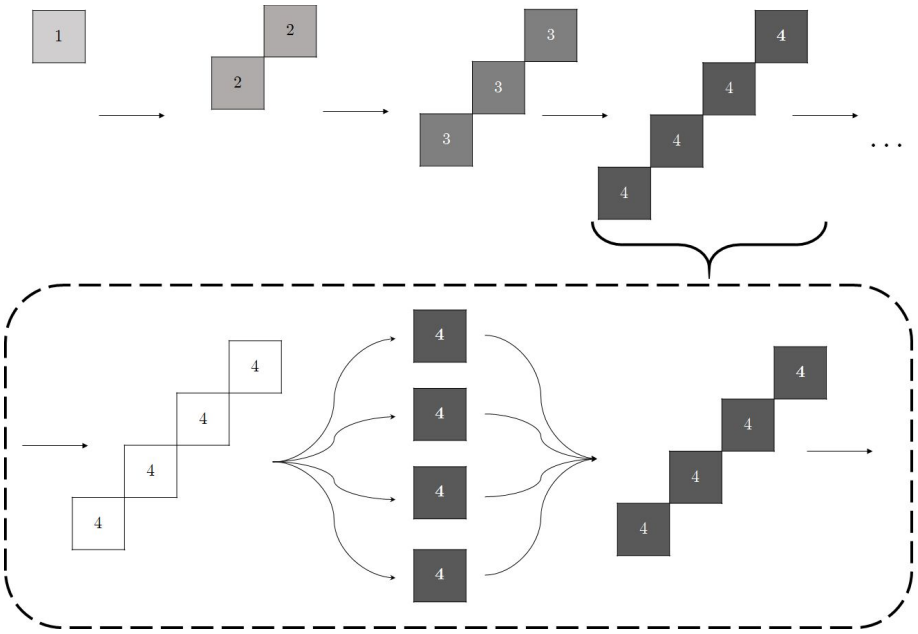


Figure 3.2.3 Execution Graph

In x10, diagonal striding was implemented based off a GeeksforGeeks 2D array diagonal traversal algorithm[5]. Parallelisation was then implemented by augmenting the for loop that is iterating over each block in a stride with finish and async statements:

```
finish for(k in 0..(numBlocks - 1)) async {...}
```

The full implementation can be found in the method `parallelMatch()` in `MemoizedSW.x10`.

Asymptotic Analysis

Assuming no resource limits and negligible cost of parallelisation, each diagonal can be completed in constant time with our method of parallelisation. Since there are $m + n - 1$ diagonals within the matrix to compute, our algorithm can achieve a theoretical time complexity of $O(m + n - 1) = O(m)$.

Optimisation with Blocks

After performing the implementation detailed above, we observed that the runtime was significantly longer than the sequential algorithm for small problem sizes (<100 length strings). We deduced that the overhead of creating threads exceeds the time saved to have each calculation on the diagonal be handled by one thread. To counteract this problem, we reduced the granularity of our decomposition.

Instead of only allocating each thread one cell to compute, we introduce a variable *cutoff* which determines a *cutoff* x *cutoff* size block of cells that will be allocated to each thread instead. Incomplete blocks after partitioning would remain on the right and bottom edges. Within each block, cells are computed sequentially as they are iterated over in row-major order. The diagonal striding pattern still exists, but the strides now take place over entire blocks (Figure 3.2.4). In Section 4, we perform experiments to determine the optimal values for *cutoff*.

---

[5] (n.d.). Zigzag (or diagonal) traversal of Matrix - GeeksforGeeks. Retrieved April 18, 2019, from https://www.geeksforgeeks.org/zigzag-or-diagonal-traversal-of-matrix/

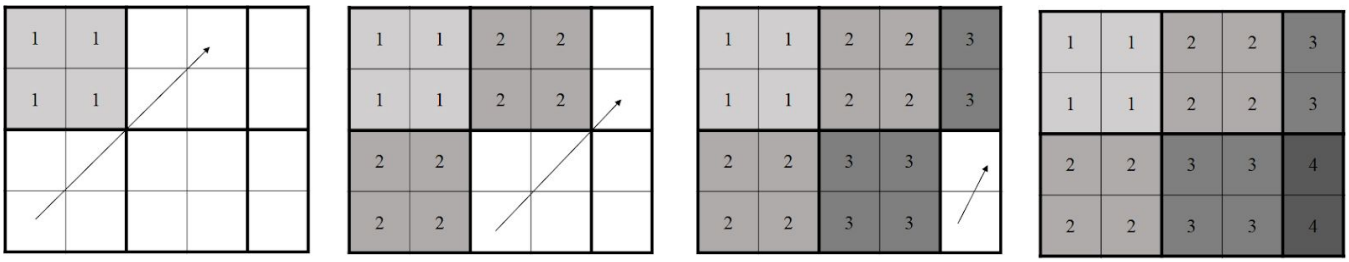


Figure 3.2.4 Diagonal Striding of a *4* x *5* Matrix with *cutoff = 2*

Determining Block Cutoff

Interestingly, a theoretical analysis would show that this enhancement would result in a longer runtime. Let us call the original algorithm the *optimistic* parallel algorithm, and the enhanced algorithm the *optimised* algorithm.

1	2	4	6
2	3	6	8
3	5	7	9
5	7	9	10

optimistic

1	2	5	6
3	4	7	8
5	6	9	10
7	8	11	12

optimised

Figure 3.2.5 Theoretical Runtime (Time Units) of *optimistic* vs *optimised* with *cutoff = 2*

Consider the runtime of the algorithm executed on a 2 core machine. Assuming each cell takes exactly 1 time unit to calculate, starting from time unit 0, the figures above shows the time unit in which that individual cell has been calculated. Grey denotes the cells calculated by core A, while white denotes the cells calculated by core B. We consider a problem of $m = n = 4$ with *cutoff = 2* for brevity.

In Figure 3.2.5, we can observe the time unit at which both versions of the algorithm complete the task by looking at the cell at the bottom right corner, as it is always the last to be calculated. We observe that there is an extra cost incurred via calculations by the *optimised* algorithm as compared to the *optimistic* of 2 time units. This extra cost of the *optimised* algorithm comes from the cores that are now idle as a result of implementing the *cutoff*. At time unit 2, for the *optimistic* implementation, core A is calculating cell *(0,1)* whilst core B is calculating cell *(1,0)*. Comparing to the *optimised* algorithm, at time unit 2, core A is calculating cell *(0,1)* but core B is idling. Core A will only start calculating *(1,0)* in the next time unit. From this example, we can see how in theory, the *optimistic* algorithm will have a more efficient utilization of the cores.

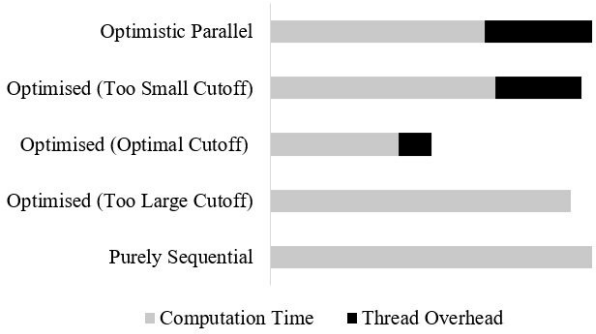


Figure 3.2.6 An Illustration on the proportion of time costs

The above discussion brings an important consideration. As the *cutoff* size increases, the theoretical efficiency of the algorithm reduces. At the same time, a *cutoff* would not yield the thread overhead savings the block optimisation was implemented for. As such, the optimal *cutoff* is essentially a balance between both factors.

Studying the extremes,

- If *cutoff = 1*, the performance will be similar to the *optimistic* algorithm, as the same number of threads will be created.
- If *cutoff* $\geq m$, the performance will be similar to the sequential implementation, as only one core will be able to execute at any time.

Figure 3.2.6 illustrates this tradeoff.

3.3 Memory Limitations

We were aware of certain memory-related shortfalls in our implementation.

1) As covered in the previous section, the memoized algorithm utilises a total of 4 *m* by *n* matrices. The relatively large space used would make an increasing impact on runtime as the problem size increases, as the amount of memory required exceeds the cache sizes.

2) The implementation of diagonal striding, in contrast with iteration in row-major order, will make the threads (and in turn CPU) perform some vertical striding down the rows between each diagonal stride. This reduces the spatial locality of our implementation and increases the memory overhead.

We have considered the possibility of implementing an optimisation that would reduce the space complexity of the parallel algorithm and address point 1) above.

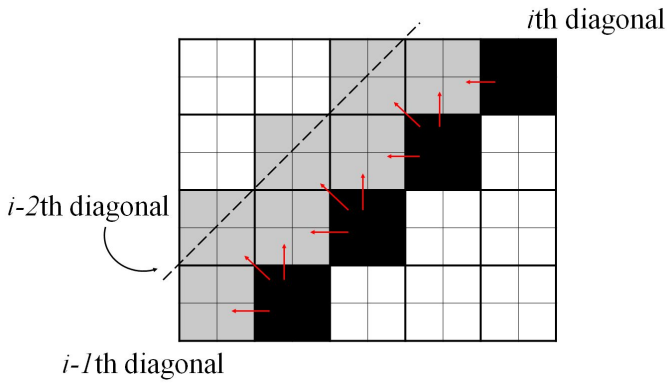


Figure 3.3.1 Dependency of *i*th Diagonal

In Figure 3.3.1, we observe that each diagonal i only depends on diagonals $i - 1$ and $i - 2$. As a result, at any point of time, we only need to retain the scores of 3 diagonals. As there are 3 separate scores to track, and each diagonal consists of $O(cutoff^2 \times n)$ cells, we thus only require $O(3 \times cutoff^2 \times n)$ space to compute the scores, compared to $O(3mn)$. As the directions array would still require $O(mn)$ of space, this optimisation would produce an algorithm with $O((m + 3 \times cutoff^2)n)$ space complexity.

However, we were not able to implement this optimisation due to time limitations. This is an area of potential expansion and future work, and successful implementation would reduce memory usage and potentially improve runtime for large problem sizes. We expect this to be so as a larger portion of the program data would be stored in cache and RAM, reducing the need to fetch data from the RAM and disk respectively.

4 Evaluation

The experiments were performed on the xcnd cluster, which utilises a Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz CPU. The cluster has a total of 12 physical cores and 24 logical cores. Each experiment is performed 3 times, with the minimum time taken recorded.

4.1 Verifying Memoized Sequential Implementation

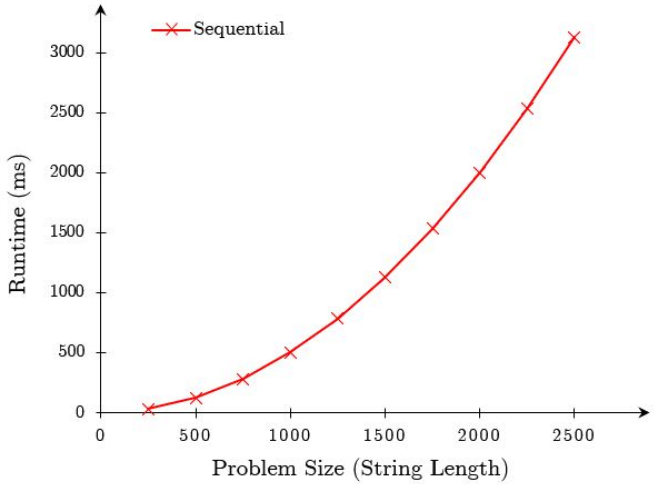


Figure 4.1.1 Runtime of Implementation against Problem Size

The memoized Smith-Waterman algorithm has a time complexity of *O(mn)*. We performed some tests by creating 2 randomised strings of the same length between 250 and 2500. The results indicate that the time taken for our sequential program indeed follow a quadratic curve as plotted in Figure 4.1.1.

4.2 Parallel Implementation Optimisation

The first experiment performed is to empirically find the value of *cutoff* that provides the best performance in the parallel implementation given the problem size. We create random string pairs of size 128 to 8192 and tested their runtime with different values of *cutoff* from 1 to 128.

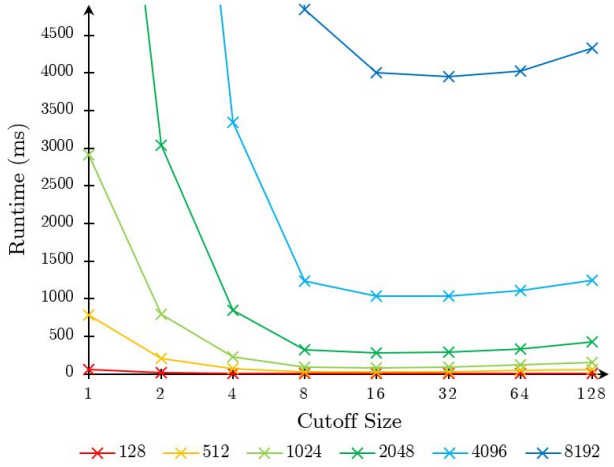


Figure 4.2.1 Runtime against Cutoff Size for $n = m$ Length Strings

A value of 1 for cutoff size represents the runtime of our unoptimised parallel algorithm.

Within the same problem size (a single line), we first note that using a value larger than 1 for *cutoff* significantly improves performance for all problem sizes measured. Secondly, the trend follows a minimum curve, where increasing *cutoff* initially decreases runtime of the parallel algorithm, but beyond a minimum point runtime slowly increases.

The initial downwards trend as *cutoff* size increases is in line with our intuition for performing optimisation with blocks. Reducing the number of threads created by increasing cutoff leads to time savings from less thread overhead, which outweighs the increase in the time units needed to finish computation on a larger *cutoff* × *cutoff* block (as explained in Section 3.2). Increasing *cutoff* past a certain point however, resulted in increasing runtime as the algorithm become increasingly sequential. With a large block size, the number of threads are reduced and each block has a greater number of cells within that are calculated sequentially. In this case, the increase in time units required for a larger *cutoff* outweighs the reduction in thread overhead.

Between the problem sizes (across the lines), we also realised that the optimal value of *cutoff* varied between different problem sizes (see table in appendix for clearer illustration). As the problem size increased, the optimal cutoff moved from between 8 and 16 to 16 and 32. This leads us to believe that *cutoff* should be dynamically set based on the problem size. However, due to constraints and difficulty formulating and proving a formula for *cutoff* calculation, we elected to run more fine-grained experiments and deduce a fixed *cutoff* for the rest of our analysis. As such, the rest of the experiments were conducted with ***cutoff = 20***.

4.3 Performance Analysis

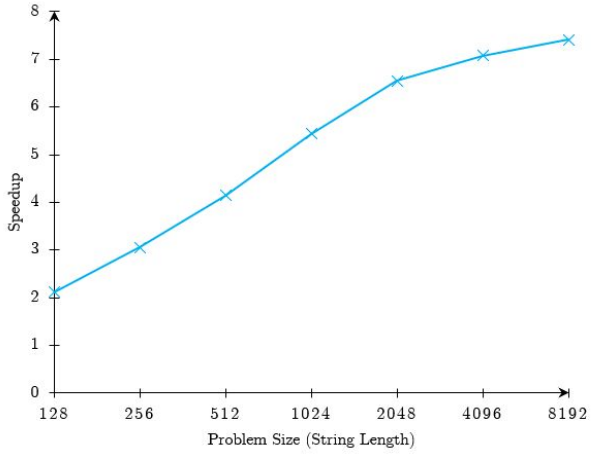


Figure 4.3.1 Speedup against Problem Size

As seen from Figure 4.3.1, overall speedup increases as problem size increases. The rate in which speedup increases is decreasing as the problem size gets larger; speedup first increases logarithmically from 2 to ~7.6 and then begins to plateau after.

When the problem size is too small, we are unable to fully realize the effects of all the cores running simultaneously. For example, take the smallest measured problem size of $m = n = 128$. Given the *cutoff* of 20, the longest diagonal will have $ceil(128/20) = 7$ blocks. This corresponds to at most 7 cores running concurrently, lower than the maximum number of cores that are available (24). For larger problem sizes, more available cores can be simultaneously utilised. The number of cores N in $S = \frac{1}{\frac{p}{N} + (1-p)}$ would increase, hence resulting in greater speedup.

Beyond a certain problem size however, the increase in speedup slows down and plateaus. At around a problem size of $m = n = 20 \times 24 = 480$, the maximum number of blocks in the diagonal is 24. All the available cores will be utilised during the calculation of this diagonal. Beyond this point, speedup no longer increases as quickly with the problem size, as diagonals with greater number of blocks cannot be executed entirely in parallel anymore. However, the parallel fraction p in $S = \frac{1}{\frac{p}{N} + (1-p)}$ would still increase, given that the number of diagonals with > 24 blocks would form a greater percentage of the problem. As the parallel fraction approaches 100%, speedup would taper off correspondingly, explaining the trend observed in the figure.

4.4 Amdahl's and Gustafson's Speedup

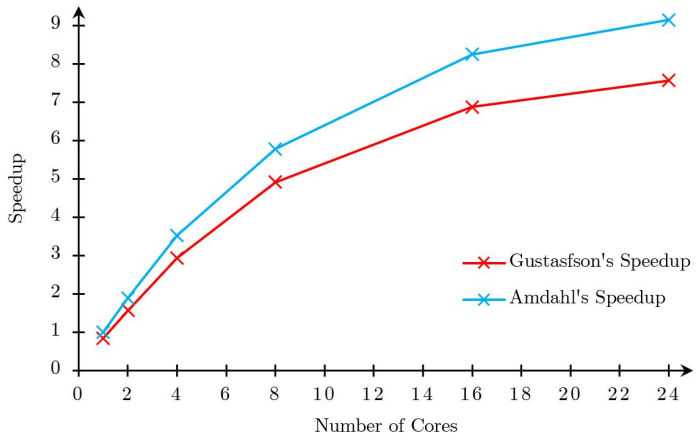


Figure 4.4.1 Speedup against Number of Cores

We experiment to find Amdahl's and Gustafson's speedup for our algorithm by setting $n = 8192$ (for Amdahl's) and a parallel runtime of *4000ms* (for Gustafson's). $n = 8192$ was chosen to measure Amdahl's speedup as we wanted a large problem size with a reasonable parallel runtime (*4000ms*). The fixed parallel runtime for measuring Gustafson's speedup was also derived from this. The number of cores was set with the Linux `taskset` command[6].

Amdahl's speedup with 24 cores used is around 9x, which is reasonable when taken in relation to the theoretical speedup possible given the hardware used, and our implementation.With 24 available cores, the expected maximum speedup achievable would be around 24 times, but discounts the fact that the virtual cores do not provide the same efficiency as a similar number of physical cores. Additionally, we would still need to account for the time cost of spawning and despawning of threads for each diagonal we have to process, which accounts for further time costs. Finally, because of our implementation with the *cutoff*, there will be instances where not all cores can be continuously active. This is due to our algorithm portioning each block to a certain core and does not allow other cores to work on it even if the other core is available and has completed its computations.

Gustafson's speedup is lower than Amdahl's in our experiments. A possible reason is that the parallel capacity of the machine we have used is not enough to allow us to test the scalability of our algorithm to large problem sizes with more CPU cores.

[6] "taskset(1): retrieve/set process's CPU affinity - Linux man page - Die.net." https://linux.die.net/man/1/taskset. Accessed 22 Apr. 2019.

4.5 Edge Case Analysis

In this section, we explore edge cases, namely if one string is significantly longer than the other, or if the strings perfectly match/mismatch.

	string1* >> *string2 **(8192 >> 128)**	***string1* << *string2*** **(128 << 8192)**
Sequential (ms)	457.598	449.395
Parallel (ms)	124.314	127.590

Figure 4.5.1 Runtime with Different Length Strings

We first run experiments if one string is much shorter at *128* and the other is much longer at *8192*, in the two different permutations (Figure 4.5.1). Between the two permutations, it appears that there are no significant differences in runtime for the sequential and parallel implementations, as the timings recorded were well within a 5% margin of error.

	Perfect Match Strings (m = n = 1024)	**No Match Strings (m = n = 1024)**
Sequential (ms)	437.133	450.292
Parallel (ms)	87.190	86.665

Figure 4.5.2 Runtime with Perfectly Matched/Mismatched Strings

We then compare runtimes when both strings perfectly match, as well as when both strings do not match in any character. Strings used were *1024* in length. As seen in Figure 4.6.2, the sequential and parallel runtimes between the two sets of strings are also within a 5% margin of error. This is expected as the same number of calculations should be done in either case, and they only differ by the runtime of traceback.

As the experiments in Figure 4.6.1 and 4.6.2 actually deal with the same overall problem size $128 \times 8192 = 1024 \times 1024$, it is interesting to note that the sequential time taken is roughly the same between the two sets of experiments. However, the time taken by the parallel algorithm is significantly longer for the first experiment with different length strings. We deduce that this observation is due to the length of the diagonals in both sets of problems. As the total number of cores that can run concurrently with our parallel algorithm is dependent on the length of the diagonal, a 128×8192 matrix will only be able to make use of $ceil(128/20) = 7$ cores concurrently for most of its execution time, given it's long, rectangular shape. This is in contrast with the 1024×1024 matrix, which due to its square shape is able to create >24 threads and utilise all 24 cores for the computation many diagonals. Thus, the longer runtime of the parallel algorithm in the first experiment reflects its poor efficiency in handling strings with big difference in lengths, a limitation the sequential algorithm does not share.

5 Conclusion

In summary, we have completed a study on the parallelisation of the Smith-Waterman local alignment algorithm. In x10, we first implemented a sequential version of the algorithm, and by studying Gotoh's implementation, we also performed memoization, allowing us to reduce the time complexity of the algorithm to *O(mn)*. We then identified a way to decompose the problem and parallelise the algorithm. In the process, we derived and implemented a working optimisation, achieving a speedup of up to ~8 times.

Looking ahead, possible improvements to our algorithm would include reducing memory usage of the algorithm and implementing a dynamic *cutoff* based on the problem size and number of cores, which would improve both performance and scalability.

6 Appendix

Problem Size	**250**	**500**	**750**	**1000**	**1250**	**1500**
	31.971	126.243	283.979	503.582	784.844	1129.742
Problem Size	**1750**	**2000**	**2250**	**2500**		
	1535.036	2001.521	2532.519	3127.185		

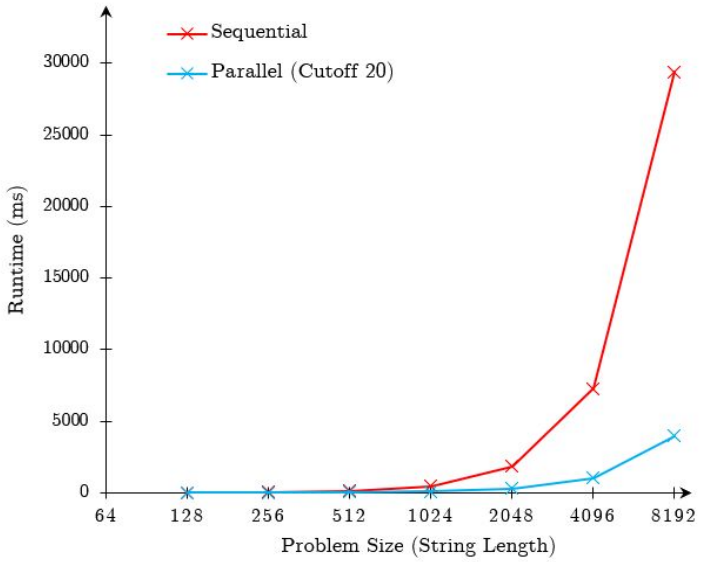
Time Taken against Problem Size [Section 4.1]

Problem Size/ Cutoff Size	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
1	61.383	200.078	777.256	2911.824	11385.031	47166.848	191805.718
2	14.404	54.082	209.206	788.412	3038.293	12241.078	49542.562
4	5.127	17.511	63.913	228.487	843.321	3345.323	13419.379
8	3.168	9.285	30.052	94.369	324.259	1230.248	4843.127
16	3.760	9.126	26.834	82.792	277.031	1036.524	4002.074
32	4.409	13.614	31.119	88.972	288.034	1033.673	3953.268
64	6.453	16.767	50.627	116.879	331.152	1103.182	4026.588
128	8.207	25.790	62.515	155.715	420.995	1245.134	4322.804

Runtime for Cutoff Size against Problem Size (highlighted is the fastest time) [Section 4.2]

Problem Size	128	256	512	1024	2048	4096	8192	16384
Sequential	7.503	29.387	114.137	455.282	1822.329	7287.230	29388.274	114028.231
Parallel (Cutoff 20)	3.536	9.602	27.524	83.798	278.242	1030.479	3967.664	14880.021
Speedup	2.122	3.061	4.147	5.433	6.549	7.072	7.407	7.663

Runtime against Problem Size [Section 4.3]



Runtime against Problem Size [Section 4.3]

No. of Cores	1	2	4	8	16	24
Fixed Problem Size (8192)	37470.037	19857.076	10634.360	6484.575	4544.034	4096.848
Fixed Parallel Time (100)	90.050 (100.386)	153.126 (100.022)	257.243 (100.698)	367.954 (100.148)	499.440 (100.716)	554.794 (100.230)
Fixed Parallel Time (4000)	3388.681 (4000.852)	6271.137 (4001.203)	11741.988 (4003.245)	19667.377 (4002.240)	27494.784 (3997.125)	30258.025 (3998.895)

Gustafson and Amdahl's Speedup [Section 4.4]