PARALLELIZING MYERS' ALGORITHM FOR THE LONGEST COMMON SUBSEQUENCE PROBLEM

Josua Cantieni, Lowis Engel, Pascal Maillard, Philippe Voinov

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The longest common subsequence is the longest possible subsequence of symbols that two input sequences have in common. Its most common applications are in finding the smallest number of differences in text files (usually source code) or aligning common regions of DNA strands.

In this report, we present our approach to parallelizing Myers' sequential algorithm. We implemented two different versions: statically distributing each DP table row across MPI workers and a priority approach that avoids idle waiting by dynamically prioritizing entries needed by other workers. We applied vectorization to further improve the performance.

We analyze the performance using both synthetic randomly generated input sequences and real DNA (nucleic acid) sequences. Overall, our row-wise implementation scales nearly linearly with the amount of workers and does not suffer from any additional overhead, as long as that the inputs are sufficiently large.

1. INTRODUCTION

Motivation. The longest common subsequence (LCS) problem is the problem of finding the longest (not necessarily contiguous) subsequence that is shared by two input strings. This problem occurs very widely in practice. For example, the two sequences $ABCD$ and $ACBAD$ have the longest common subsequences ACD and ABD. The length of the LCS is unique, although multiple subsequences with that length might exist.

LCS algorithms are at the core of comparison tools like "diff", which are ubiquitous when working with source code. These comparisons are also used extensively in version control systems. Outside of the software domain, the longest common subsequence problem is extremely important in biology - LCS is the exact solution to DNA nucleotide *sequence alignment*. However, it is important to note that heuristic methods (BLAST [1]) are often used today, since they allow fast search of very large databases of genetic sequences.

There exist various specialized LCS algorithms which achieve faster runtime or lower memory usage than general algorithms. One class of specialized algorithms provides improved runtime when both input sequences have significant similarities (edit distance not too large). This is generally the case when comparing two versions of a text document. For this reason algorithms of this class are nearly universally used by tools similar to "diff". However such algorithms are generally sequential.

We adapt the sequential LCS algorithm of Myers [2], which is by far the most commonly used LCS algorithm for similar sequences, for parallel execution on a distributed memory system.

Related work. Bergroth et al. prepared a survey [3] of various sequential LCS algorithms. They group the algorithms into three categories: "row-by-row methods", "contour methods" and "diagonal methods". The classic dynamic programming algorithm for LCS [4] is classified as a row-by-row method. Myers' algorithm [2] belongs to the class of diagonal methods. Diagonal methods are designed to be efficient when both input sequences are similar.

Most parallel algorithms for the LCS problem are based on row-by-row sequential algorithms. Lu and Lin [5] presented a parallel algorithm of this class, achieving optimal complexity with a shared-memory machine model. Yang et al. [6] designed a parallel LCS algorithm for GPUs (graphics processing units) by restructuring the classic DP (dynamic programming) table to reduce data dependencies. These algorithms are fundamentally based on row-by-row methods, which are less efficient than diagonal methods for similar input sequences.

Allison and Dix [7] present a "bit-parallel" LCS algorithm (row-by-row method). Assuming a sufficiently small alphabet, their algorithm allows calculating multiple elements of the DP table in parallel, for example by using vector instructions. Their bit-parallel technique can be applied to other algorithms. We do not incorporate it into our algorithm, but do use vectorization within each DP cell.

Since Myers' algorithm [2] is an extremely popular LCS algorithm, we attempted to parallelize it. As far as we are

aware, there are no published parallel versions of an LCS algorithm from the "diagonal methods" class, which Myers' algorithm belongs to.

2. BACKGROUND

We briefly explain the concept of edit distance and its relation to the longest common subsequence. We then present the sequential algorithm from Myers, which was the basis for our parallel implementation.

Edit distance. The edit distance refers to the minimum number of changes required to transform one string into another. In the case where we only consider insertions and deletions, and exclude substitutions, the LCS problem is equivalent to finding the edit distance. All symbols that do not appear in the LCS count towards the edit distance.

To transform the first string into the second, we perform deletions for all symbols in the first sequence that are not part of the LCS and similarly insertions for the second sequence. Those changes are referred to as an edit script.

Myers' algorithm. Myers [2] takes a greedy approach to finding the solution in the edit graph of two sequences. It has an asymptotic runtime of $O(nd)$ where n is the total length of the two sequences and d is the size of the minimum edit script. This algorithm performs well when the sequences are similar and have a small d.

For constructing the edit graph, shown in figure 1, the two sequences A and B are placed along a grid. The solution for the shortest edit script can be constructed by finding the shortest path from $(0, 0)$ to the bottom right corner. A horizontal or vertical segment represents an edit. A diagonal can only be taken if the two input sequences are identical.

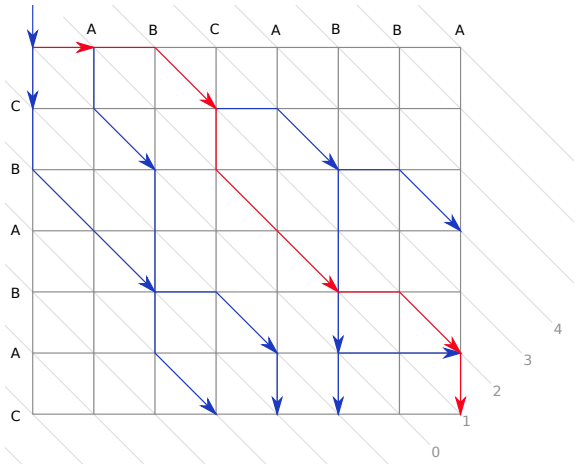

Fig. 1. Edit graph for comparison of two sequences [8]. The solution for the shortest edit script is shown in red.

Myers' algorithm, shown in algorithm 1, goes through the diagonals shown in figure 1 with an increasing edit distance and stores the coordinates of the furthest reachable point on each diagonal. The variables N and M represent the lengths of the sequences A and B.

Algorithm 1: Myers' LCS algorithm [2]

```
Constant MAX ∈ [0, M + N]
V : Integer[−MAX .. MAX]

V[1] ← 0
for d ← 0 to MAX do
    for k ← −d to d in steps of 2 do
        if k = −d or k ≠ d and V[k − 1] < V[k + 1] then
            x ← V[k + 1]
        else
            x ← V[k − 1] + 1
        y ← x − k
        while x < N and y < M and a_{x+1} = b_{y+1} do
            (x, y) ← (x + 1, y + 1)
        V[k] ← x
        if x ≥ N and y ≥ M then
            Length of shortest edit script is d
            STOP
```

To compute an entry on the diagonal k and distance d, the previous paths can connect to it either by a horizontal or vertical move from its neighbors on the diagonals $k \pm 1$. Out of those two entries from the table for a distance of $d - 1$, it takes the one with the higher x-value and follows the diagonal as long as the two sequences are identical. The new coordinates are then stored in the DP table.

The solution has been found once an entry reaches the coordinates (N, M). The edit distance d for that entry directly tells us the minimum edit distance. But the actual edit script has to be computed recursively.

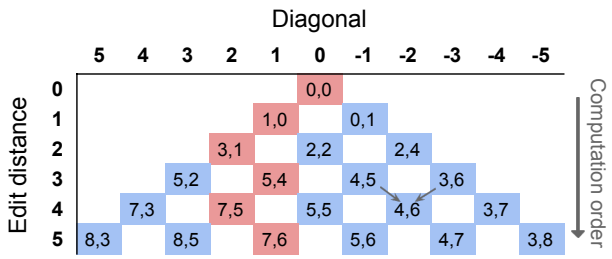

Fig. 2. The DP table represents the furthest points on each diagonal that can be reached for a given edit distance. The arrows denote the dependencies of a single entry. The solution is marked in red and has an edit distance of 5.

Every entry depends on at most two entries from the previous row in the DP table (marked as arrows in figure 2). If we only want the LCS length, it is not required to store the entire table, but only the previous row for the distance $d-1$. Thus the memory consumption is linear and no longer the limiting factor for very large input sequences. Myers presents an approach ("linear refinement") to compute the solution recursively [2]. However, we will not go further into detail because it lies outside the scope of this report.

3. METHOD

We present our two different approaches to parallelize the algorithm and their major optimizations. Our implementations use MPI ("Message Passing Interface"). We also discuss a vectorization optimization that can be applied to both versions. Our source code and test cases are publicly available [9].

Row-wise algorithm. The design of our algorithm is intentionally simple. We divide each row of our pyramid-shaped DP table into evenly spaced contiguous segments and assign exactly one to each worker. As the number of cells in each row is increased by one we increase the size of a segment by one. To ensure a balanced workload we increase each segment size in a round robin fashion.

Each worker calculates the cells in its segment from left to right. But in order to calculate the leftmost element the worker may depend on the rightmost element of the previous row from the left neighboring worker. Thus a worker has to wait for the neighboring worker to complete its segment in the previous row before the worker can start the next row. This can lead to an imbalance. In order to overcome this issue we prioritize the leftmost and rightmost elements and calculate them first and send them right away before calculating any other elements. This ensures that the waiting time is minimal.

The number of messages sent in each row is constant in the number of workers (up to two receives and one send per worker) independent of the size of the rows. This means that with a small work size per worker we get a significant communication overhead. We solve this by first increasing the segment of a worker until it reaches a certain threshold before assigning any work to the next worker. In experiments we found that a threshold of 200 cells works well.

We can calculate the edit script in a similar fashion as a sequential algorithm would. But due to the quadratic memory consumption we did not benchmark this.

Dynamic priority algorithm. In our row-wise algorithm a worker will not calculate any elements of the next row until it receives all required elements of the previous row. This blocking wait is usually not necessary - it is possible to calculate some elements of the next rows without waiting for other workers. We designed the dynamic priority algorithm to always perform some calculation instead of a blocking wait if possible. Additionally, the algorithm prioritizes calculations that will be sent to other workers, so that they are quickly unblocked.

In order to perform calculations whenever possible, it is necessary to determine which calculations are possible in a given state. Figure 3 shows the internal state that our algorithm keeps for this purpose. Every cell on the solid jagged line (*frontier*) has been calculated by some worker, and every cell on or below the dotted line (*limiting line*) can not be calculated at the moment. This means that all cells below the frontier and above the limiting line can be calculated without receiving any further messages from other workers. The frontier and the limiting lines fully determine the currently possible calculations (darker red).

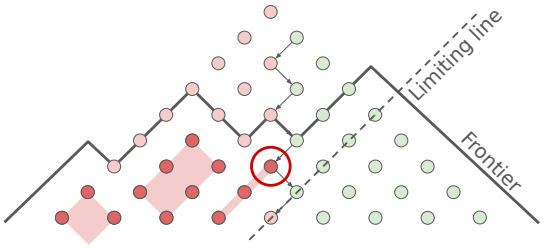

Fig. 3. Visualization of the internal state of the dynamic priority algorithm. The circles are cells in the DP table (colored by worker). There are two workers (red and green). The state is from the perspective of the red worker. The arrows represent (past or future) sends and receives. Every cell on or above the *frontier* is known to have been calculated by some worker. Every cell on or below the *limiting line* cannot be calculated by the red worker at the moment.

The shape of the frontier is a direct consequence of the dependencies in our DP table. Before a DP cell can be calculated, the cells above it (left and right) must be calculated (see arrows in figure 2). Because of transitivity, this means that all DP cells in the "upside down pyramid" ending in a cell must have been calculated before it. The frontier is the boundary of a union of such pyramids. Whenever a worker learns that a cell was calculated (because it calculated the value itself or received it from another worker) it extends its frontier by union with the pyramid ending in that cell.

Every worker has up to 2 limiting lines at any given time. These diagonal lines go through the next cell that the worker expects to receive. The worker shouldn't calculate any cells on or below a limiting line, since these cells will either be calculated by another worker (e.g. the cell that it expects to receive) or depend on values in such cells.

Once it is clear which cells are possible to calculate, it is necessary to choose a calculation order. One interesting case is outlined with a red circle in figure 3. This cell has not been calculated yet, and another worker expects to receive its value. To prevent other workers from being blocked, such cells (and their dependencies) will be calculated before cells that do not result in a send. If there are no such cells to prioritize, cells near the middle of the DP pyramid are calculated first, since this order results in a frontier with few bends, which makes planning calculations faster.

SIMD Optimization. The amount of work needed per DP cell dynamically depends on the input, since the number of loop iterations in the while-loop is determined by the number of equal elements after a change. Therefore,

the workload per worker might be imbalanced. To mitigate these imbalances and hence, reduce the time workers are waiting, we use x86's SIMD[1] vector extensions to compare multiple elements at once.

In many cases we need to compare only one value, because the first value is already unequal. In these cases we would not benefit from using SIMD instructions, they might even be more expensive. Therefore, we have a preliminary scalar check for inequality of the first element. Only if this check fails, we will use SIMD instructions.

For the vectorized comparison we first load unaligned 8-way or 4-way vectors from both input sequences (the loads need to be unaligned, since we do not know the position in the vector in advance and we compare the input sequences at different offsets). Then we use a `cmpeq` intrinsic to compare the two vectors for equality. The result is then checked for an unequal element by using the negation of the `testc` intrinsic. If there is no inequality, the next vector will be checked. Otherwise, the exact location of the inequality is determined by a scalar loop over the last up to eight elements using the same loop as in the original algorithm.

4. EXPERIMENTAL RESULTS

In this section we first describe the experimental setup for our benchmarks, including the hardware and software used, the test cases and how we performed our measurements. Then we present our results where we compare to an existing implementation and evaluate the scaling of our implementations.

Hardware & software setup. To evaluate our code with an increasing number of processing units, we ran our benchmarks on ETH's Euler cluster[2]. We used their 6th generation compute nodes with two 64-core AMD EPYC 7742 processors, which belongs to AMD's Zen 2 generation. The nodes are interconnected via a dedicated 100 Gb/s InfiniBand HDR network.

We compiled our C++ code locally using GCC 9.3.0 and Open MPI 4.0.3 with the flags `-std=c++17 -O3 -ffast-math -march=znver2`. For comparison with an existing implementation we use GNU diff utilites (diffutils) [11]. To have a fair comparison and to inject timers we recompiled it using their configure script with `CFLAGS="-O3 -march=znver2 -ffast-math"` and then ran `make`. The diffutils binary is always called with the argument `--minimal` to avoid heuristics that do not necessarily produce the smallest edit distance. Our algorithms include SIMD features which can be enabled by preprocessor macros. Unless stated otherwise, these are enabled.

[1]"Single instruction stream, multiple data streams" by Flynn's Taxonomy [10]

[2]For more information see `https://scicomp.ethz.ch/wiki/Euler`

On Euler we load the modules `openmpi/4.0.2` and `python/3.6.4`. We then use separate jobs for different processor counts and submit them to the batch system. For the job submission we use the resource requirement flag `-R 'select[model==EPYC_7742]'` to get the Euler VI nodes, `-R 'rusage[mem=512]'` to specify our memory usage of at most 512 MB per core and `-R 'span[ptile=128]'` to ensure that we always use full nodes. For exclusive use of the nodes we request more cores if necessary to reach a multiple of 128.

Test cases. For testing and benchmarking we built a script to generate random test cases. Each test case consists of two input files containing one integer per line, representing the two input sequences to compute the longest common subsequence length for. For our benchmarks we generate two completely independent files (of equal length) using a Zipf distribution [12] where the probability for an integer value $v \in [0, \text{sequence length})$ is proportional to $\frac{1}{v+1}$. We chose this distribution instead of an uniform distribution, because it models the fact that some tokens might be more frequent than others in texts written by humans.

We also used DNA nucleotide sequences for benchmarking, to check that our algorithms perform similarly as with synthetic inputs. We used two pairs of DNA which we obtained from GenBank [13], a shorter pair (streptomyces aureoverticillatus, ~171'000 base pairs, edit distance ~99'000) [14, 15] and a longer pair (synechococcus elongatus, ~2'70 0'000 base pairs, edit distance ~1'800'000) [16, 17].

Benchmarking methodology. The programs are called repeatedly on each test case from a Python script. The measurements are repeated until the confidence interval, which contains the true *median* with a probability of 95%, has a relative error of at most 5% around the current median estimate for the runtime, or a maximum of 50 repetitions is reached.

Each execution of a test case was limited to 120 seconds. If a combination of algorithm, parallelism and input size reached this time limit in any repetition, all repetitions for this combination were excluded from our results. This was done to avoid testing large inputs with low parallelism, which takes prohibitively long.

The runtime is measured using the `high_resolution_clock` from the `chrono.h` header. We measure this wall-clock time only on the dedicated MPI process which reads the input and prints the output. To measure time in diffutils we inserted calls to the `gettimeofday` function from `time.h` in `analyze.c`. In our benchmarks we focus on the computation time of the core algorithms, excluding the time needed for reading the input and writing output and also any precomputation done in diffutils. The runtimes are printed in microseconds.

Results. Each marker in all following plots represents the calculation time for a single repetition. We calculate

the median value across repetitions and connect these using linear interpolation to show trends. Unfortunately, a few data points (e.g. in the large DNA benchmark) are missing, since we had too low priority on the Euler cluster to measure them. We expect they would follow the same trend.

We first investigate the performance of our parallel algorithms when limited to a single process. Figure 4 shows the calculation time of our algorithms and diffutils. Our purely sequential baseline is faster than diffutils. This is expected: our comparison is not fair, since diffutils is calculating (but not outputting within the measured time) an edit script, whereas our algorithm only calculates the edit distance. Diffutils uses the recursive "linear space refinement" algorithm from [2] to calculate the edit script, which is "roughly twice as slow as the basic $O(nd)$ algorithm" [2] used in our baseline. Diffutils is not twice as slow as our sequential baseline - this is likely because diffutils has been well optimized over time. The MPI row-wise algorithm is faster than our sequential baseline. It is not clear why this is the case, since the core algorithm and compiler settings are identical, but the complete code for the MPI row-wise algorithm is more complex. The MPI dynamic priority algorithm seems clearly faster than all the others. This is due to the order in which it calculates the cells of the DP table. The algorithm prioritizes cells in the middle of the DP table (see 3). This makes MPI dynamic priority faster, unless there are very different amounts of additions vs deletions, in which case it would be slower than the other algorithms.

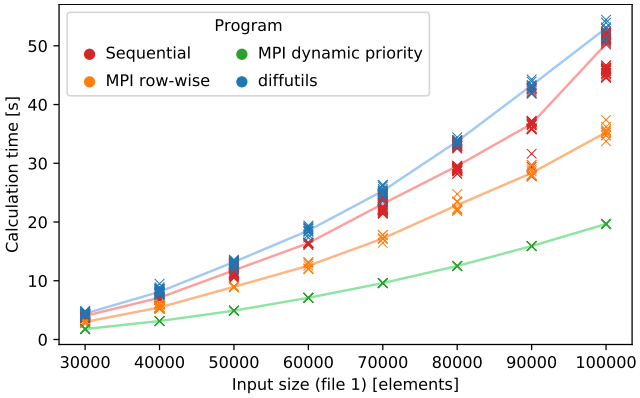

Fig. 4. Calculation time of all algorithms when limited to a single process.

To understand how our parallel algorithms scale, we ran them using various numbers of compute nodes. Figure 5 shows the calculation times from these experiments for our MPI row-wise algorithm. Using more nodes leads to a significant performance improvement. The curves for runtime seem to have approximately the ideal shape (proportional to $1/\text{nodes}$). However it is not clear from this plot exactly how much less efficient the computation becomes as more nodes are used - that is investigated in a later plot. The runtime with DNA inputs shows the same trend as with synthetic inputs. For similar input sizes, the calculation time for DNA inputs is lower than for synthetic data, since they have a smaller edit distance.

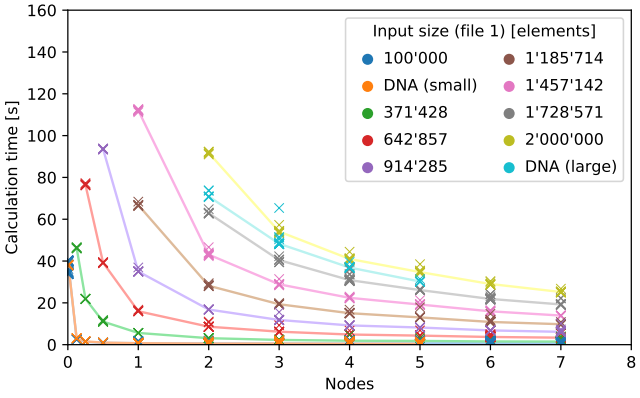

Fig. 5. Calculation time of the MPI row-wise algorithm with varying parallelism and input sizes.

Figure 6 shows the results of an identical scaling experiment for our MPI dynamic priority algorithm. With one process the MPI dynamic priority algorithm is faster than MPI row-wise (as in figure 4). With half a node both algorithms have approximately the same speed. However the MPI dynamic priority algorithm becomes slower than the MPI row-wise algorithm once more nodes are used. This is surprising since the dynamic priority algorithm was designed to avoid blocking waits. We were not able to identify the cause of this bad scaling (which is difficult, since the behavior of the algorithm is very dynamic). Note that even with this inefficiency, using more nodes (for the number of nodes we tested) still reduces the total execution time.

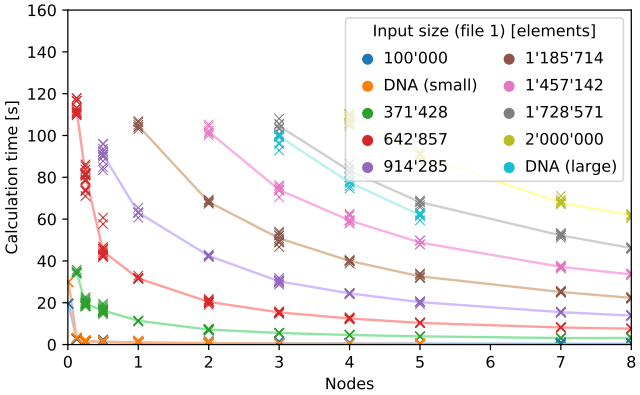

Fig. 6. Calculation time of the MPI dynamic priority algorithm with varying parallelism and input sizes.

To understand how efficiently our algorithms scale, we investigated the execution rate with a varying number of

nodes (figure 7). We calculate rate as $\frac{n \cdot d}{t}$ which is the calculation time t in seconds divided by the asymptotic runtime. For a fixed input (each curve in the plot), the rate is proportional to the speedup [18] (which we do not calculate, since we do not have sequential measurements for large inputs). The dotted reference line approximates the rate that would be achieved with ideal linear speedup. This line is calculated based on the single run which achieves the highest rate with 1 node (for any input size). For the smallest input size (30'000 elements), the rate does not increase with more nodes. The overhead (e.g. MPI communication) likely dominates, since the total execution time (figure 5) is very low. As inputs become larger, the row-wise algorithm becomes efficient. With the largest input sizes and number of nodes we tested, it scales nearly ideally.

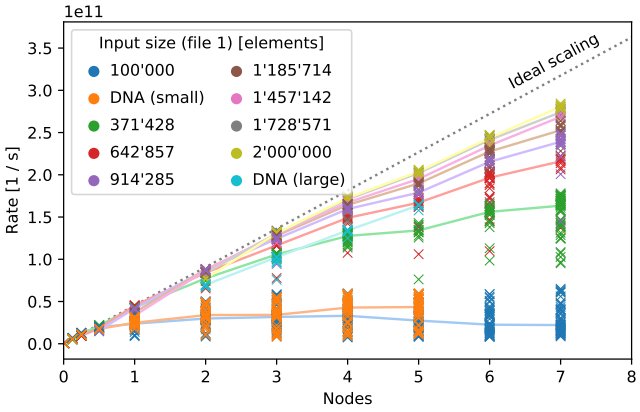

Fig. 7. Rate of calculation (inverse runtime scaled by amount of work) of the MPI row-wise algorithm with varying parallelism and input sizes

Figure 8 shows results in an identical format, but for our MPI dynamic priority algorithm. The MPI dynamic priority algorithm shows significantly worse than ideal scaling. Although the plotted curves for rate vs input size are roughly straight lines, the rate is not proportional to the number of nodes, as it would be for ideal scaling.

In all previously presented results, our algorithms had their SIMD features enabled (see "Hardware & software setup"). To investigate how this affects our algorithms, we repeated all the benchmarks from figures 5 and 6 with the SIMD features disabled. Figure 9 shows the speedup from SIMD across those benchmarks. The speedup for each plotted point is calculated from the median runtime of each combination of algorithm, node count and input size. For the MPI row-wise algorithm, SIMD typically gives a speedup of 10% - 30%. Surprisingly enabling (the same) SIMD features in the MPI dynamic priority algorithm makes it run around 10% slower. It is not clear what causes this slowdown. It may be necessary to investigate this together with the general scaling issues of the dynamic priority algorithm.

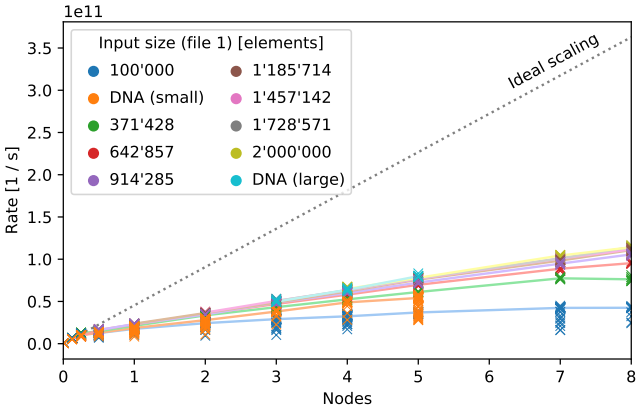

Fig. 8. Rate of calculation (inverse runtime scaled by amount of work) of the MPI dynamic priority algorithm with varying parallelism and input sizes

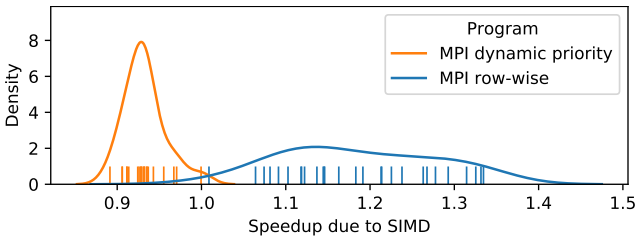

Fig. 9. Speedup (non-SIMD runtime / SIMD runtime) across all input sizes and node counts

5. CONCLUSION

In this report we presented a way to parallelize Myers' longest common subsequence algorithm and applied various optimizations. Our results showed that that our simple row-wise algorithm scales well to many processes. Our algorithms hold up against diffutils when run sequentially. Against our expectations, our dynamic priority algorithm does not scale well.

Future work. In our work on this algorithm we considered various ideas to increase performance or reduce memory consumption, which we unfortunately did not have the time to experiment with for our report.

As such, we currently do not read out the edit script due to the quadratic memory consumption that it requires with this algorithm. If our algorithm is adapted to use Myers' recursive approach for linear space refinement [2] (which calls the non-recursive algorithm), it could output the edit script with only a minor performance cost.

It is possible that using SIMD to calculate multiple cells in parallel would improve the performance further.

6. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990.

[2] Eugene W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, Nov. 1986.

[3] 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*, Sept. 2000, pp. 39–48.

[4] Robert A. Wagner and Michael J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974.

[5] Mi Lu and Hua Lin, "Parallel algorithms for the longest common subsequence problem," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 835–848, Aug. 1994, Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[6] Yang Jiaoyun, Xu Yun, and Shang Yi, "An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs," *Lecture Notes in Engineering and Computer Science*, vol. 1, June 2010.

[7] Lloyd Allison and Trevor I. Dix, "A bit-string longest-common-subsequence algorithm," *Information Processing Letters*, vol. 23, no. 5, pp. 305–310, Nov. 1986.

[8] "Image of the algorithm's edit graph," `http://simplygenius.net/Article/DiffTutorial11`, (*last accessed: 03.01.2021*).

[9] "A parallelized version of Myers' O(nd) diff algorithm," `https://github.com/tehwalris/mpi-myers-diff`.

[10] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[11] GNU, "diff utilities," `http://savannah.gnu.org/git/?group=diffutils`, commit: 7c13bbbeca620e573827636311c5e76e3e3e8da6, 15.03.2020.

[12] Encyclopedia of Mathematics, "Zipf law," URL: `http://encyclopediaofmath.org/index.php?title=Zipf_law&oldid=50751`, (*last accessed: 09.01.2021*).

[13] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank," *Nucleic Acids Res*, vol. 45, no. D1, pp. D37–D42, 01 2017.

[14] Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information, `https://www.ncbi.nlm.nih.gov/nuccore/NZ_BMSY01000016.1` [cited 2021-01-14], *Streptomyces aureoverticillatus strain JCM 4347 sequence016, whole genome shotgun sequence*, Nov 2020, Accession No. NZ_BMSY01000016.1.

[15] Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information, `https://www.ncbi.nlm.nih.gov/nuccore/NZ_BMSY01000017.1` [cited 2021-01-14], *Streptomyces aureoverticillatus strain JCM 4347 sequence017, whole genome shotgun sequence*, Dec 2020, Accession No. NZ_BMSY01000017.1.

[16] Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information, `https://www.ncbi.nlm.nih.gov/nuccore/NC_007604.1` [cited 2021-01-14], *Synechococcus elongatus PCC 7942 = FACHB-805, complete sequence*, Dec 2020, Accession No. NC_007604.1.

[17] Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information, `https://www.ncbi.nlm.nih.gov/nuccore/NC_006576.1` [cited 2021-01-14], *Synechococcus elongatus PCC 6301, complete sequence*, Dec 2020, Accession No. NC_006576.1.

[18] Kenneth Moreland and Ron Oldfield, "Formal metrics for large-scale parallel performance," in *International Conference on High Performance Computing*. Springer, 2015, pp. 488–496.