Parallel Smith-Waterman with OpenMP

ISITHA SUBASINGHE, University of Melbourne, Australia

Sequence alignment is the process of comparing and detecting similarities in biological sequences such as RNA and DNA. The idiomatic algorithm presented for this problem is inherently sequential. This report explores how the problem may be parallelised using the "wavefront" technique along with OpenMP [1] . The algorithm implemented using OpenMP was able to offer a speedup of 13 over the sequential algorithm, interestingly, the effeciency of adding more cores to the problem was shown to decrease exponentially.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**.

1 BACKGROUND

The Smith-Waterman [10] algorithm introduced about 40 years ago, is the only known algorithm that is able to find the optimal local alignment of sequences, however, it is also one of the slowest algorithms for sequence alignment. With the exponential growth of biological databases, far outpacing microprocessor improvements, the use of parallelism to speed up the Smith-Waterman algorithm is of importance to bioinformatics. The algorithm itself is not trivially parallelisable, however, recognising the dependencies for the values in the dynamic programming matrix helps in observing that the algorithm is still however, parallelisable. The dependencies of the values in the dynamic programming matrix are shown in fig. 1.

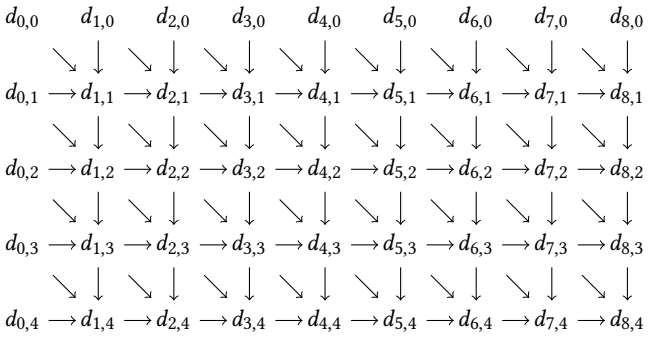

Fig. 1. Dependencies for values in Smith-Waterman matrix

2 RELATED WORK

Searching of DNA and RNA sequences is a fundamental task in bioinformatics and as mentioned previously Smith-Waterman offers optimal local alignment. However, as a result of optimal local alignment being computationally expensive, the most common solutions do not aim to provide this, instead they use heuristics to reduce the computational cost, examples of such tools are FASTA [8] and BLAST [1].

[1]See https://www.openmp.org

Early work in parallelising Smith-Waterman was based upon diagonal parallelisation, this was initially done through SIMD instructions.SIMD implementations have proven to be popular as a result of their common availability in x86 systems and have demonstrated significant speedups in several published literature [9, 12].

With the rapid accessibility of GPUs over the years following the early work on SIMD parallelism, CUDA compatible GPUs were found to be effective as hardware accelerators for the local optimal alignment problem. The CUDA implementation shown in [5] was demonstrated to be 2 to 30 times faster than other implementations on commodity hardware at the time.

Field Programmable Gate Arrays are integrated circuits that can be programmed through the user, using some hardware description language such as VHDL or Verilog. These implementations are naturally faster than their counterparts in higher level languages as the circuit itself is programmed for the specific problem, however, they are difficult to program and debug. Storaasli [11] claims that their implementation of Smith-Waterman on two Cray XD1 systems was able to obtain a 100x speedup. Interestingly, speedups of 160x have been made through the use of FPGAs as shown in [4].

3 PROPOSED APPROACH

3.1 Sequential Algorithm

The sequential algorithm used was near identical to the provided program, however certain optimisations were performed to the program, along with bug fixes. The code for this algorithm is shown below:

Algorithm 1 Sequential Smith-Waterman

```
1:  procedure SeqSmithWaterman
2:      m,n ← dimensions of matrix
3:      pgap ← gap penalty
4:      pxy ← mismatch penalty
5:      dp[m][n] ← newMatrix(m+1,n+1)
6:      x ← sequence x
7:      y ← sequence y
8:      for i ← [0 . . . m] do
9:          dp[i][0] ← i ∗ pgap
10:     for i ← [0 . . . n] do
11:         dp[0][i] ← i ∗ pgap
12:     for i ← [1 . . . m] do
13:         for j ← [1 . . . n] do
14:             if x[i-1] = y[j-1] then
15:                 dp[i][j] = dp[i-1][j-1]
16:             else
17:                 dp[i][j] = min(dp[j-1][i-1] + pxy, dp[j-1][i] + pgap, dp[j][i-1] + pgap)
        return traceback(dp)
```

3.2 Parallel Algorithm

The psuedocode for the parallel algorithm is defined as follows:

Algorithm 2 Parallel Smith-Waterman

```
1:  procedure ParSmithWaterman
2:      m,n ← dimensions of matrix
3:      pgap ← gap penalty
4:      pxy ← mismatch penalty
5:      δ_i ← tile width
6:      δ_j ← tile height
7:      outer_i ← 0
8:      outer_j ← 0
9:      diagonals ← ⌊m/δ_i⌋ + ⌊n/δ_j⌋ + { 1  m%δ_i + n%δ_j > 0
                                        { 0  m%δ_i + n%δ_j = 0
10:     dp[m][n] ← NewMatrix(m+1,n+1)
11:     x ← sequence x
12:     y ← sequence y
13:     for i ← [0 ... m] parallel do
14:         dp[i][0] ← i ∗ pgap
15:     for i ← [0 ... n] parallel do
16:         dp[0][i] ← i ∗ pgap
17:     for i ← [0 ... diagonals) do
18:         diff_i ← outer_i
19:         diff_j ← n − outer_j − 1
20:         diag_i ← 1 + ⌊diff_i/δ_i⌋
21:         diag_j ← 1 + ⌊diff_j/δ_j⌋
22:         length ← min(diag_i, diag_j)
23:         for tile ← [0 ... length) parallel do
24:             inner_i ← max(1, outer_i − (tile ∗ δ_i))
25:             inner_j ← max(1, outer_j + (tile ∗ δ_j))
26:             imax ← max(inner_i + δ_i, width)
27:             jmax ← max(inner_j + δ_j, height)
28:             for i ← [inner_i ... imax] do
29:                 for j ← [inner_j ... jmax] do
30:                     if x[i-1] = y[j-1] then
31:                         dp[i][j] = dp[i-1][j-1]
32:                     else
33:                         dp[i][j] = min(dp[j-1][i-1] + pxy, dp[j-1][i] + pgap, dp[j][i-1] + pgap)
34:         if outer_i + δ_i < m then
35:             outer_i ← outer_i + δ_i
36:         else
37:             outer_j ← outer_j + δ_j
        return traceback(dp)
```

3.2.1 Restrictions. There was a restriction in the implementation of our algorithm, namely the amount of memory available proved constraining in exploiting further opportunities for parrallelism. Computing the sequence alignment among all $\frac{k(k-1)}{2}$ pairs is "embarrassingly parallel", however, this leads to far greater memory consumption and therefore wasn't a valid optimisation given our memory restrictions.

3.2.2 NUMA awareness. NUMA (non-uniform memory access) becomes an area of concern when considering programs that use multiple cores. More accurately, when multi-core programs require access to the same memory locations as each other and this sharing is performed over an interconnect. The policy used to help reduce the cost of NUMA in the Parallel implementation is called "First Touch". "First Touch" ensures that the NUMA node to initialise some memory location, will have that page allocated on a memory device physically closer to the NUMA node.
The steps for memory allocation can be given as follows:

- dynamic memory allocation call (malloc/new)
- initial write attempt of dynamically allocated memory
- page fault triggered when write is attempted
- physical allocation of memory performed and mapped via MMU

The parallel implementation exploits this property on moderns operating systems to help reduce the cost of non-uniform memory access. This is performed by initialising the matrix used for Smith-Waterman through multiple threads, ensuring that each NUMA node has a slice of the memory and thus a lower number of memory transfers are done over the interconnect, this is shown in algorithm 3.

While this is helpful for distributing the pages among the NUMA nodes, it comes at an additional processing cost. With the GNU 11.1.0 compiler toolchain, we are able to make use of the OpenMP allocator to do this distribution of pages, additionally some further optimisations maybe performed to increase performance, such as removing thread safety from the allocator and ensuring that the allocated memory is aligned, aligning memory is useful when using SIMD as load operations are much faster.

3.2.3 Optimum tile size. The tile size presents a significant influence on the speedup of our algorithm, along with the scheduling algorithm used. The intuition regarding the dimensions of the tile size maybe to minimise it as much as possible, however, this is not always the optimial strategy. Likewise, it may seem that static scheduling maybe the most sensible given that each core would perform relatively the same amount of work, however, this is also untrue. A deeper discussion regarding the scheduling of wavefront parallelism on shared memory multiprocessors is provided in [6]. In our implementation, the tile size was determined by dividing the dimensions by a fixed number and the scheduling was performed dynamically, these details were determined through experimentation.

3.2.4 Cache optimisations. Memory latencies play a significant role in the performance of programs, ensuring false sharing is minimal and maintaining locality of reference are important objectives for any high performance application. The processing of tiles in the wavefront processing of the matrix, helps obtain the previously mentioned objectives as shown by [6]. This is as a result of the layout of memory used in the matrix itself, the entire matrix is allocated as a single contiguous slice of memory. Consider the parallel traversal of the yellow diagonals below in fig. 2. An isolated view of the potential cache lines for each core is also visible in fig. 3, observe how given sufficient width of the matrix, false sharing is extremely unlikely. Additionally, note how the tiles are traversed horizontally and then vertically, this maintains locality of reference as the rows are contiguous in memory which is crucial to high performant code.

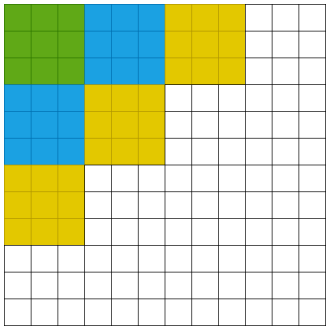

Fig. 2. Parallel processing of diagonals

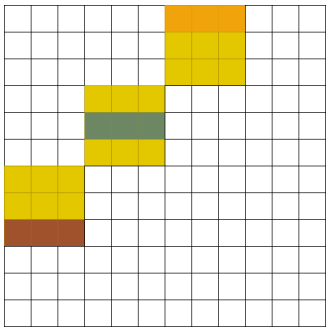

Fig. 3. Potential cache lines

3.2.5 Compiler optimisations. Compiler optimisations were heavily used to further increase the performance of the algorithm and frequent profiling of the program was done via the linux utility "perf" [2]. It is important to note that using "std::string" had a significant impact on performance when dynamically linked to our program, this is as a result of symbol lookup performed in a tight loop when traversing the dynamic programming matrix. The remedy was to statically link the C++ standard library, which lead to a significant increase in performance, almost exactly the same performance characteristics as using a simple character array for holding the sequences. Additional optimisation recommendations from [2] were evaluated and used where appropriate. A critical optimisation used was the "-march=native" flag passed onto the GNU compiler, this flag targets the exact architecture that we are compiling the program for, this results in further architecture level optimisations being possible such as usage of AVX-512 [3].

4 EXPERIMENTS

4.1 Experiment Environment

Two experimentation environments were used throughout the development process. The development environment was used as a quick way to test and filter out ideas that weren't viable, while valid ideas were tested on the Spartan High Performance Computing cloud [7].

[2]See https://perf.wiki.kernel.org/index.php/Main_Page
[3]See https://www.intel.com.au/content/www/au/en/architecture-and-technology/avx-512-overview.html

4.1.1 *Development Environment.*

- CPU - AMD Ryzen™ 7 3700X @ 3.6-4.4GHz
- Memory - 32GB
- Compiler - GNU 11.1.0

4.1.2 *Spartan Environment.*

- CPU - Intel(R) Xeon(R) Gold 6154 CPU @ 3.00-3.7GHz
- Memory - 32GB
- Compiler - GNU 10.3.0

4.2 Results

The results after running our parallel algorithm on the "mseq-big13-example" data provided are shown in fig. 4 and presented in below in section 4.2.

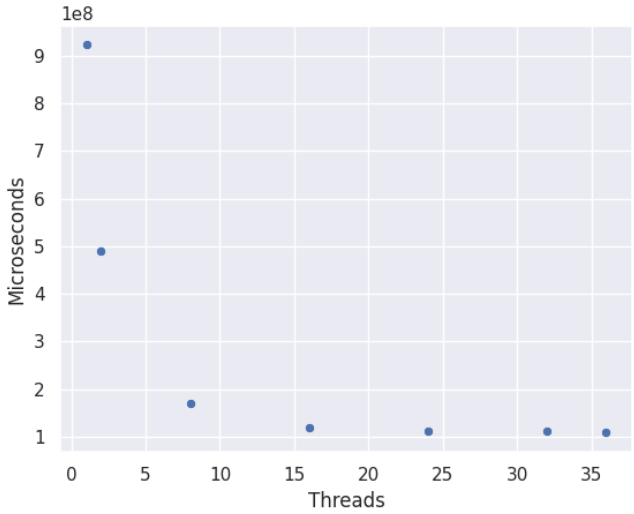

Fig. 4. Results observed

Core(s)	Walltime(microseconds)
1	923129129 μs
2	491227097 μs
8	169576852 μs
16	119517599 μs
24	112753964 μs
32	111237046 μs
36	108483498 μs

Table 1. Walltime per number of cores

4.3 Determining Efficiency

The efficiency($\eta(n)$) of our algorithm can be determined by the following formula:

$$\eta(n) = \frac{\text{Speedup(n)}}{\text{p(n)}} \tag{1}$$

The efficiency for the results shown in section 4.2 have been calculated using eq. (1) and displayed in section 4.3.

Core(s)	Efficiency
1	1.572
2	1.478
8	1.070
16	0.759
24	0.537
32	0.408
36	0.372

4.4 Exploring implications of tile size

Core(s)	Tile Size	Walltime(microseconds)
8	$(\frac{m}{8}+1, \frac{n}{8}+1)$	379606318 μs
16	$(\frac{m}{16}+1, \frac{n}{16}+1)$	382288384 μs
24	$(\frac{m}{24}+1, \frac{n}{24}+1)$	388430562 μs
32	$(\frac{m}{32}+1, \frac{n}{32}+1)$	385640454 μs
36	$(\frac{m}{36}+1, \frac{n}{36}+1)$	381233563 μs

Table 2. Walltime per number of cores and tile size

4.5 Discussion

Note: The sequential implementation was measured to have taken 1451965419 μs.

The results were extremely surprising, note that the single core parallel algorithm is far more performant than the sequential algorithm. While this was unexpected, this is likely due to the substantial amount of micro optimisations and time spent on the implementation of the parallel algorithm itself rather than the complexity properties of the parallel algorithm itself.

The choice of cores to test were not random, they were also made with the intention of determining the effectiveness of our NUMA mitigation strategy mentioned previously. The processor used consisted of 18 cores, this fact ensures that when the core count requested is above 18, we are guaranteed to encounter at least two NUMA nodes. Note that, in practice, this number was in fact much higher at 3-5 NUMA nodes per experiment. The results in section 4.2 show that our NUMA mitigation strategy was effective, although NUMA certainly would have contributed in adding latencies to our calculations as our core count went up, the increase in parallelism was able to counteract this. The decrease in efficiency as core count increased, demonstrated in section 4.3 is likely a function of NUMA costs and the limit on the parallelisability of the problem itself.

As stated previously, the tile size has a significant impact on the parallelisability of the problem. As shown in section 4.4, choosing a bad tile size may mean the parallelisability of the problem is artificially limited. The results in section 4.2, fig. 4 and section 4.3 are a result of dynamically choosing the tile size based on the dimensions of the matrix. To be exact, the dimensions of the matrix were divided by 13, determining this number was through experimentation on the development environment.

5 FURTHER WORK

5.1 Generation of tiles

The parallel algorithm implemented does not make full use of the available computing power. When tiles are iterated over, they must wait until all the tiles in the same diagonal have completed their work. This is an unnecessary restriction, as it results in a large amount of time spent on the barrier. Further work could explore how to reduce the level of granularity in our dependencies, new tiles could simply be generated after a single tiles work unit has completed. A simple algorithm would be to start processing tile $(i + \delta_i, j)$ and $(i, j + \delta_j)$ when (i, j) has completed its unit of work, this does however introduce the problem of tiles being visited more than once but this could simply be avoided through using a xorfilter to determine if a pair of indices have been visited before.

5.2 Inter-sequence SIMD parallelism

We could apply inter-sequence SIMD parallelism into our current thread-parallel implementation to obtain instruction level parallelism. While this is plausible, an important consideration that warrants investigation is determining an efficient method to move data from the heap and into the SIMD register, this technique will likely have significant speedups with AVX-512.

6 CONCLUSION

We have demonstrated how OpenMP may be used to develop a parallel algorithm for optimal local alignment of sequences. The algorithm developed showed effective use of parallelism to achieve a significant speedup of 13 over the sequential algorithm.

REFERENCES

[1] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research* 25, 17 (1997), 3389–3402.

[2] Agner Fog. 2020. Optimizing software in C++.

[3] Christoph Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013), 59–54.

[4] Isaac TS Li, Warren Shum, and Kevin Truong. 2007. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC bioinformatics* 8, 1 (2007), 1–7.

[5] Svetlin A Manavski and Giorgio Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics* 9, 2 (2008), 1–9.

[6] Naraig Manjikian and Tarek S Abdelrahman. 1996. Scheduling of wavefront parallelism on scalable shared-memory multiprocessors. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, Vol. 3. IEEE, 122–131.

[7] BERNARD MEADE, LEV LAFAYETTE, Greg Sauter, and DANIEL TOSELLO. 2017. Spartan HPC-Cloud Hybrid: Delivering Performance and Flexibility. https://doi.org/10.4225/49/58ead90dceaaa

[8] William R Pearson and David J Lipman. 1988. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences* 85, 8 (1988), 2444–2448.

[9] Torbjørn Rognes and Erling Seeberg. 2000. Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 16, 8 (2000), 699–706.

[10] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[11] Olaf O Storaasli and Dave Strenski. 2007. *Accelerating genome sequencing 100x with FPGAs.* Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[12] Mengyao Zhao, Wan-Ping Lee, Erik P Garrison, and Gabor T Marth. 2013. SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications. *PloS one* 8, 12 (2013), e82138.

A APPENDIX

Algorithm 3 Parallel Matrix initialisation

```
    procedure NewMatrix
        m,n ← dimensions of matrix
3:      dp ← malloc(sizeof(int *)*m)
        size ← m*n
        dp0 ← malloc(sizeof(int)*size)
6:      for i ← [0 . . . size) parallel do
            dp0[0] ← 0
        for i ← [0 . . . m) parallel do
9:          dp[0] ← 0
        dp[0] ← dp0
        for i ← [1 . . . m) do
12:         dp[0] ← dp[i-1] + n
        return dp
```