Parallelization and Analysis of Hirschberg Algorithm

1. Hirschberg Algorithm

The Hirschberg algorithm is a dynamic programming algorithm that finds the optimal sequence alignment between two complete strings. The Dynamic Programming approach follows the dependency structure shown in Figure 1, for a forward pass. In the forward pass a cell is completed using the three neighbors from the left, up and upper left diagonal. And for the case of a backward pass it uses the down, right and bottom right diagonal neighbors.
The algorithm uses the Levenshtein distance that basically penalizes insertion, deletion and mismatch with a cost of 1. The algorithm uses the divide and conquer approach splitting the problem in two halves and solving recursively each part. To determine where to cut the problem, the algorithm divides the biggest string in two and computes the best alignment of the shortest string in each half. One, in forward direction, and the other one backwards. Then, the algorithm finds the best punctuation in the juncture of the two halves. This point is where the problem is split. In the case the algorithm is call with one of the strings with length 1, the algorithm directly finds the best alignment of the two strings [1].

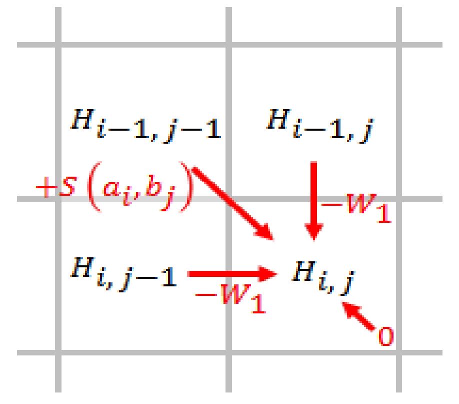


Figure 1: Dependenies of a cell [2]

The Hirschberg algorithm as the Needleman-Wunsch algorithm finds best alignment for complete sequences, while the Smith-Waterman algorithm finds the best local alignment.

One of the important characteristics of this algorithm is the time and space complexity. If x and y are the strings we want to align, and we have that lenght(x) = n and length(y) = m. Both algorithms (Hirschberg and Needleman-Wunsch) have a time complexity of O(nm). However, the former uses a linear space complexity O(min{n,m}) or equivalent O(n + m), while the latter uses quadratic space O(nm). This optimization is based in the observation that each cell can be computed only using only three adjacent cells, as we can see in the Figure 1. Instead of instantiating a full matrix of $n \times m$ cells we can use a row of $2 \times min(n,m)$ cells.

Despite the original algorithm has a space complexity cost O(n+m), this concrete implementation uses static allocation during the recursive calls. Therefore, the real space cost is O(m*log(n)). We will review this affirmation in section 5.

2. Experimental Set Up

I now present my evaluation methodology, including the execution environment, the machines used, the profiling applications and the inputs I use:

Execution Environment: For compiling my binaries I used GCC version 7.3.0. For running my experiments I used SLURM queue system to isolate the experiment from other users actions. In the case of multi socket clusters I have use numactl to pin the threads to one socket and avoiding NUMA effects

Machines: I have executed all my CPU experiments in two clusters. The first one is boada which has an Intel(R) Xeon(R) CPU E5645. The main characteristics of this CPU are its 6 cores with Westmere architecture running at 2.40GHz. The main characteristic of this CPU is that supports 128 bits SSE.4 and multi-threading. The second cluster is the a Skylake Xeon with 28 cores and 56 threads with support for AVX-512. It runs at 2.1GHz but can reach 3.8 on turbo (1-core). The last level of cache is 38.5MB. This cluster allocates jobs in nodes of two sockets.

Profiling Applicaitons: For profiling and tracing my apps I have used gprof and Extrae. For visualizing the traces generated with Extrae, I have used Paraver.

Inputfiles: The input files I have used for my applications are the originals text string that came with the original code. I have added some shorter strings for debugging purposes and two extra experiments that use larger strings. These files are Quadra_Large (200k string) and Massive (400k string).

3. Profiling Analysis

The first step in order to perform the optimization and parallelization of the application is the execution, timing, profiling and trace of the sequential code. We have named this version as “**Hseq**”. The execution and timing let us establish which is the baseline against we will compare the performance of our optimizations. The execution time of the Hirschberg application can be found in the Evaluation Section. At this point, I have executed different experiments using different flags to try to capture which ones perform better. Finally, I have choose the “-O3 -march=native“ flags.

Once I have captured the execution time of the sequential algorithm, I wanted to know where the program spends most of the execution time. First, I have tried using the gprof and compiling the code with -O0 -pg. However, this method does not extract good information. The main problem is that -O3 flag uses inline function preventing us to differentiate the number of times each function is called. And in the case of -O0 the execution time increases dramatically being spent at the function calls.

In this situation, I have opted to use Extrae and Paraver to collect custom events written by my self. This approach is good since we can re-utilize the events in future version of the code to catch the execution time spent in each function rather than the number of calls to that function. In the next table we can see the percentage of time spent in each function

Function	***% of Total ROI Time***
Sequential	1.16
nwlcost	47.43
nwrcost	50.62
nwalign	0.13
reverse	0.13

We can see that most of the time is spent in the nwlcost and nwrcost functions. These functions compute the first and second halves of the alignment matrix. After those functions, the rest of the execution time is spent along other functions like recursive calls. Finally, nwalign is the least time consuming function with reverse. One problem with this table is that it shows the aggregated times of all function calls to each function, and therefore we cannot see how the functions are called. In the Figure 2, we can see the Trace timeline of the Hirschberg Algorithm. The light red and white functions are the function calls to nwlcost and nwright. While the dark red are calls to reverse functions.

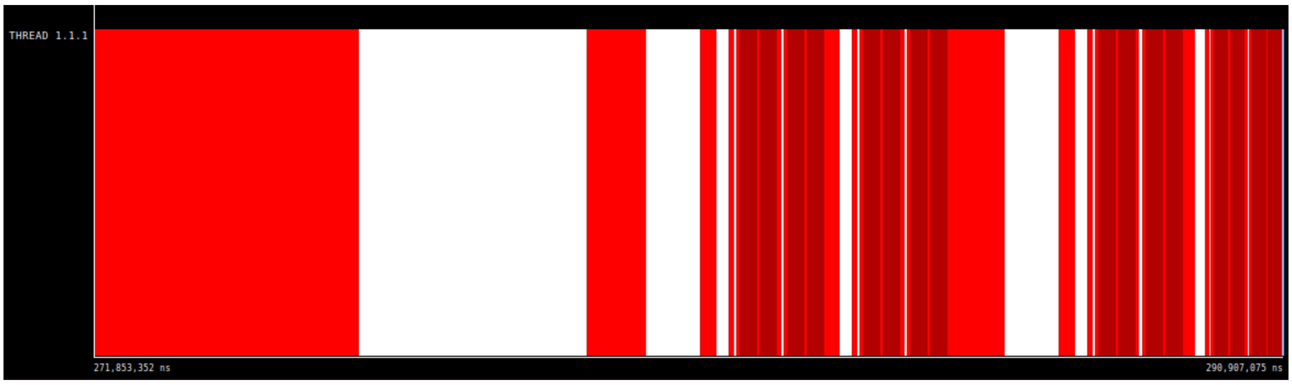


Figure 2: Trace extracted from the sequential application

4. Naive Parallelizing with OpenMP

The very first optimization I have performed is to parallelize the nwlcost and nwright functions. For this purpose I have used OpenMP task to run them in parallel. The name of this version is "**Naive_Tasks**". The maximum speed-up that this approach can achieve is computed with the Amdahl equation:

$$\frac{1}{(1-Popt)+\frac{Popt}{P}}$$

Where here, Popt is the parallelizable portion of the code (0,97), and P the number of cores (2 in this case). Therefore, the maximum speed-up is **1.961x**. The experiments show a geometric mean speed-up of 1.82x which is slightly lower than the theoretical maximum. In the case of large strings the speed-ups rises up to 1.94x.

The main problem of this approach is that does not take advantage of the multiple recursive calls that can be parallelized. The only think that prevents us from paralellizing these recursive calls is that the two recursive calls are dependent. This dependency is caused because the first call returns a pointer to the alignment string where the second should continue performing the alignment. In the next section we explain how to remove the dependency.

5. Independent Recursive Calls and Memory Optimization

As mentioned in the previous Section, the algorithm performs two recursive calls that are dependent between them. This dependence exists because each call fills an alignment string that encodes what steps are needed to align String 1 with String 2. These steps are: insert (+), remove (-), substitute (!) and match (=). This way, the second recursive call should start where the first call finished. However, we do not know where the first call will finish before performing both calls.

One possible solution to this problem could be perform each recursive call with a temporal alignment string and then perform an aggregation of both in the final alignment string. However, this implies a lot of data movement that is not necessary. A better solution can be found, if we ask ourselves: Why does exist multiple possible alignment lengths for any two string with the same length
? It is easy to see the answer to this question when we traverse the alignment matrix of the Needleman-Wunsch algorithm. It starts at the bottom right cell of the matrix. On each step we either move up, left or upper left diagonal, as can be seen in Figure 3. If the only allowed steps were only up and left the movements, the alignment would follow the Manhattan distance. This implies that all the possible paths from one cell to another would have the same length. However, the diagonal movement, that is caused by the match and replace operations, introduces the length variability.

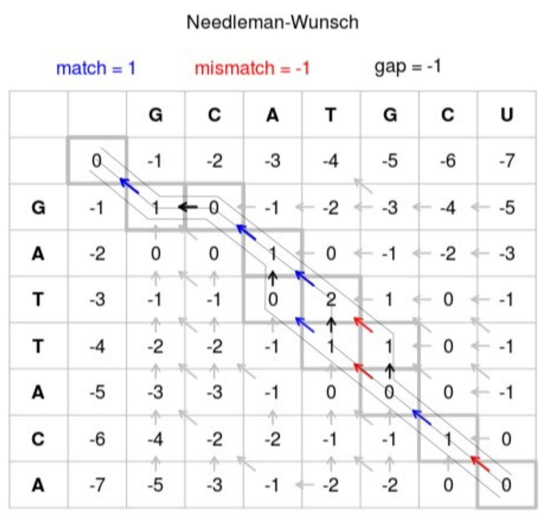


Figure 3: Alignment matrix of the Needleman-Wunsch algorithm [2]

Therefore, the solution to create a fixed length alignment algorithm is to duplicate the operations of match and replace. Thus, each time the algorithm codifies a match operation instead of using one char (=) it will use two (==). The same applies for the mismatch. With this optimization we now know the length of the alignment, which is always $n+m$.

This modification also allows us to optimize the function nwalign. This function computes which operation should be done at each step, and creates the alignment string. As explained before this is done traversing the matrix from the end to the origin. Since the string is written backwards, it needs to perform a reverse operation. Now that we know the exact size of the string we can fill the string in the final order so no extra data movement is needed.

The third and final optimization to the sequential version is to reduce the memory footprint of the code. As mentioned in the Section 1, the memory footprint O(m*log(n)) instead of O(n+m). This is happens because in each recursive call an array of length m is allocated nut not freed before calling the recursive functions. Thus, I have transformed the static data allocation into dynamic memory allocation. The total memory footprints can be seen in the following table.

Version/ Mem. Footprint	Short	Medium	Large	Double Large
Sequential Hirschberg	49.32 KB	458.0 KB	1.456 MB	2.891 MB
OptHSeq	31.99 KB	205.4 KB	647.9 KB	1.248 MB

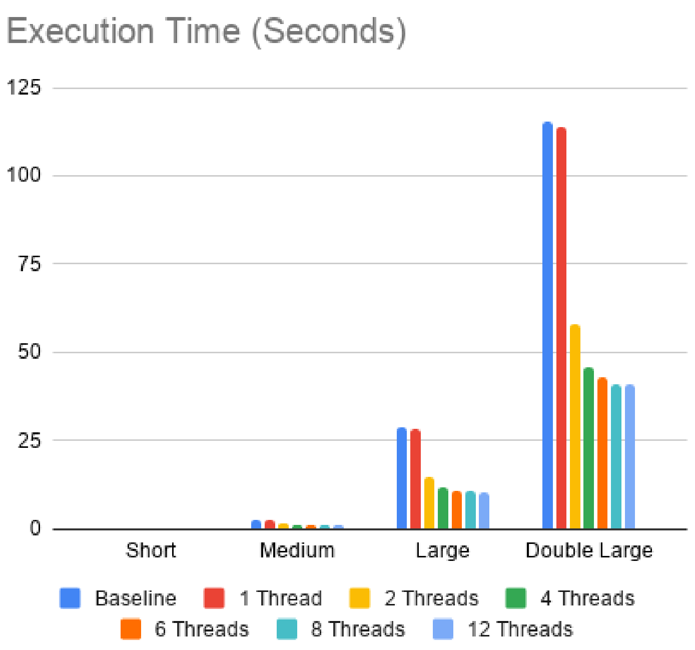
6. Recursive Parallelization

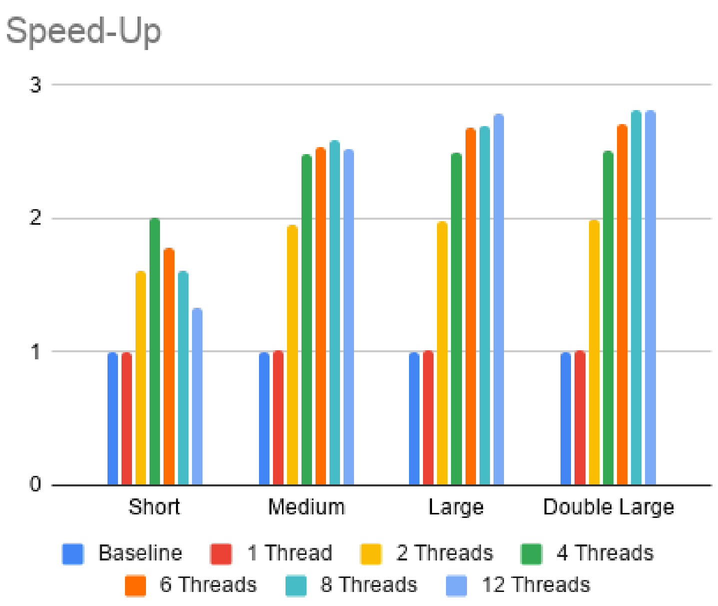
Once the code has been optimized to allow fixed length alignments, I have proceed to parallelize the recursive calls using tasks. In order to avoid too many OpenMP task for cases in which the recursion is large, I have inserted a cutoff condition to stop creating more tasks. This condition stops when the width of the three is more than 8 branches.

A part from these optimizations, I have parallelized and vectorized two extra loops that where not being parallelized. The first loop finds the minimum alignment cost in the juncture of the two sub-alignments. The second one is used to transform insertions into deletions and deletion into insertions. This is done because the algorithm can change the reference string for the query one in case the query is longer than the reference.

With the new parallelization, I have ran a scaling analysis to see what is the speed-up obtained and what is the best thread number for each workload. We can see the results in the following table and images.

Threads/Execution Time (Seconds)	Short	Medium	Large	Double Large	Geomean Speed-Up
1	0.016	2.566	28.381	113.605	1.007
2	0.010	1.319	14.502	57.954	1.873
4	**0.008**	1.041	11.498	45.908	2.361
6	0.009	1.016	10.700	42.603	**2.392**
8	0.010	**0.999**	10.678	40.941	2.365
12	0.012	1.025	**10.295**	**40.885**	2.266





7. Vectorization and Fine Multithreading

After parallelizing the recursive calls, I have decided to reanalyze the application with Extrae and Paraver. In Figure 4, we can see how the different parallel applications are distributed among the threads. Despite the taskification of recursive calls does a good work for the leaves of the tree, the main time consuming call is spent in nwlcost and nwrcost at the first recursion level. Therefore, the next step needs to target these two functions.

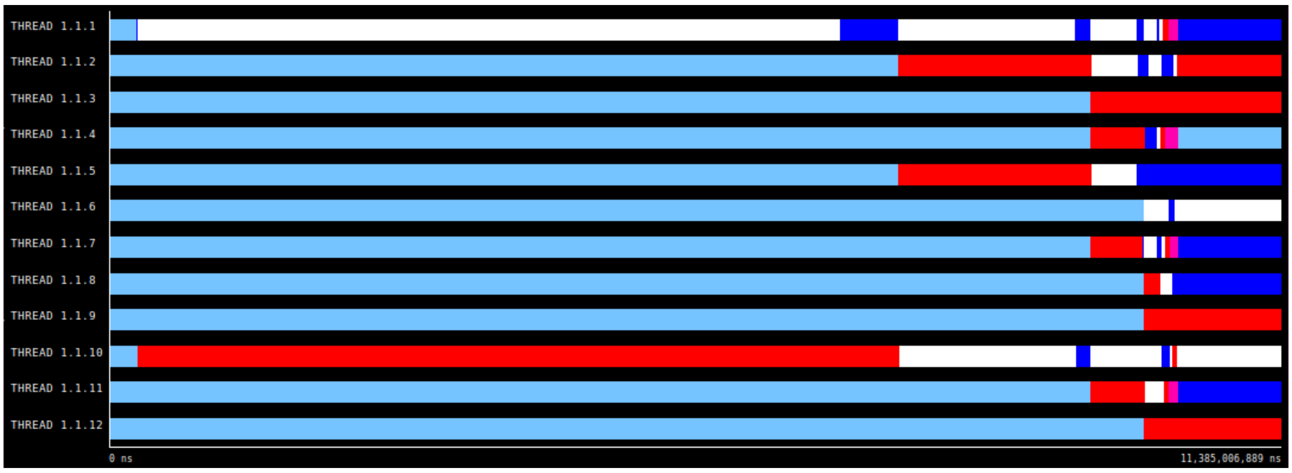


Figure 4: Trace timeline of the Parallel Recursive version

The main problem to parallelize these two functions is that they follow a dynamic programming approach that fills the cells of a matrix in a specific order. Each cell has a dependency we saw at Section 1 in the Figure 1. The original algorithm traverses the alignment cost matrix by rows and columns. Therefore, no parallelization can be applied since every iteration of the loop depends on the previous iteration. I have transformed this algorithm into the anti-diagonal traversing method or skewed traverse [3]. This method instead of compute row by row, computes the anti-diagonals one by one as can be seen in Figure 5. This method enables easy auto vectorization and also loop-unrolling. However, since It is quite complex to apply (there are different regions that must be traversed in different ways) the multithreading approach requires extra work.

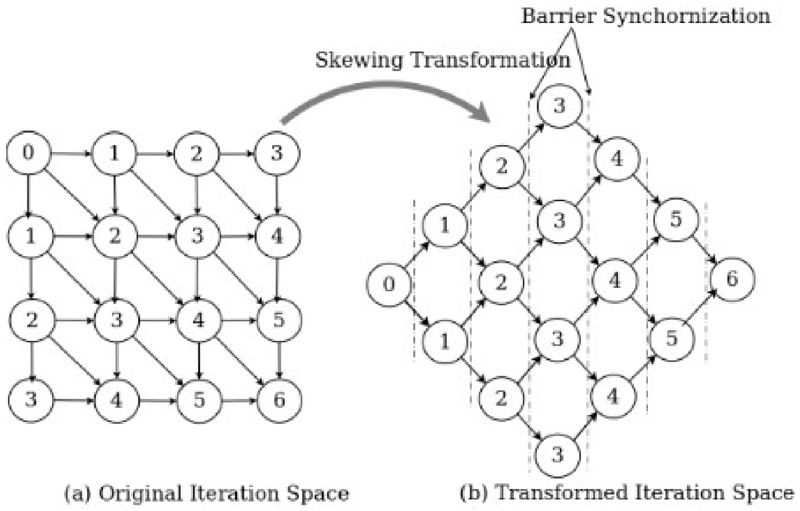


Figure 5: Skewing Transformation to allow vectorization

There are two vectorized versions, one that only applies the vectorization without the OpenMP tasks called “**HVec**” and the other has tasks called “**VecRec_Tasks**”.

Finally, I have rewritten the code to parallelize these functions. The code splits the diagonals into two segments and lets two threads to execute each segment. For the first diagonals, I have used only one thread, because the amount of work is low for two threads (See Figure 6) [4]. I have use only one parallel region to avoid multiple open and close constructs. At the end of each iteration there is a sync barrier as the one in the figure 5. The main problem with this extra parallelism is that the CPU is not able to take advantage of it, therefore I have used only two threads. I have also reduced the number of barriers in the cases in which only one thread is executing. With these approach called “**FinalOpt**”, I have obtained 99x speed-up for some benchmarks with respect to the sequential code.

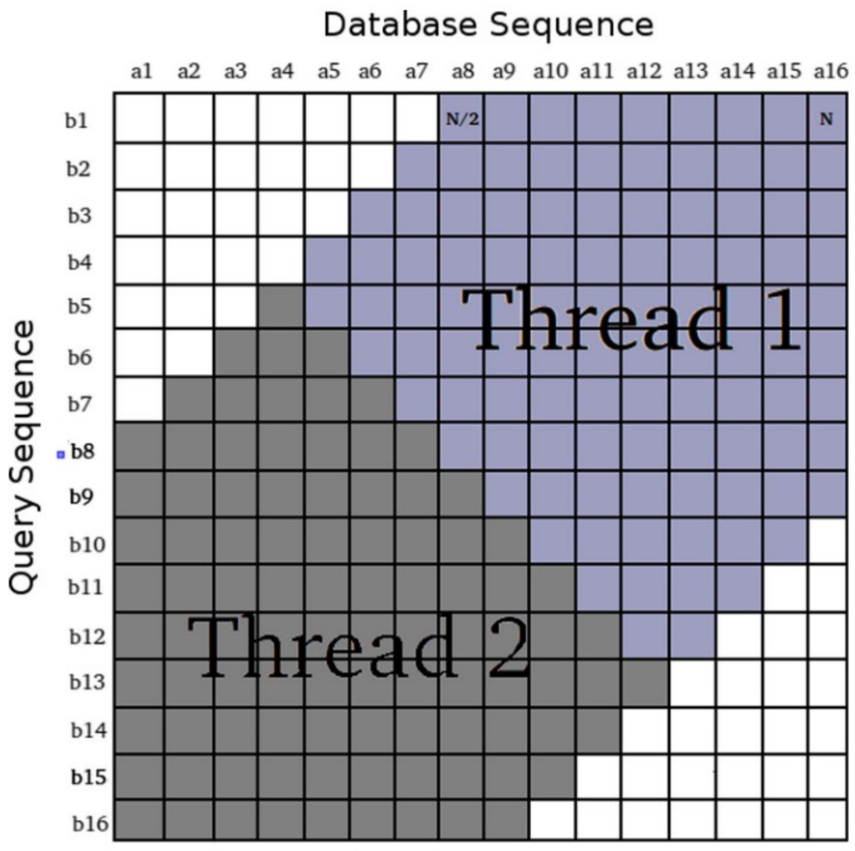

Database Sequence
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
Query Sequence
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16
N/2
N
Thread 1
Thread 2

Figure 6: Finethreading apprach

8. Evaluation

In this Section I will evaluate all the different versions in each platform. The procedure to measure the execution time is: launch multiple executions with different thread numbers, check that the output is correct, take the best time from the experiments. In addition to the execution time, I have computed the Geometric Mean of all Speed-ups I obtained with respect to the Sequential version. Finally, I have added a extra metric used in many scientific works to measure the performance, called Cell Updates Per Second (CUPS). This metric takes the amount of cells of the Needleman-Wunsch matrix and divides them by the execution time.

First, I have started in boada cluster, executing each version. We can see the results in the following Table:

Version/Execution Time (Seconds)	Short	Medium	Large	Double Large	Quadra Large	Geomean Speed-Up
HSeq	0.016	2.581	28.739	115.062	460.247	1.000
NWSeq	0.011	1.792	20.586	*	*	1.430
Naive_Tasks	0.010	1.407	14.942	59.222	235.782	1.845
Rec_Tasks	0.008	0.999	10.295	40.885	162.778	2.581
HVec	0.003	0.337	3.867	16.620	69.606	6.738
VecRec_Tasks	0.003	0.148	1.531	6.272	25.785	14.171
Final_Opt	0.008	0.136	1.003	3.783	15.740	15.739

In these experiments we can see how each version improves step by step the execution time. In the cases marked with "*", the execution did not finish because it could not allocate enough memory. Is interesting to see that vectorization alone is able to obtain more than 6x speed-up. During the executions with OpenMP most of the times 12 threads executes better than 6. Despite there are only 6 real cores. This occurs because the algorithm has a good data locality and a very High Arithmetic Intensity, which is good for hyper-threading. Finally, I have been able to obtain 30x speed-up with the final version. Next, I have computed the average CUPS of each execution. Note that only one GTX680 can obtain 119 GCUPS with CUDASW++3.0 [5]. However, the query length is restricted to 5.000 sequences.

Version / GCUPS	Short	Medium	Large	Double Large	Quadra Large
HSeq	0.073	0.099	0.100	0.100	0.100
NWSeq	0.106	0.142	0.140	-	-
Naive_Tasks	0.117	0.182	0.193	0,195	0.196
Rec_Tasks	0.146	0.256	0.280	0.282	0.284
HVec	0.390	0.760	0.747	0.695	0.664
VecRec_Tasks	0.390	1.729	1.887	1.843	1.793
Final_Opt	0.146	1.882	2.881	**3.055**	2.937

Next, I have moved to the Skylake cluster. In this case, I have computed the Sequential application speed-up with respect to boada cluster, and the followings with respect to the Sequential in Skylake. We can see in the following Table the results:

Version/Exec Time (Seconds)	Short	Medium	Large	Double Large	Quadra Large	Massive	Geomean Speed-Up
HSeq	0.006	1.287	14.176	53.855	220.510	401.062	2.172
NWSeq	0.004	1.077	11.878	37.268	*	*	1.326
Naive_Tasks	0.007	0.763	8.089	32.114	132.166	241.924	1.507
Rec_Tasks	0.004	0.556	5.519	22.112	87.620	189.036	2.208
HVec	0.001	0.099	0.996	4.057	24.251	62.843	9.742
VecRec_Tasks	0.003	0.064	0.489	1.703	9.936	27.040	15.159
Final_Opt	0.002	0.073	0.263	0.732	2.210	4.458	35.140

The sequential gain a remarkable 2x with respect to boada. The speed-up comes from the turbo mode that duplicates the frequency to 4 GHz. Again Needleman-Wunsch could not finish many experiments. We can see that the results are similar, but for the OpenMP approaches the speed-up are lower. This happens because the turbo mode cannot be sustained with multiple threads running. For vectorization, despite we quadruplicate the vector size the speed-ups of big runs are degraded from 12x to 6x. This is cause by the very well known heat diffusion problem in AVX-512.

VecRec_Task achieve an 31x for Double Large. However, the higher core count does not perform well for larger workloads. Finally, I have run the Final_Opt version obtaining spectacular 99x for the Quadra Large benchmark. For these version I have modified a the number of threads used to compute the cost functions. The massive performance of these versions is explained by a 4x increment in the vector length, ~5x more cores and the extra cache storage that is able to save the entire data structures.

Version / GCUPS	Short	Medium	Large	Double Large	Quadra Large	Massive Large
HSeq	0.194	0,199	0.204	0.214	0.209	0.230
NWSeq	0.292	0,238	0.243	0.310	-	-
Naive_Tasks	0.167	0.335	0.357	0.359	0.349	0.382
Rec_Tasks	0.292	0.460	0.523	0.522	0.527	0.489
HVec	1.169	2.586	2.901	2.849	1.906	1.471
VecRec_Tasks	0.389	4.000	5.910	6.788	4.653	3.420
Final_Opt	0.585	3.507	10.989	15.792	**20.923**	20,74

9. Future Work

In the future I would like to perform some other optimizations that I could not apply because I did not have enough time. The very first optimization consist in adding support for NUMA nodes. I have run the experiments in a multi-socket environment and I was not able to take advantage of the 28 cores present in the other socket. The basic idea will be to split each half of the problem in one socket. Other options are, use MPI to execute different sections in different nodes. But avoiding the communication overhead will be a big deal.

I would like to rewrite the output of the application since for sequences bigger than 400k sequences, I get a segmentation fault when printing the output. Maybe using an output file I could execute the algorithm with such big input.

Another interesting point will be to write an efficient nwalign version that uses vectorization and parallelism to speed-up this function. This will enable a better performance to use in cases with n dimension bigger than 1. Since there are branches of the recursion that perform several calls to this function, instead of doing only one at a higher level in the recursion tree.

10. Conclusions

From this work I have obtained a good experience that I will use in the future. The best code is not the one that goes straight to the solution, but that one that fits best in our purposes and technology. At the hour of parallelization, some times is better to question why things are done that way and try to think about new solutions out of the box. Vectorization is one of the most useful tools that should not be under estimated in favor of parallelization.

References

1. Needleman–Wunsch algorithm. Last accessed 20 Dec 2020 https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

2. Smith–Waterman algorithm. Last accessed 20 Dec 2020 https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

3. Chaudhari, Anuj & Kagathara, Deep & Patel, Vibha. (2015). A GPU based implementation of Needleman-Wunsch algorithm using skewing transformation. DOI 10.1109/IC3.2015.7346733.

4. Jararweh, Y., Al-Ayyoub, M., Fakirah, M. *et al.* Improving the performance of the needleman-wunsch algorithm using parallelization and vectorization techniques. *Multimed Tools Appl* **78,** 3961–3977 (2019). https://doi.org/10.1007/s11042-017-5092-0

5. Liu, Y., Wirawan, A. & Schmidt, B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* **14,** 117 (2013). https://doi.org/10.1186/1471-2105-14-117