Parallel Protein Structure Alignment: A Comparative Study of Two Parallel Programming Paradigms

Nada M. A. Mohammed
Faculty of Computer and Information Science, Ain Shams University

Hala M. Ebeid
Faculty of Computer and Information Science, Ain Shams University

Mostafa G. M. Mostafa
Faculty of Computer and Information Science, Ain Shams University

Mahmoud E. A. Gadallah
Modern Academy for Computer Science and Information Technology

ABSTRACT
Protein 3D structure alignment process has become the key focus of interest in structural bioinformatics. Yet, obtaining perfect alignment in a short execution time was not successful to this point. To overcome this problem, researchers tend to use parallel programming techniques to enhance the performance of the alignment process. In this article, we compare between two parallel programming paradigms for implementing a parallel version of the well-known pairwise alignment algorithm MatAlign. This parallel algorithm is implemented by using two common APIs for C++ parallel programming, which are OpenMP for multi-core CPUs and CUDA for multi-core GPUs. The results show that beside the significant improvement of the parallel implementation over the sequential one, it also shows that the multi-core GPU parallel programming model improves speedup over multi-core CPU programming model.

General Terms
Protein Structure Alignment, GPU Parallel Computing, Multi-core Parallel Programming

Keywords
MatAlign, TM-Score, GPGPU, CUDA, OpenMP

1. INTRODUCTION
Although the pairwise protein three-dimensional (3D) structure alignment plays a critical role in many molecular biology fields especially in structural bioinformatics, its complexity is categorized as non-deterministic polynomial-time hard (NP-hard) [1]. In this article, the problem of finding a good alignment on common servers is studied, which often have several processing units (CPUs and GPUs). Modern structural bioinformatics applications in different bioinformatics fields require finding fast algorithms capable of processing and aligning large volume of data [2, 3, 4, 5]. As a common solution, one can deploy a large number of processors to do the task concurrently. This article will discuss how to design and implement parallel structural alignment algorithms and will present the actual timing results for the alignment process

1.1 Parallel Programming Background
There is an urge to find the best parallel programming techniques for the benefit of performance for protein pairwise alignment process, see [6, 7, 8]. Some of the parallel computing methods tend to solve problems by using CPU hardware capabilities while as other methods use GPU's. CPUs and GPUs are significantly different which makes them suitable to perform different tasks.

Up till now, no one can determine which between CPU and GPU can produce the perfect parallel results. Both provide specific advantages for specific problems. Architecturally, GPUs have had very high arithmetic intensity. They have hundreds of cores that are designed to process an enormous amount of time-consuming operations (SIMD/Single Instruction Multiple Data). In contrast, A CPU, due to its few cores, cannot efficiently process many operations [9].

1.2 The Structural Alignment Problem
Pairwise protein structure alignment is the process of finding similar portions between two proteins based on their three-dimensional information. If the alignment result is accurate, it will be easy to predict functional relationships that may not be apparent from sequence comparison [10].

There are several methods available to perform the structural alignment. These methods are divided into two categories, 1) methods which aim to retrieve optimal alignment results and 2) methods which tend to obtain approximate alignment results. Methods in the first category are known as NP-hard problems due to the unlimited number of possible superposition of the two structures. Likewise, methods in the second category are still computationally too expensive, although they can manage the process successfully. As a consequence, most of the alignment algorithms introduced in the literature are, therefore, heuristic.

The rest of this article is organized as follows. The next section introduces a brief explanation of the MatAlign alignment procedure. Then Section 3 explains how this algorithm has been parallelized and implemented by CUDA and OpenMP APIs. In Section 4, the comparative study for the proposed parallel implementations is provided and discussed. At the end of the paper, the conclusion is presented in Section 5.

2. THE MatAlign ALGORITHM
In MatAlign [11], the pairwise protein 3D structure alignment is reached, simply, by aligning the distance matrices of the two query proteins specified by the user instead of comparing their original 3D structure. These distance matrices are built by calculating the distance between Alpha-Carbon (C_α) atoms. Technically, MatAlign applies a two-level dynamic programming approach by first mapping the protein structures into two-dimensional (2D) distance matrices and aligns them to find the initial alignment. Second, initial alignment is refined to reach the optimum alignment score [11].

Level 1: Finding Initial Alignment

Assume DM_A and DM_B represent distance matrices for two query proteins A and B respectively. In MatAlign first level, a

score matrix SM is calculated by aligning each row from DMA against each row from DMB by using a match function similar to the one used in the classical Needleman-Wunsch [12]. This match function is used to determine the matching degree between two alpha carbon atoms distance values d_1 and d_2. The match function can be defined as:

$$Match\ (d_1, d_2) = \begin{cases} \dfrac{\alpha}{|d_1 - d_2| +} & if\ |d_1 - d_2| \leq T_{Match} \\ 0 & otherwise \end{cases}$$

Where α is the score adjusting weight with value = 0.7, and T_{Match} is the difference threshold of the distances with value 1.6Å. This match function is used in the dynamic programming's selection step.

After executing the dynamic programming, the matching score of the two given rows is reached. See the row-row comparison in **Algorithm 1**.

Algorithm 1 A single-thread version of the first level of MatAlign algorithm

```
1: Procedure GetInitialAlignment(DM_A, DM_B)
2:    Let SM be the similarity matrix
3:    for row i in DM_A do
4:       for row j in DM_B do
5:          SM[i,j] ← row-row matching score of i^th row of
                DM_A and j^th row of DM_B
6:       end for
7:    end for
8:    GS ← 0  //GS is the Gap Score
9:    F ← GS  // Let F be a second similarity matrix
10:   for row i in DM_A do
11:      for row j in DM_B do
12:         F[i,j] ← Max ( F[i-1,j]+GS, F[i-1,j-1]+SM[i,j], F[i-
                1,j]+GS)
13:      end for
14:   end for
15:   GetAlignment (SM, F)
16: end procedure
```

Level 2: Alignment Refining

MatAlign used both the RMSD [13] value (Δ) and the number of aligned pairs by using the same scoring function (S) used in [14]. Since the initial alignment resulted from level 1 is not usually an optimum in terms of S, the alignment is iteratively refined until S cannot be further improved. RMSD is known as the most commonly used tool in specifying the similarity between two protein structures. Despite that, it has some defects that affect the accuracy of the comparison results. TM-score [15] overwhelms RMSD problems. And so, the TM-score is more efficient than using RMSD. In order to assess the alignment quality and balance the accuracy, we used the TM-score function instead the regular MatAlign Score (S).

Based on the heavy computations in the row-row comparison step, only the first level of MatAlign is parallelized.

3. THE PARALLEL METHODOLOGIES

As noted in Section 2, the two parallel implementations are modified versions of the basic MatAlign to gain better performance. The modified parallel algorithms are named as PTM-MatAlign [16]. Note that the prefix "PTM" denotes "Parallel algorithm enhanced using TM-Score".

There are two main time-consuming steps, in the first alignment level that affect the performance of MatAlign. First, the heavy calculation in row-row alignment at **Algorithm 1** lines 3 – 7 and second, the dynamic programming performed on the score matrix to generate the list of aligned pairs of **Algorithm 1** lines 10 – 14. PTM-MatAlign parallelizes the above two steps to accelerate the comparison process.

3.1 The CUDA implementation

In the CUDA parallel implementation, one GPU kernel is assigned to run the row-row comparison step in parallel. Each row from the first protein is aligned against each row from the second protein and stores the similarity results in the global memory. Since the total number of blocks that can concurrently execute a kernel depends on the maximum global memory size of the GPU, in this model, the total number of blocks B_t is determined in terms of the number of amino acids in query proteins, A and B, and the total number of threads T_t in each block where B_t= |B| / T_t *|A|.

In terms of memory usage, each thread requires one similarity matrix of size (|A|+1)*(|B|+1). Therefore, the total memory space needed to execute all threads in parallel is |A|*|B|*(|A|+1)*(|B|+1). This amount of data exceeds the limit of GPU local and shared memory in case of large size proteins. Therefore, the only rescue is the use of global memory to overcome the limitation of GPU memory resources.

In order to optimize the use of global memory, each thread remembers only the last two rows of the similarity matrix. This is satisfactory to determine the maximum score of the similarity matrix, which is needed to check whether the two query proteins are similar or not. **Algorithm 2** describes a pseudo-code of the CUDA parallel implementation.

Algorithm 2 The CUDA parallel implementation (Step 1)

```
1: kernel Row-RowAlignmentKernel(DM_A, DM_B)
2:    Let SM be the similarity matrix
3:    Let patch = |B| / T_t
4:    for i ← BlockId / patch to |A| in parallel do
5:       for j ← (BlockId mod patch) * T_t + ThreadId to |B| in
               parallel do
6:          SM[i,j] ← score of aligning row i in DM_A against
                row j in DM_B
7:       end for
8:    end for
9: end kernel
```

After that, another dynamic programming algorithm is applied on the score matrix SM and then traced back by a recursive algorithm to generate the initially aligned pairs. Since alignment path is needed, then not only the first two rows of similarity matrix F is needed but also the whole matrix rows have to be allocated. Consequently, from the memory view, only one global similarity matrix F is represented as a 1D vector of type double with size [(|A|+1) * (|B|+1)].

In order to decrease the number of accesses to the GPU global memory, the similarity matrix F is not calculated cell by cell but it is divided into diagonals. From the data dependency view, each element F[i,j] depends on three elements, F[i,j-1], F[i-1, j], and F[i-1, j-1]. In another word, F[i,j] depends on the data from both same and previous rows. This kind of dependency looks like a diagonal scan over the elements. This technique is called wave-front technique [17].

In fact, algorithms which are using wave-front techniques are usually developed by calling two nested loops where the outer loop represents matrix diagonals, and the inner loop represents the cells of each diagonal. This technique can be parallelized by implementing the inner loop as a parallel for

loop. This means that all cells in each diagonal run in parallel where diagonals itself are running in sequence. So in terms of data dependency, each matrix diagonal depends on the previous one.

This way of parallelism has its weakness, such as the load resulted from repetitive CPU-GPU data transfer process which is too unhelpful and will affect performance. Since, in the CUDA dynamic parallel model [18], GPU kernel can launch an inline nested kernels to eliminate the data transfer load from CPU to GPU and vice versa, this model is applied as illustrated in **Algorithm 3**.

The computation is split into two GPU kernels where the first kernel (parent) is responsible for calling diagonals in sequence and figuring out the number of blocks needed to parallelize each diagonal. It is clear that not all diagonals need the same number of blocks to run in parallel. Accordingly, the total number of blocks B_t is determined by the number of cells in each diagonal C_d where $B_t = C_d$. Afterward, this parent kernel calls a child kernel to calculate the value of each cell using dynamic programming. Once this calculation is done, the similarity matrix is moved from GPU memory to CPU memory to run a recursive algorithm to generate the initial alignment pairs. Experiments show that recursive step is much faster on the CPU than the GPU.

Algorithm 3 The CUDA parallel implementation (Step 2)

```
1: kernel ParentKernel(SM, F)
2:    Let P = max number of cells in all diagonals
3:    Let R = number of repeats of diagonals with max
            number of cells
4:    for i=1 to P do
5:      ChildKernel<<<i, 1>>>(SM, F)
6:    end for
7:    for i=1 to R do
8:      ChildKernel<<<P, 1>>>(SM, F)
9:    end for
10:   for i=1 to P do
11:     ChildKernel<<<P-i, 1>>>(SM, F)
12:   end for
13: end kernel

1: kernel ChildKernel(SM, F)
2:    calculate current cell indices i and j using threadId and
            blockId
3:    F[i,j] ← Max( F[i-1,j]+GS, F[i-1,j-1]+SM[i,j], F[i-
            1,j]+GS)
4: end kernel
```

3.2 The OpenMP implementation

The proposed OpenMP parallel implementation of the algorithm follows the same logic as explained for the CUDA implementation. These were developed by adding #pragma omp directives to our sequential C++ code (e.g. add one above the first C++ for loop in line 3 of Algorithm 1).

4. RESULTS AND DISCUSSIONS

To test the performance and correctness of the parallel algorithms, a benchmarked dataset of 68 protein pairs which is introduced by Fischer [19] is used. This dataset was selected to represent different classes according to the SCOP classification [20] such as, all alpha proteins (all α), all beta proteins (all β), alpha and beta proteins (α/β), alpha and beta proteins (α+β), multi-domain proteins (alpha and beta), membrane and cell surface proteins and peptides, coiled-coil proteins, and small proteins. Some of the query protein structures used in the evaluation are shown in **Table 1**.

Table 1. Sample of the test dataset

PDB	Length	SCOP Class
1hom_A	68	All alpha proteins
1hip_A	85	Small proteins
1ten_A	90	All beta proteins
1onc_A	104	Alpha and beta proteins a+b
2hhm_A	276	Multi-domain proteins
2cmd_A	312	Alpha and beta proteins a/b
2omf_A	340	Membrane and cell surface proteins and peptides
1gal_A	583	Alpha and beta proteins

The parallel algorithms are implemented using C++ with the two APIs: CUDA 6.5 and OpenMP 2.0. To run the CUDA program, an Nvidia GeForce GTX 860M series (Maxwell class) graphics card is used. This GTX 860M has Nvidia compute capability 5.0 and consists of 5 streaming multiprocessors. Each multiprocessor has 640 processing cores, 49 KB of shared memory per block, 65 KB of total constant memory, 65536 registers per block, and 2GB of total global memory. To run the OpenMP parallel implementation, a hyper-threaded dual-core 2.5 GHz Intel CPUs is used which provides at least 8 and up to 16 independent Pthreads.

Table 2 summaries the time in seconds for the two proposed protein structural alignment parallel implementations when the query length changed from 131 to 900. The experimental performed using 68 proteins. In the last row, the overall average running times (in seconds) are displayed. For the CUDA implementation, the I/O time for loading the distance matrices into device memory are not included. Likewise no disk I/O time for any algorithm is included. For detailed execution time, see Fig 1**.**

As noticed in **Table 2**, there is a fluctuation in the execution times. The reason behind this fluctuation is that the algorithm which is used to build the parallel implementations is divided into two computational parts, 1) finding the initial alignment and 2) refining the alignment results. Since each part has it is own time complexity as mentioned in [11], then it is possible for the same query to reach the worst case in the first part while achieving the best case in the other part. And since not all queries have the same length or the same structure, then it is not predicted for all queries to have the same time complexity in both alignment parts. Therefore, there is no clear relationship between the total query length and the relevant execution time for both CUDA and OpenMP implementations.

In general, as it is expected, the overall average running times of the sequential algorithm, MatAlign, in **Table 2** are much slower than the parallel implementations. For the parallel algorithm running on OpenMP, it two times faster than MatAlign. On the other hand, the parallel implementation running on the GPU has the best overall performance (about 4 times speed-up over the Open-MP parallel algorithm, and about 8 times speed-up over the sequential algorithm) as shown in Fig 2.

Although the CUDA algorithm provides best results, the OpenMP algorithm has some advantages that CUDA does not have. The benefit of using OpenMP over CUDA is that the memory available is larger (8GB vs. 2Gb DRAM) and much faster at data transfer rate. This makes the alignment process for the large-size proteins much faster.

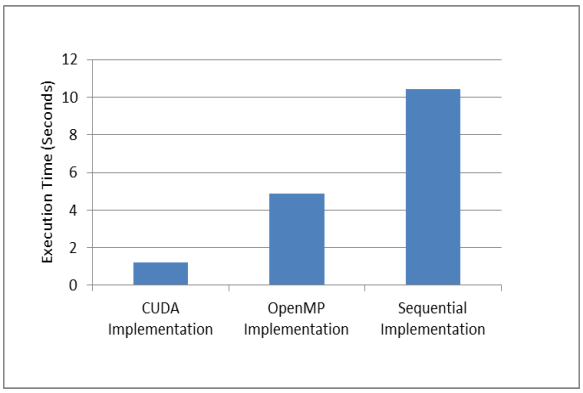

Fig 2: Average execution time of the protein structural alignment algorithm using different query length

5. CONCLUSION

In this paper, a comparative study of implementing parallel protein structure alignment using two parallel programming paradigms (CUDA and OpenMP) is presented. The execution time of both serial and parallel execution is used as the evaluation measure for the comparison. It is found that both CUDA and OpenMP based parallel implementations improve the execution time of detecting the best alignment path between two protein structures. However, for the PTM-MatAlign algorithm, it is found that the GPU implementation accelerates it more than that of the OpenMP. Though the difficulty of implementing the alignment algorithm on the GPU platform using CUDA implementation, it speedup the execution time by 3.9x on average better than the OpenMP implementation and 8.4x on average better than the sequential implementation. In general, it is recommended to use GPU than OpenMP for problems with massive amount of calculations. For future work, it is expected to parallelize different alignment algorithms and compare the results with those obtained from PTM-MatAlign.

6. REFERENCES

[1] Alexandrov, N., and Fischer, D. 1996. Analysis of topological and nontopological structural similarities in the PDB: New examples with old structures. Proteins: Structure, Function, and Bioinformatics, 25(3), 354-365.

[2] Singh, A. P., and Brutlag, D. L. 2000. Protein Structure Alignment: A Comparison of Methods. Bioinformatics.

[3] Aung, Z., and Tan, K. 2006. MatAlign: Precise protein structure comparison by matrix alignment. Journal of Bioinformatics and Computational Biology, 4(06), 1197-1216.

[4] Clark, M. 2012. Introduction to GPU Computing, s.l.: Developer Technology Group, nVIDIA.

[5] Daniel, F., Arne, E., Danny, R., and David, E. 1996. Assessing the performance of fold recognition methods by means of a comprehensive benchmark. In Proceedings of Pacific Symposium on Biocomputing 397, 300-318.

[6] Dhraief, A., Issaoui, R., and Belghith, A. 2011. Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability. In The 1st International Conference on Advanced Communications and Computation (INFOCOMP).

[7] Godzik, A. 1996. The structural alignment between two proteins: is there a unique answer?. Protein Science: A Publication of the Protein Society, 5(7), 1325-1338.

[8] Halperin, I., Ma, B., Wolfson, H., and Nussinov, R. 2002. Principles of docking: An overview of search algorithms and a guide to scoring functions. Proteins: Structure, Function, and Bioinformatics 47(4), 409-443.

[9] Hung, C.-L., and Lin, Y.-L. 2013. Implementation of a Parallel Protein Structure Alignment Service on Cloud. International Journal of Genomics.

[10] Jones, S., 2012. Introduction to Dynamic Parallelism. In GPU Technology Conference Presentation S 338, p. 2012.

[11] Koehl, P. 2001. Protein structure similarities. Curr Opin Struct Biol. 11, 348-353.

[12] Mrozek, D., Brożek, M., and Małysiak-Mrozek, B. 2014. Parallel implementation of 3D protein structure similarity searches using a GPU and the CUDA. J Mol Model, 20(2), p. 2067.

[13] Murzin, A. G., Brenner, S. E., Hubbard, T. & Chothia, C., 1995. SCOP: a structural classification of proteins database for the investigation of sequences and structures. J Mol Biol 247(4), 536–540.

[14] Needleman, S. B., and Wunsch, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol, 48(3), 443-53.

[15] Samudrala, R., and Hung, L.-H. 2012. Accelerated Protein Structure Comparison using TM-Score-GPU. Bioinformatics, 28(16), 2191-2192.

[16] Mohammed, N. M., Ebeid, H. M., Mostafa, M. G., and Gadallah, M. E., 2016. PTM-MatAlign: A Fast GPU-Based Algorithm for Pairwise Protein Structure Alignment, submitted to International Journal of Computational Biology, (2016).

[17] Shin, D. H., Hou, J., Chandonia, J. M., Das, D., Choi, I. G., Kim, R., and Kim, S. H. 2007. Structure-based inference of molecular functions of proteins of unknown function from Berkeley Structural Genomics Center. Journal of Structural and Functional Genomics, 8(2-3), 99-105.

[18] Xu, Y., Xu, D., and Liang, J. 2007. Computational Methods for Protein Structure Prediction and Modeling.

[19] Zhang, Y. & Skolnick, J., 2004. Scoring function for automated assessment of protein structure template quality. Proteins, 57(4), 702-10.

[20] Zhang, C., and Lai, L. 2011. Towards structure-based protein drug design. Biochemical Society Transactions, 39(5), 1382-138

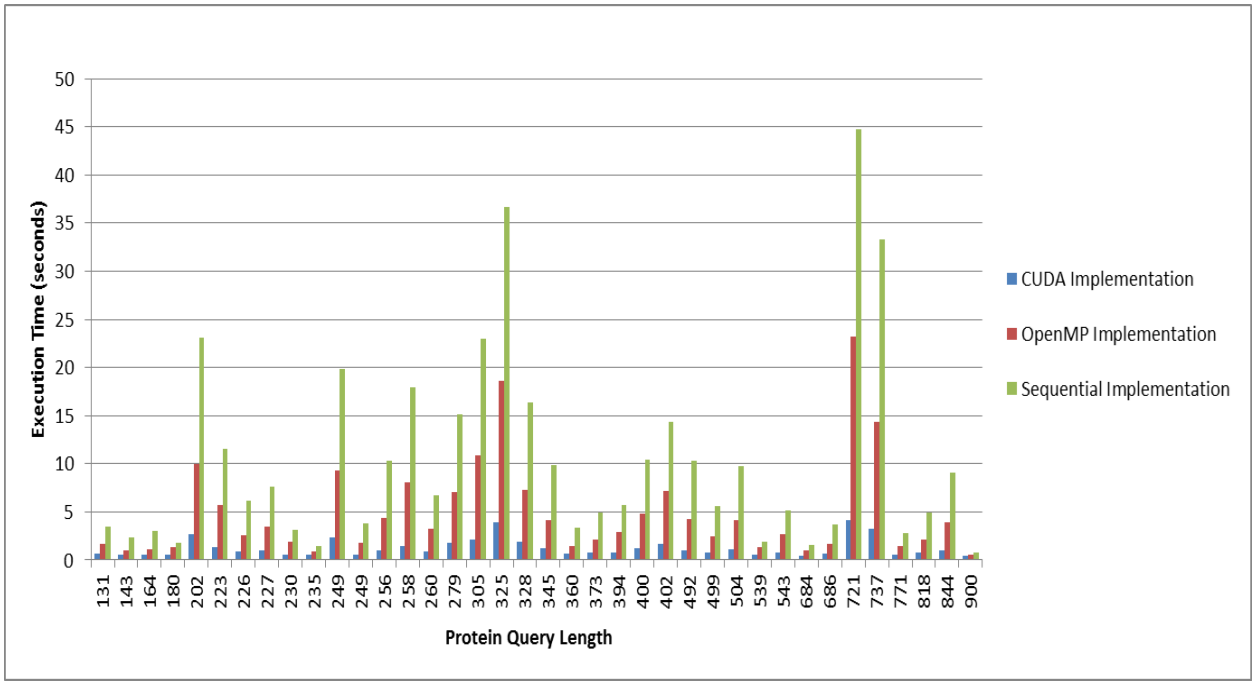

Fig 1: Comparison between execution time using CUDA, OpenMP, and sequential implementations for the protein structural alignment algorithm

Table 2: Execution times (in seconds) of the protein structural alignment algorithm using CUDA, OpenMP, and sequential implementations for different query length

Query Length	PTM-MatAlign (CUDA)	PTM-MatAlign (OpenMP)	MatAlign (Sequential)
131	0.689	1.7	3.487
143	0.541	0.98	2.4
164	0.564	1.144	3.035
180	0.5	1.3	1.75
202	2.7	9.945	23.1
230	0.6	1.89	3.09
235	0.5	0.859	1.485
249	2.3	9.284	19.82
249	0.6	1.817	3.824
256	1	4.311	10.355
258	1.5	8.019	17.9559
260	0.9	3.208	6.67
279	1.8	7.07	15.096
305	2.1	10.918	22.976
325	3.9	18.555	36.715
328	1.9	7.31	16.33799
345	1.2	4.097	9.9
360	0.63	1.467	3.301
373	0.76	2.169	4.973
394	0.82	2.878	5.7
400	1.28	4.79	10.429
402	1.71	7.141	14.294
492	0.972	4.29	10.353
499	0.82	2.491	5.634
504	1.14	4.106	9.698
539	0.502	1.318	1.922
543	0.801	2.683	5.122
684	0.48	0.9888	1.53
686	0.71	1.685	3.746
721	4.1	23.158	44.723
737	3.2	14.359	33.35
771	0.54	1.5	2.8

818	0.74	2.1	4.921
844	1.02	3.94	9.09
900	0.401	0.543	0.817
Average	1.240263158	4.889968421	10.413155