On the Parallelization of the SProt Measure and the TM-Score Algorithm

Jakub Galgonek, Martin Kruliš, and David Hoksza

Departement of Software Engineering, Charles University in Prague
Malostranske nám. 25, 118 00 Praha 1, Czech Republic
{galgonek,krulis,hoksza}@ksi.mff.cuni.cz

Abstract. Similarity measures for the protein structures are quite complex and require significant computational time. We propose a parallel approach to this problem to fully exploit the computational power of current CPU architectures. This paper summarizes experience and insights acquired from the parallel implementation of the SProt similarity method, its database access method, and also the wellknown TM-score algorithm. The implementation scales almost linearly with the number of CPUs and achieves 21.4× speedup on a 24-core system. The implementation is currently employed in the web application http://siret.cz/p3s.

Keywords: protein structure similarity, parallel, optimization.

1 Introduction

Measuring the similarity of protein structures has many applications in various fields of computational proteomics. Based on the hypothesis that the proteins sharing similar structure often share also other properties, the measurement of structural similarity could be used, for example, to study biological functions of proteins [1]. It could be also used to identify related proteins from the evolutionary point of view [2]. Structural similarity can reveal distant protein relations even in cases which cannot be recognized on the basis of sequential similarity since evolution tends to preserve the structure (and thus the function) of the protein rather than its sequence [3].

Computing the measure of structural similarity could be very time consuming. This computational cost become very important especially in search queries to the protein database when similarity between a query protein structure and many structures from the database must be computed.

Some of the approaches are trying to avoid this problem by designing measure along with an efficient indexing method. The *ProtDex2* method [4] represents protein structure as a set of feature vectors describing spatial properties of secondary structure element pairs. Feature vectors of the query are compared with feature vectors of the database proteins which are organized in inverted file. Hence, the time complexity of the search algorithm is dependent on the number of different feature vectors rather than on the number of structures in the database. Another approach presents the *3D-Blast* method [5]. It defines a

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 238–247, 2013.

small structural alphabet and corresponding substitution matrix, which allows to employ the well-established algorithms of the BLAST sequence method. The *Vorometric* method [6] tries to utilize metric access methods. It represents each amino acid by a so-called *contact string*, for which the metric similarity is defined. Contact strings from the query are searched in the database of contact strings and found hits are used as seeds that are extended into alignments.

Combining the design of similarity measure with the design of indexing and access method allows the creation of efficient search system. However, this approach may also lead to some restrictions that cause a decrease in effectiveness of the similarity measure. Therefore, we have designed our similarity measure *SProt* [7] without any type of indexing method and the indexing method based on metric properties was added after the measure was designed. Even though the indexing improves performance significantly, the methods is still quite slow for real-time applications. We have employed parallelism in order to speed up the search process further.

As the processors developed in time, we have been allowed to compute larger and more complex biological tasks. The processor frequency has grown steadily for a few decades. This trend has reached an impasse recently due to the physical limitations of the silicon-based chips. The CPU development turned and focused on parallel processing instead. This major change in hardware architectures forced us to adjust our approach to high-performance computing as we can no longer rely solely on the compiler to produce optimal code for the processor. In order to achieve optimal performance, programs and algorithms must be rewritten in a way that embraces the parallel nature of the current processor architectures.

The paper is organized as follows. We briefly revise the principles and algorithms of SProt and TM-score measures (Section 2). Our parallel implementation is described in Section 3. Section 4 discusses the parallelization benefits from the perspective of experimental results and Section 5 concludes the paper.

2 Methods and Algorithms

Our method consist of two parts – the similarity measure and the access method that provides means for performing the similarity search on a database of protein structures. We will briefly describe both parts in the following section.

2.1 SProt Measure

Similarity measure SProt [7] is based on comparing local spacial neighborhoods of the amino acids present in a protein. A neighborhood of an amino acid is defined by amino acids present in a sphere with center in the given amino acid (called central amino acid) and with radius 9Å. Defining similarity on these small parts of the protein structure is much easier than working with the structure as whole. Continuous parts of the amino acid backbones included in the neighborhoods that span over central amino acids of the spheres are short enough so they

can be aligned without gaps. This will provide sufficiently long seed alignment from which the local superposition can be computed. The superposition is then used to align remaining amino acids present in the neighborhoods. This local alignment expresses the similarity of the neighborhoods.

Global alignment for whole protein structures is defined consequently by application of Needleman-Wunsch dynamic programming algorithm [8]. It uses similarity of amino acid neighborhoods as scoring function in combination with constant gap penalty model. Finally, the alignment quality is measured by well-established TM-score algorithm [9]. The algorithm tries to find the superposition that maximizes the following formula:

$$\text{TM-score} = \frac{1}{L_T} \sum_{i=1}^{L_A} \frac{1}{1+\left(\frac{d_i}{d_0(L_T)}\right)^2}, \tag{1}$$

where L_A is the length of the alignment, L_T is the size of the query structure, d_i is the distance of the ith pair of the aligned amino acids according to given superposition, and $d_0(L_T)$ is a scale function.

The TM-score expresses the quality of the alignment very well and the TM-score algorithm is quite resistant to misaligned pairs (unlike, for example, the RMSD method for which the outliers presents serious problem). However, in contrast to RMSD [10], there is no efficient algorithm that would allow us to find the exact superposition maximizing the formula quickly. Therefore, the suboptimal heuristic approach is the only solution.

The heuristic algorithm is based on the idea that two at least partially similar protein structures are likely to have structure subsets that will be very similar. For such subsets the RMSD superposition would not be much different from optimal TM-superposition. Therefore, the algorithm tries computing RMSD superpositions for various subsets of the alignment called *cuts*. A TM-score value is computed for each RMSD superposition and the maximal value is returned as the result of the algorithm.

Algorithm 1. TM-score algorithm

Input: $X \in (\mathbb{R}^3)^L, Y \in (\mathbb{R}^3)^L, L \in \mathbb{N}$
Result: $score \in \mathbb{R}, U_{max} \in \mathbb{R}^{3\times3}, T_{max} \in \mathbb{R}^3$
Parameters: $max_c = 20$

```
1  begin
2      score_max ← 0
3      foreach l ∈ {L, L/2, L/4, ..., 4} and i ∈ {0, ..., L − l} do
4          cut ← {i, ..., i + l − 1}
5          for c = 0 to max_c do
6              (U, T) ← RMSD(X[cut], Y[cut])
7              score ← TM(X, UY + T)
8              if score > score_max then score_max ← score; (U_max, T_max) ← (U, T)
9              cut' ← cut; cut ← {}
10             for i = 0 to L − 1 do
11                 if |X[i] − (UY[i] + T) < 3d_0| then cut ← cut ∪ {i}
12             if cut = cut' then break
13     Result: score_max, (U_max, T_max)
```

Algorithm 2. LAESA access method

```
   Input: q ∈ 𝒟, k ∈ ℕ
   Result: R ⊂ D
   Data: D ⊂ 𝒟, P ⊂ D, d_i : D × P → ℝ_0^+
1  begin
2      S ← D; R ← {}; ∀o ∈ D : e(o) ← 0; threshold ← ∞; s ← arbitrary member of P
3      while S ≠ {} do
4          S ← S\{s}
5          dist(s) ← d(q, s)                                   // distance measuring
6          R ← k-nearest objects from set R ∪ {s}
7          if |R| = k then threshold ← max{dist(o); o ∈ R}
8          d_s ← ∞
9          foreach o ∈ S do                          // approximation and estimation loop
10             if s ∈ P then e(o) ← max{|d_i(s, o) − dist(s)|, e(o)}     // approximation
11             if e(o) > threshold then S ← S\{o}                        // elimination
               if e(o) < d_s then d_s ← e(o); s' ← o
12         s ← s'
13     Result: R
```

The TM-score method is outlined in Algorithm 1. It receives two sequences of coordinates X and Y, of the length L, representing coordinates of aligned pairs of amino acids for the first and the second protein respectively. The result is the value of computed TM-score and the corresponding superposition. The *outer loop* (lines 3-12) of the algorithm takes all continuous parts of the alignment of lengths $L, L/2, \ldots, 4$ as the initial cuts. Each iteration of the *internal loop* (lines 5-12) computes the superposition for the current cut (line 6) and constructs a new cut that contains only those amino acid pairs that are closer than the $3d_0$ threshold after the superposition (lines 10-11). The internal loop continues until the cut is stabilized or the maximal number of iterations is reached. After that the outer loop continues by picking another initial cut for testing.

2.2 SProt Access Method

The similarity measure SProt achieved excellent results at information retrieval experiments. However, the procedure is very time consuming. We have designed specific access method based on LAESA metric indexing [11,12] in order to increase the performance of the measure.

Metric access methods [13] are being used for similarity search that employs similarity measure d (called distance), that complies with metric axioms. These axioms allow us to estimate the distance lower bounds between two objects based on known distances to another objects. Let us have query object q and database object o for which we are trying to determine their similarity. If we precompute the distance between o and another database object p (usually called pivot) and the distance from q to p is computed as well, we can use triangle inequality to estimate the lower bound for the distance between q and o:

$$d(q, o) \geqq |d(q, p) - d(o, p)| \tag{2}$$

If the estimated lower bound is greater than some predefined range or the distances to the best result objects found so far, we can quickly dismiss object o

Table 1. Required computational time

Method Subpart	Time (min:sec)	Relative Time (in percents)
Query Model Loads	0:03	0.06%
Alignments	96:12	85.91%
Alignments: Score Matrix	95:43	85.48%
TM-score	13:41	12.22%
LAESA: Approximation and Estimation	2:01	1.81%
Total Query Evaluation	111:58	100.00%

without computing potentially expensive distance $d(q, o)$ as the o will certainly not be present in the results.

The same approach employs the LAESA metric access method. The method is being used to find k database structures that are the most similar to the query structure. It takes a subset (called *pivots*) of structures $P \subseteq D$ from given database D and precomputes table d_i of distances $d(p, o)$ for each pivot $p \in P$ and database structure $o \in D$. When the distance between query structure q and any of the pivots p is computed during the query evaluation process, this distance can be used to update lower bound estimates for the remaining structures and prune them using triangular inequality described earlier.

The fundamentals of this method are summarized in the Algorithm 2. It expects only the query structure q and number k (beside the database itself) as input and yields closest k structures from the database as output. The algorithm iteratively process a set of candidate structures S and keeps a set R of intermediate results (closest k structures). Initially the S contains the whole database D and the algorithm stops when S becomes empty. The algorithm also maintains an array of lower bound estimates $e(o)$, that contains the greatest lower bound estimate for each candidate o. In each iteration (lines 3-12), a structure $s \in S$ with the lowest $e(s)$ value is taken and also removed from S (line 4). The distance $d(q, s)$ is computed (line 5) and intermediate results in R are updated accordingly (line 6). If s is a pivot ($s \in P$), the lower bound estimates $e(o)$ for the candidates remaining in S are updated using values from the precomputed pivot table and the triangle inequality (line 10). If any estimate $e(o)$ is greater than threshold value $t = max\{d(q, x), x \in R\}$, structure o is pruned from S (line 11). When the algorithm terminates, set R contains the final result.

Since the SProt measure does not conform to all metric axioms, the access method has to be slightly modified and works only as an approximation [7]. The lower bound estimation formula has been altered for asymmetric measures and some new attributes has been incorporated into the method to increase precision of measures that does not fully conform to the triangle inequality axiom. However, the main idea of the algorithm was preserved.

3 Parallel Implementation

There are many approaches to the parallel processing of our task. The most straightforward would be to process multiple queries concurrently. In such case,

each query is evaluated by separate task while tasks are processed concurrently by available processors, however, the task itself is processed in serial manner. This approach is very easy to implement and, since the tasks are completely independent, it is also very efficient. Unfortunately, sufficient number of queries must be submitted to the system at the same time in order to occupy all available processors. Furthermore, if there are a few long lasting queries amongst a block of short queries, the system will be left with a few big tasks that will linger (occupying only one core each) while remaining processors become idle.

Second possible approach is to modify the method itself, so that one query evaluation utilizes multiple cores. We have profiled our code to determine, how long does each part of the algorithm take. Times measured in the profiling run and their relative contribution are summarized in Table 1. The ProtDex2 dataset [4] containing 34,055 structures and 108 queries was used for the profiling. The search yielded 20 the most similar structures for each query.

The profiling results clearly show that the alignment computation (especially the score matrix computation) time dominates the entire evaluation. Therefore, this part would benefit the most from the concurrent execution. However, according to Amdahl's law [14], serial parts create serious scalability issues, thus we parallelize the TM-score computation and the LAESA approximation and estimation as well. We do not parallelize the query model loading as the I/O operations are serial in nature and the loading itself takes insignificant part of the overall time.

We can also choose completely different method and try a data parallel approach. In such case, the database is divided into fragments, which are treated as independent (small) datasets and queried concurrently. Each fragment yields a local result and local results of all fragments are merged into the final result. Even though this method has minimal scheduling overhead and does not depend on the number of queries available, it is not suitable for our algorithm. Our search measure employs the LAESA method to prune the number of computed distances (structure similarities). This method performs better on larger datasets and with larger number of pivots. If the database fragments are too small, the effect of the LAESA is significantly diminished. In the worst case, it can deteriorate into simple scan of the entire database. For these reasons, we have abandoned this approach and focus solely on the previous two approaches.

Concurrent Alignment Computation. According to profiling results, the alignment algorithm spends its almost entire time by computing the scoring matrix. Since each item of the matrix is computed independently, the computation can be easily modified to use two-dimensional parallel for-loop instead of its sequential version. The matrix is evenly divided into tiles and each tile is processed as a concurrent task. The size of the tiles is chosen carefully so that one task computing one tile will consist of at least 10^5 CPU instructions[1]. The tile-wise approach is preferred to both row-wise and column-wise approaches as it is better optimized for CPU caches.

[1] Tasks containing 10^5 to 10^6 of CPU instructions were observed as optimal [15].

Concurrent TM-Score Computation. In the case of TM-score, we chose to parallelize the outer-loop only. Internal loops are rather short, thus, the overhead of task scheduling would be too great. The outer loop iterates over initial cuts. An initial cut can be defined by its length and the index of the first pair of aligned amino acids (we denote the length-index tuple the *initial configuration*). Our approach first generates all initial configurations to an array and then traverses the array by the parallel for-loop. Hence, each configuration is processed independently as a separate task. In order to eliminate explicit synchronization, each thread keeps its own maximum value of computed scores and the corresponding superposition. The best score is picked from the local values when the parallel for-loop terminates. Concurrent computation of initial cuts introduces nondeterminism into the process, since we arbitrarily change the order of score comparisons. Even though this does not affect the method, a deterministic approach is preferred by the users and required for the verification of our parallel implementation. We have modified the score comparison, so that the configuration index is used as secondary key in case two score values are equal. This modification ensures that the choice of the best superposition is deterministic despite the concurrent processing.

Our implementation generates the initial configurations sequentially and then processes them concurrently. In theory, it would be better to generate the configurations by one thread and dispatch them immediately to other available threads. We can simply utilize a parallel while-loop [15] provided with an initial configuration generator. However, this approach requires significantly more thread synchronization, thus having greater overhead. Empirical results show that it is not as efficient as concurrent processing of configurations, which have been pregenerated sequentially.

LAESA Approximation and Estimation Loop. The main loop of the LAESA method cannot be effectively parallelized since each iteration updates the set of candidates as well as lower bound estimates which are required by next iteration. Fortunately, we are still able to parallelize the internal loop, called also the *approximation and estimation loop* (see Algorithm 2: lines 9-11). The internal loop updates the estimate values $e(o)$, filters the candidate set S, and finds the candidate s with the lowest estimate $e(s)$. To avoid synchronization, the threads work on different parts of S (and $e(o)$ respectively). When the candidate s needs to be found, each thread t has its own candidate s_t and after the entire set S is processed concurrently, the candidate s is quickly found from the set $s \in \{s_t | t \in Threads\}$. Since we are parallelizing the internal loop, a barrier must be present at the end of the main loop. Therefore, the whole process is slightly less efficient in comparison with other parts of the algorithm.

4 Discussion

The experiments were performed on three different versions of the parallel implementation (all based on the *Intel Threading Building Blocks* library [15]) and

compared with the performance of the serial implementation. The first one (denoted *Query-Parallel*) employs the first parallelization approach – queries are evaluated concurrently, but each query runs serially. The second one (*Intern-Parallel*) utilizes the second approach to parallelism – each query is parallelized inside, but queries are submitted for evaluation sequentially. The third one (*Full-Parallel*) combines previous two methods to achieve even higher scalability.

Each of these versions points out different aspects of the parallel processing. The Query-Parallel version is the most efficient since it reduces the overhead of task spawning, scheduling, and synchronization. However, it presumes that there are enough tasks to keep available CPU cores occupied. The Intern-Parallel implementation better reflects the real world situation when only one query is being processed and we still want to exploit parallelism to expedite the evaluation of this query. Combination of these two methods provides the ultimate solution (Full-Parallel) that reaches the limits of the optimal scalability.

We measure the efficiency of parallelization by the speedup factor:

$$S_p = \frac{T_{seq}}{T_p}, \tag{3}$$

where T_{seq} is the execution time of the sequential algorithm and T_p is the execution time of the parallel version running on p CPU cores. Note that we do not distinguish between two physical CPUs and two CPU cores on one die. Even though that there are some differences like NUMA factor or L3 cache-sharing issues, these hardware properties had negligible impact on overall performance in our case. On the other hand, we do not consider two logical cores mapped to one physical core using hyper-threading technology to be independent cores and all tests were performed on independent physical cores only.

We have used the Dell M910 server containing four Intel Xeon E7540 CPUs, six cores clocked at 2.0 GHz each (i.e., 24 cores total). The server was equipped with 128 GB of RAM organized as 4-node cache coherent NUMA. The experiments were conducted using the ProtDex2 dataset [4] containing 34,055 structures and 108 testing queries. The speedup of all three implementations measured for up to 24 cores is depicted in Figure 1. We have used an approximation curve that demonstrates the speedup of the algorithm. The approximation curve minimizes the squared differences from the measured values.

Figure 1 (left part) indicates that Query-Parallel version exhibits lower scalability and larger variance of measured results than other versions. This is due to large granularity of the tasks (as we execute only 108 queries), thus the performance strongly depends on the task distribution over available threads. It is safe to assume that the performance will be better if the number of the tasks is significantly larger than number of available processors. The Intern-Parallel version performs better than Query-Parallel as it produces more fine-grained tasks. However, this approach still suffers from the synchronization between two subsequent queries – evaluation of one query must be completed before another query is started. These synchronization points between queries create performance gaps when CPUs are heavily underutilized, thus the speedup is still less than optimal (16× on 24 cores). On the other hand, the Intern-Parallel version

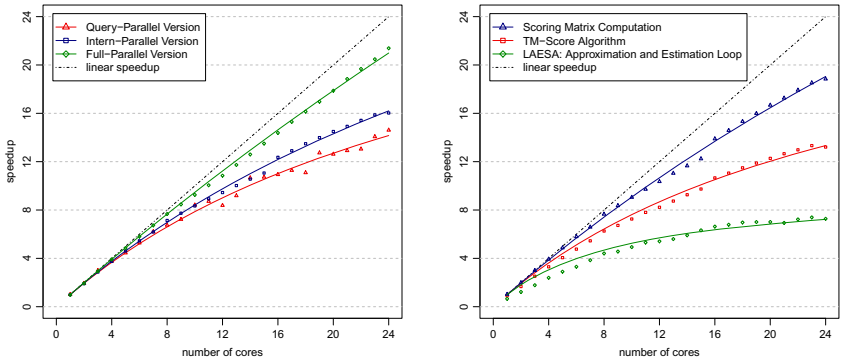

Fig. 1. Measured speedup of the implementation – comparison of the versions on the left and the algorithm parts overview on the right

simulates the situation when the queries are provided by the user in sequential manner. We have observed that significant speedup can be achieved on multiple cores even when single query is processed.

If we combine both approaches, we preserve the benefit of fine-grained tasks and we eliminate the synchronization points between queries. The Full-Parallel version reached 21.4× speedup on 24 cores. Considering the inevitable overhead of the task scheduling and the presence of data loading parts which are sequential in nature, it is safe to say that the scalability of the Full-Parallel version is almost optimal. This version is particularly useful when multiple users search the database and we use it in our web application, which is available at http://siret.cz/p3s.

To compare the efficiency of each part of the parallel implementation, we have also measured their individual speedups (see Figure 1, right part). The results match our expectations. The scoring matrix computation is the most independent, thus exhibits the best speedup. The TM-score algorithm, which is often adopted by other similarity methods, scales also exceptionally well. The weakest part of the algorithm is the LAESA method since it is iterative and only the internal loop was parallelized. We have observed smaller speedup due to synchronization within the main loop and small size of the tasks being dispatched to the threads.

5 Conclusion

We have compared three approaches to the parallel implementation of the SProt measure and its access method. Our best version reached excellent speedup and scales almost linearly with the number of CPU cores. Furthermore, we believe, that achieved speedup is almost optimal, since there is some inevitable overhead and sequential parts that cannot be parallelized. We have also presented concurrent version of the TM-score superposition algorithm. Since the TM-score

is being used in other methods as well, its concurrent version may find other applications beyond the realms of the protein structure similarity measures.

Acknowledgement. This work was supported by the Grant Agency of Charles University [project Nr. 430711]; the Czech Science Foundation [project Nr. 202/11/0968]; and the Specific Academic Research [project Nr. SVV-2012-265312].

References

1. Orengo, C.A., Michie, A.D., Jones, S., Jones, D.T., Swindells, M.B., Thornton, J.M.: CATH–a hierarchic classification of protein domain structures. Structure (London, England: 1993) 5(8), 1093–1108 (1997)
2. Balaji, S., Srinivasan, N.: Use of a database of structural alignments and phylogenetic trees in investigating the relationship between sequence and structural variability among homologous proteins. Protein Eng. 14(4), 219–226 (2001)
3. Chothia, C., Lesk, A.M.: The relation between the divergence of sequence and structure in proteins. The EMBO Journal 5(4), 823–826 (1986)
4. Aung, Z., Tan, K.L.: Rapid 3D protein structure database searching using information retrieval techniques. Bioinformatics 20(7), 1045–1052 (2004)
5. Tung, C.H.H., Huang, J.W.W., Yang, J.M.M.: Kappa-alpha plot derived structural alphabet and BLOSUM-like substitution matrix for rapid search of protein structure database. Genome Biol. 8(3), R31 (2007)
6. Sacan, A., Toroslu, H.I., Ferhatosmanoglu, H.: Integrated search and alignment of protein structures. Bioinformatics 24(24), 2872–2879 (2008)
7. Galgonek, J., Hoksza, D., Skopal, T.: SProt: sphere-based protein structure similarity algorithm. BMC Proteome Science 9(suppl. 1), S20 (2011)
8. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 48(3), 443–453 (1970)
9. Zhang, Y., Skolnick, J.: Scoring function for automated assessment of protein structure template quality. Proteins 57(4), 702–710 (2004)
10. Kabsch, W.: A solution for the best rotation to relate two sets of vectors. Acta Crystallogr. A 32(5), 922–923 (1976)
11. Micó, M.L., Oncina, J., Vidal, E.: A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. Pattern Recognition Letters 15(1), 9–17 (1994)
12. Moreno-Seco, F., Micó, L., Oncina, J.: Extending LAESA Fast Nearest Neighbour Algorithm to Find the k Nearest Neighbours. In: Caelli, T.M., Amin, A., Duin, R.P.W., Kamel, M.S., de Ridder, D. (eds.) SSPR and SPR 2002. LNCS, vol. 2396, pp. 718–724. Springer, Heidelberg (2002)
13. Chávez, E., Navarro, G., Baeza-Yates, R.A., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
14. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM (1967)
15. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc. (2007)