FROST: Revisited and Distributed

Vincent Poirriez*
Université de Valenciennes,
59313 Valenciennes, France
vpoirriez@univ-valenciennes.fr

Antoine Marin
Mathématique, Informatique et Génome,
INRA, 78352 Jouy-en-Josas, France
antoine.marin@jouy.inra.fr

Rumen Andonov
IRISA, Campus de Beaulieu,
35042 Rennes Cedex, France
randonov@irisa.fr

Jean-François Gibrat
Mathématique, Informatique et Génome,
INRA, 78352 Jouy-en-Josas, France
gibrat@jouy.inra.fr

Abstract

FROST (Fold Recognition-Oriented Search Tool) [6] is a software whose purpose is to assign a 3D structure to a protein sequence. It is based on a series of filters and uses a database of about 1200 known 3D structures, each one associated with empirically determined score distributions. FROST uses these distributions to normalize the score obtained when a protein sequence is aligned with a particular 3D structure. Computing these distributions is extremely time consuming; it requires solving about $1,200,000$ *hard combinatorial optimization problems and takes about 40 days on a 2.4 GHz computer. This paper describes how FROST has been successfully redesigned and structured in modules and independent tasks. The new package organization allows these tasks to be distributed and executed in parallel using a centralized dynamic load balancing strategy. On a cluster of 12 PCs, computing the score distributions takes now about 3 days which represents a parallelization efficiency of about 1.*

Key words: protein threading; parallel algorithms; large scale problems;

1. Introduction

The protein folding problem can be simply stated in the following way: given a protein sequence, which is a string over the 20-letter amino acid alphabet, determine the positions of each amino acid atom when the protein assumes its 3D folded shape. Although simply stated, this problem is extremely difficult to solve and is widely recognized as one of the most important challenges in computational biology, today [11, 12, 13, 14, 7].

In case of remote homologs, one of the most promising approaches to the above problem is protein threading, i.e., one tries to align a query protein sequence with a set of 3D structures to check whether the sequence might be compatible with one of the structures. This method relies on three basic facts:

- 3D structures of homologous proteins are much better conserved than their amino acid sequences. Indeed, many cases of proteins with similar folds are known, although having less than 15% sequence identity.
- There is a limited, relatively small number of protein structural families (figures vary between 1,000 and 10,000 according to different estimations [15, 22]).
- Different types of amino acids have different preferences for occupying a particular structural environment. These preferences are at the basis of the empirically calculated score functions that measure the fitness of a given sequence for a 3D structure.

Fold recognition methods based on threading are complex and time consuming computational techniques consisting of the following components:

1. a database of 3D structural templates;
2. an objective function which evaluates any alignment of a sequence to a template structure;
3. a method for finding the best (with respect to the score function) possible sequence-structure alignment;
4. a statistical analysis of the raw scores allowing the detection of the significant sequence-structure alignments.

* This work has been partially supported by the GenoGRID project (ACI GRID, Ministère de la Recherche).

The third point above is related to the problem of finding the optimal sequence-to-structure alignment and is referred as protein threading problem (PTP). From a computer scientist's viewpoint this is the most challenging part of the threading methods. Until recently, it was the main obstacle to the development of efficient and reliable fold recognition methods. In the general case, when variable-length alignment gaps are allowed and pairwise amino acid interactions are considered in the score function, PTP is NP-hard [4]. Moreover, it is MAX-SNP-hard [1], which means that there is no arbitrary close polynomial approximation algorithm, unless P = NP. In this context the progress done by the computational biology community in solving PTP during the last few years is really remarkable [16, 8, 2, 17, 18, 20, 19, 3]. However the empirical results clearly show that the problem is easier in practice than in theory and that it is possible to solve real-life (biological) instances in a reasonable amount of time. One of the most promising approaches in solving this problem is using advanced mathematical programming (Mixed Integer Programming, MIP) models for PTP [8, 2, 17, 18, 21]. A further step in this direction of research is the development of special-purpose algorithms for solving MIP models instead of using general-purpose branch-and-bound algorithms based on linear programming (LP) relaxation. Impressive computational results reported in [3] show that MIP models can be successfully solved using Lagrangian relaxation. This approach is today the default algorithm for solving a PTP instance in FROST.

This paper focuses on the 4th point above. Despite the significant recent progress in solving PTP, this component is still an extremely time consuming part of the threading methods. The underlying score normalization procedure involves threading a large set of queries against each template and requires solving millions of PTP [6]. Accelerating computations involved in this component is crucial for the development of efficient fold recognition method.

The organization of the papers is as follows. In section 2.1 we present a computer scientist's vision of FROST, i.e. as a succession of functions and procedures, yielding dependent and/or independent tasks. This new vision permits to redesign FROST and to organize it in modules which presents numerous advantages. This is discussed in section 2.2. We also distinguish independent tasks and discuss how to distribute them on a cluster of PCs in order to obtain an efficient parallelization (section 3.1). In section 4, the performances of the proposed algorithm are experimentally validated on the entire FROST database and corresponding running times are given.

2. FROST: a computer science vision

2.1. Description of the original FROST

FROST (Fold Recognition-Oriented Search Tool) is intended to assess the reliability of fold assignments to a given protein sequence (hereafter called the query sequence or query for short) [5, 6]. This tool is based on a series of filters, each one possessing a specific scoring (or fitness) function used to measure the adequacy between the query sequence and template structures. There are currently about 1200 template structures. For the time being, FROST employs only two filters. The first filter is based on a fitness function whose parameters involve only a local (degenerate) description of the 3D structure: a given structural state is assigned to each amino acid in the sequence. On the other hand, parameters of the second filter fitness function require the knowledge of the interactions between residues in contact in the 3D structure, thus making use of spatial information. Hereafter, these filters will be called 1D and 3D filters, respectively.

When aligning a given query sequence to a set of 3D structures it is not possible to directly use the raw scores to rank the 3D structures since these scores strongly depend on the query and template lengths and also, in a complicated way, on the particular features of the 3D structures. In addition, the query sequence may correspond to none of the existing folds. Therefore one must have a mean to evaluate the significance of an alignment score. This is done by empirically calculating a distribution of scores for each core, using a set of sequences not related to it. We then compare the alignment score between the query and a core to the corresponding distribution. The more far away the score from the corresponding distribution, the more significant it is because the more far away the score from the bulk of not related sequences scores, the better the chance the sequence has to be related to the template structure.

Because we do not know the analytical form of the distribution we use the following scheme: the raw score (RS) is normalized (NS) using the first and third quartiles of the distribution (q_{25} and q_{75} respectively) according to: $NS = \frac{q_{75} - RS}{q_{75} - q_{25}}$.

As the scores are highly dependent on sequences lengths, for each template, we compute five distributions (for five different sequence lengths corresponding to -30%, -15%, 0%, +15% and +30% of the template length, as explained below). We then linearly interpolate the corresponding quartile values according to the actual query length. Thus, the whole threading procedure is composed of two phases, the first one is the computation of scores distributions (hereafter called phase D) and the second one is the alignment of the sequence of inter-

est with the dataset of templates (hereafter called phase E for evaluation) making use of the previously calculated distributions. These two phases are repeated for each filter (1D and 3D) but some tricks have been developed to accelerate the whole procedure. First, as a template represents a fold, we search for a global match between a query and a template and we thus do not consider queries and templates when their sequences lengths differ from more than 30%. This strategy usually reduces the number of alignments from 1200 to about 300 for the 1D filter. The second is that after the 1D filter, the templates are ranked and only the 10 best are passed to the 3D filter which is more computationally expensive. The algorithm used by FROST in the 1D filter, denoted here by `Ali1D(Q,C)`, is based on dynamic programming and has a quadratic complexity for a fixed query Q and a template C. On the other hand, the algorithm for the 3D filter, here denoted by `Ali3D(Q,C)`, has to solve an alignment problem which is proven to be NP-complete [4]. Even using the fastest algorithm currently available for solving the underlying combinatorial optimization problem [3], computing the score distributions for all the templates takes more than a month when performed sequentially.

The whole procedure requires the following computations:

1. Phase D: align non homologous sequences in order to obtain the scores distributions for all templates and all filters (two for now). Since five distributions are associated to any template, and there are about 200 sequences for each distribution, this procedure needs solving about 1,200,000 quadratic problems `Ali1D` and the same amount of NP-complete problems `Ali3D`.

2. Phase E: align the query with the dataset of templates which requires solving several hundreds of quadratic problems `Ali1D` and N NP-complete problems `Ali3D` (where N is usually ten).

To give an idea of the amount of computation required by the 3D filter, Figure 1 shows the distribution of the $\approx 10^6$ alignment problems solved during phase D w.r.t the number of possible alignments. The latter can be as large as $6.6\ 10^{77}$.

Figure 2 shows the plot of the mean cpu time required to solve the 3D problems involved in phase D as a function of the number of possible alignments.

The purpose of the procedure proposed in the next section is to distribute all these tasks.

Note that phase D needs to be repeated each time the fitness functions or the library of templates change, which is

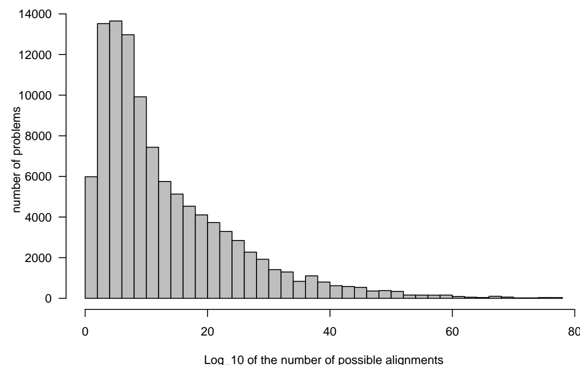

Figure 1. Populations of 3D problems solved during phase D as a function of the $\log_{10}$ of the number of possible alignments (the size of the search space).

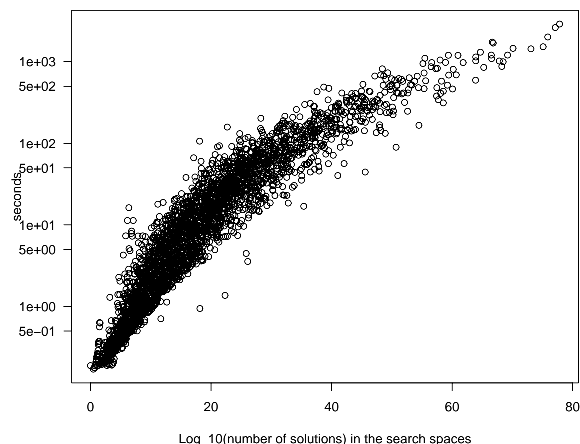

Figure 2. Mean CPU time required to solve the Phase D 3D problems, partitioned w.r.t. the $\log_{10}$ of the number of possible alignments (the size of the search space).

almost always the case when the program is used in a development phase.

2.2. Dividing up FROST into modules

The first improvement in the distributed version (DFROST) compared to the original FROST consists in clearly identifying the different stages and operations in order to make the entire procedure modular. The process of computing the scores distributions is dissociated from the alignment of the query versus the set of templates. We therefore split the two phases (D and E) which used to be interwoven in the original implementation. Such a decomposition presents several advantages. Some of them are:

- Phase D is completely independent from the query, it can be performed as *a preprocessing stage* when it is convenient for the program designer.
- The utilization of the program is *simplified*. Note that only the program designer is supposed to execute phase D, while phase E is executed by an "ordinary" user. From a user's standpoint DFROST is *significantly faster* than FROST, since only phase E is carried out at his request (phase D being performed as a preprocessing step).
- The program designer can *easily carry out different operations* needed for further developments of the algorithm or for database updating such as: adding new filters, changing the fitness functions, adding a new template to the library, etc.
- This organization of DFROST in modules is very *suitable for its decomposition in independent tasks* which can be solved in parallel.

The latter point is discussed in details in the next section.

3. Parallelization

We distinguish two kinds of atomic independent tasks in DFROST: the first is related to solving an instance of a problem of type `Ali1D`, while the second is associated with solving an instance of an `Ali3D` problem[1].

Hence phase D consists in solving 1,200,000 independent tasks of type `Ali1D`, and of type `Ali3D`, while phase E consists in solving several hundreds of independent tasks `Ali1D` and ten independent tasks `Ali3D`. The final decision requires sorting and analysis of the ten best solutions of type `Ali1D` and the ten best solutions of type `Ali3D`.

In the next section we describe how these tasks can be distributed and executed in parallel on a cluster of processors.

1 In reality this problem can be further decomposed in subtasks. Although non independent, these subtasks can be executed in parallel as show in [2, 8]. This parallelization could be easily integrated in DFROST if necessary.

3.1. Parallel Algorithm

There is a couple of important observations to keep in mind in order to obtain an efficient parallel implementation for DFROST. The first is that the exact number of tasks is not known in advance. Second, which is even more important, the tasks are irregular (especially tasks of type `Ali3D`) with unpredictable (for now) and largely varying execution time. In addition, small tasks need to be aggregated in macro-tasks in order to reduce data broadcasting overhead. Since the complexity of the two types of tasks is different, the granularity for macro-tasks `Ali1D` should be different from the granularity for macro-tasks `Ali3D`.

The parallel algorithm that we propose is based on *centralized dynamic load balancing*: macro-tasks are dispatched from a centralized location (pool) in a dynamic way. The work pool is managed by a "master" who gives work *on demand* to idle "slaves". Each slave executes the macro-tasks assigned to it by solving sequentially the corresponding subproblems (either `Ali1D` or `Ali3D`). Note that dynamic load balancing is the only reasonable task-allocation method when dealing with irregular tasks for which the amount of work is not known prior to execution.

In phase E the pool contains initially several hundreds of tasks of type `Ali1D`. The master increases the work granularity by grouping `gran1D` of them in macro-tasks. These macro-tasks are distributed on demand to the slaves that solve the corresponding problems. The solutions computed in this way are sent back to the master and sorted by it locally. The templates associated to the ten best scores yield ten problems of type `Ali3D`. The master groups them in batches of size `gran3D` and transmits them to the slaves where the associated problems are solved. The granularity `gran1D` is bigger than the `gran3D` granularity. Finally the slaves send back to the master the computed solutions.

The strategy in phase D is simpler. The master only aggregates tasks in macro-tasks of size either `gran1D` or `gran3D`, sends them on demand to idle slaves (where the corresponding problems are sequentially solved), and gathers finally the distributions that have been computed. The master processes the library of templates in a sequential manner. First, it aims at distributing all the tasks for a given template to the slaves. However, when the list of tasks for a given template becomes empty and there is at least one slave demanding work, the master continues to distribute

tasks from the next template. This strategy allows to reduce globally the idle time of the processors.

4. Computational experiments

The numerical results presented in this section were obtained on a cluster of 12 Intel(R) Xeon(TM) CPU 2.4 GHz, 2 Gb Ram, RedHat 9 Linux, connected by 1 Gb ethernet network. The behavior of DFROST was tested by entirely computing the phase D of the package, i.e. all the distributions for 1125 templates for both filters.

In the case of 3D filter, solving 1,104,074 alignments required **3 days 3 hours 20 minutes** (wall time of the master). We were not in single-user mode but there were very few other users during this period. We then added the running times reported in the log files of the slaves and obtained a total sequential time equal to **37 days 5 hours 11 minutes**. Therefore, for this very representative set of instances, DFROST exhibits a **speedup of 11.9** with an efficiency close to one. Details from this execution are presented in table 1. The value of the parameter `gran3D` was experimentally fixed to 10.

In the case of 1D filter, solving 1,107,973 alignments required **31 minutes and 20 seconds** (wall time of the master). When we added the running times reported in the log files of the slaves we obtained a total sequential time equal to **4 hours 12 minutes and 55 seconds**. The total time to compute, sequentially, the distributions for the 2 filters was **37 days 9 hours and 24 minutes**, while computed by DFROST on twelve processors the same amount of work required **3 days 3 hours and 55 minutes**.

These significant results, obtained on such a large data set, justify the work done to distribute FROST and prove the efficiency of the proposed parallel algorithm.

4.1. Statistical analysis

Using this parallel algorithm we were able to compute all the distributions for the entire FROST templates library. This was never done before, because of large templates like `1BGLA0` with sequences as long as 528 amino acids, leading to a number of possible alignments as large as 6.647E+77 (see Figure 3 for computation times). We observed that for 188 templates the computation of the distributions requires more than one hour CPU time. Statistical details concerning the running time of the four most time consuming templates are presented in table 2. Remember, that a PTP instance (i.e. the query and the 3D structure are fixed) is considered as an atomic independent task in the current parallel strategy and, as shown in [2, 8], such an instance can be further decomposed in subtasks that can be executed in parallel. We were wondering about the necessity of implementing this parallelization for our application. However, in view of: i) the huge number of independent tasks when computing FROST distributions; ii) the running time presented in tables 1 and 2, as well their statistical recapitulations in figure 3 showing clearly that really hard PTP instances are very rare; iii) the very satisfactory speedup reported in section 4, we decided to stay for now with the current parallel algorithm.

5. Conclusion

Solving the protein threading problem remains time consuming even though we are able to achieve peak rates as high as 10^{74} equivalent threadings per second. For the purpose of testing new developments of the FROST algorithm we are often led to recompute the score distributions which involves carrying out millions of threading alignments. This is not easily tractable with a single computer. In this paper we have presented a simple, but very efficient, load balancing algorithm enabling us to use FROST on a cluster of computers. Using this implementation we were able to compute the score distributions for the whole set of templates in about 3 days on a cluster of 12 computers instead of more than 40 days on a single computer. This parallelization procedure achieved an efficiency of about 1.

References

[1] T. Akutsu and S. Miyano. On the approximation of protein threading. *Theoretical Computer Science*, 210:261–275, 1999.

[2] R. Andonov, S. Balev and N. Yanev, Protein Threading Problem: From Mathematical Models to Parallel Implementations, INFORMS Journal on Computing, 2004; 16(4): Special Issue on Computational Molecular Biology/Bioinformatics, Eds. H. Greenberg, D. Gusfield, Y. Xu, W. Hart, M. Vingro

[3] Stefan Balev, Solving the Protein Threading Problem by Lagrangian Relaxation, WABI 2004, 4th Workshop on Algorithms in Bioinformatics, Bergen, Norway, September 14 - 17, 2004

[4] R. Lathrop, The protein threading problem with sequence amino acid interaction preferences is NP-complete, Protein Eng., 1994; 7: 1059-1068

[5] A. Marin, J.Pothier, K. Zimmermann and J-F. Gibrat, Protein threading statistics: an attempt to assess the significance of a fold assignment to a sequence, Protein structure prediction: bioinformatic approach, I. Tsigelny Ed, International University Line: La Jolla, California, 2002

[6] A. Marin, J.Pothier, K. Zimmermann, J-F. Gibrat, FROST: A Filter Based Recognition Method, Proteins, 2002 Dec 1; 49(4): 493-509

[7] J.C. Setubal, J. Meidanis, Introduction to computational molecular biology, 1997, Chapter 8: 252-259, Brooks/Cole Publishing Company, 511 Forest Lodge Road, Pacific Grove, CA 93950

[8] N. Yanev and R. Andonov, Parallel Divide and Conquer Approach for the Protein Threading Problem, Concurrency and Computation: Practice and Experience, 2004; 16: 1-14

[9] D. Fischer, http://www.cs.bgu.ac.il/ dfishcer/CAFASP3/, Dec 2002

[10] R: A language and environment for statistical computing, R Foundation for Statistical Computing, Vienna, Austria, 2004, http://www.R-project.org

[11] C. Branden and J. Tooze. *Introduction to protein structure*. Garland Publishing, 1999.

[12] H. J. Greenberg, W. E. Hart, and G. Lancia. Opportunities for combinatorial optimization in computational biology. *INFORMS Journal on Computing*, 16(3), 2004.

[13] T. Head-Gordon and J. C. Wooley. Computational challenges in structural and functional genomics. *IBM Systems Journal*, 40:265–296, 2001.

[14] T. Lengauer. Computational biology at the beginning of the post-genomic era. In R. Wilhelm, editor, *Informatics: 10 Years Back - 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 341–355. Springer-Verlag, 2001.

[15] C. Chothia. Proteins. one thousand families for the molecular biologist. *Nature*, 357:543–544, 1992.

[16] R.H. Lathrop and T.F. Smith. Global optimum protein threading with gapped alignment and empirical pair potentials. *J. Mol. Biol.*, 255:641–665, 1996.

[17] J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. Protein structure prediction by linear programming. In *Proceedings of The 7th Pacific Symposium on Biocomputing (PSB)*, pages 264–275, 2003.

[18] J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. RAPTOR: optimal protein threading by linear programming. *Journal of Bioinformatics and Computational Biology*, 1(1):95–118, 2003.

[19] Y. Xu and D. Xu. Protein threading using PROSPECT: design and evaluation. *Proteins: Structure, Function, and Genetics*, 40:343–354, 2000.

[20] Y. Xu, D. Xu, and E. C. Uberbacher. An efficient computational method for globally optimal threading. *Journal of Computational Biology*, 5(3):597–614, 1998.

[21] J. Xu. Speedup LP approach to protein threading via graph reduction. In *Proceedings of WABI 2003: Third Workshop on Algorithms in Bioinformatics*, volume 2812 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 2003.

[22] C.A Orengo, T. D. Jones, and J. M. Thornton. Protein superfamilies and domain superfolds. *Nature*, 372:631–634, 1994.

Template	DFROST	CPU tot	Cpu av	NAli
1BGLA0	15455	107569	113	945
1ALO_0	9565	96579	97	995
1CXSA0	5988	55808	58	960
1DIK_0	4506	46855	47	977
(...)				
1AYL_0	1807	18961	19	973
1EUT_0	1753	18883	18	995
1CTN_0	1535	16670	16	1000
1ECL_0	1439	15589	16	953
(...)				
1LYLA0	782	8335	8	990
1BIF_0	657	7129	7	948
1AD3A0	629	6669	6	1000
1DNPA0	776	6580	6	960

Table 1. Execution times in seconds for calculating the 3D score distributions. The templates for which the distributions are calculated are listed in the first column. The second column gives the parallel time (the execution time for the master) on a cluster of 12 processors. The third column shows the CPU sequential time (obtained by adding the CPU times from the slaves). The fourth column reports the average CPU time per threading and the last column shows the actual number of sequences that have been threaded to calculate the distributions. The value of the granularity was fixed to 10.

	Nb Sol	NAli	Min	Q_1	Med	Mean	Q_3	Max
1BGLA0	$5.4\ 10^{27}$	55	0.95	0.96	0.98	0.97	0.98	1.02
	$1.2\ 10^{35}$	56	0.95	0.96	0.97	0.97	0.98	1.01
	$3.5\ 10^{58}$	192	35.6	39.9	42.2	45.2	50.0	73.2
	$1.3\ 10^{70}$	199	102.4	116.3	131.0	145.7	164.6	510.0
	$6.6\ 10^{77}$	150	203.8	229.7	252.6	291.7	327.5	797.4
1QBA_0	$1.6\ 10^{3}$	58	1.82	1.83	1.83	1.84	1.84	1.89
	$8.3\ 10^{37}$	57	1.82	1.83	1.83	1.84	1.84	1.89
	$5.2\ 10^{57}$	197	27.1	30.2	32.5	36.3	39.8	76.6
	$2.8\ 10^{68}$	200	68.4	77.5	86.9	101.4	116.0	354.8
	$7.2\ 10^{75}$	200	130.1	154.7	178.3	207.0	239.8	789.8
1ALO_0	$3.1\ 10^{33}$	57	0.85	0.87	0.87	0.87	0.88	0.89
	$6.0\ 10^{33}$	57	0.85	0.86	0.87	0.87	0.87	0.89
	$2.5\ 10^{57}$	190	25.8	29.3	36.1	40.8	46.7	135.2
	$1.6\ 10^{69}$	200	67.4	86.3	113.2	123.2	134.8	397.6
	$1.3\ 10^{77}$	200	139.9	175.7	231.0	262.2	303.4	735.0
1YGE_0	$3.4\ 10^{23}$	61	0.39	0.40	0.41	0.41	0.41	0.43
	$2.8\ 10^{45}$	59	0.40	0.41	0.41	0.41	0.42	0.42
	$2.1\ 10^{55}$	192	34.8	39.9	43.1	47.5	48.9	139.8
	$6.5\ 10^{61}$	173	71.2	80.5	89.5	102.0	115.9	365.1
	$4.4\ 10^{66}$	199	120.2	138.5	158.3	178.2	208.9	443.7

Table 2. Sequential times in seconds for computing the 3D score distributions of four templates selected for their “difficulty” (search space size). For a given template the 5 rows represent alignment of sets of non related sequences having length respectively equal to: -30%, -15%, 0%, +15%, +30% of the template length. Nb Sol is the number of possible alignments that can be generated with the sequences and the template. This gives an indication of the difficulty of the problem to solve. NAli is the number of alignments (sequences) in the corresponding set. The last six columns report diverse running time characteristics obtained when aligning the set of sequences with the corresponding 3D structure: Min is the minimum value, Q_1 is the time at the 1st quartile position, Med. is the time at the median position, Mean is the average time, Q_3 is the time at the 3rd quartile position and Max is the maximum value.

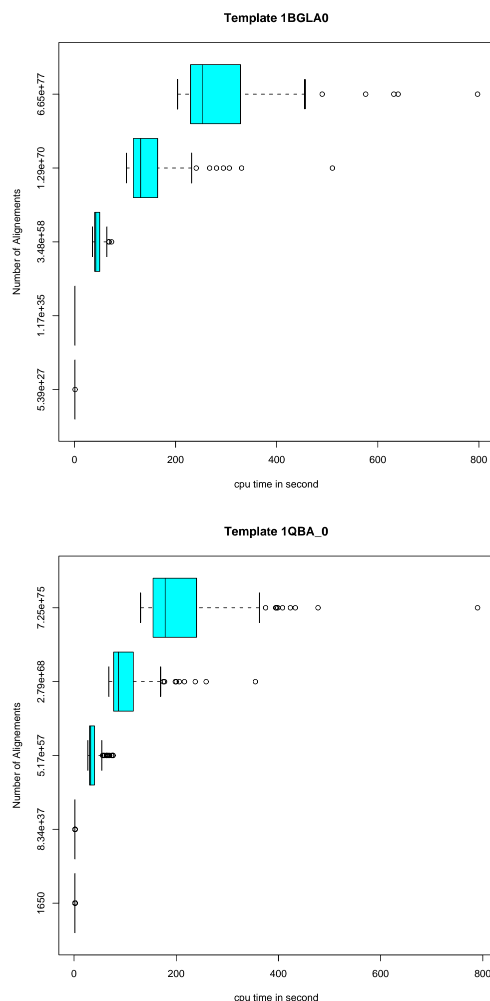

Figure 3. The two templates for which the distribution calculations are the most computer intensive, 1BGLA0 and 1QBA_0, are selected from table 2 and the corresponding boxplots of the running time distributions are plotted using the statistical package R [10]. The box contains the middle half of the data, i.e., the left and right ends of the box are at the lower and upper quartiles and the middle line corresponds to the median of the distribution. Vertical lines, usually called “whiskers”, go left and right from the box to the extreme of the data (here defined as 1.5 times the interquartile range). Points outside the whisker lines are plotted by themselves. Note that the distribution is not symmetric and exhibits a heavy tail for longer CPU times.