Sequential and Parallel Implementation of a Constraint-based Algorithm for Searching Protein Structures

Sascha Hunold #, Thomas Rauber #, Georg Wille *

#Department of Mathematics and Physics
University of Bayreuth, Germany
{hunold,rauber}@uni-bayreuth.de

*Institut für Biophysik
Goethe-Universität Frankfurt am Main, Germany
wille@biophysik.org

***Abstract*— Data mining in biological structure libraries can be a powerful tool to better understand biochemical processes. This article introduces the LISA algorithm which enables the researcher to search substructures in PDB files describing the 3D structure of protein molecules. The use of constraints such as atomic distances, torsion angles, or the distance of residues within the linear amino acid sequence, allows for great flexibility in defining and searching specific structures, which could not be found with other tools.**

Data mining in biological databases, e.g. scanning the entire PDB database for structures that match user-defined criteria, is a massively computation-intensive task. Thus, we present a parallel implementation of LISA and show that the algorithm achieves good parallel efficiency on homogeneous clusters.

I. Introduction

As of March 2007, the Protein Data Bank[1] contains approximately 42000 3D structures of biological macromolecules [1]. Well over 100 new entries are being added each week, and this rate of growth is still accelerating. Increasingly, these structures result from structural genomics approaches with the consequence that often the structure of a protein is known before its function.

Automated analysis tools are needed to make full use of the vast amount of information contained within these structures, and many are available today, e.g. for sequence comparison, secondary structure analysis or fold recognition and classification. Tools are also available to search the PDB for structures matching certain geometric criteria [2], [3], but these are not yet as flexible as would be desirable, e.g. when asking questions about sterical requirements within the active sites of enzymes.

We here introduce a new application, LISA, which allows the user to define a search pattern consisting of chemical, geometrical, and protein sequence restraints, thereby allowing for more flexibility than previous approaches involving more or less static 3D point pattern matching techniques, which it could nevertheless emulate. LISA can be used to find such diverse targets as secondary structure elements like α-helices, the catalytic triad of serin proteases, conformationally strained cofactors, specifically liganded metal ions, or amino acid clusters of certain defined compositions.

The price for this improved generality is a less then optimal performance in any particular case, i.e., for each specific problem a program could be written that finds its structural matches in less time. While LISA already tries to create a search tree that takes as little time as possible to traverse, run times for certain problems can still be long.

This paper is organized as follows: Section II gives an overview of the sequential structure search algorithm called LISA. In Section III we present a parallel implementation which is evaluated in IV. Section V discusses related work and Section VI concludes the article.

II. The sequential structure search algorithm – LISA

In this section, the LISA algorithm is introduced by a description of the primary goals of LISA. We also discuss performance-critical decisions which are crucial for obtaining high performance in the sequential case.

A. Motivation

The basic idea behind LISA is to search a user-specified structure in one or more PDB files. The user defines the search pattern by passing a query file that contains the target atoms and several constraints which have to be satisfied. An overview of the software modules of LISA is shown in Figure 1. The LISA program takes two input arguments, the query definition file and a PDB file. The LISA parser reads the query file and the LISA main program starts searching the specified structure in the PDB file. A matching sub-structure can be saved into a new PDB file.

Since there are numerous libraries and tools available for Bioinformatics, we wanted to build upon existing software like

[1]http://www.pdb.org

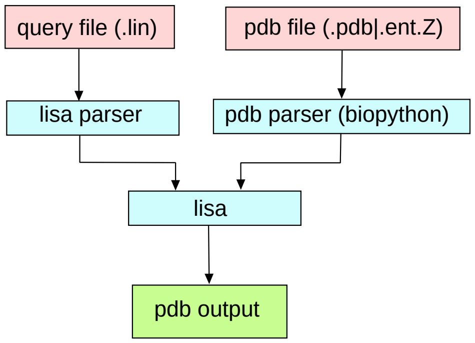

Fig. 1. Data flow of LISA.

Biopython. The Biopython Project[2] is a collection of freely available Python libraries and tools for computational molecular biology [4]. Biopython combines the rapid-development approach of Python but also provides a very mature API. It comes with out-of-the-box PDB file support containing a PDB parser, linear algebra functions for PDB atoms, a PDB file writer, and more. Moreover, Biopython provides an implementation of the kd-tree data structure which allowed us to build a fast distance-constraint filter (neighbor search algorithm [5]).

B. Query language of LISA

This section introduces the query language which is used in LISA. In a query file the user specifies the protein structure in which he is interested. Query files are basically separated into two section. One is the definition of the atoms that a PDB substructure should contain. In the other section, the user defines the constraints between the atoms.

Atoms can be defined by their name and optionally by their residue name. The name of an atom or the residue name can be defined as regular expressions as shown below. The variable name 'id' is a user-defined label for an atom which is used as unique identifier within the entire query file.

- `<id>.def.atom=<regex>`
  This line is used to define the name of the atoms.
- `<id>.def.residue=<regex>`
  This command defines the name of the residue for atom `id`.

The following list contains the currently supported constraints.

- `dist.<id1>.<id2>=<value>`
  Specifies a distant constraint between the atoms `id1` and `id2`.
- `dist_from_plane.<id1>.<id2>.<id3>.<id4>`
  `=<value>`
  Is used to define the distance of atom `id4` from the plane which is defined by the other three atom IDs.
- `angle.<id1>.<id2>.<id3>=<value>`
  Defines a constraint which checks if the angle between the two vectors`id1`→`id2` and `id2`→`id3` is "`value`" degrees.
- `torsion.<id1>.<id2>.<id3>.<id4>=<value>`
  Similar to angle constraint. It defines the angle between the planes `id1-id2-id3` and `id2-id3-id4`.
- `diff_resnum.<id1><id2>=value`
  Defines the distance of residues within the linear amino acid sequence of the protein to which the two different atoms `id1` and `id2` belong.
- `same_chain=[ <list of ids> ]`
  This constraint is used to define the atom IDs which should be part of the same chain.
- `same_residue=[ <list of ids> ]`
  Similar to `same_chain`, this constraint defines which of the atoms must belong to the same residue.

An example LISA query file is given in Figure 2. This query defines five query atoms which are nuc, his_n1, his_n2, asp_o, and his_cg. As mentioned above, these names can be chosen arbitrarily and are used as identifiers within a query file.

Each of these query atoms defines a PDB atom type, e.g. `nuc` is a placeholder for an `OG`-atom in residue '`SER`'. The remaining query file contains the definition of the constraints. Line 18, for example, fixes the distance between an `OG` in '`SER`' and an `N` in '`HIS`' to 2.8 Å. It is also possible to set up a tolerance range for each constraint, e.g. line 26 specifies that the distance of `asp_o` to the plane of '`his_n1-his_n2-his_cg`' may be bigger than zero (line 25) but should be at most 1 Å ($0 <= d <= 1$).

C. Algorithmic details of LISA

The algorithmic structure of LISA is presented in Algorithm 1. As mentioned in Section II-A LISA gets a list of query atoms and a list of PDB atoms from the parser. The defined constraints create virtual dependencies between query atoms, i.e. if two query atoms are part of a distance constraint, then we call them dependent atoms. Such a dependency is later utilized to make a good choice when selecting a query atom for the next computational step (building the search tree). The dependency graph is used to order the query atoms (line 3). This is done by using heuristics which is described later in this article.

The algorithm starts off with an empty allocation (statement 4). In our notation, an allocation is a mapping of query atoms to PDB atoms. A candidate list is generated for the first query atom (q_0) in the atom list. This list contains all PDB atoms which match the name and residue definition of the q_0. The candidate list of q_0 and the empty allocation is then passed as arguments to the recursive function `check_allocation`. This function first checks if the current allocation matches the specification of the query file, i.e. it checks that all defined constraints which can be applied to PDB atoms in the allocation are satisfied. Only if the allocation matches these constraints, the allocation is extended by one atom. This procedure of cutting the search tree as early as possible is the key for a fast branch-and-bound approach. A match is found when the allocation contains a mapping for each query atom to exactly one PDB atom. If the allocation does not contain a mapping for each query atom, the algorithm extends the allocation by one atom which is a PDB atom of the candidate list (line 6 in function `check_allocation`). After the PDB atom is fixed, the PDB atom candidate list for the next query atom is generated. To create a matching allocation,

[2]`www.biopython.org`

```
nuc.def.atom=OG
nuc.def.residue=SER

his_n1.def.atom=N..
his_n1.def.residue=HIS

his_n2.def.atom=N..
his_n2.def.residue=HIS

asp_o.def.atom=OD.
asp_o.def.residue=ASP

his_cg.def.atom=CG
his_cg.def.residue=HIS

tolerance.dist = 0.2

dist.nuc.his_n1=2.80
dist.his_n1.his_n2=2.12
dist.his_n2.asp_o=2.65

dist_from_plane.his_n1.his_n2.his_cg.nuc = 0.0
tolerance.dist_from_plane.his_n1.his_n2.his_cg.nuc = 1

dist_from_plane.his_n1.his_n2.his_cg.asp_o = 0.0
tolerance.dist_from_plane.his_n1.his_n2.his_cg.asp_o = 1

constraint.same_residue=[his_n1, his_n2, his_cg]
```

Fig. 2. Example query file for searching serine proteases.

the LISA algorithm only needs to consider all the PDB atom candidates which match the user-defined constraints. Thus, the PDB candidate list (`next_candidate_list`) is filtered by applying all constraint rules. In general, filtering PDB candidates leads to a much shorter list of PDB candidates which results in a faster search due to a reduction of the width of the search tree.

The function `check_allocation` is then called recursively with the new allocation and the filtered candidate list for the next query atom as parameters.

An important performance factor is the order in which the query atoms are visited/traversed. The following section discusses our approach for ordering the query atoms.

D. Building the search tree efficiently

The speed of the implementation strongly depends on the search tree. Since we use a branch-and-bound-strategy it is very important to cut off as many leaves as possible and as early as possible. In particular, whenever a constraint can be applied to the current atom allocation this constraint rule has to be exploited for reducing the number of nodes in the search tree.

Another important criterion is the structure of the search tree, i.e. the configuration of each level. To find a substructure of a protein, e.g. A_0, A_1, A_2 (A_i are arbitrary atoms), it does not matter in which order the algorithm checks the atoms, e.g. A_0-A_1-A_2 or A_1-A_2-A_0. This means that the result is independent of the search order. However, the search speed does depend on the search order. In Figure 3 shows two different search trees which could be generated for one query. In the tree on the left-hand side, 100×1000 nodes (number of possible checks) would be cut off if stepping down into node A_1 was avoided. In the right-hand tree, the algorithm could

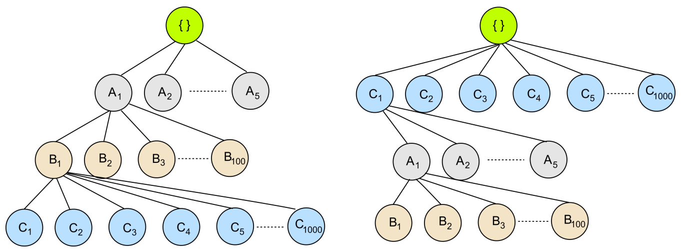

Fig. 3. Two possible search trees for the same query file.

cut off at most 5×100 nodes below C_1. Thus, for reducing the search time the number of nodes per level should increase with tree depth.

E. Fast reduction of PDB atom candidates

To achieve a fast search (quick response time), the primary objective of the algorithm must be to minimize the number of nodes at each tree level. As shown above, LISA uses constraint-based filters to reduce the number of candidates for an allocation of PDB atoms.

The distance constraint is one of the possibilities to filter the candidate list. In order to avoid computing the distance between each pair of PDB atoms A_i and A_j which match a corresponding query atom pair Q_i and Q_j, LISA exploits the neighbor search algorithm that is bundled with Biopython. The neighbor search returns all PDB atoms which are located in the sphere with center A_i and the radius which is defined by the distance constraint. Internally, the neighbor search stores PDB atoms in a *kd*-tree data structure which is an efficient data structure for range queries. The neighbor search of LISA proceeds as follows. For a distance constraint d_j between atoms Q_i and Q_j, LISA uses the neighbor search algorithm to

Algorithm 1 LISA

```
1: input ⇐ { $PQ$ – list of atoms in PDB file, $QQ$ – list of atoms in query file }
2: build dependency graph of atoms in $QQ$
3: $QQ$ ⇐ create ordered list of atoms in $QQ$        // defines structure of search tree
4: allocation ⇐ {}
5: $q_0 \Leftarrow QQ[0]$
6: candidate_list ⇐ get_PDB_candidates( $q_0$ )
7: check_allocation( allocation, candidate_list, 0 )

function check_allocation( allocation, candidate_list, query_atom_id )
1: if is_match(allocation) then
2:    if length(allocation) == length($QQ$) then
3:       print "match found" + allocation
4:    else
5:       for all PDB_atom in candidate_list do
6:          allocation[ query_atom_id ] ⇐ PDB_atom
7:          next_query_atom ⇐ $QQ$[ query_atom_id + 1 ]
8:          next_candidate_list ⇐ get_PDB_candidates( next_query_atom )
9:          next_candidate_list ⇐ filter_candidate_list_by_constraints(
                      next_candidate_list, query_atom, next_query_atom )
10:         check_allocation( allocation, next_candidate_list, query_atom_id + 1)
11:      end for
12:   end if
13: end if
```

determine the PDB atoms which are in the sphere with center A_i and radius $d_j +$tolerance. In a second step, the PDB atoms which are located in the sphere at A_i and radius $d_j -$tolerance and which do not match type Q_j are erased from the list. The remaining PDB atoms are the possible candidates for A_j. This procedure leads to a faster reduction of the candidate list by exploiting the distance constraints.

III. Parallel framework for LISA

The sequential algorithm takes a lot of computation time if user-defined constraints do not help to reduce the list of PDB candidates for a given query atom. Furthermore, the entire PDB database contains about 7 GB of protein data. Hence, a parallel version of LISA would reduce the search time of protein structures tremendously which could make the program more attractive for the daily use by researchers.

The remaining section describes the selection of the target platform and the approach for parallelizing LISA.

a) The granularity question: As in most parallel implementations of algorithms, the granularity of the parallel task plays an important role to achieve good parallel speedup.

There are basically two ways to parallelize the sequential LISA algorithm:

1) A single processor processes a single LISA request for multiple PDB files. The number of PDB files is variable. Each participating processors receives a portion of the data set (PDB files).
2) All processors work simultaneously on searching a query pattern in one PDB file.

The second option has a higher granularity, but it is a real challenge to find a well-scalable solution to this problem. It is difficult to equally balance the workload of the search considering the irregular access pattern in the tree. The prototype implementation of this option did not show good parallel scalability for many PDB files because of the extra communication for exchanging sub-results.

Therefore, this article presents the parallel implementation of option (1).

b) Choosing the target platform: We had two major concerns for the parallel implementation of LISA; it should be well-scalable, and it should also be easy to install and easy to use for researchers. We decided to use PC clusters as target platform. At first glance, the problem seems to be embarrassingly parallel which raises the question of grid applicability. The major drawback, however, is the size of the PDB database. For a concurrent computation in the grid, data from the PDB database has to be fetched by the computation nodes which is very costly. Another problem is the use of Python/Biopython for the sequential LISA implementation. Python is a good choice for rapid development but it is not completely platform-independent since most of the used Python modules (numpy, kdtree) are implemented in C. In contrast, in a single cluster environment the availability of the necessary libraries and also a shared access to the PDB database from the computation nodes can be easily ensured.

c) The parallel algorithm in depth: The parallel algorithm uses a master/slave approach to obtain the result. The master node contains the information about all PDB files and distributes files among the requesting slaves. In the current implementation, the master and the slave nodes share one file system. Hence, the master node only distributes the names of the PDB files. On the one hand, the communication amount of the master slave is massively reduced. On the other hand, the IO bandwidth of the shared file system is a potential bottleneck if all slave nodes are accessing the PDB database at the same time.

A centralized approach (master and slaves) has to fulfil two criteria to be efficient in such a cluster environment. First, the time for executing a task must be large enough to avoid steady requests by the clients (slaves). For slaves, the time to receive a task should be as short as possible to achieve good parallel efficiency.

In this approach, the master node is responsible for finding well-suited task sizes to obtain a good load balance and hence, a good parallel speedup. As mentioned before, the minimum task granularity in our implementation is one PDB file. The question is how many and which of the available PDB files should be sent to a requesting client. After experimenting with different scheduling strategies we observed that a guided self-scheduling approach leads to best results [6]. A first approach for filling tasks is to select n random files of the PDB database, where n is determined by guided self-scheduling. In this case, clients which request data within a short time frame would retrieve almost the same number of PDB files. The problem is that the collective file size of both sets may strongly differ. Since the computation time for one file is in general proportional to its file size, this approach will lead to work imbalance. Hence, a task is not determined simply by the number of remaining files but by considering their file sizes as well.

Thus, the master node holds a list of PDB files sorted by file size. Whenever a client is requesting PDB files, the master node determines the task size by using guided self-scheduling considering the overall size of the PDB database and the size of the remaining PDB files. After computing the task size, the master node has to fill the task with PDB files and has to ensure that the task size is not exceeded. For efficiency reasons, we introduce two additional parameters. One is the minimum task size MTS, the other is the task filling factor TFF. The MTS is simply a lower bound for the task size which is introduced to avoid that the guided self-scheduling approach creates too many very small tasks at the end of the computation. The TFF is used to stop searching when the task has been filled up to a certain amount of data. The parameter TTF prevents from iterating over the entire list of remaining PDB files once a certain task capacity has already been reached (e.g. 90%).

d) Implementation details: The parallel algorithm was implemented using MPI and Python. The Python implementations of MPI are neither standardized nor complete. However, bindings exist and it was possible to find workarounds for problems which were caused by missing functionality (like the absence of non-blocking directives). We decided in favor of PyMPI [7] which has shown good communication performance and compatibility with the installed MPICH in evaluation tests.

IV. Experimental results

The experiments have been carried out on a 64 processor Opteron-cluster running Linux (2.6.11). The nodes were connected by a GBit Ethernet switch. We used MPICH 1.2.7, Python 2.4.4, Biopython 1.42, and pyMPI 2.4b4.

In the first experiment we tested the parallel algorithm by searching serine proteases in a specific subset of the PDB database. For this experiment, we downloaded all proteases with enzyme classification 3.4.x.y from `www.pdb.org` with close homologues removed at the level of 95% sequence identity. At the time of the experiments, this PDB subset contained about 290 proteases (148 MB in total). This subset was mainly used because

1) a set of 290 files has enough parallel potential for evaluation purposes, and
2) the results can be validated by hand, and
3) it is important to know in advance that the subset must contain several matches if the algorithm works correctly.

Figure 4 illustrates the search pattern by presenting a matching sub-structure for the query file of Figure 2.

The speedup of the parallel version of LISA which has been obtained for searching the protease subset of 290 files is shown in Figure 5.

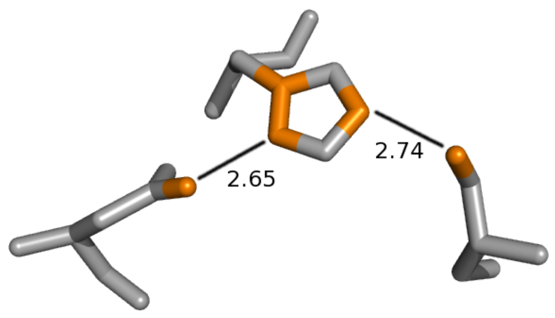

Fig. 4. Example: matching structure (serine protease) in `4cha.pdb`.

In this experiment, the time for searching the subset has dropped from 148 seconds on one processor to about 8 seconds on 64 processors. Even though there is a significant time gain the parallel speedup and therefore the parallel efficiency is not optimal. This is due to fact, that the subset of PDB files used in this test is relatively small, and so the inherited overhead (e.g. the task retrieval contention overhead when all hosts are requesting tasks at program start) has a bigger impact on the overall performance of the parallel implementation.

In another experiment, we evaluated the parallel efficiency and applicability by performing a PDB query against the entire PDB database. The entire PDB library contains about 42000

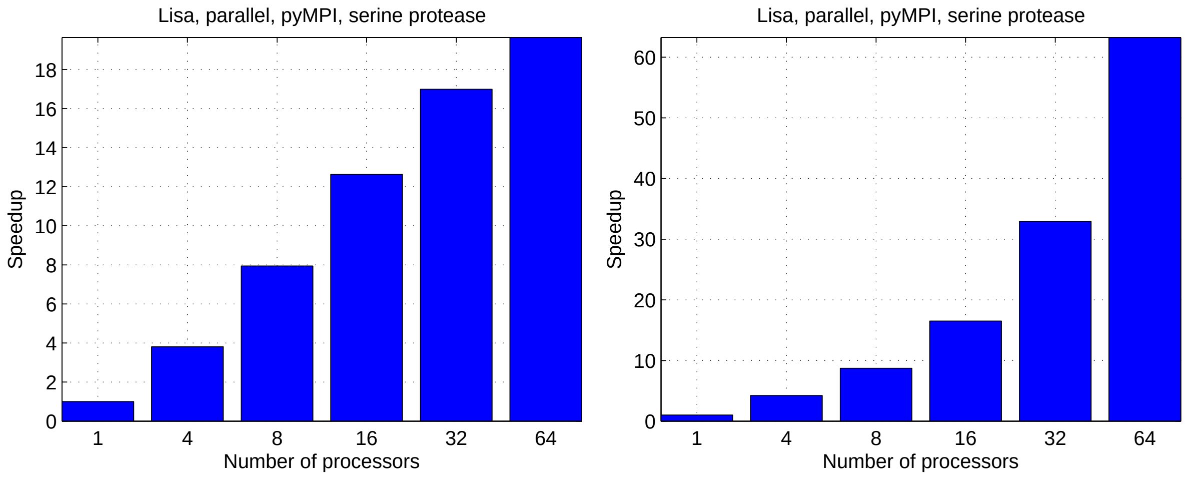

Fig. 5. Speedup of the parallel LISA algorithm for different data sets, left: subset of 290 PDB files, right: entire PDB database

structures and requires about 6.9 GB disk space. The following set of parameters has been used in this experiment:

- minimum task size $(MTT) = 1024\,\text{kB}$,
- task filling factor $(TFF) = 0.9$.

The chart on the right hand side of Figure 5 shows the speedup obtained for searching the proteins in the entire PDB library which match the serine protease pattern. In this experiment, the time for processing the query was reduced from more than 500 minutes on a single processor to about 8 minutes on 64 processors. For up to 32 processors the algorithm achieves almost a perfect speedup and for 64 processors the speedup is still bigger than 63. According to these numbers we can say that the parallel implementation of the LISA algorithm scales well for homogenous PC clusters. This high parallel efficiency can only be reached with a good scheduling strategy. During the experiments we have seen that the guided self-scheduling approach as described above (using the file size as basic unit) leads to best results (speedup).

V. Related work

Previous work has mostly been focussed on the matching of given protein structures to more or less rigid 3D sets of atoms or "virtual atoms", the latter derived from amino acid functional groups or even representing entire amino acids (e.g. [2], [8], [9] , [10]). To our knowledge, only one approach, JESS, would have eventually allowed the inclusion of arbitrary geometric restraints in the search pattern ([3]), but at the moment this project seems to be dormant, without having seen the completion of the essential extensible search pattern generator.

Therefore, no program is currently available that allows the search for easily defined flexible patterns like an arrangement of atoms A-B-C-D, where the torsion angle around the central connection is completely free - even though such cases with strict local geometry restraints (like bond lengths A-B, B-C and C-D, or angles A-B-C and B-C-D), but relaxed remote restraints (like the spatial relation of A and D in the above example), are quite relevant for chemical reaction mechanisms.

VI. Conclusions

In this article, we have presented a novel approach for searching protein sub-structures based on constraints. We have shown that LISA achieves good sequential performance and have also presented a parallel implementation of LISA. The experimental results have shown that the parallel version of LISA obtains a good parallel speedup for large PDB data sets on homogeneous PC clusters.

For the future, we plan to extend the LISA query language with further constraint types within a new plug-in framework that also enables user-defined constraints. Moreover, we will make the serial and parallel version available for public use.

References

[1] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The Protein Data Bank," *Nucleic Acids Res*, vol. 28, no. 1, pp. 235–242, January 2000. [Online]. Available: http://nar.oxfordjournals.org/cgi/content/abstract/28/1/235

[2] A. C. Wallace, N. Borkakoti, and J. M. Thornton, "TESS: a geometric hashing algorithm for deriving 3D coordinate templates for searching structural databases. Application to enzyme active sites," *Protein Science*, vol. 6, no. 11, pp. 2308–2323, November 1997.

[3] J. A. Barker and J. M. Thornton, "An algorithm for constraint-based structural template matching: application to 3D templates with statistical analysis," *Bioinformatics*, vol. 19, no. 13, pp. 1644–1649, 2003.

[4] B. Chapman and J. Chang, "Biopython: Python tools for computational biology," *SIGBIO Newsl.*, vol. 20, no. 2, pp. 15–19, 2000.

[5] T. Hamelryck and B. Manderick, "PDB file parser and structure class implemented in Python." *Bioinformatics*, vol. 19, no. 17, pp. 2308–2310, 2003.

[6] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers." *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1425–1439, 1987.

[7] P. Miller, "pyMPI An Introduction to Parallel Python Using MPI," http://www.llnl.gov/computing/develop/python/pyMPI.pdf, September 2002.

[8] G. J. Kleywegt, "Recognition of spatial motifs in protein structures," *Journal of Molecular Biology*, vol. 285, no. 4, pp. 1887–1897, 1999.

[9] M. Jambon, O. Andrieu, C. Combet, G. Deleage, F. Delfaud, and C. Geourjon, "The SuMo server: 3D search for protein functional sites," *Bioinformatics*, vol. 21, no. 20, pp. 3929–3930, October 2005. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bti645

[10] J.-C. Nebel, "Generation of 3D templates of active sites of proteins with rigid prosthetic groups," *Bioinformatics*, vol. 22, no. 10, pp. 1183–1189, 2006.