PocketMatch (version 2.0): A parallel algorithm for the detection of structural similarities between protein ligand binding-sites

Deepesh Nagarajan (Presenting author)
Graduate Student, Biochemistry Department.
Indian Institute of Science,
Bangalore, India.
deepesh@mrdg.iisc.ernet.in

Dr. Nagasuma Chandra (Corresponding author)
Associate Professor, Biochemistry Department.
Indian Institute of Science,
Bangalore, India.
nchandra@biochem.iisc.ernet.in

Abstract **--- Knowledge of protein-ligand interactions is essential to understand several biological processes and important for applications ranging from understanding protein function to drug discovery and protein engineering. Here, we describe an algorithm for the comparison of three-dimensional ligand-binding sites in protein structures. A previously described algorithm, PocketMatch (version 1.0) is optimised, expanded, and MPI-enabled for parallel execution. PocketMatch (version 2.0) rapidly quantifies binding-site similarity based on structural descriptors such as residue nature and interatomic distances. Atomic-scale alignments may also be obtained from amino acid residue pairings generated. It allows an end-user to compute database-wide, all-to-all comparisons in a matter of hours. The use of our algorithm on a sample dataset, performance-analysis, and annotated source code is also included.**

Index Terms **--- Structural bioinformatics, 3D sub-structure matching, molecular recognition, drug-discovery.**

I. INTRODUCTION

Biology is increasingly becoming a data-driven science. A fundamental requirement of contemporary biology involves recognizing similarities and deriving functional relationships among protein molecules. Molecular machines underlying biological processes are understood at atomic level detail in a number of cases. Structural bioinformatics serves as an important bridge area that transforms experimentally derived protein structures to biological knowledge for all related molecules. The exponential increase in both protein sequence and 3D structural data has increased the need for high-throughput, accurate data analysis tools. Such tools, including protein structural comparison algorithms, form the rapidly-growing foundation of this discipline. Structural comparisons help better understand the nature and function of macromolecules. The 'meaning' of a protein, or its biological function, can be inferred by comparisons of functional regions in protein molecules. Ligand binding sites (pockets) contain a few amino acid residues critical to the function of the entire protein. Understanding pockets using techniques including database-wide comparisons can yield insights into diverse processes such as enzyme kinetics, molecular mechanics, drug resistance, and protein design.

Two different approaches exist for the creation of pocket comparison algorithms. The first involves computationally intensive but accurate pocket superimpositions and root-mean-squared- distance (RMSD) calculations. The second approach involves a high-throughput but approximate comparison of quantifiable 'factors' that influence the nature of a pocket. The exponential growth of protein and ligand databases has led to far more raw-data available than can be processed by current methods. Therefore, faster algorithmic performance is always desirable. Here we combine the best of both approaches. Initially, a high throughput screening of shape descriptors is carried out, followed by the generation of residue pairings that can yield atomic-level alignments.

Available programs for pocket comparison include *SitesBase*[1], that employs a *geometric hashing* algorithm. *PINTS*[2] detects local similarities using a depth-first traversal algorithm. *SPASM, RIGOR*[3] compare residue-distributions from binding-site centroids. *CavBase*[4] identifies maximal shared sub-graphs between atom-points.

PocketMatch (version 2.0) is an algorithmic extension, optimisation, and low-level reimplementation of a previously described algorithm: PocketMatch[5][6] (version 1.0). The original version compares *shape descriptors*, or chemically tagged interatomic distances and returns a score that quantifies overall extent of similarity between pockets. The algorithm suffered two drawbacks. Firstly, although designed for high-throughput data analysis, it was implemented in multiple high-level programming languages, greatly increasing memory access times. Secondly, it did not produce residue pairings, and was incapable of structurally superimposing pockets.

We have overcome both these drawbacks. Firstly, the algorithm was re-implemented in the C programming language, and parallelised using MPI libraries for C. This greatly improved performance and allowed for easy scalability on any multiprocessor system. Secondly, an optional module (Sphinx) allows end-users to generate residue pairings for structural superimpositions, at the cost of moderately increased run-time. Addition of this feature

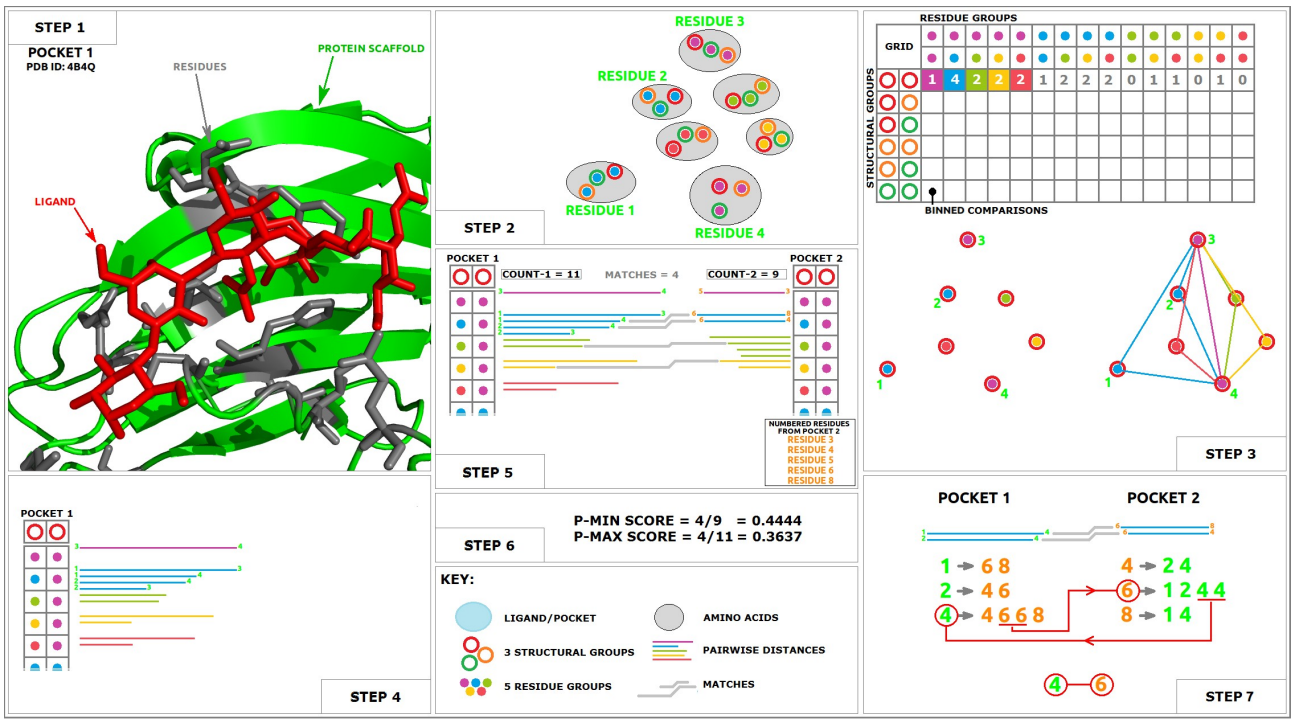

Fig. 1. The underlying PocketMatch algorithm involves conceptualising a pocket's structure as a network, followed by classification, binning, and matching distance elements. **Step 1-6:** The original algorithm (version1.0). Similarity scores quantify pocket structural relationships. **Step 7:** A newly introduced (version 2.0) module (Sphinx) also features pairwise-association mappings, which can be used for structural superimpositions.

is a major improvement, opening up huge protein sequence and structure databases to rapid and accurate computational analysis via structural superimpositions.

Performance evaluation of PocketMatch (version 2.0), and benchmarking against the original version was carried out using a sample dataset of vitamin B-12 binding pockets. Annotated source code can be found on our webserver (http://proline.physics.iisc.ernet.in/pocketmatch/).

II. THE POCKETMATCH(v2.0) ALGORITHM

The first 6 steps of PocketMatch (version 1.0) is described under **Steps 1-6**. For the sake of clarity. The Sphinx module of PocketMatch (version 2.0) is discussed under **Step 7.**

The underlying algorithm consists of conceptualising a pocket's structure as a network of *shape descriptors*, or interatomic distances classified according to amino-acid physicochemical properties. These classified distances are compared between pockets for the generation of a final similarity score, ranging from 0 (not similar) to 1 (very similar). The entire algorithm is illustrated *in Figure 1*.

Classification:
The 20 amino acids are then classified into five ***residue-groups*** depending on their physicochemical nature.
-Group 0 (A,V,I,L,G,P,M) : Contains aliphatic, non-polar, uncharged amino acids.
-Group 1 (K,R,H): Contains positively charged amino acids.
-Group 2 (D,E): Contains negatively charged amino acids.
-Group 3 (Y,F,W): Contains aromatic, uncharged amino acids.
-Group 4 (C,S,T,Q,N): Contains aliphatic, polar amino acids, containing a hydroxy or mercapto group.

The five residue-groups are sub-divided into three ***structural-groups***.
-Group 1: This group consists of all C-α atoms.
-Group 2: This group consists of all C-β atoms.
-Group 3: This group consists of all centroids of atoms in the amino acid's side chain. The centroid is computed as the mean 3-dimensional position of all side chain atoms.

A unique combination of one residue-group and one structural-group is referred to as a ***point-type***. There are (5*3 = 15) possible point-types.
Many individual ***points*** in space can belong to one point-type. Comparisons between point types are referred to as ***binned-comparisons***. 90 binned-comparisons are used by

the algorithm. The real-world distance between any two point-types is referred to as a ***pairwise-distance***.

Step 1: Extraction
The algorithm extracts all amino acid residues that are within 4Å of the ligand molecule. This distance was chosen because important interactions occur within a 4Å range. The input file is in .pdb format[7].

In *Figure 1, step 1*, a ligand is represented as a blue circle. Seven interacting amino acid residues (21 points) are primed for extraction.

Step 2: Implementation of classification
This classification scheme is implemented on the extracted binding-site amino acid residues.
In *Figure 1, step 2*, the pocket's amino acids are shown.

All residue-groups and structural-groups are represented.

Step 3: Pairwise-distance calculation
Pairwise distance calculations are carried out for all residue-groups and structural-groups. Since there are five residue-groups, there are ($5\mathbf{C}_2$+5 = 15) different combinations possible. All combinations are illustrated in *Figure 1*, *step 3* (grid, residue-groups). Since there are three structural-groups, there are ($3\mathbf{C}_2$+3 = 6) different combinations possible. All combinations are illustrated in *Figure 1*, *step 3* (grid, structural-groups).

The resulting number of combinations used here, for all residue-groups and structural-groups is (15*6 = 90) different binned-comparisons. All binned-comparisons used here are illustrated in *Figure 1*, *step 3* (grid, binned-comparisons). In *step 3*, an example calculation is carried out on a single structural-group (red). The number of pairwise-distances for all residue-group types is given along the first grid-line. The identity of each pairwise-distance is represented as coloured lines connecting different points.

Step 4: Pairwise-distance sorting
Pairwise-distances are then sorted in ascending order. This step reduces the complexity of a single run of the rate-limiting step (step 5) from **O**(*nlog(n)*) to **O**(*n*), greatly increasing performance. In *Figure 1*, *step 4*, an example sort is shown for a single residue-group (purple). The horizontal lines represent the magnitudes of each pairwise-distance. The sorted pairwise-distances are stored as *alpha-files* for further processing.

Step 5: Pairwise-distance matching
This is the rate-limiting step. Steps 1-4 will have to be run independently for different ligand-pockets before attempting step 5. A *greedy strategy* is then used to match different pairwise distances obtained from different pockets. The matching algorithm looks for pairwise-distances that differ by 0.5Å or less, comparing them in a sequential manner. If such pairwise distances are found, an occurrence of a ***match*** is noted. The pairwise-distances must be from the same point-type-comparison on different pockets. This portion of the algorithm is best described in C-like pseudo-code (Appendix*: Pseudo code*).

Step 6: P-min and P-max calculations
The values P-min and P-max are expressions of the degree of similarity between two pockets.
-P-min is the value obtained by dividing the number of matches by the total number of pairwise-distances of the smaller pocket.
-P-max is the value obtained by dividing the number of matches by the total number of pairwise distances of the larger pocket.
Conventionally, P-max is used for further calculations.

Step 7: The Sphinx module (version 2.0)
A further addition to this algorithm involved accounting for pairwise-distance based residue associations. Every pairwise distance is derived from two points, which may be atomic or centroid in nature. This association may be retained through the new implementation's pipeline by attaching two parent residue-numbers to every pairwise-distance. While calculating the number of matches, inter-pocket residue-number associations for matched pairwise distances can also be retained. If a given pocket-1 residue, ***X*** is maximally associated with a pocket-2 residue, ***Y***, and if *Y* reciprocates the maximal associationwith *X*, then a ***pairwise-association*** (residue pairing) occurs. These associations can then be used to corroborate the P-max score, or as input for superimposition-based RMSD calculations using Kabsch's algorithm[8]. This portion of the algorithm is best described in C-like pseudo-code (*Figure 2*).

```
variable: counter=0;
for(iᵗʰ binned-comparison out of 90)  {
  variable: a=0, b=0;
  variable: m=no. of pairwise-distance entries in pocket-1, in bin (i);
  variable: n=no. of pairwise-distance entries in pocket-2, in bin (i);
  vector: S₁[?]=all pairwise-distance entries in pocket-1;
  matrix: M₁[?][2]=residue-pairings corresponding to S₁ pairwise distances;
  vector: S₂=all pairwise-distance entries in pocket-2;
  matrix: M₂[?][2]=residue-pairings corresponding to S₂ pairwise distances;
  data-struct:  M₁pairings,  M₂pairings;
  while(a<m AND b<n)  {
    if(|S₁[a]-S₂[b]|<=0.5Å) then  {
      a match has occurred:
      a = a+1;
      counter = counter+1;
      M₁[a,1:2] is paired with M₂[b,1:2].The remember function can store
      these pairs in a dynamically allocated custom data-structure:
      remember(M₁[a,1] is-associated-with M₂[b,1:2] → M₁pairings[M₁[a,1]]);
      remember(M₁[a,2] is-associated-with M₂[b,1:2] → M₁pairings[M₁[a,2]]);
      remember(M₂[b,1] is-associated-with M₁[a,1:2] → M₂pairings[M₂[b,1]]);
      remember(M₂[b,2] is-associated-with M₁[a,1:2] → M₂pairings[M₂[b,2]]);
      M₂pairings[X] does not store a single value. Rather, it points to a
      dynamic list of all M₂ elements associated with M₁[X], and vice-versa.
      }
    else  {
      if(S₁[a]<S₂[b]) then a = a+1;
      else b = b+1;
      }
    break after S₁ or S₂ has run out of entries.
    }
  }
let m>n for this example
P-max = counter/m;
P-min = counter/n;

variable: residue1_store, residue2_store;
variable: pairwise_association=0;
for(iᵗʰ residue in length(M₁pairings))  {
   The mode function finds the most common M₂ residue-number
   associated with the iᵗʰ residue number of M₁ ,and vice-versa.
   residue1_store = mode(M₁pairings[i]);
   residue2_store = mode(M₂pairings[residue1_store]);
   if(i==residue2_store) then  {
    pairwise_association =  pairwise_association+1;
    output-pairwise-association;
    }
  }

Key:
Black text: Standard PocketMatch algorithm, implementation independent.
Green text: Sphinx module, available in version 2.0
Grey  text: Comments and annotations
```

Fig. 2. C-like pseudo-code that better describes step 5 and step 7 of the PocketMatch algorithm.

III. IMPLEMENTATION, COMPLEXITY AND PERFORMANCE ANALYSIS ON A SAMPLE DATASET

The new implementation was tested on a sample dataset and the performance analysis was carried out on a quad-core, 64-bit, Intel(R) Core(TM) i7-2600 CPU (3.40GHz). All analyses were carried out in a Linux environment (Ubuntu 10.10 x86), under the bash shell (GNU bash, version 4.1.5(1)-release (i686-pc-linux-gnu)) using the standard C/C++ compiler (gcc/g++ (Ubuntu/Linaro 4.4.4-14ubuntu5.1) 4.4.5.). The system *time* (*real* output) command was used for measuring program execution-time.

Performance analysis was carried out on a dataset of crystal structures of vitamin B12-binding proteins extracted from the Protein Data Bank (PDB). A dataset of 53 structures containing a total of 113 B12-binding pockets was grouped into 16 structural/functional classes based on sequence-similarity, SCOP classification[9], EC-

numbers and manual annotation based on associated literature (*Table I*). From each of these classes, one or more representative structures were chosen. Multiple representatives were chosen in cases where sequence similarity was low. A single B12-binding pocket was chosen from each representative structure, comprising all residues within 4.5Å of the B12 ligand. These 16 representatives were used for performance analysis tests of all PocketMatch implementations.

During alpha-file generation, the calculation of all pairwise-distances within a pocket is an $\mathbf{O}(m^2)$ problem, where m is the number of residues present within the pocket. $\mathbf{O}(m)$ binning and $\mathbf{O}(mlog(m))$ sorting complexity can be ignored. $\mathbf{O}(m^2)$ complexity applies to the creation of a single alpha file. In the downstream match function, multiple alpha files are compared with $\mathbf{O}(n^2)$ complexity, where n is the number of input files.

Class *(Figure 4)*	PDB code[7]	Name of protein crystallised	Sequence Similarity 95.00%	70.00%	50.00%	E.C. Number
	1DIO	DIOL DEHYDRATASE				4.2.1.28
	1E1C	methylmalonyl-CoA mutase (h244a mutant)				5.4.99.2
	1G64	ATP:corrinoid adenosyltransferase	1G64	1G64	1G64	2.5.1.17
	1I9C	glutamate mutase	1I9C	1I9C	1I9C	5.4.99.1
	1IWB	diol dehydratase	1IWB	1IWB	1IWB	4.2.1.28
	1IWP	glycerol dehydratase	1IWP			4.2.1.30
	1K7Y	MetH C-terminal fragment				2.1.1.13
	1K98	meth C-terminal fragment				2.1.1.13
	1MMF	glycerol dehydratase				4.2.1.30
	1REQ	methylmalonyl-CoA mutase	1REQ	1REQ		5.4.99.2
	1XRS	Lysine 5,6-aminomutase	1XRS	1XRS	1XRS	5.4.3.4
	2BB5	transcobalamin	2BB5			NA
	2BB6	transcobalamin	2BB6	2BB6	2BB6	NA
	2BBC	transcobalamin				NA
	2H9A	corrinoid iron-sulfur protein	2H9A	2H9A	2H9A	NA
	2PMV	human intrinsic factor	2PMV	2PMV	2PMV	NA
	2REQ	methylmalonyl-CoA mutase				5.4.99.2
	2V3N	transcobalamin				NA
	2V3P	transcobalamin				NA
	2XIJ	methylmalonyl-CoA mutase	2XIJ	2XIJ	2XIJ	5.4.99.2
	2XIQ	methylmalonyl-CoA mutase				5.4.99.2
	2YCL	corrinoid,iron-sulfur protein				NA
	3ABO	ethanolamine ammonia-lyase				4.3.1.7
	3ABQ	ethanolamine ammonia-lyase	3ABQ	3ABQ	3ABQ	4.3.1.7
	3ABR	ethanolamine ammonia-lyase				4.3.1.7
	3ANY	ethanolamine ammonia-lyase				4.3.1.7
	3AO0	ethanolamine ammonia-lyase				4.3.1.7
	3AUJ	diol dehydratase				4.2.1.28
	3BUL	meth C-terminal fragment	3BUL	3BUL	3BUL	2.1.1.13
	3CI1	PduO-type ATP:co(I)rrinoid adenosyltransferase				2.5.1.17
	3CI3	PduO-type ATP:co(I)rrinoid adenosyltransferase	3CI3	3CI3	3CI3	2.5.1.17
	3GAH	PduO-type ATP:corrinoid adenosyltransferase				2.5.1.17
	3GAI	PduO-type ATP:corrinoid adenosyltransferase				2.5.1.17
	3GAJ	PduO-type ATP:corrinoid adenosyltransferase				2.5.1.17
	3IV9	B12-dependent Methionine Synthase (MetH)				2.1.1.13
	3IVA	B12-dependent methionine synthase (MetH)				2.1.1.13
	3KOW	ornithine 4,5 aminomutase				5.4.3.5
	3KOX	ornithine 4,5 aminomutase				5.4.3.5
	3KOY	ornithine 4,5 aminomutase				5.4.3.5
	3KOZ	ornithine 4,5 aminomutase				5.4.3.5
	3KP0	ornithine 4,5 aminomutase				5.4.3.5
	3KP1	ornithine 4,5 aminomutase	3KP1	3KP1	3KP1	5.4.3.5
	3KQ4	intrinsic factor				NA
	3O0N	ribonucleotide reductase				1.17.4.1
	3O0O	ribonucleotide reductase	3O0O	3O0O	3O0O	1.17.4.1
	3REQ	methylmalonyl-CoA mutase				5.4.99.2
	3SOM	methylmalonic aciduria and homocystinuria type c (mmachc)	3SOM	3SOM	3SOM	NA
	4DJD	folate-free corrinoid iron-sulfur protein (cfesp)	4DJD	4DJD		NA
	4DJE	folate-bound corrinoid iron-sulfur protein (cfesp				NA
	4REQ	methylmalonyl-CoA mutase				5.4.99.2
	5REQ	methylmalonyl-CoA mutase				5.4.99.2
	6REQ	methylmalonyl-CoA mutase				5.4.99.2
	7REQ	methylmalonyl-CoA mutase				5.4.99.2

TABLE I: Annotated list of B12-binding proteins used in this study. A classification scheme based on manual annotation was implemented in order to determine the correctness of PocketMatch. This scheme is represented as different colours, with every colour representing a different class of pocket. Within different classes, a single pocket was chosen as representative, based solely on crystal resolution. Red dotted suqares represent higher resolution structures when multiple reprenentatives of differing sequence similarity are chosen for each class. Sequence similarity based and E.C. number based classification corroborated the manual annotation. All pockets and classes are similarly represented in Figure 4.

This trivialises the complexity of alpha-file generation, which can be expressed with complexity $k_1\mathbf{O}(n)$; where k_1 is the the average number of residues per pocket (*Figure 5.2*)

Following alpha-file generation, an all-to-all comparison and PocketMatch score calculation for n files is an $\mathbf{O}(n^2)$ problem. MPI-based parallelisation can reduce runtime by p-fold (*Figure 5.3, Figure 5.4*). This is achieved using the MPI_File_write() function to output PocketMatch results in parallel to a given MPI file-pointer.
The default algorithm, used without the Sphinx module (*Figure 5.3*) has a complexity of $\mathbf{O}(n^2)$. Although the complexity remains constant, PocketMatch (version 1.0) (*Figure 5.1*) and PocketMatch (version 2.0) show vastly different execution times. The average execution-time of a single iteration of PocketMatch (version 2.0) via the *match* function (*Figure 1, steps 5,6*) is 82 µs . This represents a 52-fold runtime increase over the previous (version 1.0) implementation.

The Sphinx module (*Figure 5.4*) uses double-hashing, adding $\mathbf{O}(m+\alpha)$ complexity, where α is the hash-collision rate. Assuming m, α and PocketMatch distributions remain constant over large datasets, $\mathbf{O}(m+\alpha)$ can be reduced to an external constant k_2, keeping the overall complexity at k_2 $\mathbf{O}(n^2)$. The average execution-time of a single iteration of PocketMatch (version 2.0) using the module is 400 µs .
While the Sphinx module is considerably slower, it nevertheless performs better than the previous (version 1.0) implementation, representing a 10-fold increase in runtime.

Validation of this analysis is presented in *Figure 5*. Machine-specific runtimes calculated over limited input ranges fit the complexity models very well. Machine and input specific linear and quadratic fitted parameters are also given. A full complexity analysis is presented in *Figure 3*, in (1), (2), and (3).

The neighbour-joining algorithm[8] was applied to the PocketMatch results generated from the vitamin B12 sample dataset (*Figure 4)*. Visualisation of the results confirmed near-perfect similarity-scoring. All pockets were appropriately clustered, with the exceptions of pockets of proteins (PDB IDs) 2YCL and 2H9A.

IV. CONCLUSION

We have presented PocketMatch (version 2.0), a fast, robust, algorithm capable of database-wide pocket similarity searches. Its parallel MPI-implementation and easy scalability reported here will expand its applicability to future high-throughput bioinformatics and computational-biology studies that require peta/exascale computing. An example application is the analysis of an entire pathogenic bacterial species *pocketome*. Such an analysis would require approximately 300 million comparisons, which could be completed in less than 2 hours on a single processor using the current implementation, without using the Sphinx module.

Future enhancements could include multi-platform compatibility, spanning of multiple operating systems hardware architectures, to include CUDA architectures. The Sphinx module may be used for rapid, parallel, 3D structural alignments, which would be far more informative than similarity scores alone.

m: number of residues within a given pocket
n: number of pockets/cabbage-files used as input
p: number of processors used for parallel computations
In serial execution, **p** = 1

For generation of a single alpha file:

For **m** residues, $\frac{m*(m+1)}{2}$ total distance elements are created

$\frac{m*(m+1)}{2}$ results in $\mathbf{O(m^2)}$ complexity

$$\mathbf{O(m^2)} + \mathbf{O(m)} + \mathbf{O(m\log(m))} = \mathbf{O(m^2)} \quad ..(1)$$

pairwise-distance calculation complexity | binning complexity | sorting complexity | total complexity

If m distributions remain constant over large datasets:

$$\mathbf{O(m^2)} \approx \mathbf{k_1}$$

Then for generation of n cabbage files:

total complexity $\approx \mathbf{k_1 O(n^2)}$ ..(2)

For calculation of PocketMatch scores (version 1/2):

For **n** cabbage-files, $\frac{n*(n+1)}{2}$ total PocketMatch scores are calculated

$\frac{n*(n+1)}{2}$ results in $\mathbf{O(n^2)}$ complexity

For augmented PocketMatch scores the Sphinx module (version 2 only):
-If m distributions remain constant over large datasets:
-If PocketMatch distributions remain constant over large datasets:

$$\mathbf{O(m\log(m))} \approx \mathbf{k_2}$$

Then, the PocketMatch score complexity is estimates as:

$$\mathbf{O(n*p)} + \frac{\mathbf{k_2 O(n^2)}}{\mathbf{p}} = \frac{\mathbf{k_2 O(n^2)}}{\mathbf{p}} \quad ..(3)$$

file-input complexity | PocketMatch complexity | total complexity

Fig. 3. Complexity equations governing the current implementation of PocketMatch (version 2.0).

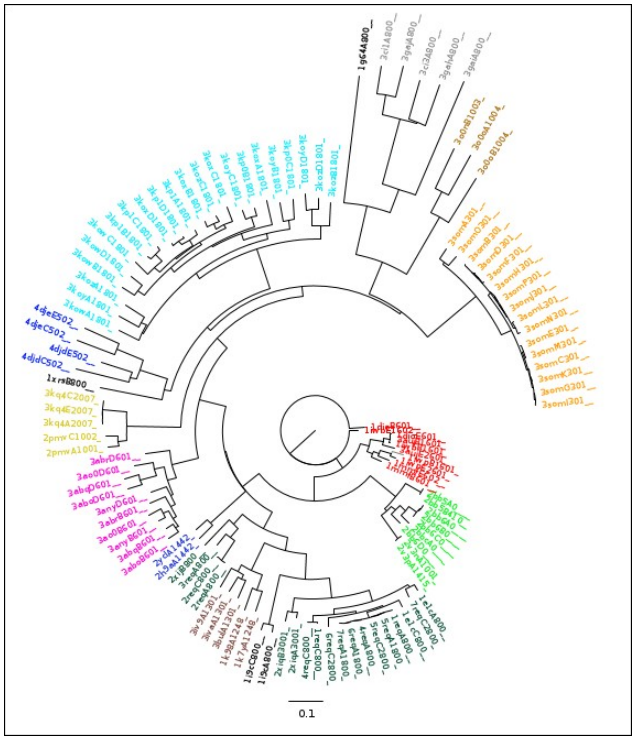

Fig. 4. Application of the Neighbour-joining algorithm to PocketMatch scores of the sample (vitamin B12-binding pocket) dataset revealed near-perfect performance. Pockets from proteins 2YCL and 2H9A are the only wrongly-clustered elements. A detailed description is given in *Table I.*

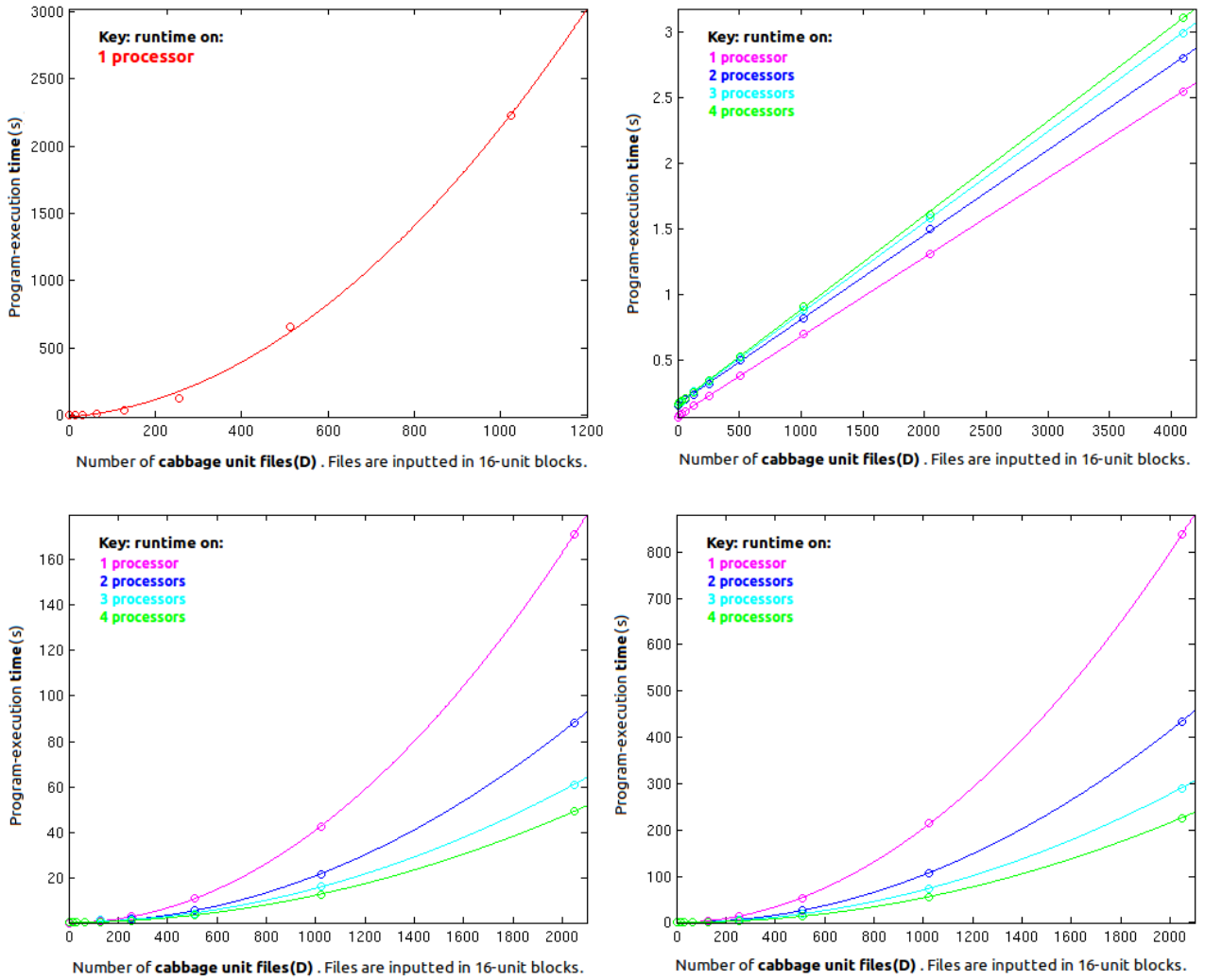

Fig. 5.1. (above, left). Performance analysis of the original (version 1.0) implementation of PocketMatch. While the complexity remains approximately O(n^2) for all cases, this version shows runtimes that are an order of magnitude slower.
Fig. 5.2. (above, right). Performance analysis of the data input segment of the MPI implementation of the (version 2.0) algorithm (excluding the match function). Data once inputted has to be transferres serially to multiple slave processors, accounting for an additional O(n) complexity layer.
Fig. 5.3. (below, left). Performance analysis of the standard (version 2.0) algorithm. Runtimes change based on number of processors used and size of data-input. Data analysis occurs in O(n^2) time.
Fig. 5.4. (below, right). Performance analysis of the (version 2.0) algorithm. This figure involves runtimes when using the optional Sphinx module. Data analysis occurs in approximately O(n^2) time.

V. REFERENCES

[1]: Liu, Z., *et. al.*, **"Bridging protein local structures and protein functions"**, *Amino acids* (2008) v35(3) p627-650.
[2]: Alexander, S., Russel, R.B., **"Annotation in three dimensions. PINTS: Patterns in Non-homologous Tertiary Structures"**, *Nucleic Acids Research* (2003) v31(13) p3341-3344.
[3]: Kleywegt G.J., **"Recognition of Spatial Motifs in Protein Structures"**, *J. Mol. Biol.* (1999) v285 p1877-1897.
[4]: Kuhn D., *et. al.*, **"Functional classification of protein kinase binding sites using Cavbase"**, *ChenMedChem* (2007) v2(10) p1432-1447.
[5]: Yeturu K., Chandra N., **"PocketMatch: a new algorithm to compare binding sites in protein structures"**, *BMC Bioinformatics* (2008) v9 p543.
[6]: Anand, P., Yeturu, K., Chandra, N., **"PocketAnnotate: towards site-based function annotation"**, *Nucleic Acids Research* (2012) Jul;40 (Web Server issue):W400-8.
[7]: Berman, H.M., *et. al.*, **"Announcing the worldwide Protein Data Bank"**, *Nature Structural Biology* (2003) v10(12) p980.
[8]: W. Kabsch **"A solution of the best rotation to relate two sets of vectors"**, *Acta Crystallographica* (1976) A32, 922-923.
[8]: Felsenstein, J., **"PHYLIP - Phylogeny Inference Package (Version 3.2)"**, *Cladistics* (1989) v5 p164-166.
[9]: Murzin A.G., *et. al.*, **"SCOP: a structural classification of proteins database for the investigation of sequences and structures"**, *J. Mol. Biol* (1995) v247 p536-540.