Improving the discovery and clustering of three-dimensional protein patterns with OpenMP

Alejandro Valdés-Jiménez
Depto. Sistemas de Información
Universidad del Bío-Bío
Concepción, Chile
avaldes@ubiobio.cl

Miguel Reyes-Parada
CIBAP, Facultad de Ciencias Médicas
Universidad de Santiago de Chile
Santiago, Chile
miguel.reyes@usach.cl

Gabriel Nuñez-Vivanco
Dept. of Natural Sciences and Technology
University of Aysen
Aysén, Chile
gabriel.nunez@uaysen.cl

Fabio Durán-Verdugo
Depto. de Bioinfomática
Universidad de Talca
Talca, Chile
fduran@utalca.cl

Daniel Jiménez-González
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya (UPC Barcelona Tech)
Barcelona, Spain
djimenez@ac.upc.edu

***Abstract*—The discovery of conserved three-dimensional (3D) amino-acid patterns among a set of protein structures can be useful, for instance, to predict the functions of unknown proteins or for the rational design of multi-target drugs. There are several applications that perform a three-dimensional search of patterns in the structures of proteins. However, discovering conserved 3D patterns in a set of proteins with no other baseline patterns is a challenge. In this paper, we analyze and improve a state-of-the-art algorithm, 3D-PP, that implements this discovery. In this algorithm, the 3D patterns are detected and clustered using the root mean square deviation value, measured among each pair of 3D patterns (topological variability indicator). Even when 3D-PP deals with this task, the simultaneous processing of high amounts of proteins becomes a computational challenge with the size and the number of proteins to be evaluated. In this work, we present and analyze different shared memory parallel strategies of 3D-PP, using OpenMP. Those strategies improve the overall performance of the original implementation by reducing parallel load unbalance among threads and overall increasing parallelism. The results show significant performance improvements compared to the original version, achieving up to 13x speedup for a small number of proteins and 17.7x for a larger set.**

***Index Terms*—OpenMP, performance optimization, three-dimensional protein patterns, drug-design**

I. Introduction

Polypharmacology, which refers to the ability of a molecule to simultaneously interact with multiple target proteins, has emerged as an alternative to the classical "one-drug-one-target" drug discovery process paradigm [3], [4]. This is based on the observation that robust pathological phenotypes, such as those observed in psychiatric or cardiovascular diseases, often result from a complex network of molecular events rather than changes in the function of a single target. Therefore, it is expected that acting on several nodes of these biological networks should lead to compounds with better profiles regarding both efficacy and side effects as compared to more selective drugs [5], [6]. One plausible approach to searching for polypharmacological agents is to detect similar/conserved binding sites in different proteins, where one drug might bind [7] [8]. However, this is not an easy task, particularly when the drugs are aimed to act at proteins with highly diverse structures and functions. In this context, 3D-PP was developed, an algorithm for the discovery and recognition of all similar 3D amino acid patterns among a set of protein structures [1]. This algorithm does not require any previous structural knowledge about ligands and/or protein sequence similarity. Thus, 3D-PP is a reliable and flexible tool to identify conserved structural motifs (3D-patterns) among a wide range of different proteins, which could be relevant for the discovery of novel polypharmacological drugs. In this work, a parallel version of the 3D-PP algorithm is presented using the OpenMP [2] parallel programming model for shared memory. This version: i) significantly improves performance over the original version; ii) is aware of the possible load imbalance, the NUMA architecture of the system, and the possible overheads of data sharing and synchronization between threads, and iii) is open source (available at https://gitlab.com/amvaldesj/3d-pp). In Section II, details about the 3D-PP are described and an analysis of the sequential version profile is detailed. The parallel algorithms are presented in Section III and finally, in Section IV, the performance results of the presented parallel version of 3D-PP are shown.

II. 3D-PP Algorithm

In this section, the main functions of 3D-PP are briefly described, as well as the results of the analysis of the sequential baseline version. Algorithm 1 presents the main structure of 3D-PP. Each pdb file is parsed getting all the chains (*all_chains*). Each of these chains is then processed, generating a list of patterns (*all_patterns*), each with its list of sites found. This list of patterns is filtered, leaving only those that meet the minimum coverage percentage. Finally, the sites of each pattern are clustered and the results are saved. The following parameters are received: i) Spacing Threshold:

This value is used to create the virtual grid of coordinates and defines, how broad and rigorous will be the exploration of 3D-patterns; ii) *Radius Threshold*: This term represents the limits of the size of the 3D-patterns searched; iii) *RMSD Threshold*: This value is used for clustering the 3D-patterns detected and represents a measure of structural variability for the sites composing each 3D-pattern; iv) *Minimum Coverage*: This value allows displaying only 3D-patterns with a given coverage; v) *Displacement Threshold*: This value is used to expand the size and shape for the exploration of the 3D-patterns; vi) *Minimum number of residues*: This value defines the minimum number of residues that make up a site; vii) *List of pdb files*: The list of the *pdb* files.

Algorithm 1 Main function.

```
1: function MAIN(params)
2:     pdbs ← load_pdb_ids(params)
3:     all_chains ← parses_pdbs(pdbs, params)
4:     all_patterns ← process_chains(params, all_chains)
5:     all_patterns ← filter_patterns(params, all_patterns)
6:     all_patterns ← cluster_sites(params, all_patterns)
7:     save_parameters(params)
8:     save_results(params, all_patterns)
```

A. Parsing the proteins

This function is responsible for parsing each of the protein structures to obtain information about the chains contained in them. Algorithm 2 shows this process. A set of protein identifiers and parameters defined by the user are received for the processing, returning a list of chains (*all_chains*). Assuming that *N* is the number of proteins to analyze, this algorithm has a complexity of *O(N)*.

Algorithm 2 Parse PDB files.

```
1: function PARSES_PDBS(pdbs, params)
2:     all_chains ← [ ]
3:     for pdb in pdbs do
4:         chains ← parse_pdb(params, pdb)
5:         all_chains.add(chains)
6:     return all_chains
```

B. Processing the chains

Once the list of all the chains is obtained (*all_chains*), each one of them is processed identifying all the possible sites (arrangements of structurally related amino acids). For each chain the following actions are carried out: i) a *kdtree* [9] structure is generated, used to search for neighboring atoms in 3D space; ii) the geometric center of each amino acid is calculated; iii) the coordinates of the limits *x, y, z* are obtained (Figure 1A); iv) the grid of virtual coordinates is generated (Figure 1B), the sites are searched from each virtual coordinate, and their pattern is generated; v) the patterns in the chain are merged, and vi) these patterns are merged with the patterns found in the other chains. Finally, a list of the patterns discovered (*all_patterns*) with its sites is returned. If *N* is the number of chains found in all proteins, this algorithm has a complexity of *O(N)*. Algorithm 3 shows this function. Every valid site must contain at least four amino acids. A list of amino acids is defined for each site. Then, the sites are transformed into a representation of components through a sorted alphabetical string which contains the one-letter code of the amino acid and the number of occurrences of the same amino acid (e.g., the site *CYS15:CYS24:CYS30:HIS34* is transformed into the pattern *3C1H*).

Algorithm 3 Process chains.

```
1: function PROCESS_CHAINS(params, all_chains)
2:     all_patterns ← [ ]
3:     for chain in all_chains do
4:         chain.kdtree ← create_kdtree(chain)
5:         calc_geom_center_of_residues(chain)
6:         calc_max_min_coordinates(chain)
7:         find_sites(chain, params)
8:         merge_patterns_in_chain(chain)
9:         merge_all_patterns(chain.patterns, all_patterns)
10:    return all_patterns
```

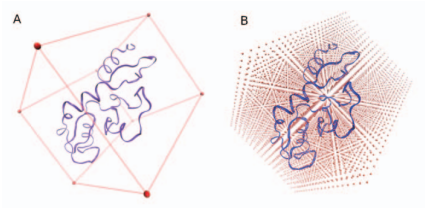


Fig. 1. A Grid of virtual Coordinates (GvC) is generated for each chain. From each virtual coordinate, the sites are searched.

C. Finding/discovering the sites

With the limit values in each dimension (minimum and maximum), the number of points to be generated in the virtual grid (*maxux, maxuy, and maxuz*) is calculated. This calculation uses the *params.step* parameter to indicate the distance between the points. Next, the nearby residues (at *params.radius* distance) are searched for each virtual coordinate. Each site must have a minimum of residues (*params.nres*) and must not be repeated (Algorithm 4). If *MAXUX, MAXUY*, and *MAXUZ* are the number of points on the x, y, and z axes, respectively, then this algorithm has a complexity of *O(MAXUX×MAXUY×MAXUZ)*.

Algorithm 4 Find sites.

```
1: function FIND_SITES(chain, params)
2:     maxux = (chain.xmax − chain.xmin)/params.step + 1
3:     maxuy = (chain.ymax − chain.ymin)/params.step + 1
4:     maxuz = (chain.zmax − chain.zmin)/params.step + 1
5:     for (ux = 0; ux < maxux; ux + +) do
6:         for (uy = 0; uy < maxuy; uy + +) do
7:             for (uz = 0; uz < maxuz; uz + +) do
8:                 x ← chain.xmin + params.step ∗ ux
9:                 y ← chain.ymin + params.step ∗ uy
10:                x ← chain.zmin + params.step ∗ uz
11:                pt ← {x, y, z}
12:                list ← search_neighbors(chain.kdtree, pt, params.radius)
13:                if (list.size() >= params.nres) then
14:                    new_site ← create_site(list)
15:                    if (new_site not in chain.sites) then
16:                        chain.sites.add(new_site)
```

D. Clustering the sites

The last step is clustering the sites in each pattern. Before proceeding with the clustering, it is necessary to filter the

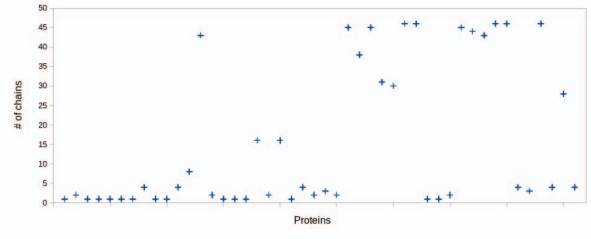
list of all the patterns and leave only those that meet the minimum percentage of coverage (percentage of proteins in which a pattern appears). Depending on this parameter (*Minimum Coverage*), which is defined by the user, the number of patterns to cluster can decrease considerably. Algorithm 5 shows how the clustering is done. The list of filtered patterns (*all_patterns*) with its sites is received and then each site of each pattern is evaluated. In order to get identical results on each run, the sites of each pattern are ordered by *site-string, chain, and protein id*. For example, if you change the order of the list of the same entered proteins, the results should not be different. If the pattern does not have a cluster or any site does not match a cluster, a cluster is created and added to the pattern. A unique cluster name is generated and the site is added to this cluster as a base site. However, if the pattern has at least one cluster, then it is necessary to find which cluster the site matches (the first match). To determine if a site matches a cluster, the *root-mean-square deviation (rmsd)* [10] between the site and the base site of the cluster is measured, and if the value is less than or equal to the *RMSD Threshold (params.diffrmsd)* parameter, the site is added to that cluster. Finally, each pattern has its sites clustered. If N is the number of patterns, M is the maximum number of sites in a pattern, and P is the maximum number of cluster in a pattern, then the complexity of this algorithm is $O(N \times M \times P)$.

Algorithm 5 Cluster the sites.

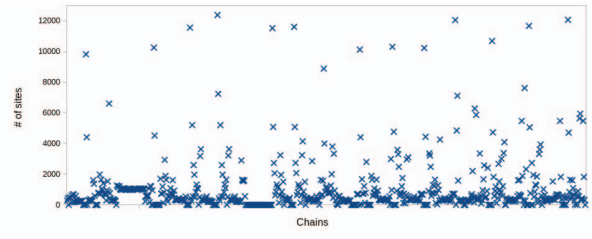
```
1:  function CLUSTER_SITES(params, all_patterns)
2:    for pattern in all_patterns do
3:      sort(pattern.sites)
4:      for site in pattern.sites do
5:        matched ← false
6:        nclu ← pattern.clusters.size()
7:        for (c = 0; c < nclu and matched == false; c++) do
8:          site_base ← pattern.clusters[c].sites[0]
9:          rmsd ← rmsd(site, site_base)
10:         if rmsd <= params.diffrmsd then
11:           pattern.clusters[c].sites.add(site)
12:           matched ← true
13:       if matched == false then
14:         cluster.name ← concat(pattern.name, nclu + 1)
15:         cluster.sites ← [ ]
16:         cluster.sites.add(site)
17:         pattern.clusters.add(cluster)
18:   return all_patterns
```

E. Protein data

The protein data set has 46 proteins [1]. Figure 2(a) shows each protein and its number of chains. It is possible to see that there is an irregular distribution of the chains among the proteins, some proteins have only one chain but others have more than forty chains. In addition, proteins can be globular in shape, while others can be elongated (fibrous). The same irregularity is observed in the number of sites found in each chain of the protein set (Figure 2(b)). This is because the number of amino acids that make up each chain is also variable. If a chain has few amino acids, then the number of virtual coordinates to generate (see Figure 1) for the search of sites can be few too, however, if the number of amino acids increases, then the number of virtual coordinates probably also increases. If it is considered that the depth of the search can be managed by the user through the *Spacing Threshold* parameter, then the greater the number of virtual coordinates that are generated, the greater the amount of time required to complete the analysis. However, not only the number of amino acids that make up the chain can influence the processing time, but also the three-dimensional shape of the protein, they can be globular or fibrous in shape. It is important then to consider that an incorrect data distribution strategy could cause a load imbalance and affect the performance of the parallel program.



(a) Distribution of the number of chains.



(b) Distribution of the number of sites found in the 718 chains.

Fig. 2. Number of chains and sites found in the set of 46 proteins.

F. Baseline version profiling

The original version of 3D-PP was developed with the Python programming language (version 2.7) using the *threading* module as a parallel strategy, which is well known to use the *GIL* [17] that limits exploit parallelism for CPU-bound programs. A new sequential version in C/C++ based on the original program has been developed and is used in this work as the baseline version. Table I shows the profiling details (performed on the IBM Power9 cluster using the SLURM batch queuing system, see Section IV-A) of the applications using a standard input data set. The table shows the elapsed execution time of the main functions and their percentage of the total execution time of the application. For processing chain (*process_chains()*), with more than 99% of the execution time, the information of their main called functions (*find_sites()*, *merge_all_patterns()*, and *merge_patterns_in_chain()*) are included. It is possible to see that the *find_sites()* function covers the highest percentage (≈90%) of the total elapsed time, next, the *merge_all_patterns()* function covers ≈7.7%, and finally the function *merge_patterns_in_chain()* covers ≈1.2%. All these functions are part of the *process_chains()* function (Algorithm 3). If we add the percentages of time used for the functions *find_sites()*, *create_kdtree()*, *calc_geom_center_of_residues()*,

calc_max_min_coordinates(), *merge_patterns_in_chain()*, and *merge_all_patterns()*, we have approximately 99% of the total elapsed time. In this algorithm, the processing of each chain can be parallelized. Many of their operations are independent, however, they share the *all_patterns* variable where the patterns found in each chain are consolidated, specifically in the function *merge_all_patterns()*. Therefore, access to this variable must also be synchronized. An important detail is the time (≈7.7%) used to consolidate the patterns found, since this percentage of time is not fully parallelizable, which certainly affects performance expectations. In addition, although *parses_pdbs()* function (Algorithm 2) is not relevant for the profiling shown, it may be important to consider it for inputs with a large number of proteins. Finally, clustering in each pattern can be done completely independently and in parallel, there is no data access restriction. In this way, in the event of having to perform clustering on a large number of patterns, the parallelization of this algorithm would improve the overall performance.

TABLE I
BASELINE PROFILE. 46 PROTEINS EVALUATED. USER PARAMETERS: *Spacing Threshold*=0.8Å, *Radius Threshold*=3Å, *RMSD Threshold*=5.0Å, *Min. Coverage*=80%, *Displacement Threshold*=0, *Min. # of residues*=4.

functions	elapsed time (seconds)	% of the total time
main()	463.1	100
process_chains()	**460.2**	**99.2**
find_sites()	418.4	90.3
merge_all_patterns()	35.9	7.7
merge_patterns_in_chain()	5.9	1.2

III. PARALLEL ALGORITHMS

In this section, we present different OpenMP parallelization of the most time-consuming function, *process_chain()*, but also for those potential functions that may become a speedup limitation for large data sets (*parses_pdbs()*) or large number of patterns to cluster or process (*cluster_sites()*), once the process chain is properly parallelized. Function *find_sites()* was not parallelized since the parallelization is done at *process_chains()* function level (caller), allowing good load balance and reducing extra synchronization and scheduling overheads due to very fine grain parallelism.

A. Baseline Proposal

Algorithms 6, 7, and 8 show the parallelized versions of the *parses_pdbs()*, *process_chains()*, and *cluster_sites()* algorithms, respectively. The parallel function *parses_pdbs()* (Algorithm 6) processes each protein in parallel and independently. A shared *chain_thd* array of size *params.nthreads* is used to temporarily store a list of chains per thread, necessary to avoid synchronization overhead on reducing chains over a shared variable *all_chains* in the parallel region. After parallel processing is complete, all chains are reduced sequentially into the *all_chains* array. A *#pragma omp parallel num_threads(params.nthreads)* directive is used to define the parallel work team, and then, the *pdb ids* are distributed to this team using the directive *#pragma omp for schedule(runtime)*. The number of threads to use and the specific scheduler are defined at the runtime. The possibility of having a false sharing in the *chains_thd* array update is not significant since we understand that what will be updated are the lists associated with each thread, which will not affect the consecutive positions of the *chains_thd* array, which could be in the same cache line.

Algorithm 6 Parallel parse of PDB files. Baseline version.

```
1:  function PARSES_PDBS(pdbs, params)
2:    all_chains ← [ ]
3:    chains_thd[params.nthreads]
4:    #pragma omp parallel num_threads(params.nthreads) private(id, i, tmp_chains)
      shared(chains_thd)
5:    id ← omp_get_thread_num()
6:    #pragma omp for schedule(runtime)
7:    for (i = 0; i < pdbs.size(); i++) do
8:      tmp_chains ← parse_pdb(params, pdbs[i])
9:      chains_thd[id].add(tmp_chains)
10:   for (id = 0; id < params.nthreads; id++) do                ▷ reduction.
11:     all_chains.add(chains_thd[id])
12:   return all_chains
```

The parallel function *process_chain()* (Algorithm 7) distributes the chains stored in the *all_chains* array to different threads to be executed independently. Each thread will work on its own set of chains, temporarily storing its patterns in the *patterns_thd* array. After performing all operations on the chains, all found patterns are reduced sequentially to the *all_patterns* array outside the parallel region, avoiding synchronization overhead when updating that array. The number of threads used and the specific scheduler are also defined at runtime. Function *cluster_sites()* is also parallelizable (see Algorithm 8). Each pattern, with its sites, can be clustered independently. Only the *#pragma omp parallel for num_threads(params.nthreads) schedule(runtime)* directive is used to define the number of threads and the scheduler. Same to the others algorithms, the number of threads to use and the specific scheduler are defined at the runtime.

Algorithm 7 Parallel process of chains. Baseline version.

```
1:  function PROCESS_CHAINS(params, all_chains)
2:    all_patterns ← [ ]
3:    patterns_thd[params.nthreads]
4:    #pragma omp parallel num_threads(params.nthreads) private(id, c)
      shared(all_chains)
5:    id ← omp_get_thread_num()
6:    #pragma omp for schedule(runtime)
7:    for (c = 0; c < all_chains.size(); c++) do
8:      all_chains[c].kdtree ← create_kdtree(all_chains[c])
9:      calc_geom_center_of_residues(all_chains[c]))
10:     calc_max_min_coordinates(all_chains[c])
11:     find_sites(all_chains[c], params)
12:     merge_patterns_in_chain(all_chains[c], params)
13:     patterns_thd[id].add(all_chains[c].patterns)
14:   for (id = 0; id < params.nthreads; id++) do                ▷ reduction.
15:     merge_all_patterns(patterns_thd[id], all_patterns)
16:   return all_patterns
```

B. Overlapping Computational Unbalance With Final Reduction

Previous baseline parallel proposals of *parses_pdbs()* and *process_chains()* presented important load balance problems (later evaluated) when distributing chains and their processing.

Algorithm 8 Parallel cluster of sites. Baseline version.

```
1: function CLUSTER_SITES(params, all_patterns)
2:   #pragma omp parallel for num_threads(params.nthreads) schedule(runtime)
     private(i, n_sites, s, site, matched, nclu, c, site_base, rmsd, cluster)
     shared(all_patterns)
3:   for (i = 0; i < all_patterns.size(); i++) do
4:     sort(all_patterns[i].sites)
5:     n_sites ← all_patterns[i].sites.size()
6:     for (s = 0; s < n_sites; s++) do
7:       site ← all_patterns[i].sites[s]
8:       matched ← false
9:       nclu ← all_patterns[i].clusters.size()
10:      for (c = 0; c <= nclu and matched == false; c++) do
11:        site_base ← all_patterns[i].clusters[c].sites[0]
12:        rmsd ← rmsd(site, site_base)
13:        if rmsd <= params.diffrmsd then
14:          all_patterns[i].clusters[c].sites.add(site)
15:          matched ← true
16:      if matched == false then
17:        cluster.name ← concat(all_patterns[i].name, nclu + 1)
18:        cluster.sites ← [ ]
19:        cluster.sites.add(site)
20:        all_patterns[i].clusters.add(cluster)
21:  return all_patterns
```

This means that the sequential reduction performed by the master thread has to pay for all that balance before it starts sequentially processing the results of the parallelized loop. For this reason, it was decided that threads did not have to stop at a global synchronization and perform a reduction in the global variable after the loop ended. On the one hand, doing it outside the loop is to avoid synchronization at each iteration, and on the other hand, the threads that have been able to finish their work will be able to make their contribution to the global variable while there are others that are still doing calculations. The reduction operation is performed with a *critical* section to avoid data race conditions. These new versions of the *parses_pdbs()* and *process_chain()* functions (Algorithms 9 and 10, respectively) uses the *critical* section to try to reduce the time of its sequential parts, specifically in the reduction of the chains and the patterns. The *nowait* clause has been used in the loop to avoid synchronization between threads. This way, if one thread finishes its work before the others, it can continue to reduce its data. The parallel *cluster_sites()* function remained the same (Algorithm 8).

Algorithm 9 Parallel parse of PDB files. Overlapping Computational Unbalance.

```
1: function PARSES_PDBS(pdbs, params)
2:   all_chains ← [ ]
3:   chains_thd[params.nthreads]
4:   #pragma omp parallel num_threads(params.nthreads) private(id, i, tmp_chains)
     shared(chains_thd, all_chains)
5:   id ← omp_get_thread_num()
6:   #pragma omp for nowait schedule(runtime)
7:   for (i = 0; i < pdbs.size(); i++) do
8:     tmp_chains ← parse_pdb(params, pdbs[i])
9:     chains_thd[id].add(tmp_chains)
10:  #pragma omp critical                                   ▷ reduction.
11:  all_chains.add(chains_thd[id].get_chains)
12:  return all_chains
```

IV. Performance Evaluation

This section presents the experimental setup, the performance of the new parallel versions of 3D-PP, and details

Algorithm 10 Parallel process of chains. Overlapping Computational Unbalance.

```
1: function PROCESS_CHAINS(params, all_chains)
2:   all_patterns ← [ ]
3:   patterns_thd[params.nthreads]
4:   #pragma omp parallel num_threads(params.nthreads) private(id, c)
     shared(all_chains, patterns_thd, all_patterns)
5:   id ← omp_get_thread_num()
6:   #pragma omp for nowait schedule(runtime)
7:   for (c = 0; c < all_chains.size(); c++) do
8:     all_chains[c].kdtree ← create_kdtree(all_chains[c])
9:     calc_geom_center_of_residues(all_chains[c]))
10:    calc_max_min_coordinates(all_chains[c])
11:    find_sites(all_chains[c], params)
12:    merge_patterns_in_chain(all_chains[c], params)
13:    patterns_thd[id].add(all_chains[c].patterns)
14:  #pragma omp critical                                   ▷ reduction.
15:  merge_all_patterns(patterns_thd[id], all_patterns)
16:  return all_patterns
```

of the different loop schedulers tested (*static, static,1, and dynamic,1*).

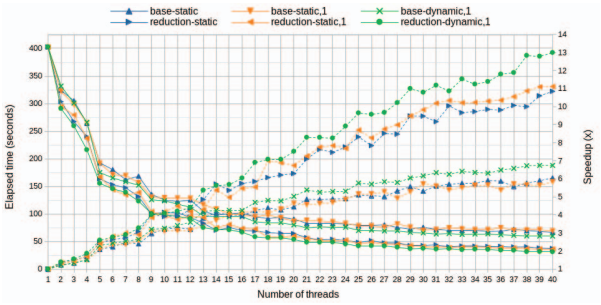
A. Experimental Setup

The experiments have been carried out on one node of the IBM Power9 cluster [16] available at the BSC (Barcelona Supercomputing Center). One compute node is a NUMA system with 2 x IBM Power9 8335-GTH @2.4GHz with 20 cores in each socket and total main memory of 512GB, with 32KB of L1 cache, 512 KB of L2 cache, and 10 MB of L3 cache that is shared by each pair of cores. For local storage, it has 2 x 1.9TB SSDs and uses the GPFS cluster file system via one fiber link 10 GBit to distribute and manage data across multiple nodes. Our application has been implemented in C++11 and has been compiled with gcc++ 10.1.0 version, with the full support of OpenMP 5.0 parallel programming model. The compilation includes the *-O3 -mcpu=power9 -mtune=power9* flags optimization options, allowing maximum optimization and automatic vectorization, and *-fopenmp* flag. Using the *-fopt-info-loop-optimized* and *-fopt-info-vec-optimized* developer options [15] it was possible to see some loop vectorizations. The operating system is Red Hat Enterprise Linux Server 7.5 alternative. For each experiment, at least 3 runs have been performed and the mean time is used. Executions have been done using a batch SLURM queue system version 21.08.8-2, with OMP_PLACES equal to *cores* and OMP_PROC_BIND equal to *spread* to allow maximum performance in the NUMA system. The results of parallel versions have been compared with the sequential baseline C/C++ version implemented. Two sets of proteins have been used: 46 structures [1] containing the PROSITE Zinc finger C3H1-type motif [11] and a subset of 8,344 from FDA-approved human drugs [12]. With the user parameters: *Spacing Threshold*=0.8Å, *Radius Threshold*=3Å, *RMSD Threshold*=5.0Å, *Displacement Threshold*=0, *Min. # of residues*=4, *Min. Coverage*=80% for the set of 46 proteins, and *Min. Coverage*=70% for the set of 8,344 proteins.

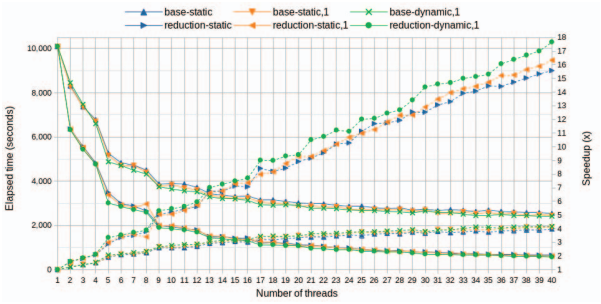
B. Results

The parallel version has been validated and obtains the same results in terms of 3D patterns found, clusters, and

sequence alignments. Up to 40 threads were used (one for each physical core) since the efficiency, and even the performance, worsened if we increased the number of threads per core. Figures 3(a) and 3(b) show the performances obtained for both sets of proteins, by both versions and using the *static*, *static,1*, and *dynamic,1* schedule policies. Solid lines represent elapsed times and dashed lines represent speedups. We can see that the second parallel approach (*reduction-**) shows much better performance on both sets of proteins and on all three schedulers. The *dynamic,1* scheduler always shows better performance than the *static* variants, although, in the smaller set of proteins, the difference was much more significant. In general, the performances of the *static* variants are similar. Table II shows the general results using only 40 threads. It is possible to see, for the small set of proteins (46), how the second parallel approach using the *dynamic,1* scheduler is ≈2.0 times faster than the first parallel approach, going from 6.7x to 13.0x speedup. With this acceleration, it was possible to reduce the sequential time from 402.4 to 30.9 seconds. For the larger set of proteins (8,344), the difference in speedup using the *dynamic,1* scheduler was more significant, it was ≈4 times faster, going from 4.2x to 17.7x speedup. The sequential version elapsed time was reduced from 10,094.2 to 571.3 seconds. Furthermore, it can be seen how the performance of both parallel versions using the *static* scheduling variants is very similar. For the purpose of completing the results, we also show results using *guided,1*. Although in some cases it performs better than *static-** versions, *dynamic,1* schedule policies show better load balance and performance.



(a) 46 proteins.



(b) 8,344 proteins.

Fig. 3. Performances of both versions for the two sets of proteins. Up to 40 threads are used. Elapsed times and speedups are displayed for the *static*, *static,1*, and *dynamic,1* schedulers.

TABLE II
GLOBAL RESULTS USING 40 THREADS AND DIFFERENT SCHEDULERS.

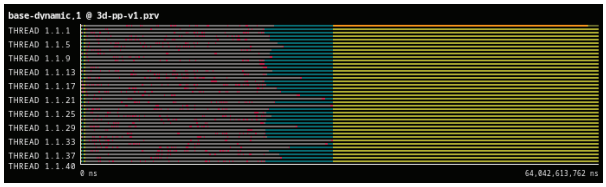
Set of proteins	46		8,344	
Sequential baseline C/C++ time	402.4 secs		10,094.2 secs	
Versions	secs	acc	secs	acc
Baseline Proposal (*static*)	66.5	6.0x	2,542.2	4.0x
Baseline Proposal (*static,1*)	68.8	5.8x	2,455.3	4.1x
Baseline Proposal (*dynamic,1*)	**59.6**	**6.7x**	**2,422.0**	**4.2x**
Baseline Proposal (*guided,1*)	65.3	6.1x	2,706.7	3.7x
Overlapping Computational Unbalance With Final Reduction (*static*)	37.0	10.9x	647.8	15.6x
Overlapping Computational Unbalance With Final Reduction (*static,1*)	36.2	11.1x	617.9	16.3x
Overlapping Computational Unbalance With Final Reduction (*dynamic,1*)	**30.9**	**13.0x**	**571.3**	**17.7x**
Overlapping Computational Unbalance With Final Reduction (*guided,1*)	33.7	11.9x	753.4	13.3x

C. Load Balance Analysis

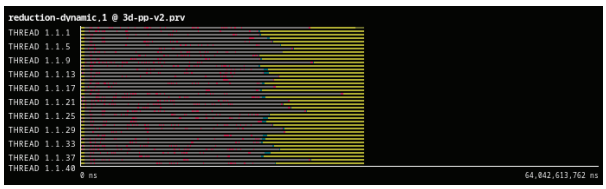
To understand why the *dynamic,1* scheduler showed better performance than the *static* schedulers, software instrumentation was performed using the different schedulers. Forty threads and the set of 46 proteins were used. Extrae [13] 4.0 and Paraver [14] 4.10 tools are used to obtain traces and then analyze them. Figures 4(a) and 4(b) show the captured traces of the two parallel versions. The images show a timeline for each running thread (each horizontal line represents a thread). The different colors show the states of the threads (Figure 4(c) shows the color map). A thread can be executing some function or remain idle. It is possible to see the different functions (*parser_pdbs()*, *find_sites()*, *merge_patterns_in_chains()*, and *merge_all_patterns()*) running in the threads. Also, when worker threads join the master thread (*OMP worksharing join*). The timelines show the same scale (≈64 seconds), from the beginning to the end of the process, to appreciate the differences between both approaches. The workload distribution is better than in other schedule policies, but there are a few threads that still have a larger workload than others, reducing the overall speedup. There are some exceptions due to the fact that some chains are very large and therefore the *find_sites()* function requires more time. This situation affects significantly the performance of the first parallel version because the final sequential reduction of the patterns (*merge_all_patterns()* function) does not start until all threads finish their work (*find_sites()* function). This is clearly reflected in Figure 5(a), where a large amount of synchronization (red bars) is performed. The second parallel version addresses this problem by parallelizing the final reduction (*merge_all_patterns()* function) across all threads. Figure 5(b) shows how the synchronization times have been reduced considerably (red bars). However, even though the execution time is reduced, the characteristics of the chains, size, and shape (see Section II-E), still have a negative effect on performance.

V. CONCLUSION AND FUTURE WORK

3D-PP, unlike other applications, is an algorithm that allows discovering and clustering three-dimensional patterns of amino acids in a set of proteins, without the need to define a known



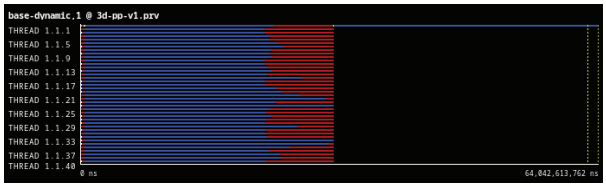
(a) base-dynamic,1



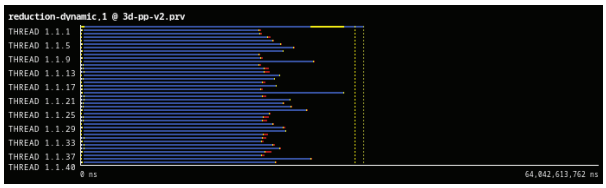
(b) reduction-dynamic,1

(c) legend

Fig. 4. Traces of both version using the *dynamic,1* scheduler. Forty threads are used. Main functions are shown.



(a) base-dynamic,1



(b) reduction-dynamic,1

Fig. 5. Traces of both version using the *dynamic,1* scheduler. Forty threads are used. The running state (blue bars) and synchronization state (red bars) are shown.

search pattern. Thus, 3D-PP is a flexible tool to identify conserved structural motifs among a wide range of different proteins, which could be relevant for the discovery of new polypharmacological drugs. Several algorithms that make up 3D-PP have great potential for parallelism, in some cases, it is embarrassing parallelism. However, the workload per iteration can vary greatly, and that means that the imbalance can significantly affect the final speedup. To avoid this, we have analyzed different types of schedules to favor different granularities that allow both statically and dynamically to improve the imbalance. On the other hand, having to perform a global reduction of results, different reduction strategies have been explored together with the possibility of avoiding global synchronizations in parallelized loops. All this has led to obtaining, for a small number of proteins (46) an acceleration of 13x and for a larger set (8,344) of proteins an acceleration of 17.7x. In this work, we have focused on the performance in a node with shared memory, as a baseline work of the distributed and hybrid memory version that is currently being designed and developed.

Acknowledgment

This research was funded by DICYT-USACH grant 5392102RP-AC and the Fondo Nacional de Desarrollo Científico y Tecnológico (FONDECYT) grants 1191133, Proyecto Fondecyt (Chile) 1220656 (MR-P), and by the Spanish Government (Grants PCI2021-121964 - TEXTAROSSA, PID2019-107255GB-C21/AEI/10.13039/501100011033, PID2019-107255GB-C22 - UPC-COMPUTACION DE ALTAS PRESTACIONES VIII and CEX2021-001148-S), and by Generalitat de Catalunya (2021 SGR 01007). Additional tests were performed in the cluster obtained with the grant CONICYT-FONDEQUIP-EQM160063.

References

[1] A. Valdés-Jiménez, J.-L. Larriba-Pey, G. Núñez-Vivanco, and M. Reyes-Parada, "3D-PP: A Tool for Discovering Conserved Three-Dimensional Protein Patterns," International Journal of Molecular Sciences, vol. 20, no. 13, p. 3174, Jun. 2019, doi: 10.3390/ijms20133174

[2] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," in IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46-55, Jan.-March 1998, doi: 10.1109/99.660313

[3] J.-L. Medina-Franco, M.-A. Giulianotti, G.-S. Welmaker, R.-A. Houghten, "Shifting from the single to the multitarget paradigm in drug discovery", Drug Discovery Today, Volume 18, Issues 9-10, 2013, Pages 495-501, ISSN 1359-6446, doi: 10.1016/j.drudis.2013.01.008.

[4] B. Ravikumar, T. Aittokallio, "Improving the efficacy-safety balance of polypharmacology in multi-target drug discovery", Expert Opinion on Drug Discovery, 2018, 13:2, 179-192, doi: 10.1080/17460441.2018.1413089

[5] M. L. Bolognesi, A. Cavalli, "Multitarget Drug Discovery and Polypharmacology", ChemMedChem 2016, 11, 1190, doi: 10.1002/cmdc.201600161

[6] A. Hopkins, "Network pharmacology: the next paradigm in drug discovery", Nat Chem Biol 4, 682–690 (2008). doi: 10.1038/nchembio.118

[7] M. Naderi and others, "Binding site matching in rational drug design: algorithms and applications", Briefings in Bioinformatics, Volume 20, Issue 6, November 2019, Pages 2167–2184, doi: 10.1093/bib/bby078

[8] J. Konc, "Binding site comparisons for target-centered drug discovery", Expert Opinion on Drug Discovery, 2019, 14:5, 445-454, doi: 10.1080/17460441.2019.1588883

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching", Commun. ACM 18, 9 (Sept. 1975), 509–517. doi: 10.1145/361002.361007

[10] E. Coutsias and M. Wester, "RMSD and Symmetry", Comput. Chem. 2019, 40, 1496-1508. doi: 10.1002/jcc.25802

[11] Zinc finger C3H1-type profile. https://prosite.expasy.org/PDOC50103 (accessed Aug. 24, 2023).

[12] DrugBank. https://go.drugbank.com/releases/latest (accessed Aug. 24, 2023).

[13] Extrae instrumentation package. http://tools.bsc.es/extrae (accessed Aug. 24, 2023).

[14] Paraver: A Tool to Visualize and Analyze Parallel Code. https://tools.bsc.es/paraver (accessed Aug. 24, 2023).

[15] GCC Developer Options. https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html (accessed Aug. 24, 2023).

[16] BSC IBM Power9 cluster. https://www.bsc.es/user-support/power.php (accessed Aug. 24, 2023).

[17] Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock (accessed Aug. 24, 2023).