Exploring the Non-Uniform Memory Access Parallel Architecture Applied to the Protein Structure Prediction Problem

1st Felipe Marchi
Graduate Program in Applied Computing
Santa Catarina State University
Joinville, Santa Catarina, Brazil
felipe.r.marchi@gmail.com

2nd Rafael Stubs Parpinelli
Graduate Program in Applied Computing
Santa Catarina State University
Joinville, Santa Catarina, Brazil
rafael.parpinelli@udesc.br

***Abstract*—Proteins are base molecules present in live organisms. The study of their structures and functions is of considerable importance for many application fields, particularly for the pharmaceutical area. However, predict the structure of a protein is considered a complex problem. As optimizing methods for this problem have high execution time, a parallel algorithm is proposed. However, just employing parallelization is not enough to guarantee the efficient use of the available computational resources. In this work, the proposed Protein Structure Prediction (PSP) optimizer was executed in a system with Non-Uniform Memory Access (NUMA) architecture. To demonstrate the effects of this architecture on the execution of an algorithm with simple parallel model, experiments were carried. Results shows that the improper execution of a parallel algorithm in this architecture may lead to performance loss.**

I. Introduction

It is well-known that proteins spatial structure determines many of their essential biological functions [1]. Among possible applications involving protein structure prediction, it is possible to highlight drugs synthesis for specific uses and an in-depth analysis of diseases and their possible treatments.

The main structure generation methods are in laboratories and are costly and slow [1]. An alternative to the classical methods is computational prediction, which allows the simulations of protein structures in computers. Many studies and algorithms were developed through the years to make feasible the prediction of protein structures of any scale [2]. However, even with all the efforts made, the Protein Structure Prediction (PSP) problem is still viewed as a considerably complex problem without a practical and scalable solution.

The PSP problem is approached in this work with *ab initio* optimization using a multi-objective model. The *ab initio* PSP problem is usually optimized using a well-defined potential energy function as optimization objective. However, these functions are composed of several terms, and it is known that some of these terms may be conflicting objectives [3]. These conflicts happen because optimizing one objective does not implicate optimizing the others. As such, grouping these conflicting terms in a single objective will harm the optimization process.

Another possibility for a multi-objective model is to add different types of information that are not related as separated objectives. Considering the PSP context, these objectives could be high-level level information about the protein. In this work, the multi-objective model is composed of three objectives: energy function, secondary structure information, and contact map information.

Because of the costly energy functions used to evaluate the structures, computational parallelism can be employed in the algorithm. However, just employing parallelization is not enough to guarantee the efficient use of the available computational resources. There are different architectures of computers, and each of them may have some important characteristics that must be considered when developing and using parallel algorithms [4]. In the multi-core processor architecture, a single processor has multiple processing units, which can be used in parallel [5]. Multiples computers can be connected to work as single computer through cluster computing, allowing infrastructure heterogeneity and scalable growth. Another possible type of parallel architecture is the general-purppose computing on GPUs [6].

Some works in the literature have explored the multi-objective PSP problem with a parallel approach [7] [8] [9]. These works employ parallels models such as the master-slave model [7] and the island model [8]. None of them explores a specific computer architecture for parallel computing.

In this work, the proposed PSP optimizer is executed in a system with Non-Uniform Memory Access (NUMA) architecture. The objective is to demonstrate how the NUMA architecture modeling may influence the algorithm efficiency. To demonstrate these effects, experiments are proposed and executed to analyze the processing time and speedup of the parallel PSP predictor.

The remainder of this paper is organized as follows. Section II provides main concepts about what will be used in this work, finishing with an exposition of related works. The methodology is explained in Section III, which describes the algorithm used. Section IV describes the experiments and Section V provides the results and analysis of the experiments.

Section VI concludes this work.

II. Background

A. Parallel Computing

In general, computer programs are developed with a sequential flow of instructions. This linear flow of execution uses only a single computational processor, and is not able to fully use the resources of modern parallel computer architectures [10]. To be able to better utilize these resources, techniques and tools of parallel computing can be used.

One of the most common types of parallelism is CPU parallelism. Considering traditional operational systems, such as systems of the UNIX family, employing CPU parallelism usually means using processes and threads [5]. Although they display some similarities regarding parallelism, they have some important distinctions.

Threads are logical units that represent some flow of execution [5]. A thread is composed of a set of instructions, a call stack, and some private data. Threads usually can communicate with each other and share access to the same memory.

Processes are threads with a private address and a larger associated state [5]. Because of this characteristic, processes exhibit a greater computational cost than common threads. Also, the communication between processes is different from the communication between threads as processes do not share access to the same memory space.

By using the concept of threads and processes, some sequential programs can be divided into units that can be executed in parallel. The performance of a parallel algorithm will depend on the fraction of the algorithm that can be divided into parallel units and the number of available processors [10]. The allocation of threads to available processors can be configured through affinity. Thread affinity allows the placement of threads in specific cores [11].

Regarding processors, it is important to consider the concepts of physical and virtual cores. Some processors may use some type of virtualization technology (such as Intel™ *Hyper-Threading*), where a single physical core may be divided into multiple logical cores [11]. Although these cores can be seen as independent units, using multiples virtual cores of a single physical core does not have the same performance of using multiple physical cores [11].

Another important point to consider is multiprocessor architectures. These architectures can be defined by their memory access time, which can be classified as Uniform Memory Access (UMA), where the time to access some memory address is the same for all processors, and Non-Uniform Memory Access (NUMA) when the time to access some memory address will differ between the processors. Examples of these architectures are shown in Figures 1 and 2.

The main difference between these architectures is the division of memory to each processor. For single applications, the UMA architecture is interesting as it allows the use of multiple processors in a single memory space [12]. The NUMA architecture reduces the problem of having multiple processors trying the use the same memory, which can be useful for scenarios with spread data, such as servers [13].

Fig. 1: Example of UMA architecture

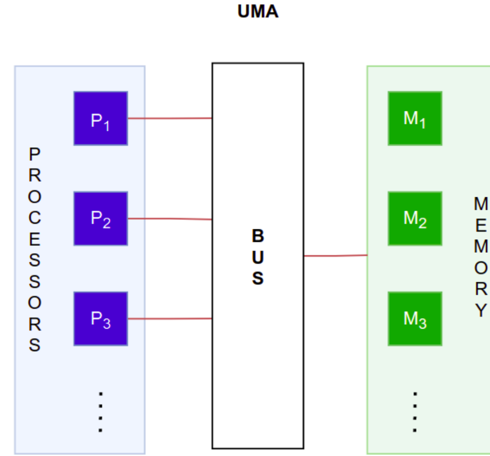
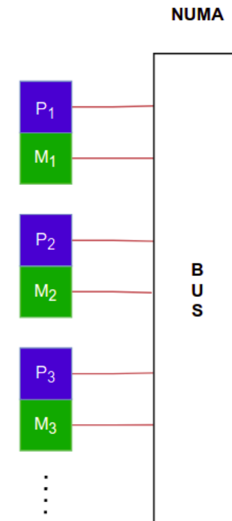


Fig. 2: Example of NUMA architecture



This classification is relevant, as the cost of memory access may impact the overall computational performance of the algorithm [11]. This becomes evident in algorithms with considerable amounts of inter-thread communication. Considering this, the modeling of algorithms with CPU parallelism has to consider not only threads and processes but also the processors themselves.

B. Protein Structure Prediction

Proteins are chains of amino acids, which are biological molecules identified by their side chains. It is possible to describe proteins by enumerating all their atoms and bonds. With all their atoms and bonds, such complete representations are usually computationally expensive to manipulate, and

simplified representations (e.g., using only torsion angles) are preferred.

Besides the structure representation, it is also necessary to define a function to evaluate a structure's quality. Energy functions integrate several physical and chemical interactions between the molecules that form the protein and environment molecules [1]. Although energy functions can model interactions that are interesting for realism, the computational complexity to evaluate such functions renders their use impracticable. As such, simplified functions composed of experimental data are used to evaluate structures and to guide optimizations.

C. Biased Random-Key Genetic Algorithm

The *Biased Random-Key Genetic Algorithm* (BRKGA) [14] is an optimization method of the evolutionary algorithms class. Evolutionary algorithms use biological evolution as a metaphor to describe generic optimization frameworks, also known as meta-heuristics. These frameworks utilize the concept of population, where each individual of the population is a candidate solution. These individuals are evolved through selection mechanisms, guided by their fitness and the combination of different biological operators, such as reproduction and mutation.

One main characteristic of this algorithm is the clear separation between the problem-dependent and independent parts. With this, it is possible to develop problem-specific methods by changing only the method's problem-dependent part. In the BRKGA, this separation occurs in the solution coding, where a decoding function is used to transform a problem-independent codification into a problem solution.

To approach the problem at hand, a multi-objective BRKGA was developed, named MO-BRKGA. The proposed algorithm uses the base structure of the original BRKGA. The main modifications were to allow the algorithm to optimize multi-objective problems. To achieve multi-objective optimization, modifications were made to the problem-independent parts. The elitist selection operator from NSGA-II [15] was used to generate the elite part of the population.

An archive (set of solutions) is maintained and updated at each generation. This archive has the same size as the population, and it is updated using the non-dominated sorting with crowding. At the end of the algorithm, the archive is returned and contains the best Pareto set found, although it may also contain dominated solutions as the best Pareto set may not occupy the entire archive. Maintaining dominated solutions is interesting as some problems may benefit from having sub-optimal solutions. In the case of the PSP, as the information used as the objectives is not exact, the optimal structure may not belong to the non-dominated set.

III. Method

In this work, a multi-objective model for the PSP problem is optimized. The optimization algorithm employed is a variant of the BRKGA with support to multi-objective problems. This algorithm, denominated MO-BRKGA, was also developed considering parallelism. This parallelism is promoted in the fitness evaluation, which is described below.

Evolutionary algorithms are inherently parallel, with multiple solutions being optimized concurrently through implicit parallelism [16]. As such, it is straightforward to parallelize these algorithms, with the most trivial parallelism being the fitness evaluation. The fitness evaluation of a candidate solution is independent of the others, which implies that individuals can be evaluated in parallel.

This fitness parallelism was implemented in the proposed MO-BRKGA. The algorithm was implemented in the language C++ and was parallelized using the OpenMP [1] 4.5 library. This library simplifies the development of parallel code, with the fitness evaluation in the MO-BRKGA being implemented as:

```
#pragma omp parallel for
for (int index = 0; index < population_.size(); ++index) {
    population_[index].evaluateFitness();
}
```

In this piece of code, the fitness evaluation of the entire population (*for* loop) was parallelized by just using the OpenMP pragma directive. Despite the simplicity, this directive effectively divides and distributes the loop among available processors. For a problem with costly fitness evaluation, such as the PSP, this parallelization is important. It is not uncommon for a single execution of a PSP optimizer to take hours to finish.

The proposed algorithm implements CPU parallelism. Although the performance scales with the number of processors, caution must be taken regarding the memory access architecture, as discussed on Section II-A. The improper use of these architectures may result in performance loss, even if all processors are fully utilized by the application.

To demonstrate the impact of the memory access architecture in the performance of algorithms with CPU parallelism, experiments were performed. A system with NUMA architecture was employed for these experiments. The objective is to show the evidence that improper use of this architecture may result in performance loss.

Three scenarios were considered for experiments:

1) Serial execution
2) Trivial parallel execution
3) Controlled parallel execution

The serial execution considers the serial version of the proposed algorithm. The trivial parallel version employs parallelism disregarding the memory access architecture. Finally, the controlled version considers the architecture by controlling the use of the processors.

The trivial parallel version does not uses any kind of configuration other than the C++ directive specified above. As such, the system is free to map the created threads to any available processors. The controlled version, however, specifies how these threads should be allocated.

To control the distribution of threads into processors, two mechanisms were utilized. The first mechanism was to use specific OpenMP control options. These options are:

[1] https://www.openmp.org/

- `OMP_PROC_BIND = close`: this option changes the thread affinity of the application. The idea is to keep threads from the same process close, which effectively means using the same processor unit.
- `OMP_PLACES = cores`: by using this option, the application indicates that physical cores should be used for each thread. This options is important as usually threads can run in multiple virtual cores of the same physical core simultaneously.

The second mechanism used was the `numactl`[2] tool. This program allows to control how a specific application will use the current NUMA architecture. With this, it is possible to run a process in a single NUMA node, with the application using only the processors of this node.

IV. Experiments Setup

The experiments were performed in Ubuntu 18.04 system, with Intel Xeon E7-8860 processors and 1TB RAM. This system has a NUMA architecture with 4 nodes, each composed of a single processor with 10 physical cores and 10 virtual cores (1 virtual for each physical).

To test the scenarios specified in Section III, four proteins were selected to be optimized by the proposed MO-BRKGA method. These proteins are displayed in Table I. Also, for each protein the algorithm was executed with 1, 2, 4, 8, 10, 20, 40, and 80 threads. The algorithm was executed 20 times for each case, using 1,000,000 fitness evaluations each run.

TABLE I: Proteins utilized

Protein	Size
1ZDD	34
1GB1	56
1AIL	73
1HHP	99

For each run of the algorithm the execution time was measured. The average time and mean deviation of each scenario was calculated for all proteins. These results were analyzed in Section V.

V. Results and Analysis

In this section the results of each scenario will be presented and analyzed. The average time and speedup were measured for each protein.

A. Serial execution

The average time and standard deviation in seconds for the serial execution of the algorithm are shown in Table II. It is possible to see that the execution time is directly proportional to the protein size. This relation appears to be close to a linear relationship, with approximately 25 minutes of execution time per 100 amino acids.

[2]https://linux.die.net/man/8/numactl

Fig. 3: Plot of the average time for the trivial parallel execution

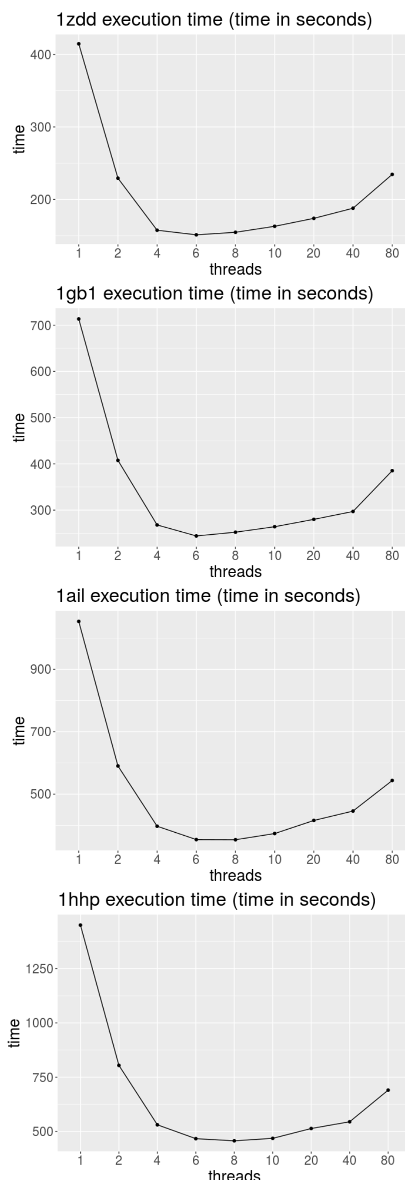


Fig. 4: Plot combining the average time of all proteins for the trivial parallel execution

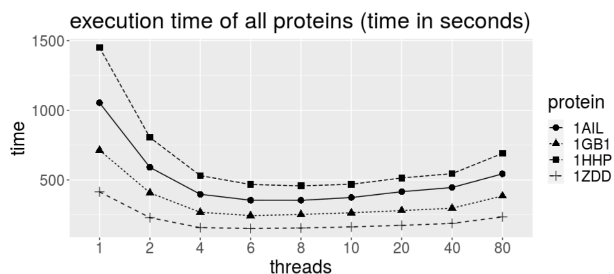


Fig. 5: Plot of the average speedup for the trivial parallel execution

Fig. 6: Plot combining the average speedup of all proteins for the trivial parallel execution

Fig. 7: Plot of the average time for the controlled parallel execution

Fig. 8: Plot combining the average time of all proteins for the controlled parallel execution

Fig. 9: Plot of the average speedup for the controlled parallel execution

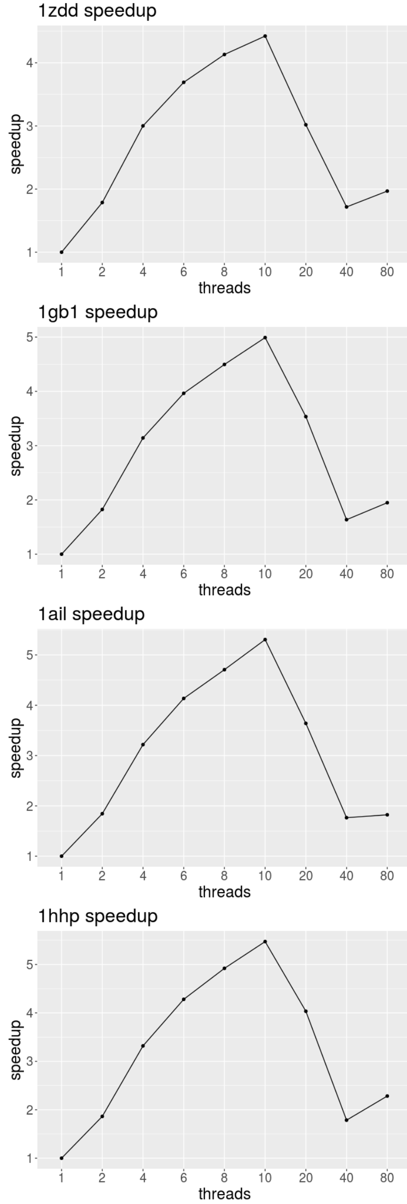


Fig. 10: Plot combining the average speedup of all proteins for the controlled parallel execution

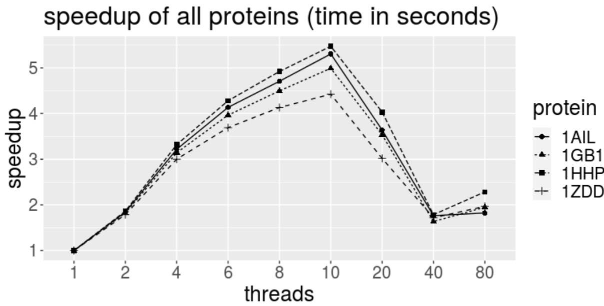


TABLE II: Average execution time in seconds for the serial algorithm

Protein	Time
1ZDD	414.54 ± 1.28
1GB1	713.43 ± 3.89
1AIL	1053.48 ± 15.57
1HHP	1449.87 ± 17.64

B. Trivial parallel execution

In this experiment, the parallel version of the algorithm was run without explicit control of the available processors. The average execution time for each protein is displayed in Figure 3, Figure 4, and in Table III. Figures 5 and 6 show the average speedup for each protein.

TABLE III: Average time in seconds for the trivial parallel execution

	1ZDD	1GB1	1AIL	1HHP
1	414.5 ± 1.2	713.4 ± 3.8	1053.4 ± 15.5	1449.8 ± 17.6
2	229.3 ± 1.2	407.6 ± 4.9	590.2 ± 9.2	804.6 ± 14.9
4	157.6 ± 1.7	267.9 ± 3.4	397.1 ± 8.3	531.1 ± 9.6
6	151.3 ± 2.3	244.1 ± 3.6	354.2 ± 7.0	467.4 ± 11.7
8	154.8 ± 1.5	252.2 ± 6.3	353.8 ± 9.0	457.6 ± 12.0
10	163.0 ± 1.7	264.1 ± 8.1	373.6 ± 6.4	468.9 ± 14.8
20	174.0 ± 2.9	280.0 ± 9.9	415.6 ± 10.3	514.2 ± 11.2
40	187.9 ± 2.0	297.1 ± 2.4	445.8 ± 13.7	545.2 ± 15.3
80	234.5 ± 7.5	385.2 ± 3.0	543.6 ± 15.4	690.6 ± 21.5

It is possible to see from the plot in Figure 6 the linear relationship between the protein size and execution time. All the graphs show a similar behavior regarding the number of employed threads. The graphs indicate that the best acceleration was found between 6 and 8 threads.

Regarding the speedup, the graphs demonstrate that the maximum acceleration was approximately a factor of 3. Even though each processor has 10 physical cores, the algorithm was not able to use efficiently these resources.

C. Controlled parallel execution

In this experiment, the parallel algorithm was executed using the control options presented in Section IV. The average execution time for each protein is displayed in Figure 7, Figure 8, and in Table IV. Figures 9 and 10 show the average speedup for each protein.

TABLE IV: Average time in seconds for the controlled parallel execution

	1ZDD	1GB1	1AIL	1HHP
1	414.5 ± 1.2	713.4 ± 3.8	1053.4 ± 15.5	1449.8 ± 17.6
2	232.0 ± 1.0	391.0 ± 2.9	571.2 ± 9.7	778.1 ± 11.5
4	138.1 ± 0.5	227.0 ± 1.5	327.3 ± 3.3	436.8 ± 11.2
6	112.3 ± 0.7	179.9 ± 1.1	254.7 ± 3.8	338.7 ± 5.4
8	100.3 ± 0.4	158.6 ± 1.1	223.8 ± 2.7	294.6 ± 5.4
10	93.7 ± 0.4	201.8 ± 2.0	198.5 ± 1.4	264.9 ± 6.8
20	137.3 ± 0.9	142.9 ± 1.2	289.4 ± 3.4	359.4 ± 6.9
40	241.4 ± 70.4	436.1 ± 60.3	596.8 ± 101.1	812.0 ± 100.6
80	210.6 ± 44.7	366.1 ± 86.9	577.9 ± 135.7	634.9 ± 133.8

Observing the graphs, it is clear that the algorithm was able to better use the available infrastructure. The maximum acceleration was found using 10 threads, which is the number of physical cores of a single processor. By using the OpenMP options, the algorithm was able to map threads to physical cores of the same processor, minimizing the communication cost between threads. With this, the performance of the same parallel algorithm increased almost two-fold.

With this setup, the algorithm was able to run 5 times faster than the serial counterpart. Although the maximum performance occurred with 10 threads, greater numbers of threads reduced this performance considerably. This makes sense, as the maximum number of physical cores in a single processor in this experiment is 10. If more than 10 threads are necessary the algorithm will either use virtual cores or use cores from other processors, increasing the overhead and resulting in performance loss.

Moreover, running the algorithm in more than one processor will always result in this performance loss. As such, one way to effectively use this architecture is to run multiples executions of the algorithm in parallel. As each process of the algorithm is independent, it is possible to map them to different processors.

Considering the context of this work, this mapping was done using the `numactl` program presented in Section IV. Using this program, it is possible to control the thread affinity, mapping four processes to four processors. As each process and processor is independent, it is possible to reach a linear increase of speedup. Considering that the best speedup found was approximately 5 by using 10 threads, by applying this distribution of processes to each processor, it should be possible to reach a speedup of approximately 20. With this, the final parallel methodology is able to execute the proposed algorithm approximately 20 times faster than the correspondent serial algorithm.

VI. Conclusion and Future Work

This work had the objective to demonstrate the impact and importance of efficient parallelization of algorithms. The algorithm used is a predictor for the PSP problem. As PSP optimizers are known to be slow, parallel computing is important to decrease high execution times.

In this work, the CPU parallelization model was employed. In this model, concepts such as threads, processes and memory access architectures are important.

Experiments were conducted considering the serial versions of the proposed algorithm, and an trivial and controlled execution of the parallel algorithm. These experiments were performed on a system with NUMA architecture. Results showed that not taking in consideration the available infrastructure may result in performance loss. The controlled parallel execution is considerably more efficient than the trivial counterpart, as it was able to better utilize the available computational resources.

For future works, other types of architecture could be explored. In this work, a master-slave parallel model was employed in the algorithm. There are, however, others models that could better utilize a NUMA architecture, such as the island model.

Acknowledgment

The authors would like to thank the Santa Catarina State University for the financial support.

References

[1] J. Gu and P. E. Bourne, *Structural bioinformatics*. John Wiley & Sons, 2009, vol. 44.

[2] M. Dorn, M. B. e Silva, L. S. Buriol, and L. C. Lamb, "Three-dimensional protein structure prediction: Methods and computational strategies," *Computational biology and chemistry*, vol. 53, pp. 251–276, 2014.

[3] V. Cutello, G. Narzisi, and G. Nicosia, "A multi-objective evolutionary approach to the protein structure prediction problem," *Journal of The Royal Society Interface*, vol. 3, no. 6, pp. 139–151, 2006.

[4] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[5] C. Lin *et al.*, *Principles of parallel programming*. Pearson Education India, 2008.

[6] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 1. IEEE, 2004, pp. 805–808.

[7] J. C. C. Tudela and J. O. Lopera, "Parallel protein structure prediction by multiobjective optimization," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2009, pp. 268–275.

[8] D. Becerra, A. Sandoval, D. Restrepo-Montoya, and F. N. Luis, "A parallel multi-objective ab initio approach for protein structure prediction," in *2010 IEEE international conference on bioinformatics and biomedicine (BIBM)*. IEEE, 2010, pp. 137–141.

[9] J. C. Calvo, J. Ortega, and M. Anguita, "Comparison of parallel multi-objective approaches to protein structure prediction," *The Journal of Supercomputing*, vol. 58, no. 2, pp. 253–260, 2011.

[10] S. H. Roosta, *Parallel processing and parallel algorithms: theory and computation*. Springer Science & Business Media, 2012.

[11] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[12] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*. John Wiley & Sons, 2005, vol. 42.

[13] Z. Majo and T. R. Gross, "Memory system performance in a numa multicore multiprocessor," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011, pp. 1–10.

[14] J. F. Gonçalves and M. G. Resende, "Biased random-key genetic algorithms for combinatorial optimization," *Journal of Heuristics*, vol. 17, no. 5, pp. 487–525, 2011.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[16] A. E. Eiben, J. E. Smith *et al.*, *Introduction to evolutionary computing*. Springer, 2003, vol. 53.