Protein Structure Prediction with Parallel Algorithms Orthogonal to Parallel Platforms

Matheus Henrique Junqueira Saldanha[1], **Paulo Sérgio Lopes de Souza**[1]

[1]Institute of Mathematics and Computer Sciences
University of São Paulo – São Carlos, SP – Brazil

mhjsaldanha@gmail.com, pssouza@icmc.usp.br

Abstract. *The problem of Protein Structure Prediction (PSP) is known to be computationally expensive, which calls for the application of high performance techniques. In this project, parallel PSP algorithms found in the literature are being accelerated and ported to different parallel platforms, producing a set of algorithms that it is diverse in terms of the parallel architectures and parallel programming models used. The algorithms are intended to help other research projects and they have also been made publicly available so as to support the development of more elaborate prediction algorithms. We have thus far produced a set of 16 algorithms (mixing CUDA, OpenMP, MPI and/or complexity reduction optimizations); during its development, two algorithms that promote high performance were proposed, and they have been written in an article that was accepted in the International Conference on Computational Science (ICCS).*

1. Introduction

Proteins are biological macromolecules consisting of a chain of smaller monomers, the *amino acids*. They are omnipresent in living beings, and are essential to their correct functioning, so much that problems in their synthesis are directly related to *proteases* like Alzheimer's and Parkinson's diseases [Bourne and Weissig 2003]. From a more positive point of view, proteins are also closely related to chemical phenomenons of interest, such as bioluminescence (luciferins and luciferases), degradation of PET bottles (petases) and conversion of light to electrical signals (photopsins). In this light, a better understanding of proteins and their synthesis process seems to be of medical, biological and environmental interest; for example, if petases could be improved, this could help cleaning PET bottles, whose degradation process is unsustainably long, from the environment, especially the sea.

The function of a protein depends on its three dimensional structure which, in turn, is known to be uniquely determined by its amino acid sequence. Protein Structure Prediction (PSP) is the study of how computers can be used to predict the 3D structure of a protein whose amino acid sequence is given. This can avoid the high cost and time requirements of *in vitro* experiments, and also circumvent the difficulty of simulating *in vivo* aspects, such as agents (e.g., chaperones) that interact with proteins during the folding process [Balchin et al. 2016]. There exist many ways to perform such prediction computationally, which can be grouped as 1) *template-based*, which use knowledge that is available about existing proteins in openly available data banks; and 2) *ab initio* algorithms, whose input is solely the amino acid sequence of the protein to predict. Each

group has its own utility, and PSP algorithms used in global competitions such as the CASP may combine results of both types [Moult et al. 2016].

This Scientific Initiation project aims to contribute to the area of scheduling different parallel PSP algorithms over multiple available hardware devices whose computation power can differ a lot from each other. The problem of efficient scheduling is NP-complete in many of its formulations [Norman and Thanisch 1993], and a better understanding of each algorithm's performance characteristics can help finding a computational model that is more easily dealt with [Norman and Thanisch 1993, Kwok and Ahmad 1999]. This requires a set of parallel PSP algorithms that are diverse in terms of the parallel platforms they can exploit, which are not publicly available in a sufficient amount; our project would help solve this deficiency.

The objectives of our project would be threefold: 1) implement parallel PSP algorithms for usage by other research projects; 2) port the implemented algorithms to exploit different kinds of parallelism; and 3) search for opportunities to accelerate them. We have thus far implemented a total of 16 algorithms: four algorithms found in the literature (two sequential and two parallelizations thereof); and twelve versions that resulted from this project by accelerating the first four, either by porting them to use different combinations of cluster, multicore and/or GPU platforms, or by reducing the computational complexity of a sequential procedure. Algorithms we have devised were written in an article that was accepted in the International Conference on Computational Science (ICCS) (already available as preprint [Saldanha and de Souza 2019]).

By fulfilling these objectives, we expected to help research in the area of PSP, by providing a set of algorithms that were diverse in terms of supported parallel platforms and performance characteristics. The implemented algorithms have been documented and made publicly available[1], so they can also be used by students interested in learning about PSP or by other projects that attempt to orchestrate multiple PSP algorithms to achieve a more ambitious objective.

We begin narrating the present project by exposing methods used for investigation (Section 2), and then describe the course followed to pursuit the defined objectives (Section 3). In Section 4 are shown two algorithms that we proposed, and in Section 5 the set of algorithms produced is described by means of experimental results.

2. Utilized Methods

Experimental analysis was mainly guided by two laws. Amdahl's law gives the ideal speedup of parallelizing a program over p processors, namely $\frac{1}{f+(1-f)/p}$ where f is the portion of execution time used by routines that will remain sequential; this guides us to focus on parallelizing portions that consume most of the execution time. Gustafson's law expands the ideal speedup analysis on the dimension of the problem size [Rauber and Rünger 2013], saying that the ideal speedup might differ a lot between small and big problem sizes depending on the computational complexity of internal procedures. This leads the analysis to take the expected problem size into consideration and apply Amdahl's law with an extra level of care. Supported by these laws, algorithms are then profiled in order to determine their performance characteristics on different problem sizes

[1]More information at `https://mjsaldanha.com/sci-projects/1-psp-project-1/`.

or on a single, reasonable one. Theoretical analysis was guided by computational complexity concepts and the notions of *depth-* and *work-complexity* [Blelloch 1996], which are specific for parallel algorithms.

For the design of algorithms we used Foster's PCAM methodology [Foster 1995], in which the analyst goes through four phases. In the *partition* phase, the algorithm is split into as many tasks as possible, each of which consists of instructions and a small private memory. In *communication* the necessary inter-task communication is established, and in *agglomeration*, tasks are joined together in order to minimize communication. Finally, in *mapping*, tasks are assigned to physical processors, possibly using a scheduling algorithm. In this project, two PCAM analyses were performed and enabled the proposal of many algorithms, the most fruitful of which are commented in Section 4.

3. Course of Investigation

We initially gathered, from the literature, ten articles that propose parallel PSP algorithms, then ranked them according to number of citations and relevance of the publishing conference or journal. We chose to investigate the second and third best ranked ([Chu and Zomaya 2006] and [Benítez and Lopes 2010] respectively), since the first one's complexity and dependency on external libraries seemed to leave the project's scope.

Investigation began with article [Benítez and Lopes 2010], which applies Artificial Bee Colony Optimization in solving the PSP problem and proposes a distributed version thereof, using MPI library for parallelization. After implementing both the sequential and parallel versions ourselves, experiments were performed to profile them. We found that two procedures consumed most of the execution time: 1) counting collisions among beads in the 3D space, which represent the protein's subunits; and 2) given a sequence of relative movements (e.g., *front–left–up–up*) reconstruct the protein as a set of beads in the 3D space. However, complying with Gustafson's law (see Section 2) we also identified that counting collisions is a $O(N^2)$, whereas reconstruction is $O(N)$.

PCAM analysis was then applied to these two procedures, being more fruitful for collision counting, to which we devised the two algorithms elaborated in Section 4. For reconstruction, the analysis yielded that it could be formulated as a *reduction* operation, which is not nearly as parallel as collision counting, would be more complex to implement and would result in diminishing returns for bigger problem sizes (due to Gustafson's law and the lower complexity compared with collision counting); for all these reasons, we did not investigate it further.

We followed a similar course for investigating the second article, namely [Chu and Zomaya 2006], which applies Ant Colony Optimization to the PSP problem. In this case, profiling showed that execution time was well spread over multiple procedures; at a higher level, however, most of it was evenly spread over the work performed by each of the N ants in the colony. PCAM analysis was then applied, focusing in how ants could operate in parallel; then parallelizations were proposed and implemented. Results are shown in Section 5.

4. High Performance Algorithms for Collision Counting

As mentioned previously, the investigation of PSP algorithms led to the proposal of algorithms for counting collisions or, more generally, summing symmetric pairwise interactions (SPI). This problem considers a set of objects $\{b_1, ..., b_N\}$ upon which is defined a commutative operation $b_i \circ b_j$; this could return, for example, gravitational forces between planets, or a boolean value representing whether b_1 and b_2 collide or are in contact. In such framework, if the interest resides in the sum over the set $\{b_i \circ b_j : 1 \leq i < j \leq N\}$ then the algorithms discussed here apply.

The problem as formulated above appears frequently in many areas: in computer graphics and virtual reality, collision among objects or friction among hair strands (as done in [Selle et al. 2008]) has to be calculated on a per-frame basis; in simulations, gravitational, electrical or magnetic forces are often of interest; in robotics, it is important to calculate intersection relations between the robot's current direction and each visible object. This problem is known to be a major bottleneck in many applications [Lin and Gottschalk 1998], as there are $O(N^2)$ interactions to be evaluated, so it has been extensively investigated. However, most approaches focus on pruning the set of objects within which sum of SPI is performed (mainly by spatial division [Elseberg et al. 2012] and bounded volume hierarchies [Stich et al. 2009]), or in approximating the forces of distant objects (mainly by means of the Fast Multipole Method [Greengard and Rokhlin 1987]). In any case, within these algorithms the usual SPI counting algorithm is still used, so accelerating it implies improvement in all aforementioned methods.

The straightforward algorithm for counting SPI is presented in Algorithm 1. It can be seen that all iterations of the nested loops would be fully data-parallel if it were not for the reduction variable *interactions*. This is often parallelized to GPU in two phases: first the interactions are calculated by each GPU thread, and then are *reduced* to a single value. The reduction operation has known efficient parallel implementations; for the first phase, however, the straightforward parallelization would consist of assigning each iteration of the outer *for-loop* to one GPU thread. This parallelization indeed allows for intensive usage of the GPU's computational resources and available memory bandwidth, even though the outer *for*'s iterations are not balanced in terms of work performed. The GPU manages to hide most of the inefficiency that could be caused by threads performing a different amount of work compared with "neighbor" threads; in particular, the separation of threads in groups (called *warps*) of 32 prevents most of the idle work that could be caused by this imbalance.

Algorithm 1: Standard algorithm for calculating SPI.

```
for (i = 0 to N-1)
  for (j = i+1 to N-1)
    interactions += interact(obj[i],
                             obj[j]);
```

Algorithm 2: Proposed algorithm for calculating SPI.

```
for (i = 0 to N-1)
  for (j = 1 to (N-1)/2)
    interactions += interact(obj[i],
                             obj[(i+j)%N]);
```

Within a warp, though, such inefficiency still happens. In Algorithm 2 is shown our proposal, which is a small alteration of the sequential algorithm that, when parallelized to the GPU, overcomes this inefficiency. In this case, each outer iteration is also assigned to a thread, so that thread i compares bead b_i with exactly $(N-1)/2$ following

beads (in a circular way, so bead b_1 "follows" b_N). This is in contrast to the previous parallelization, in which thread i compared bead b_i with all following beads up to b_N, hence threads with higher index performed fewer comparisons.

In [Saldanha and de Souza 2019] this proposal is proved to be equivalent to the original algorithm, using modular arithmetic. In short, there are $N(N-1)/2$ distinct symmetric pairwise interactions to be evaluated, which is precisely what the original algorithm considers. We first prove that the proposed algorithm evaluates $N(N-1)/2$ interactions, and then prove they are different, concluding the proof of equivalence.

Both algorithm versions were implemented, with shared memory being used in the same way and using the same reduction function, and experiments were performed with an NVIDIA Tesla P100 with 16GB of memory, 3584 CUDA cores and 56 multiprocessors. The results are presented in Figure 1 (left), in which it can be seen that the proposed approach is faster than the straightforward one. For all problem sizes higher than $525\,000$ the proposed approach is at least 12% faster ($p < 0.01$ using a t-test assuming unknown and different variances).

In contrast with the first proposal, which considers beads in $\mathbb{R}^m$, the second proposal imposes a restriction on such space. Here beads must be in an $\mathcal{S}^3$ such that

$$\begin{cases} \mathcal{S}^3 = \mathcal{S} \times \mathcal{S} \times \mathcal{S} \\ \mathcal{S} = \{-a,\ -a+1,\ \ldots,\ a-1,\ a\} \subset \mathbb{Z} \end{cases}$$

which means that the space is discrete and bounded. Besides this, it is also required that the interaction of interest is either collision ($b1 = b2$) or contact ($||b1 - b2|| = 1$). For counting contacts among beads in such space, we propose Algorithm 3, which receives as argument the vector of beads and a pointer to a memory region that has one element for each point in $\mathcal{S}^3$. For simplicity, we assume the pointer can be dereferenced with negative indices; for now we also assume it is zero-initialized, which is elaborated later.

Algorithm 3: Counting the number of contacts among a vector of beads.

```
1  int countContacts(point3D beads[], int space[][][]){
2      int contacts = 0;
3      for (b in beads) space[b.x][b.y][b.z] += 1;
4      for (b in beads){
5          contacts += space[b.x+1][b.y][b.z] + space[b.x-1][b.y][b.z];
6          contacts += space[b.x][b.y+1][b.z] + space[b.x][b.y-1][b.z];
7          contacts += space[b.x][b.y][b.z+1] + space[b.x][b.y][b.z-1];
8      }
9      return contacts / 2;
10 }
```

The algorithm begins by incrementing memory elements associated with each bead's position (line 3). After this process, each memory element holds the number of beads located in the associated position. Following, the algorithm iterates over the vector of beads again: for each bead, it reads the number of beads located in each neighboring position, and adds it to the *contacts* variable. A problem arises here because each contact is being counted twice: if b_i and b_j are in contact, then it is being counted in iterations

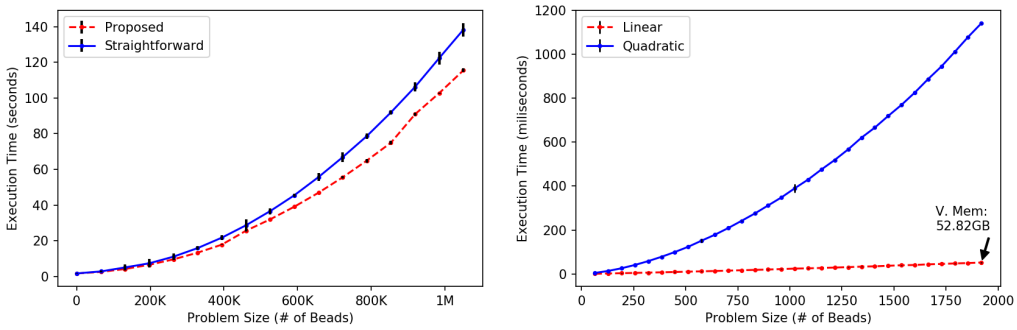

Figure 1. Experimental results. Each sample point is the mean taken from 100 executions, and the vertical length of the black error bars equals 4 standard deviations. (*left*) Comparison of the straightforward parallelization for GPU with the proposed one. (*right*) Comparison of the sequential approaches: the usual algorithm and the proposed one with $O(N)$ complexity.

for both b_i and b_j. Consequently, the function returns half of the counted contacts. Initialization of the `space` memory region is done by iterating over the vector of beads one more time, initializing the six memory elements associated with neighbor positions of each bead.

It is readily seen that the proposed algorithm has $O(N)$ complexity, though it can consume a large amount of memory. Using a computer with an Intel i7-4790 3.6GHz and 32GB of primary memory, we performed experiments with the usual and proposed approaches for calculating contacts among beads of multiple randomly generated proteins, obtaining results shown in Figure 1 (right). Bigger proteins require a larger memory region, which is why the proposed approach allocated about 52GB of virtual memory at problem sizes of about 1900 beads. In [Saldanha and de Souza 2019] we argue that many problems, such as one of the PSP algorithms investigated, will perform contact counting multiple times, so the memory region can be reused. Further, the probability distribution of the beads' positions may be concentrated in a certain region, as happens in PSP where compact proteins are favored by the optimization algorithm. This prevents swapping and allows a good speedup to be obtained when compared with the $O(N^2)$ approach.

5. Heterogeneous Platform PSP Algorithms

Complying with the objectives initially defined for the project, we have produced 16 algorithm versions, 14 of which use some sort of high performance artifice: CUDA, MPI, OpenMP and/or sequential optimizations; therefore they are algorithms that can exploit GPUs, multicore CPUs and/or clusters of computers.

In [Benítez and Lopes 2010] is proposed the first PSP algorithm we investigated. The authors apply the Artificial Bee Colony optimization algorithm to predict the structure of a protein modelled according to their proposed model, in which each amino acid is represented by two relevant mass centers (the backbone and side-chain centers). They also propose an MPI parallelization, where the available processing nodes are divided in groups called *hives*, each running the optimization algorithm in a master-slave fashion, and the masters of each hive periodically exchange proteins with each other in a ring topology. We have implemented their sequential and parallel versions; let them be called

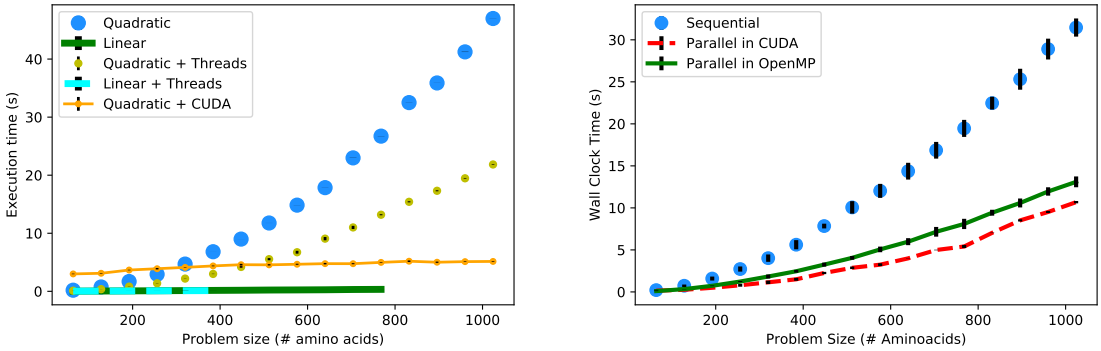

Figure 2. Execution time of the PSP algorithms produced. Each sample point is the mean of 10 executions, and the black error bar's vertical length equals 4 standard deviations. (left) Algorithms from the first investigated algorithm. (right) From the second investigated algorithm.

the S_1 set of versions. Each of these was enhanced in this project by using other acceleration techniques. First, the contact and collision counting procedures were changed to the linear version elaborated in Section 4, generating a new set S_2. Second, both sets S_1 and S_2 were parallelized to evaluate different proteins simultaneously in multiple threads, using OpenMP, generating set S_3 of versions. Finally, set S_1 had its counting procedures parallelized to CUDA, using the efficient algorithm that we proposed (see Section 4).

Experiments to evaluate these versions were performed in a computer with a GPU GeForce 940M, an Intel i7-5500U 2.40GHz and 8GB primary memory. Since the MPI parallelization is not a product of this project, and due to the large number of versions, we experimented only with the original sequential version and its enhancements proposed in this project. Results are shown in Figure 2 (left), where it can be seen that the initial sequential version (big blue dots) is the slowest. Its parallelization with OpenMP (small yellow dots) yielded a speedup slightly above 2.0, which is reasonable for a machine with two physical cores (four virtual). The linear approach (thick green solid line) provided the best results (reasons are given in Section 4), although the machine could only provide enough virtual memory up to problem size 768. Its parallelization for two threads (thick cyan dashed line) resulted in a 1.30 speedup, much less than the ideal, likely due to competition between both threads for accessing memory. In this case, the virtual memory was exhausted at problem size 384 because each thread requires its own memory region. Finally, the CUDA parallelization (thin orange solid line with small dots) initially performs worse than all other versions, due to the high fixed cost of data transfer between primary and GPU memory. At larger problem sizes, however, this version becomes better than both other versions that use $O(N^2)$ procedures for collision and contact counting.

The second investigated PSP algorithm comes from [Chu and Zomaya 2006], where Ant Colony Optimization (ACO) is applied to the PSP problem, and four distributed versions thereof are also proposed. We implemented, and then enhanced, the authors' sequential approach and the proposed distributed version named "round robin – multiple colonies". In this version, the available processors execute the ACO independently, periodically exchanging best solutions with neighbor processors in a ring topology. As mentioned in Section 3, the execution time for this algorithm is well spread over multiple procedures, so we opted to parallelize the whole activity of an ant, which comprises multiple procedures. This was also an attempt to make the GPU version provide speedups

even for small problem sizes, which was not the case with the first algorithm.

Experiments with the versions produced in this project were performed in the same conditions as the first algorithm; the results are shown in Figure 2 (right). Again, as the MPI parallelization is not ours, we here show experiments only with the original sequential versions and enhancements thereof made in this project; all versions are available in public repositories[2]. The original versions were parallelized with OpenMP (green solid line), by distributing ants to each thread and synchronizing their access to the pheromone matrix. Besides that, we also implemented the ant colony in the CUDA programming model (red dashed line), which involved migrating hundreds of lines of CPU code to run in GPU. As seen in Figure 2, both produced versions are faster than the original algorithm. The OpenMP parallelization resulted in 2.39 speedup, which is reflects the machine having 4 virtual cores (2 physical). The CUDA version was the fastest one for all problem sizes, which is a positive characteristic when compared with results from the first algorithm investigated. However, this version resulted in 2.95 speedup for the largest problem size tested, which does not seem to be good considering the amount of energy consumed and the computational power provided by the GPU. Investigation of this version has not ended, and we are now attempting to devise ways of increasing such speedup.

The set of algorithms produced is not without limitations. The area of PSP is very large, and there are multiple ways to tackle the problem of predicting protein structures. Some algorithms use data available in international protein data banks, others consider more aspects of protein folding, such as the interaction of the protein with the surrounding fluid or the ribosome during folding. Moreover, PSP algorithms used in competitions such as the CASP [Moult et al. 2016] may also combine results from multiple algorithms with diverse characteristics. In this project, both investigated PSP algorithms are part of a class called *ab initio*, which predicts the structure solely based on the protein's amino acid sequence; not using data available in data banks is seen as a better way of predicting proteins to which there does not exist similar "templates" (similar proteins). Also, both algorithms use HP (hydrophobic-polar) models, in which each amino acid is represented by a small number of beads. It could instead model all of the protein's atoms, which would characterize a *full-atom* representation; in this case, the protein can be encoded, for example, with a few beads per amino acid (representing relevant mass centers) and the angle between amino acids [Xu and Zhang 2012]. Full-atom models tend to have more mathematical calculations, so they take longer than HP models, but may result in more precise predictions. One limitation of our project is thus having investigated only HP models.

6. Conclusion

As the performance growth of a single processor core decreases, it is all the more important to use high performance computing (HPC) techniques to accelerate programs that, despite their great importance, take too long to execute. This project brings a contribution to the area of Protein Structure Prediction (PSP) by applying parallel programming and other algorithmic optimizations into existing PSP algorithms. The implemented algorithms are available for other researchers in the area of PSP, who could use them to

[2]Available at `https://mjsaldanha.com/sci-projects/1-psp-project-1/` under Section "Public Resources".

perform predictions more quickly or to compose a bigger algorithm that combines the output of multiple algorithms to produce a better prediction.

The investigation also resulted in a contribution that reaches many areas other than PSP. As said in Section 4, the proposed algorithms for counting interactions could be used in computer graphics, virtual reality, robotics and simulations of natural phenomena. This contribution resulted in an article [Saldanha and de Souza 2019] in which one of the algorithms is mathematically shown to be correct, and both algorithms are shown experimentally to yield benefits in terms of execution time.

The CUDA parallelization of the second algorithm is on the process of being investigated, aiming to use shared memory more efficiently or reduce thread divergence caused by ants (which are executed by a thread) that end up following different control flow paths, which impacts performance in the GPU. Future work should also focus on applying HPC in other kinds of PSP algorithms, such as those that use full-atom representations of proteins. Finally, we believe that all objectives established upon this project's conception were achieved, and the obtained results and contributions exceeded our initial expectations.

7. Acknowledgement

We thank São Paulo Research Foundation (FAPESP) for funding this research project (grant 2017/25410-8, associated to 2013/07375-0), and the Center for Mathematical Sciences Applied to Industry (CeMEAI) for providing access to powerful computational resources.

References

Balchin, D., Hayer-Hartl, M., and Hartl, F. U. (2016). In vivo aspects of protein folding and quality control. *Science*, 353(6294):aac4354.

Benítez, C. V. and Lopes, H. (2010). Parallel artificial bee colony algorithm approaches for protein structure prediction using the 3dhp-sc model. *Intelligent Distributed Computing IV*, pages 255–264.

Blelloch, G. E. (1996). Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97.

Bourne, P. E. and Weissig, H. (2003). *Structural bioinformatics*, volume 44. John Wiley & Sons.

Chu, D. and Zomaya, A. (2006). Parallel ant colony optimization for 3d protein structure prediction using the hp lattice model. In *Parallel Evolutionary Computations*, pages 177–198. Springer.

Elseberg, J., Magnenat, S., Siegwart, R., and Nüchter, A. (2012). Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12.

Foster, I. (1995). *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Reading.

Greengard, L. and Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348.

Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471.

Lin, M. and Gottschalk, S. (1998). Collision detection between geometric models: A survey. In *Proceedings of IMA conference on mathematics of surfaces*, volume 1, pages 602–608.

Moult, J., Fidelis, K., Kryshtafovych, A., Schwede, T., and Tramontano, A. (2016). Critical assessment of methods of protein structure prediction: Progress and new directions in round xi. *Proteins: Structure, Function, and Bioinformatics*, 84:4–14.

Norman, M. G. and Thanisch, P. (1993). Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302.

Rauber, T. and Rünger, G. (2013). *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media.

Saldanha, M. H. J. and de Souza, P. S. L. (2019). High performance algorithms for counting collisions and pairwise interactions. *arXiv preprint arXiv:1901.11204*. Available at `https://arxiv.org/abs/1901.11204`.

Selle, A., Lentine, M., and Fedkiw, R. (2008). A mass spring model for hair simulation. *ACM Transactions on Graphics (TOG)*, 27(3):64.

Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13. ACM.

Xu, D. and Zhang, Y. (2012). Ab initio protein structure assembly using continuous structure fragments and optimized knowledge-based force field. *Proteins: Structure, Function, and Bioinformatics*, 80(7):1715–1735.