Understanding the I/O Impact on the Performance of High-Throughput Molecular Docking

Stefano Markidis[1,2], Davide Gadioli[2], Emanuele Vitali[2], Gianluca Palermo[2]
[1] *KTH Royal Institute of Technology*, Stockholm, Sweden
[2] *Politecnico di Milano*, Milano, Italy

***Abstract*—High-throughput molecular docking is a data-driven simulation methodology to estimate millions of molecules' position and interaction strength (ligands) when interacting with a given protein site. Because of its data-driven nature, the high-throughput molecular docking performance depends on how fast we can ingest data into the processing pipeline and how efficiently we can write molecular docking results to a shared file. In this work, we characterize the I/O performance of a high-performance high-throughput molecular docking application, called Docker-HT, running on a supercomputer up to 512 computing nodes with two different parallel I/O configurations. We show that a tuned I/O configuration can improve the overall parallel efficiency from 71% to 90% on 512 nodes and identify and solve a performance degradation observed when running on 16 and 32 nodes.**

***Index Terms*—High-Throughput Molecular Docking, Parallel I/O, Darshan, I/O Profiling**

I. Introduction

High-throughput molecular docking, which is a part of *virtual screening*, is the usage of High-Performance Computing (HPC) systems to investigate and examine large data sets of chemical compounds to determine and discover potential drug candidates [1]. The basic building block of virtual screening is the molecular docking of a ligand (a small molecule with typically less than a hundred atoms) to a given protein location, called *pocket*. In a nutshell, molecular docking estimates and scores the ligand's three-dimensional pose and binding with a target protein pocket for a high number of ligands. In the context of drug discovery, the ligand with a high score from the molecular docking procedure is a potential drug candidate, while the protein pocket is the drug target. In virtual screening, million of ligands from chemical compound databases are evaluated against the same protein pocket. Virtual screening has been one of the primary tools to support discovering potential drugs against the Coronavirus disease (COVID-19) [2], [3].

From an HPC perspective, high-throughput molecular docking is essentially an embarrassingly parallel data-driven problem. Because we can process each ligand independently, high-throughput molecular docking is data-parallel and can be parallelized without requiring communication across processes or threads. For instance, each process or thread can compute and score a specific ligand in parallel. While the number of pockets that we want to evaluate is limited (in this study, we study one pocket), the chemical library of ligands can be arbitrarily large. Therefore, it is crucial to define how we read ligands in input and write the results. However, because of its data-driven nature, the high-throughput molecular docking performance depends on how fast we can ingest data (often from a very large shared file) into the processing pipeline and how efficiently we can write molecular docking results. The I/O performance not only depends on the particular I/O setup and implementation but also how the I/O stage interacts with the rest of the application pipeline. I/O can become a major performance bottleneck. While the acceleration of the molecular docking stage via accelerators has received a lot of attention by the HPC community, the I/O performance and its impact on the application's overall performance has not been the focus of studies. To address this lack of studies, we investigate the performance of parallel I/O in high-throughput molecular docking application, called *Docker-HT*.

The contributions of this paper are the following:

- We characterize the I/O parallel performance of a large-scale high-throughput molecular docking code, Docker-HT, up to 512 nodes. We use two configurations with different read and write buffer and queue sizes, number of parallel writers, and Lustre stripe counts. We show that a tuned I/O configuration is crucial for the performance
- By monitoring the overall high-throughput molecular docking performance, we evaluate the impact of I/O performance on the overall application performance and identify a parallel efficiency issue.

II. High-Throughput Molecular Docking and I/O with Docker-HT

In this work, we evaluate the I/O impact on performance high-throughput molecular docking using the Docker-HT code. The Docker-HT code development stems from the LiGen workflow [4], the LiGenDocker code [5] and tuning of functional parameters with the LiGen mini-app, GeoDock-MA [6]. In LiGen, each stage of the workflow is a one-purpose application (OPA), a single application that carries out a single task, e.g., docking a molecule or score a pose. The user is then responsible for composing the OPAs to define the workflow of interest for a given problem. By definition, each OPA reads the target ligands from the standard input and writes the result to the standard output. This allows using UNIX pipes to create workflows easily. Differently from LiGen, the Docker-HT code combines the OPAs in

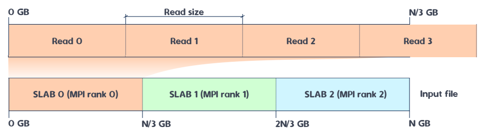

Fig. 1. Example on how three MPI processes, represented by a different color, split an input file of N GB and on how they perform the read operations.

one monolithic application implementing the entire workflow. In addition, Docker-HT has been specifically designed for HPC systems, including supercomputers with heterogeneous hardware. Docker-HT is written in C++17 and uses pthreads for on-node parallelism, CUDA for Nvidia GPU nodes, and MPI for inter-node communication. Within each node, we use pipeline parallelism and work-stealing to process the ligands.

This paper focuses on how Docker-HT reads ligands from the input file and how it stores the results in the output file. In the current version, we read ligands encoded using a custom binary format inspired to the Tripos `Mol2` format. The output file is a `CSV`-like file where we store the ligand's SMILE representation, which encodes the molecule in a string and its score against the target pocket. If we want to evaluate a ligand using multiple pockets, we need to run Docker-HT on each pocket. How to combine the results to select the most promising ligands is out of the scope of this paper.

A. *Read operations*

To carry out the dock and score operation, we need to have information about the ligand and the target pocket. Since Docker-HT considers the pocket constant, it reads the related files once at the beginning of the execution using an MPI IO collective read [7]. Therefore, the cost of this operation is greatly amortized over the lifespan of the application. On the other hand, the processes that contribute to the elaboration need to agree on how they process the ligands contained in the input file. Figure 1 shows an example where Docker-HT reads from an input file of N GB with three MPI processes. The idea is to divide the input file into a number of slabs equal to the number of MPI processes based on the file size. In the example, each process will elaborate $1/3$ of the file. Starting from the beginning of the slab, each process reads chunks of data with a granularity that the user can configure, i.e., the *Read Buffer Size*. One problem that we need to consider is that the size of a ligand description is not fixed but depends on the molecule's number of atoms and bonds. Therefore, it seldom happens that a ligand description starts with the beginning of a slab and it stops at the ending of a slab. We use the convention that each process elaborates the ligand whose description starts after its slab begins, while it stops reading the file when it finds the start of a ligand past the slab end. Thus, it is possible that a process reads more content after the slab end for the following two reasons: the *Read Buffer Size* is not a divisor of the slab size; and because we need to complete the description of the last ligand that we need to process. We implemented the reader stage using MPI I/O non-collective read operations. The frequency of the read operations depends on the throughput of the slowest stage of the computation pipeline, which is the stage that docks and scores a ligand.

B. *Write operations*

For each ligand that we read from the file, we need to store a line in the `CSV`-like output file that relates the molecule with its score. The order in which we store the lines does not matter. Therefore, we can efficiently aggregate data before issuing the write operation on the file system. When implementing this strategy with MPI I/O and support MPI_THREAD_MULTIPLE, we incurred into technical problems with MPI I/O implementations [7]. For this reason, we implement the write operation using a two-step approach inspired by the MPI I/O collective write. We expose to the end-user two parameters: the number of MPI processes that issue the write operation (*Parallel Writers*), and the maximum amount of data that they write (*Write Buffer Size*).

Figure 2 shows an example where six MPI processes write to the output file. In this example, we also assume that the user set the number of *Parallel Writers* to two. In the writer initialization, we assign each MPI process to a writing group based on the number of MPI processes and Parallel Writers. Suppose the number of MPI processes is not a multiple of the number of Parallel Writers. In that case, we enlarge the size of the first groups to include the spare ones to minimize the difference between the groups' sizes. We use the convention that the writer is the MPI process with the lowest rank in its group.

While the application pipeline is running, the writer stage of each MPI process appends the output of each ligand in an accumulation buffer. The maximum size of this buffer is equal to the *Write Buffer Size* divided by the number of MPI processes in the group. When appending the current line of the `CSV` would increase the size of the accumulation buffer over its maximum size, or there is no more ligand to elaborate, the writer stage initiates the write operation.

The first phase aims at aggregating all the accumulation buffers of a group to the writing buffer of the writer. It is crucial to note that the actual size of each accumulation buffer is different because it depends on the computed ligands. Therefore, before issuing the gather operation, there is a synchronization between the MPI processes of a group to agree on the writing buffer size and displacement. In this phase, each group operates in parallel. In the second phase, all the writers exchange information about their writer buffers and agree on the offset for all the write operations. Eventually, each writer will issue the actual I/O operation in parallel.

The last issue that the writer stage must consider is the termination problem: the amount of output that an MPI process generates depends on properties of the ligands that it elaborates, which are impossible to foresee. This means that each writer can issue a different number of write operations. For this reason, after each write operation, the writer will broadcast the maximum amount of data written by all the writers to the MPI processes of its group. We use the convention that if the latter

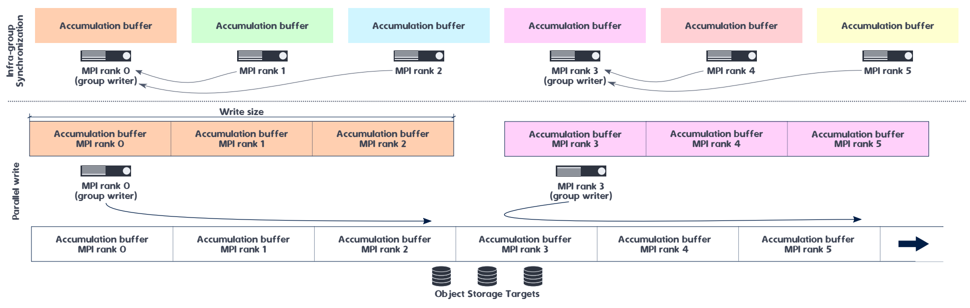
Intra-group Synchronization
Accumulation buffer
MPI rank 0 (group writer)
MPI rank 1
MPI rank 2
MPI rank 3 (group writer)
MPI rank 4
MPI rank 5
Write size
Parallel write
Accumulation buffer MPI rank 0
Accumulation buffer MPI rank 1
Accumulation buffer MPI rank 2
Accumulation buffer MPI rank 3
Accumulation buffer MPI rank 4
Accumulation buffer MPI rank 5
Object Storage Targets

Fig. 2. Overview of the two-phases writing procedure, using an example with six MPI processes, represented by a different color, that write the output file. We omit from the picture all the synchronizations required to gather the data and to perform the parallel write.

is equal to zero (no data were actually written), the global computation is completed. Otherwise, the writing stage of the MPI process needs to re-start another write operation even if it has no more ligand to process and its accumulation buffer is empty. This implementation yields significant benefits. On the main hand, it does not require a single operation that involves all the MPI processes but only a subset of them. On the other hand, it exposes two intuitive parameters that can significantly change the access pattern of the application. Moreover, the semantic is similar to the Stripe Count and Size of the Lustre parallel file system.

III. Experimental Setup

We evaluate the Docker-HT I/O performance on the Beskow supercomputer at KTH Royal Institute of Technology. Beskow is a Cray XC40 system, with 2,060 compute nodes, equipped with two Xeon E5-2698v3 Haswell 2.3 GHz CPUs (16 cores per CPU) per node and high-speed network Cray Aries. The storage employs a Lustre parallel file system (client v2.5.2) with 165 OST servers. The OS is SUSE LINUX (Release 11). Docker-HT is built with the Cray Compiler Environment 10.0.1, Cray MPICH 7.7.14, and the Boost library version 1.50. Docker-HT uses hybrid parallelism with pthreads and CUDA on node and MPI for inter-node parallelism. We use a pool of 32 pthreads per node (as the number of cores per node on Beskow) and a number of MPI processes equal to the number of nodes. We perform simulation up to 512 nodes: in this configuration, we use 512 MPI processes and a total of 16,384 pthreads.

As ligand data set, we use the Covid-19 MolEcular DockIng AT homE (MEDIATE) [1] that includes 3.4 million ligands. The original data set is 10.3 GB large in the `.sdf` format. We use `OpenBabel` [8] and a LiGen pre-processing tool to convert the original `.sdf` file first to the `.mol` and then to a binary format for Docker-HT. The final binary file, including all the ligand information, is 2 GB large. We use the execution time, bandwidth for I/O operations, and the per-node number of ligands docked per second (ligand/s) as main performance figures of merit.

[1]https://mediate.exscalate4cov.eu/downloads/Commercial_MWlower330T.zip

	Default	Advanced
Read Buffer Size	20 MiB	1MiB
Lustre Stripe Count - Read	1	4
Write Buffer Size	20 MiB	1 MiB
Parallel Writers	1	14
Write Queue Size	1	20000
Lustre Stripe Count - Write	1	14

TABLE I
Default and advanced configurations for parallel I/O.

We use the Darshan profiler (version 3.3.1) to measure the I/O performance. Darshan is a low-overhead tool to investigate the I/O performance of parallel applications [9]. We extract the bandwidth and amount of moved data from Darshan logs for each file (both read and written). In addition to the Darshan tool, we post-process Docker-HT log files, retrieving information about the moving average of the per-node performance when the read and write operations are performed.

In our tests, we evaluate the impact of scaling the number of nodes on the I/O nodes. We use two Docker-HT configurations with different read and write buffer sizes, Lustre stripe counts, and number of parallel writers. We summarize the values for two different configurations in Table I. We call default configuration the standard Docker-HT I/O set-up without setting any flag. We set the advanced configuration using Docker-HT flags to target simulations on large-scale systems, such as a supercomputer and parallel I/O. In particular, the advanced configuration uses small buffer sizes (1 MiB), several parallel writers, a large queue size, and Lustre strip count equal to the number of parallel writers.

IV. Results

We first evaluate the Docker-HT performance varying the number of computing nodes, from four to 512, using the default and advanced configurations. Figure 3 shows the total Docker-HT execution time, the relative (to four-node performance) parallel speed-up, and efficiency. The Docker-HT execution time on four nodes is for the default and advanced configuration time is 3,882 and 4,071 seconds, respectively. The default configuration exhibits better performance than the advanced configuration for a small number of nodes. On 512 nodes, the advanced configuration outperforms the default

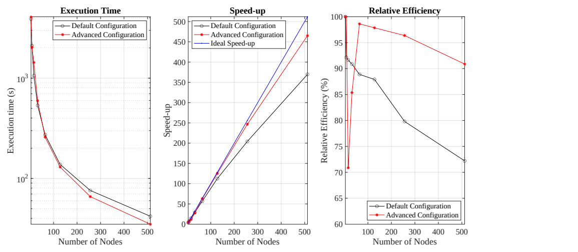

Fig. 3. Docker-HT execution time, relative parallel speed-up and efficiency varying the number of nodes.

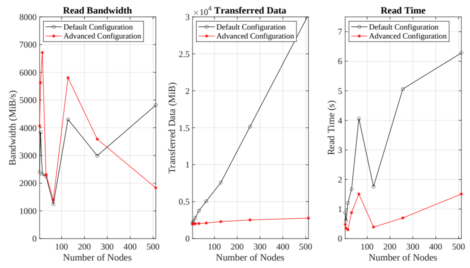

Fig. 4. Docker-HT read bandwidth, transferred data and execution time for the reading of the ligand file.

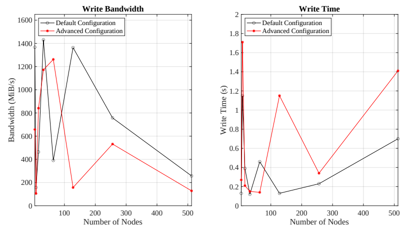

Fig. 5. Docker-HT write bandwidth and time.

configuration with an execution time of 35 seconds vs. 43 seconds.

The relative parallel efficiency on 512 nodes is 72% and 90% for the default and advanced configuration, respectively. We find that the default I/O configuration performs better than the advanced configuration when using less than 64 nodes. In particular, the advanced configuration performs poorly when running on 16 and 32 nodes with a parallel efficiency of 71% and 85%. On the contrary, the default configuration exhibits a parallel efficiency of 91% and 90% on 16 and 32).

To understand the difference of Docker-HT performance given in two I/O configurations, we investigate first the I/O performance by instrumenting Docker-HT and extracting bandwidth, read/written data size, and the execution time spent in I/O. We first focus on the reading operation of the ligand file occurring at the beginning of the high-throughput molecular docking operation. We show the performance for the reading operation in Figure 4. The maximum read bandwidth we recorded is 6.7 GiB/s on 16 nodes in the advanced configuration. The minimum read bandwidth occurs when running on 64 nodes: 1.2 GiB/s and 1.4 GiB/s for the default and advanced configuration. As a reference, the highest read bandwidth, obtained with the IOR benchmark [10] on four nodes of the Beskow system reading from a shared file, is 4.1 GiB/s. The most interesting plot in Figure 4 is the amount of the read data: in both configurations, the size of the read data increases linearly with the number of nodes with the increase rate determined by the read buffer size. With a large read buffer size of 20 MiB (as in the default configuration), the amount of the read data on 512 nodes is 30 GBi (roughly 15 times the size of the input ligand data set). With a smaller read buffer size of one MiB (as in the advanced configuration), the read data on 512 nodes is 2.7 GBi (roughly 25% more than the size of the input ligand data set). The total time for Docker-HT reading operations is largely due to the size of the read data. A configuration with a read buffer size of one MiB is always faster than a configuration with 20 MiB (default configuration) in all our experiments (see the rightmost panel in Figure 4).

We note that poor scalability of Docker-HT in the default configuration is in part due to the read operation and, in particular, to the oversized read buffer of 20 MiB: on 512 nodes, the read operation takes 6.3 seconds, approximately 15% of the total execution time. In the advanced configuration, the reading time accounts for 4% of the whole execution time.

To further understand the impact of I/O on high-throughput molecular docking, we investigate the I/O performance of parallel writing of the ligand score to a shared file. In all our experiments, the size of written data is constant, as the number of ligands is constant and equal to 182 MiB. Figure 4 shows the write bandwidth and time for the default and advanced configuration. The maximum write bandwidth is 1.4 and 1.3 GBi/s achieved on 32 and 128 nodes in the default configuration with one writer and a queue size of one. The minimum bandwidth is 128 MBi/s on 512 nodes with the advanced configuration with 14 parallel I/O writers. For reference, the maximum write bandwidth obtained with IOR [10] running on four nodes of the Beskow system is 1.6 GBi/s. For more than 64 nodes, the default configuration with one parallel writer leads to higher bandwidth and faster execution time than simulation with advanced configuration. On 512 nodes, the parallel write operations take 0.7 and 1.41 seconds in the default and advanced configuration, respectively. Interestingly, the default configuration exhibits higher parallel write performance than the advanced configuration in terms of bandwidth and execution time. These results are expected as the default configuration has a larger write buffer size, leading to an increased usage of the available bandwidth and reduced write operations. However, these results do not explain the increased scalability of the advanced I/O configuration and the poor parallel efficiency on 16 and 32 nodes with the advanced configuration (see Figure 3).

To gain more insight into the impact of I/O on the whole high-throughput molecular docking performance, we investigate the Docker-HT performance variation in time and correlate with the times the parallel write operations occur.

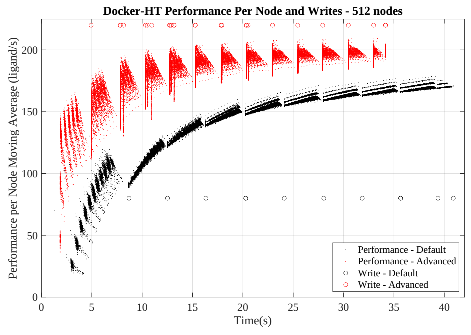

Fig. 6. Moving average of per-node performance on the 512 nodes in the default (black dots) and advanced (red dots) configuration together with the times writing operations occur (black and red circles).

Figure 6 shows the moving average of per-node performance evolution in time for the default (black dots) and advanced (red dots) configurations. Different dots at a given time represent the performance on different nodes: the dot spread represents a performance variation across the nodes. To help correlate Docker-HT performance with the write operation, we superimpose black and red circles to denote the time the write operations occur in the simulation. On 512 nodes, the average per-node performance of the advanced configuration is approximately 13% higher than the performance with the default configuration. This result is due to the pipelining of molecular docking and I/O operations and parallel writers (the advanced configuration uses a queue length of 20,000 and 14 parallel writers). We note that the performance variation across nodes is larger in the advanced configuration. A performance drop occurs when a write operation is performed (the circles are vertically aligned with the performance drops). Writes are synchronizing operations and lead to a performance drop. The performance drop is more drastic in the case of advanced configuration, up to a maximum of 20%.

To investigate the poor parallel efficiency and identify the problem with the advanced configuration on 16 and 32 nodes, we study the per-node performance moving average for this configuration on 16 nodes, as presented in Figure 7. In this picture, the red dots and circles represent the performance of the default configuration. It is clear from inspecting the plot that the per-node performance drops by 50% when write operation occurs. However, the main reason for the poor parallel efficiency is the final phase of the high-throughput molecular docking starting after approximately 1,000 seconds. This poor behavior is due to the unbalance between the writer groups' size. In particular, with 16 nodes and 14 writers, we have 12 writing groups composed of one node and two groups consisting of two nodes. Since the accumulation buffer on each node is equal to the *Write Buffer Size* divided by the group size, we have four nodes with an accumulation buffer that is half of the others, but the node's throughput is similar. Thus, the two groups with two nodes will fill the accumulation buffer and initiate the write operation earlier than the other groups. Once the writing stage queue is complete, the whole node will be idle until the other groups join the writing process since it involves synchronization calls between the writers.

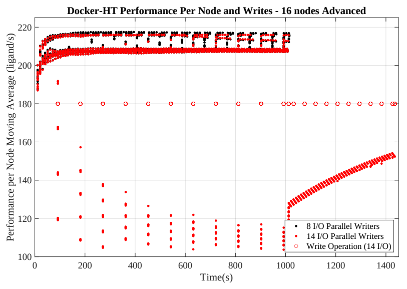

Fig. 7. Moving average of per-node performance on 16 nodes with 14 (red) and 8 (black) parallel writers and write times for 14 parallel writers.

V. Related Work

Several approaches and tools have been designed for performing molecular docking on HPC systems [11]–[13]. AutoDock4 [14] and AutoDock Vina [14] are among the most used tools for molecular docking. These tools use OpenMP for multithreaded molecular docking on a single node. Autodock has also port to use accelerator with OpenCL [15] , Nvidia CUDA [3] and OpenACC [16], [17]. VinaMPI is an MPI wrapper of Autodock Vina to use MPI on extreme scale supercomputers [18]. VinaLC [19] is another Autodock Vina extension to use MPI and large supercomputers. In this work, we use the Docker-HT code and focus on I/O performance and impact of high-throughput molecular-docking.

VI. Discussion and Conclusion

This paper investigated the I/O performance and impact on an HPC high-throughput molecular-docking code, called Docker-HT. High-throughput molecular-docking is a data-intensive application stressing the I/O system for reading many ligands and writing the score information for each ligand to a shared file. We showed that parallel read performance considerably depends on the read buffer size. On 512 nodes, a reduced read buffer size of 1 MBi led to a significant improvement in reading execution time. When investigating the write performance, we noted that the default I/O configuration (one parallel writer and queue size equal to one) exhibits better I/O performance in terms of bandwidth and time spent in writing than the advanced configuration. To assess the overall impact of I/O, we traced the high-throughput molecular docking performance. We found that while the usage of large queue size and several parallel writers showed worse I/O performance, it led instead to a better overall application performance as it provided increased parallelism. We also identified a parallel efficiency issue by tracing the application performance when selecting the number of parallel writers for the computation on 16 and 32 nodes. We plan to extend this study to large input data sets and systems with accelerators as future work.

Acknowledgments. We acknowledge funding from the European Commission H2020 program, Grant Agreement No. 956137 (LIGATE). All computation needed for this paper was enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC).

References

[1] W. P. Walters, M. T. Stahl, and M. A. Murcko, "Virtual screening- an overview," *Drug discovery today*, vol. 3, no. 4, pp. 160–178, 1998.

[2] A. Acharya, R. Agarwal, M. B. Baker, J. Baudry, D. Bhowmik, S. Boehm, K. G. Byler, S. Chen, L. Coates, C. J. Cooper *et al.*, "Supercomputer-based ensemble docking drug discovery pipeline with application to covid-19," *Journal of chemical information and modeling*, vol. 60, no. 12, pp. 5832–5852, 2020.

[3] S. LeGrand, A. Scheinberg, A. F. Tillack, M. Thavappiragasam, J. V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez *et al.*, "Gpu-accelerated drug discovery with docking on the summit supercomputer: porting, optimization, and application to covid-19 research," in *Proceedings of the 11th ACM international conference on bioinformatics, computational biology and health informatics*, 2020, pp. 1–10.

[4] A. R. Beccari, C. Cavazzoni, C. Beato, and G. Costantino, "Ligen: a high performance workflow for chemistry driven de novo design," 2013.

[5] C. Beato, A. R. Beccari, C. Cavazzoni, S. Lorenzi, and G. Costantino, "Use of experimental design to optimize docking performance: The case of ligendock, the docking module of ligen, a new de novo design program," 2013.

[6] D. Gadioli, G. Palermo, S. Cherubin, E. Vitali, G. Agosta, C. Manelfi, A. R. Beccari, C. Cavazzoni, N. Sanna, and C. Silvano, "Tunable approximations to control time-to-solution in an HPC molecular docking mini-app," *J. Supercomput.*, vol. 77, no. 1, pp. 841–869, 2021.

[7] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[8] N. M. O'Boyle, M. Banck, C. A. James, C. Morley, T. Vandermeersch, and G. R. Hutchison, "Open babel: An open chemical toolbox," *Journal of cheminformatics*, vol. 3, no. 1, pp. 1–14, 2011.

[9] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[10] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008, pp. 1–12.

[11] J. Biesiada, A. Porollo, P. Velayutham, M. Kouril, and J. Meller, "Survey of public domain software for docking simulations and virtual screening," *Human Genomics*, vol. 5, no. 5, pp. 497–505, 2011.

[12] E. Yuriev, J. Holien, and P. A. Ramsland, "Improvements, trends, and new ideas in molecular docking: 2012-2013 in review," *Journal of Molecular Recognition*, vol. 28, no. 10, pp. 581–604, 2015.

[13] N. S. Pagadala, K. Syed, and J. Tuszynski, "Software for molecular docking: a review," *Biophysical Reviews*, vol. 9, no. 2, pp. 91–102, 2017.

[14] O. Trott and A. J. Olson, "Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *Journal of computational chemistry*, vol. 31, no. 2, pp. 455–461, 2010.

[15] D. Santos-Martins, L. Solis-Vasquez, A. F. Tillack, M. F. Sanner, A. Koch, and S. Forli, "Accelerating autodock4 with gpus and gradient-based local search," *Journal of Chemical Theory and Computation*, vol. 17, no. 2, pp. 1060–1073, 2021.

[16] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, and C. Silvano, "Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes," *The Journal of Supercomputing*, vol. 75, no. 7, pp. 3374–3396, 2019.

[17] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, and C. Silvano, "Accelerating a geometric approach to molecular docking with openacc," in *Proceedings of the 6th International Workshop on Parallelism in Bioinformatics*, 2018, pp. 45–51.

[18] S. R. Ellingson, J. C. Smith, and J. Baudry, "Vinampi: Facilitating multiple receptor high-throughput virtual docking on high-performance computers," 2013.

[19] X. Zhang, S. E. Wong, and F. C. Lightstone, "Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines," *Journal of computational chemistry*, vol. 34, no. 11, pp. 915–927, 2013.