Enhancing large-scale docking simulation on heterogeneous systems: an MPI vs rCUDA study

Baldomero Imbernón[a], Javier Prades[c], Domingo Giménez[b], José M. Cecilia[a], Federico Silla[c]

[a]*Polytechnic School, Catholic University of San Antonio of Murcia (UCAM), 30107, Murcia, Spain*
[b]*Department of Computing and Systems, University of Murcia, 30071, Murcia, Spain*
[c]*Departament d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, 46002 Valencia, Spain*

Abstract

Virtual Screening (VS) methods can considerably aid clinical research by predicting how ligands interact with pharmacological targets, thus accelerating the slow and critical process of finding new drugs. VS methods screen large databases of chemical compounds to find a candidate that interacts with a given target. The computational requirements of VS models, along with the size of the databases, containing up to millions of biological macromolecular structures, means computer clusters are a must. However, programming current clusters of computers is no easy task, as they have become heterogeneous and distributed systems where various programming models need to be used together to fully leverage their resources. This paper evaluates several strategies to provide peak performance to a GPU-based molecular docking application called *METADOCK* in heterogeneous clusters of computers based on CPU and NVIDIA Graphics Processing Units (GPUs). Our developments start with an OpenMP, MPI and CUDA *METADOCK* version as a baseline case of cluster utilization. Next, we explore the virtualized GPUs provided by the *rCUDA* framework in order to facilitate the programming process. rCUDA allows us to use remote GPUs, i.e. installed in other nodes of the cluster, as if they were installed in the local node, so enabling

Email addresses: `bimbernon@ucam.edu` (Baldomero Imbernón), `japraga@gap.upv.es` (Javier Prades), `domingo@um.es` (Domingo Giménez), `jmcecilia@ucam.edu` (José M. Cecilia), `fsilla@disca.upv.es` (Federico Silla)

access to them using only OpenMP and CUDA. Finally, several load balancing strategies are analyzed in a search to enhance performance. Our results reveal that the use of middleware like rCUDA is a convincing alternative to leveraging heterogeneous clusters, as it offers even better performance than traditional approaches and also makes it easier to program these emerging clusters.

Keywords:
Virtual Screening, HPC, rCUDA, Metaheuristics, Heterogeneous Computing

1. Introduction

Drug discovery and development may take more than a decade from discovery of a candidate drug to patient treatment [1]. There are several stages that a candidate drug must successfully go through. Among them, we would highlight the basic research of drug discovery, pre-clinical stages, clinical trials, and final review by associations like FDA (Food and Drug Administration) in the USA. The use of Virtual Screening (VS) methods can tremendously improve the drug discovery process, saving time, money and computational resources [2].

VS methods are computational techniques that analyze large libraries of small molecules (a.k.a. *ligands*) to search for structures most likely to bind to a target drug, typically a protein receptor or enzyme [3]. These libraries of chemical compounds may contain up to millions of ligands [4], given that analyzing larger databases exponentially increases the chances of generating hits. However, current VS methods, such as docking [5], fail to make good toxicity and activity predictions, since they are constrained by their access to computational resources; indeed, the fastest VS methods cannot process large biological databases in reasonable times.

The use of high performance computing in order to enhance virtual screening methods is therefore necessary to fulfill pharmaceutical industry expectations, and a lot of research is been carried out in this regard. Methods like Autodock [6], Autodock VINA [7], Glide [8], LeadFinder [9], SurFlex [10], ICM+ [11], FMD [12] or DOCK [13] use multithreading programming at the node level in order to leverage multicore architectures, and some of them even distribute their computations among the CPUs of several nodes by means of the Message Passing Interface (MPI) library. However, we are currently witnessing a steady transition to heterogeneous computing systems [14], with heterogeneity representing systems where nodes combine traditional multicore architectures (CPUs) with accelerators such as Graphics Processing Units (GPUs). Programs such as BUDE [15], AMBER [16] or BINDSURF [17] use GPUs to overcome this problem by dividing the whole protein surface into independent regions (or spots). However, heterogeneity may limit system growth as it can no longer be addressed in an incremental way. Indeed, several computational challenges come up with such heterogeous systems [18], like scalability, programmability or data management, to mention just a few.

In addition to the use of heterogeneous systems, virtualization techniques may provide significant improvements, as they enable a larger resource utilization by sharing a given hardware among several users, thus reducing the required amount of instances of that particular device. As a result, virtualization is being increasingly adopted in data centers. Some of the most extended virtualization techniques are based on software solutions, such as the VMware [19] (by VMware Inc.) or Xen [20] hypervisors. These solutions virtualize the entire system, providing a whole virtual computer to each user. However, although using virtual machines is appealing in many cases, even for high performance computing, when the goal is to make use of GPUs, these solutions introduce an unacceptable overhead due to the strong limitations they present with respect to the shared use of accelerators. In this regard, current virtual machine approaches are unable to concurrently share a GPU among several virtual machine instances.[1] Therefore, instead of virtualizing the entire computer, an altenative approach would be to virtualize specific resources, such as the GPU.

rCUDA [21] is a framework that enables remote concurrent use of CUDA-compatible GPUs. To enable remote GPU-based acceleration, this framework creates virtual CUDA-compatible devices on machines without local GPUs. These virtual devices represent physical GPUs located in a remote host offering GPGPU (General-Purpose Computing on Graphics Processing Units) services. Thus, all nodes in a cluster are able to access the whole set of CUDA accelerators concurrently. Moreover, a single-node shared-memory

[1]Notice that the GRID GPU by NVIDIA is designed to be shared among VMs. However, the shared usage of this GPU is limited to desktop virtualization. When GRID GPUs are used as CUDA accelerators they cannot be shared among VMs.

application could access all the GPUs in the cluster without using the MPI library, which potentially reduces the programming complexity. Additionally, given that real GPUs are concurrently shared among several applications, energy would be saved at the same time that a lower hardware investment is required. Furthermore, this approach would still deliver an acceptable performance, as shown in [22].

In this paper, we analyze the current computational landscape by applying heterogeneous clusters based on NVIDIA GPUs and CPUs to a challenging problem such as molecular *docking* computational methodology, called $METADOCK$ [23], where the interaction between two molecules (a macromolecule known as receptor and a small molecule referred to as ligand) is simulated by minimizing a scoring function (affinity between the two molecules) that models the chemical process behind molecular interaction. The METADOCK methodology has two main characteristics: (1) the user can configure the *optimization procedure* at compile time from among a wide set of metaheuristics (i.e, algorithms like Genetic Algorithm, Scatter Search or local search methods), and (2) the calculation of the computationally expensive *scoring function* is offloaded to GPUs. With this in mind, major contributions of this paper include the following:

1. We develop a new version of $METADOCK$ to perform large-scale simulations on heterogeneous computer clusters based on CPUs and NVIDIA GPUs. The implementation is developed using a traditional approach based on MPI, OpenMP and CUDA.
2. We evaluate $rCUDA$ as a framework to leverage virtualized GPUs and also to facilitate the programming. This implementation only requires us to deal with OpenMP and CUDA APIs.
3. Several load balancing strategies are also evaluated in both configurations (virtualized and non-virtualized GPUs) to pursue the performance into unprecedented levels.
4. Finally, we check whether the search for performance is translated into an actual benefit in the quality of the results (reductions in execution time do not necessarily mean a better affinity quality). In this paper, the search procedure of $METADOCK$ is configured to use three different metaheuristics (genetic algorithm, scatter search and local search) in order to analyze the evolution of the fitness along with the performance improvements.

The rest of the paper is structured as follows. Section 2 includes the background about Virtual Screening methods and the scoring problem we are working on and describes some relevant knowledge about HPC architectures. Next, our metaheuristic-based virtual screening technique is introduced in Section 3. The parallel strategies used for the efficient application of these techniques in heterogeneous clusters are explained in Section 4, whereas the experimental results are presented and analyzed in Section 5. Related work are reviewed in Section 6. The final section summarizes the conclusions and gives some directions for future work.

2. Background

This section introduces the main characteristics of Virtual Screening methods and heterogeneous clusters to better understand the rest of the paper.

2.1. Virtual Screening methods

We draw on our description of Virtual Screening (VS), which was first given in [17, 23]. VS methods are computational techniques used in several scientific areas, such as catalysts and energy materials [24], and mainly drug discovery [5], where experimental techniques can benefit from computational simulation.

VS methods search for libraries of small molecules that can potentially bind to a drug target, typically a protein receptor or enzyme, with high affinity. They actually "dock" small molecules into the structures of macromolecular targets. Moreover, they look for (i.e., score) the optimal binding sites by providing a ranking of chemical compounds according to the estimated affinity or *scoring* [25]. In general, VS methods optimize *scoring functions*, which are mathematical models used to predict the strength of the non-covalent interaction between two molecules after docking [26]. Indeed, these candidate molecules will continue the drug discovery process road-map that goes from in-vitro studies to animal investigations and, eventually, to human trials [27].

Although VS methods have been used for many years and have identified several compounds to be used in drugs, VS has not yet fulfilled all its expectations. Neither the VS methods nor the scoring functions used are sufficiently accurate to identify high-affinity ligands reliably. To deal with large numbers of potential candidates (many databases comprise hundreds of thousands of ligands), VS methods must be very fast and still they would require hundreds of CPU hours for each ligand, and, according to [28], even thousands of CPU hours for each ligand when simulation strategies are used to compute absolute binding affinities.

2.2. Metaheuristics

A wide range of optimization problems, like VS methods, are NP-hard and cannot afford to compute all possible solutions. In such scenarios, metaheuristics provide an abstraction layer for good enough solutions which are found in a reduced search space focused just on promising areas [29]. Metaheuristics can be specially useful in VS methods.

Metaheuristics of interest to us fall into two prominent classes:

- *Distributed metaheuristics.* These metaheuristics search within the entire solution space, and work with populations that are combined to improve solutions progressively. Examples of this group include Ant Colony, Particle Swarm Optimization, Genetic Algorithms and Scatter Search.

- *Neighborhood metaheuristics.* These metaheuristics work with an element in the solution space and search for better elements in its neighborhood. Examples include Guided Local Search, Hill Climbing, Simulated Annealing, Tabu Search, Variable Neighborhood and GRASP (Greedy Randomized Adaptive Search Procedures). GRASP is a metaheuristic close to one of the parameter configurations of the metaheuristic later used in the experiments in this paper.

All the previous metaheuristics can be combined among them in order to improve the quality of the methodology, although in this paper they will not be combined. We will focus on separately using metaheuristics similar to Genetic Algorithms, Scatter Search and GRASP.

Diversity in metaheuristics [30] is worth investigating. We can first define a subset of alternatives, and then apply a tuning process which is fuzzy, or even blind, for the effects of certain values in the experimental praxis. This paper sheds some light on scenarios guided by computational criteria: minimize execution time and fitness using parameters which give similar results. Even so, the procedure may be different for each application area and, hence, our experimental analysis (Section 5) focuses more on quantifying potential gains. An artificial intelligent system or human expert can then use our findings to complete the selection process with clear benefits.

Algorithm 1 A parameterized metaheuristic schema to generate several types of Metaheuristics

```
Initialize(S,ParamIni)
while no End condition(S) do
    Select(S,Ssel,ParamSel)
    Combine(Ssel,Scom,ParamCom)
    Improve(Scom,ParamImp)
    Include(Scom,S,ParamInc)
end while
```

Many metaheuristics follow similar patterns, particularly those based on populations share six basic functions (see Algorithm 1): *Initialize*, *End condition*, *Select*, *Combine*, *Improve* and *Include* [31, 32]. These functions work with several populations: *S*, *Ssel* and *Scom*. *S* represents the set of candidate solutions, where some selected solutions (*Ssel*) are combined to generate a new set of elements, *Scom*.

Within the above template, programmers can provide different instances and/or implementations, and the final set of candidate solutions (S) uses population insights to guide the search. Local search metaheuristics are also within the schema ($|S| = 1$).

Unified metaheuristic schemes can be enriched by introducing parameters for each function [33, 34, 35] (see Param- prefixes in Algorithm 1), and hybrid metaheuristic schemes can be considered too, with computational complexity increasing continuously. In the benchmarks throughout this work, we use the parameterized schema with two configurations of the parameters which give metaheuristics close to Genetic Algorithms and Scatter Search.

2.3. Programming heterogeneous clusters

Since the early days, computer architects have relied on technology scaling to provide increased performance. Heterogeneous architecture design is now seen as the only solution to continue Moore's law scaling through innovation alone [14, 36], with systems where nodes combine traditional multicore architectures (CPUs) and accelerators (mostly GPUs) [37].

Traditional parallel implementations are not always efficient when ported to heterogeneous systems. They are often inherited from scalable supercomputers, where all nodes in the cluster have the same compute capabilities, and they therefore lack the ability to distinguish computational devices with asymmetric computational power and energy consumption. Differences are not limited to fundamental hardware design (CPUs vs. GPUs), but also occur within the same family of processors. Therefore, programmers play a fundamental role in this heterogeneous context as they have to deal with different programming models such as OpenMP [38], MPI [39] and OpenCL [40] or CUDA [37] to fully leverage all computing resources in current clusters. In this paper we address different programming models and techniques to accommodate our VS application to an increasingly heterogenous underlying hardware.

2.3.1. The rCUDA middleware

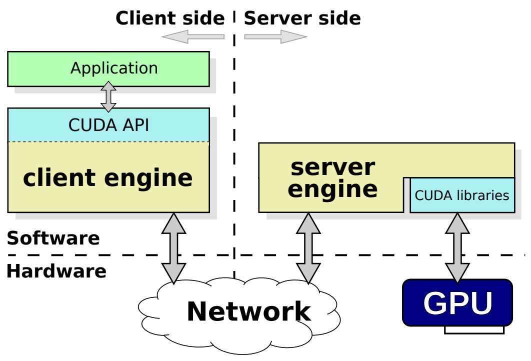

Figure 1: Architecture of the rCUDA framework

Figure 1 depicts the architecture of the rCUDA framework, which follows a client-server distributed approach. The client part of rCUDA is installed in the cluster node executing the application requesting GPU services, whereas the server side runs in the computer owning the actual GPU. The client side of the middleware offers the same application programming interface (API) as does the NVIDIA CUDA API. In this manner, the client receives a CUDA request from the accelerated application and appropriately processes and forwards it to the remote server. In the server node, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and provides the execution results to the server middleware. In turn, the server sends back the results to the client middleware, which forwards them to the initial application, which is not aware that its request has been served by a remote GPU instead of a local one.

rCUDA is binary compatible with CUDA 8.0 and implements the entire CUDA Runtime and Driver APIs (except for graphics functions). It also provides support for the libraries included within CUDA (cuBLAS, cuFFT, etc). Additionally, it supports several underlying interconnection technologies by making use of a set of runtime-loadable, network-specific communication modules (currently TCP/IP, RoCE and InfiniBand). Independently of the exact network used, data exchange between rCUDA clients and servers is pipelined in order to attain high performance. Internal pipeline buffers within rCUDA use preallocated pinned memory, given the higher throughput of this type of memory [41].

The InfiniBand communication module is based on the IB Verbs (IBV) API. This API offers two communication mechanisms: the channel semantics and the memory semantics. The former refers to the standard send/receive operations typically available in any networking library, while the latter offers RDMA operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy transfers with minimum involvement of the CPUs. rCUDA employs both IBV mechanisms, selecting one or the other depending on the exact communication to be carried out.

3. Population-based metaheuristics for Virtual Screening: METADOCK

METADOCK [23] is a virtual screening (VS) application that simulates the interaction between two molecules. Indeed, it attempts to predict non-covalent binding of molecules or, more frequently, of a macromolecule (receptor) and a small molecule (ligand). This prediction is computationally performed through an iterative procedure that tries to minimize a scoring function that models such molecular interaction. Therefore *METADOCK*, as a VS application, has two main ingredients. First, the *optimization algorithm* is based on a metaheuristic schema (see Algorithm 1) to be able to select a metaheuristic that will guide the search procedure. Second, the scoring function calculation is offloaded to GPUs in order to speedup execution time.

With this in mind, we now provide some details about *METADOCK* and we refer the reader to [23] for further insights. *METADOCK* divides the whole receptor surface into arbitrary and independent regions (or spots) where the optimization process is performed independently. This enables the so-called *blind docking* that offers the opportunity to find novel binding sites, but drastically increasing the computational cost. The optimization at each spot consists of looking for the best ligand conformation that interacts with the lowest fitness value (it is a minimization problem). The fitness is given by the scoring function that mathematically represents the chemical interaction between the protein receptor and ligand conformation. In our case, the scoring function is based on the relevant non-bonded potentials used in VS calculations, which are the Coulomb, or electrostatic term, and the Lennard-Jones potentials, since they describe very accurately the most important short and long-range interactions between atoms of the protein-ligand system [42]. The calculation of the scoring function requires the highest percentage of the overall execution time, and it is offloaded to the GPU to be accelerated.

The simulation starts by minimizing the value of the scoring function by continuously making random or predefined perturbations of the initial population (S) at each spot. Particularly, each candidate solution is a conformation ligand that differs in the application of some movement (translating and/or rotating) with respect to a given region. Then, the new value of the scoring function for each candidate solution is obtained, being eventually accepted according to metaheuristic criteria.

Table 1: The seventeen metaheuristic parameters used in the unified parameterized metaheuristic schema for $METADOCK$.

Metaheuristic Parameters	Description
$INEIni$	Number of initial ligand conformations.
$PEIIni$	Percentage of the best conformations that are improved in the function Initialize.
$IIEIni$	The intensification of the improvement in the function Initialize.
$PBEIni$	Percentage of best conformations to be included in the initial set for the next iterations.
$PWEIni$	Percentage of worst conformations to be included in the initial set for the next iterations.
$PBESel$	Percentage of the best conformations to be selected for combination.
$PWESel$	Percentage of the worst conformations to be selected for combination.
$PBBCom$	Percentage of best-best conformations to be combined.
$PWWCom$	Percentage of worst-worst conformations to be combined.
$PBWCom$	Percentage of best-worst conformations to be combined.
$PMUCom$	Percentage of best conformations of the combination to be muted.
$IMUCom$	The intensification of the mutation of elements generated by combination.
$PEIImp$	Percentage of best conformations of the combination to be improved.
$IIEImp$	The intensification of the improvement of elements generated by combination.
$PBEInc$	Percentage of best conformations to be included in the reference set.
$NIREnd$	Maximum number of steps without improvement.
$MNIEnd$	Maximum number of iterations with or without improvement.

As previously explained, $METADOCK$ is able to configure the search procedure at compile time. This is performed by setting values to different parameters, which are listed in Table 1. These parameters are introduced in several functions of the general computational pattern (schema) that many population-based metaheuristics have in common (see Algorithm 1). The functions are briefly explained:

- **Initialize** returns an initial set of solutions. `INEIni` conformations are generated randomly. In this first generation, conformations are created with a different position and orientation around each spot. Then a percentage (`PEIIni`) of the initial conformations of each spot is improved. The intensification of the improvement is indicated by `IIEIni`. This improvement is a local search in which each conformation is modified within its neighborhood in the solutions space; i.e., it is translated or rotated with respect to its corresponding protein spot. Then, the scoring function is calculated to evaluate those new conformations. Finally, (`PBEIni`+`PWEIni`)*`INEIni` conformations from each spot are selected for the execution of the subsequent functions. `PBEIni` and `PWEIni` represent the percentage of best and worst conformations according to the scoring function. The best conformations are those with the best fitness, and the "worst" conformations are randomly selected from the remaining ones. $METADOCK$ does not select only the best conformations so as to diversify the search and prevent falling in local optima.

- **End condition** determines the stop criteria for METADOCK, which is either `MNIEnd` (maximum number of iterations), or `NIREnd` (maximum number of steps without improvement of the best solution among all the spots).

- **Select** chooses working conformations for subsequent phases. A percentage of the best (`PBESel`) and worst (`PWESel`) conformations relative to each spot are selected.

- **Combine** mixes conformations in pairs depending on their scoring. Three parameters represent the percentage of best-best, worst-worst and best-worst conformations to be combined: `PBBCom`, `PWWCom` and `PBWCom`, respectively. Combinations are performed among conformations at the same spot. In each combination, two conformations are generated with a different orientation and located on the line connecting the two elements to be combined.

- **Mutation** maintains the diversity of conformations after the *Combine* stage. For those conformations affected by the mutation, their position in the space or its orientation is modified randomly around the spot they are associated to. Two parameters are involved in this function: `PMUCom`, to define the percentage of conformations that the mutation procedure receives as an input, and `IMUCom`, the intensification of improvement of the elements obtained by mutation.

- **Improve** performs a local search within the neighborhood of some of the conformations previously generated by *Combine*. As in the improvement after initialization, two metaheuristic parameters are considered for each spot: `PEIImp`, for the percentage of conformations the local search is applied to, and `IIEImp`, for the number of trials for the local search. Hence, METADOCK can generate hybrid metaheuristics with different degrees of intensification.

- **Include** updates the reference set for the next iteration of the schema. Here, `PBEInc` establishes the percentage of best conformations associated to each spot to be included in its reference set. The remaining conformations in the reference set are randomly selected and contribute to diversify the search, so avoiding stalling in local minima.

4. Targeting heterogeneous clusters

This section introduces the parallelization strategy of our docking methodology presented in Section 3 for a heterogeneous cluster based on CPUs-GPUs. First, our algorithm defines an MPI process for each existing node in the cluster where we run our simulation. The number of spots is sent to each node, where supporting data structures are also created to avoid communication overhead. Then, each MPI process creates as many OpenMP threads as GPUs are available at the node, which is easily obtained by querying the GPU properties at runtime using `cudaGetDeviceCount`.

Algorithm 2 shows the parallelization schema we use to leverage heterogeneous nodes with shared-memory multiprocessors and multiple GPUs. OpenMP is used to manage several CPU threads, where each thread is responsible for controlling a GPU (lines 2 and 3). The targeted GPU for the actual CPU process is selected (line 4) and, from that point, all operations will be related to a different GPU. Some functions in Algorithm 2 work with various sets or populations (*Rhost* and *Rdevice*). These sets represent the receptor molecule on the CPU and GPU memories respectively. Line 5 copies *Rhost* into each GPU's device memory. Note that the whole receptor

Algorithm 2 Scoring function computation for multiGPU on each node

```
1:  omp_set_num_threads(number_GPUs)
2:  #pragma omp parallel for
3:  for i=1 to number_GPUs do
4:      Select_device(Devices[i].id)
5:      Host_To_GPU(Rhost,Sdevice)
6:      Conformations=Devices[i].conformations
7:      threads=Devices[i].Threadsblock
8:      stride=Devices[i].stride
9:      Calculate_scoring<Conformations/threads,threads>
        (Rdevice+Devices[i].stride)
10:     GPU_To_Host(Rhost,Rdevice)
11: end for
```

molecule is copied in all GPU memories. Although the computation of the scoring function at each spot is distributed among the different GPUs, all atoms of the receptor are needed to calculate the Lennard-Jones potentials (see [23]). Moreover, an additional structure, called *Devices*, is created to manage several configuration parameters. This structure stores, among other things, the number of conformations (line 6) assigned to each GPU; i.e., the number of different ligand configurations that will be executed at each spot. *Devices* also includes information about the ligand compound and, with that information, the different ligand conformations will be generated (translating or rotating this information) on the GPU. *Devices* also have some GPU runtime parameters such as memory, grid size, maximum threads per block (line 7), stride on set of population (line 8) and so on. Then, each GPU calculates the scoring function (line 9) for a set of conformations (i.e., candidate solutions). In our homogeneous implementation, those conformations are equally distributed among GPUs in form of CUDA thread blocks. Actually, we associate each conformation to a CUDA warp, and warps are grouped into blocks depending on the CUDA thread block granularity.

Parallel runs do not incur any communication overhead, and the final solution is chosen from all the independent executions, given the stochastic nature of metaheuristics. The execution time of each independent execution can differ, as it depends on (1) the underlying GPU each metaheuristic instance runs on, which is actually unknown at compile-time, and (2) the number of conformations (the same in principle for each computing unit, but the execution time is affected by GPU heterogeneity). Given that the slowest GPU will determine the overall execution time, our mission in the next steps is to make use of the idle time offered by the most powerful GPUs. This requires the implementation of a load balancing strategy that can somehow distribute a higher number of conformations to the most powerful GPUs. Indeed, there is a trade off between performance and overhead introduced in the design of this load balancing strategy. Here, we propose two different alternatives. A load balancing strategy based on the features offered by the manufacturer for each device (*Theoretical Distribution*) and a load-balancing strategy that explores the application performance on each GPU before the computation is actually carried out. Indeed, the former is straightforward as peak performances provided by manufacturer are under "ideal" conditions but it does not require an additional computational time. The latter, however, would introduce an overhead but it will theoretically obtain better application performances. This overhead is mainly due to a *warm-up* phase where you can get the real performance of each GPU for our problem. This phase is common for all metaheuristics, and it is done to establish performance differences among all targeted GPU by running the scoring function for a few simulated solutions. This phase measures the execution time of a small number of iterations in order to detect these differences. Importantly, at this stage, the algorithm is not trying to *solve* the docking problem in any meaningful sense (five to ten iterations are not enough to do this), but these runs do allow us to calculate the performance differences between GPUs. The execution times in this *warm-up* phase in all GPUs are reduced to obtain the maximum value using `mpi_Reduce`. Each node then calculates the number of conformations to deal with based on this information.

5. Evaluation

This section shows an experimental evaluation of our three different virtual screening strategies running on a heterogeneous cluster based on Intel CPUs and NVIDIA GPUs. First of all, we briefly introduce the hardware and software environment where the experiments are carried out. Afterwards, three different studies are carried out: (a) two different runtime environments are evaluated, namely the traditional MPI based programming approach and the rCUDA approach[2]; (b) different load balancing strategies are also analyzed in order to get peak performance of the system; (c) we carry out an analysis of the quality of our applications in terms of fitness.

5.1. Hardware environment and Benchmarking

Hardware and software environment: Experiments have been conducted in a cluster based on five 1027GR-TRF Supermicro nodes. Each node contains two Intel Xeon E5-2620 v2 processors, and has a Mellanox ConnectX-3 VPI single-port InfiniBand adapter (FDR InfiniBand). The nodes are connected by a Mellanox switch SX6025 with FDR compatibility (a maximum rate of 56Gb/sec). Two different GPUs, NVIDIA Tesla K20m and NVIDIA Tesla K40m, are available for acceleration purposes at nodes K1 and K2. In nodes Gtx1 and Gtx2 one GeForce GTX 590 with 2 GPUs is available in each one. Additionally, one SYS7047GR-TRF Supermicro server, referred to as node K3 with identical processors is available with 4 NVIDIA Tesla K20m GPUs and 128 GB of DDR3 SDRAM memory at 1600MHz. The CentOS 6.4 operating system and the Mellanox OFED 2.4-1.0.4 were used at the servers along with the NVIDIA driver 346.96 and CUDA 7.0. The rCUDA version is 15.10 and the MPI configuration is based on MVAPICH2 2.0.

Table 2 gives insights about each GPU architecture found on these nodes. The experimental environment is summarized in Tables 3 and 4. The former shows each GPU location and Device Identifier. In other words, Table 3 shows the node where each GPU is located and the identifier assigned to each computational component. Table 4, however, shows the tag used in the experiments. For instance, the simulations performed with 2 GPUs use devices 0 and 1, which means K40m and K20m in the node K1, while simulations with 4 GPUs involve 0, 1, 2 and 3 GPUs, which means two nodes (K1 and K2).

Metaheuristics: Three different configurations of our metaheuristics are

[2]Remember that the rCUDA approach consistis of providing the application GPUs located anywhere in the cluster. In this way, the application is programmed as a shared-memory application without having to use MPI in order to make use of the GPUs located in other cluster nodes. Furthermore, the application code does not need to be specifically designed for rCUDA but it is just designed to use multiple GPUs.

Table 2: GPU architectures involved in the experimental study.

Feature	GTX 590	K20m	K40m
GPU generation	Fermi	Kepler	Kepler
Year released	2011	2013	2014
Raw computational power			
Number of cores	512	2496	2880
Core frequency (MHz)	1215	706	745
Peak processing (GFLOPS)	2x 1244	3520	4290
CUDA Compute Capability	2.0	3.5	3.5
Memory			
Size (GB)	2x 1.5	5.2	12
Frequency (MHz)	2x 1215	2x 2600	2x 3004
Width (bits)	384	320	384
Bandwith (GB/s)	163.8	208	288
Cache			
Shared memory / multipr.	48 KB	64 KB	64 KB
L2 cache	768 KB	1.5 MB	1.5 MB

Table 3: Hardware location and models of the experimental environment.

Device	**Model**	**Node**	**Device**	**Model**	**Node**
0	K40m	node K1	**6**	K20m	node K3
1	K20m	node K1	**7**	K20m	node K3
2	K40m	node K2	**8**	GTX 590	node Gtx1
3	K20m	node K2	**9**	GTX 590	node Gtx1
4	K20m	node K3	**10**	GTX 590	node Gtx2
5	K20m	node K3	**11**	GTX 590	node Gtx2

Table 4: Relation between experiment tag and hardware location shown in Table 3.

Tag (Number of GPUs)	**Devices used (IDs from Table 3)**	**Tag (Number of GPUs)**	**Devices used (IDs from Table 3)**
2 GPUs	0,1	**8 GPUs**	0,1,2,3,4,5 6,7
4 GPUs	0,1,2,3	**10 GPUs**	0,1,2,3,4,5 6,7,8,9,10
6 GPUs	0,1,2,3,4,5	**12 GPUs**	0,1,2,3,4,5 6,7,8,9,10,11,12

under study (referred to as M1, M2 and M3 in Table 5). The first hybrid metaheuristic (M1) is close to a *Genetic Algorithm* with a population of 2048 individuals for each spot. Elements are selected from the best ones for combination, and half of the resulting elements are mutated. This metaheuristic does not include local search to improve the conformations. The second metaheuristic (M2) is also an evolutionary method but, in this case, it is closer to a *Scatter Search* algorithm with a population of 512 individuals. In this case, all the elements are improved after the initial generation and the combination, and the improvement is a local search in the neighborhood of each element to obtain better solutions. The last metaheuristic (M3) is a method of search in the neighborhood. A local search is carried out at each spot by changing position and orientation of the conformations. Notice that we have used different population sizes for our metaheuristics because we are interested in the scalability of our system and, therefore metaheuristics are designed to have different populations, which means having different computing intensity. All individuals are considered for selection and combination to perform the algorithm previously explained in Section 2.2.

Table 5: Metaheuristics used for experimentation.

	COMBINATIONS		
	M1	M2	M3
$INEIni$	2048	512	2048
$PEIIni$	0	100	100
$IIEIni$	0	20	100
$PBEIni$	100	4	0
$PWEIni$	0	4	0
$PBESel$	2	50	0
$PWESel$	0	50	0
$PBBCom$	100	100	0
$PWWCom$	0	100	0
$PBWCom$	0	100	0
$PMUCom$	50	0	0
$IMUCom$	20	0	0
$PEIImp$	0	100	0
$IIEImp$	0	20	0
$PBEInc$	100	70	0

Databases: A set of benchmark instances from the well-known Directory of Useful Decoys (DUD) [43] was used for testing. Surface screening was performed on proteins GPB, SRC and COMT and their corresponding crystallography ligands. Table 6 shows the size of each complex protein-ligand, with the number of atoms of each component and the number of spots of the receptor. They have different sizes to test the scalability of the methods implemented.

Table 6: Size of receptor and crystallography ligand (number of atoms) used for performance comparison, and the number of spots for each receptor.

Targets	Number of spots	Receptor size (number of atoms)	Crystallography ligand size (number of atoms)
GPB	813	13,261	52
SRC	452	7,158	67
COMT	214	3,419	29

5.2. Runtime evaluation

This section shows the performance evaluation of our VS methodology on a heterogeneous cluster based on CPU+GPU. We compare our MPI + OpenMP + CUDA version to leverage the heterogeneous cluster, which requires a larger programming effort, with an OpenMP + CUDA version that uses rCUDA as underlying runtime system. We ensured that simulations in both systems featured the same starting point by setting the seed in the random number generator. The idea after this comparison is that four GPUs are actually the maximum number of GPUs that are physically installed within the same node in our cluster. Therefore, the only way to use all GPUs in the cluster would be to use libraries like MPI. This introduces an extra programming effort to access all the available computational resources. Another way to do that, bypassing the MPI requirement, is to use a runtime system such as rCUDA that transparently “brings” all GPUs available in the cluster to a given node where the computation is carried out. The question that comes up here is if there is any overhead related to the use of runtime systems like rCUDA instead of a hand-made MPI code. Figure 2 shows the execution time of $METADOCK$ using both alternatives. In particular, we use a GRASP metaheuristic ($M3$) The number of the overall number of GPUs is incremented on the X-axis and different protein-ligand conformations are simulated (see Table 6) to analyze the system scalability. Indeed, the $METADOCK$ performance increases with the number of GPUs. As previously explained, the metaheuristic runs simultaneously at each spot and the spots are equally distributed among GPUs. These results justify the use of multiple GPUs to speedup our simulations. Execution times of the MPI and rCUDA versions are very close. Actually, a small performance gain is reported in the rCUDA version.

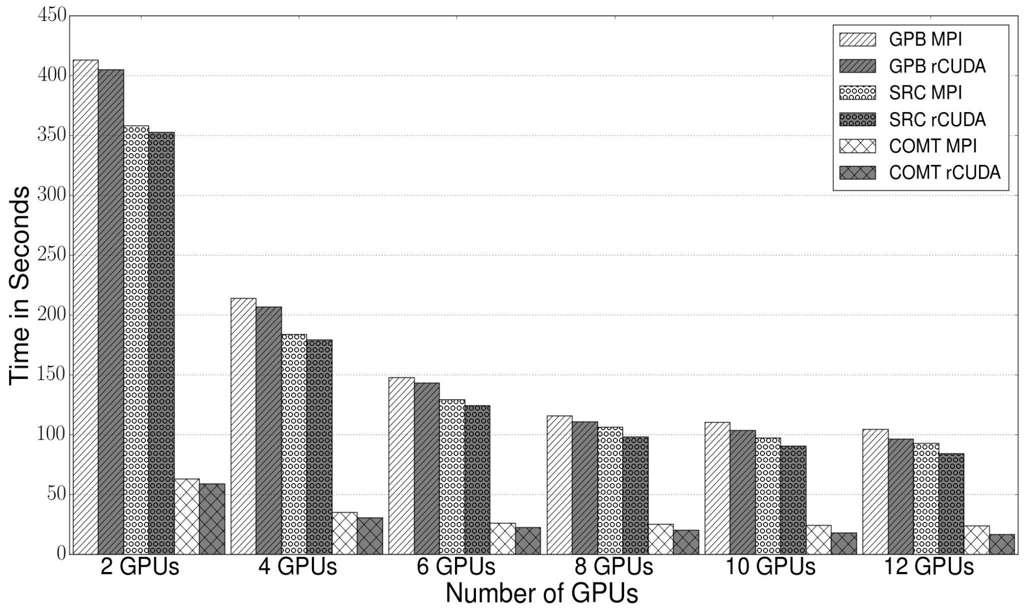

Figure 2: Execution time (in seconds) with COMT, GPB and SRC complexes from two to twelve GPUs with M3 for our MPI + OpenMP + CUDA implementation vs. OpenMP + CUDA with rCUDA middleware.

The reason for the performance gain with rCUDA is the following. In a classical approach, programming interfaces like MPI are used to involve several nodes to solve a given problem. In contrast, rCUDA avoids the use of MPI, which eases the development and avoids the MPI runtime overhead by using a very efficient communication pipeline between client and server nodes [21]. In addition to make the program more complex from the programming point of view, MPI also presents a non-negligible overhead when dealing with exchanged messages. This means that a shared-memory application may present a better performance than an MPI one for the same amount of leveraged CPU cores. As a result of the combination of both factors, our application provides better performance when it accesses GPUs in other nodes of the cluster by using rCUDA instead of MPI.

5.3. Load-balancing strategy evaluation

Table 7: Workload (in percentages) for each GPU with two distributions (Homogeneous and Theoretical) for COMT, GPB and SRC complexes.

Numbers of GPUs	**Homogeneous Distribution device / workload**		**Theoretical Distribution device / workload**	
2 GPUs	0 - 50%	1 - 50%	0 - 55%	1 - 45%
4 GPUs	0 - 25%	2 - 25%	0 - 27.5%	2 - 27.5%
	1 - 25%	3 - 25%	1 - 22.5%	3 - 22.5%
6 GPUs	0 - 16.6%	3 - 16.6%	0 - 19.3%	3 - 15.3%
	1 - 16.6%	4 - 16.6%	1 - 15.3%	4 - 15.3%
	2 - 16.6%	5 - 16.6%	2 - 19.3%	5 - 15.3%
8 GPUs	0 - 12.5%	4 - 12.5%	0 - 14%	4 - 12%
	1 - 12.5%	5 - 12.5%	1 - 12%	5 - 12%
	2 - 12.5%	6 - 12.5%	2 - 14%	6 - 12%
	3 - 12.5%	7 - 12.5%	3 - 12%	7 - 12%
10 GPUs	0 - 10%	5 - 10%	0 - 13%	5 - 11%
	1 - 10%	6 - 10%	1 - 11%	6 - 11%
	2 - 10%	7 - 10%	2 - 13%	7 - 11%
	3 - 10%	8 - 10%	3 - 11%	8 - 4%
	4 - 10%	9 - 10%	4 - 11%	9 - 4%
12 GPUs	0 - 8.3%	6 - 8.3%	0 - 12%	6 - 10%
	1 - 8.3%	7 - 8.3%	1 - 10%	7 - 10%
	2 - 8.3%	8 - 8.3%	2 - 12%	8 - 4%
	3 - 8.3%	9 - 8.3%	3 - 10%	9 - 4%
	4 - 8.3%	10 - 8.3%	4 - 10%	10 - 4%
	5 - 8.3%	11 - 8.3%	5 - 10%	11 - 4%

The existence of different families of GPUs (see Table 2) in the same cluster requires to adapt the amount of work to be carried out by the different kernels to the different computing power of the devices in the cluster.

Table 8: Workload (in percentages) for each GPU with heterogeneous distributions for COMT, GPB and SRC complexes.

Numbers of GPUs	**COMT Heterogeneous Distribution device / workload**		**GPB Heterogeneous Distribution device / workload**		**SRC Heterogeneous Distribution device / workload**	
2 GPUs	0 - 55.7%	1 - 44.3%	0 - 55.1%	1 - 44.9%	0 - 55.5%	1 - 44.5%
4 GPUs	0 - 28.1%	2 - 27.4%	0 - 27.1%	2 - 27.4%	0 - 27.7%	2 - 27.9%
	1 - 21.9%	3 - 22.6%	1 - 22.6%	3 - 22.9%	1 - 22.3%	3 - 22.1%
6 GPUs	0 - 19.1%	3 - 15.3%	0 - 18.9%	3 - 15%	0 - 18.7%	3 - 15.3%
	1 - 15.1%	4 - 15.7%	1 - 15.9%	4 - 15.9%	1 - 15.8%	4 - 15.7%
	2 - 18.3%	5 - 16.5%	2 - 17.9%	5 - 16.4%	2 - 18.2%	5 - 16.3%
8 GPUs	0 - 14.4%	4 - 11.8%	0 - 15%	4 - 11.1%	0 - 14.7%	4 - 11.5%
	1 - 12.1%	5 - 11.9%	1 - 11.8%	5 - 11.9%	1 - 11.7%	5 - 11.7%
	2 - 15%	6 - 11.7%	2 - 14.6%	6 - 11.8%	2 - 14.8%	6 - 11.9%
	3 - 11.6%	7 - 11.5%	3 - 12.1%	7 - 11.7%	3 - 11.9%	7 - 11.8%
10 GPUs	0 - 14.1%	5 - 10.5%	0 - 14%	5 - 11.2%	0 - 13.9%	5 - 11.0%
	1 - 11.6%	6 - 10.7%	1 - 11.2%	6 - 10.6%	1 - 11.1%	6 - 10.7%
	2 - 14.1%	7 - 11%	2 - 14.3%	7 - 10.4%	2 - 14.1%	7 - 10.8%
	3 - 11.1%	8 - 3.2%	3 - 10.8%	8 - 3.1%	3 - 10.9%	8 - 3.4%
	4 - 10.6%	9 - 3.1%	4 - 11%	9 - 3.4%	4 - 10.8%	9 - 3.3%
12 GPUs	0 - 13%	6 - 10.1%	0 - 13%	6 - 10.4%	0 - 12.8%	6 - 10.3%
	1 - 10.5%	7 - 10.7%	1 - 10.6%	7 - 10.8%	1 - 10.6%	7 - 10.5%
	2 - 13.1%	8 - 2.5%	2 - 13.1%	8 - 2.6%	2 - 12.9%	8 - 2.7%
	3 - 10.4%	9 - 2.9%	3 - 10.3%	9 - 2.9%	3 - 10.4%	9 - 3.0%
	4 - 10.2%	10 - 3.1%	4 - 10.2%	10 - 2.8%	4 - 10.3%	10 - 3.0%
	5 - 10.6%	11 - 2.9%	5 - 10.6%	11 - 2.7%	5 - 10.7%	11 - 2.8%

We have adapted $METADOCK$ to work with heterogeneous architectures and massively parallel techniques applied to areas of intensive computing, distributing the work onto all the available devices. The independence of a conformation with respect to the others when calculating their potential makes it easier to take advantage of the heterogeneity by dividing computation.

This section evaluates three different load-balancing techniques:

- In the *Homogeneous Distribution* all the devices receive the same workload.
- In *Theoretical Distribution* the workload received by each device is calculated from the theoretical peak performance provided by the manufacturer.
- In the *Heterogeneous Distribution* the workload is established through a warm-up phase, previously explained in Section 4.

Tables 7 and 8 show the homogeneous, theoretical and heterogeneous distribution used for three protein-ligand complexes, varying the number of GPUs. In Table 7, the first and second columns show the percentage assigned to each device with the homogeneous distribution, and the percentages with the theoretical distribution are shown in columns three and four. With the homogeneous distribution the workload is equally distributed among all the GPUs, independently of the relative performance of the devices. The theoretical distribution uses the device performance provided by the manufacturer to obtain the percentages. In this case, Kepler GPUs are assigned more workload than Fermi GPUs. Table 8 shows the distribution obtained for the three complexes with the heterogeneous distribution. In this case, a higher percentage of workload is assigned to the GPUs that offer better performance for our problem. Some conclusions are drawn: (1) The theoretical distribution for the Kepler family is close to the heterogeneous distribution which actually explores the real computational differences among GPUs. (2) The workload for the three complexes in the heterogeneous distribution is very similar and does not depend on their sizes.

Table 9 shows the execution time for the warm-up phase using MPI. The warm-up phase in the MPI case explores the computational capabilities and distributes the work within each node; i.e. an MPI process is launched to each node where the warm-up phase is performed. Therefore, this is done

Table 9: Execution time (in seconds) of the warm-up phase for the MPI + OpenMP + CUDA implementation with three complexes (COMT, GPB and SRC).

Number of GPUs	**2 GPUs**	**4GPUs**	**6 GPUs**	**8 GPUs**	**10 GPUs**	**12 GPUs**
COMT	2.53	2.49	2.51	3.36	5.28	5.55
GPB	5.13	5.11	5.06	5.98	6.91	6.87
SRC	8.28	8.45	8.35	8.86	9.98	9.79

Table 10: Execution time (in seconds) of the warm-up phase for the OpenMP + CUDA implementation with rCUDA middleware for three complexes (COMT, GPB and SRC).

Number of GPUs	**2 GPUs**	**4GPUs**	**6 GPUs**	**8 GPUs**	**10 GPUs**	**12 GPUs**
COMT	1.08	2.21	3.42	5.57	8.01	10.823
GPB	2.07	3.62	4.21	6.34	9.71	12.51
SRC	2.26	4.24	4.71	7.19	11.25	14.43

in parallel at each node, and thus the execution time reported in Table 9 corresponds to the slowest node in computing this warm-up stage. The slowest GPUs targeted (GeForce GTX 590) are involved in the execution for 10 and 12 GPUs (see Tables 3 and 4), and they determine the overall execution time. Table 10 shows the execution time for the warm-up phase using rCUDA. Here, all the GPUs are virtually on a node and so the warm-up is not parallelized like in the MPI case. The progressive increase in execution time with the number of GPUs reflects this idea. The warm-up phase introduces an overhead in both cases, that is higher for rCUDA whenever many GPUs are targeted. After evaluating the workload percentages assigned to each GPU, we observe in Table 8 that small and large complexes have similar workloads. This leads us to think about a clever strategy to reduce the overhead. The warm-up phase could be executed once at the installation stage, when compounds of several sizes could be evaluated, so storing meta information for future executions. It is important to bear in mind that, when solving a real problem, many iterations are carried out so representative values are obtained, and the overhead incurred in the *warm-up* phase is less significant.

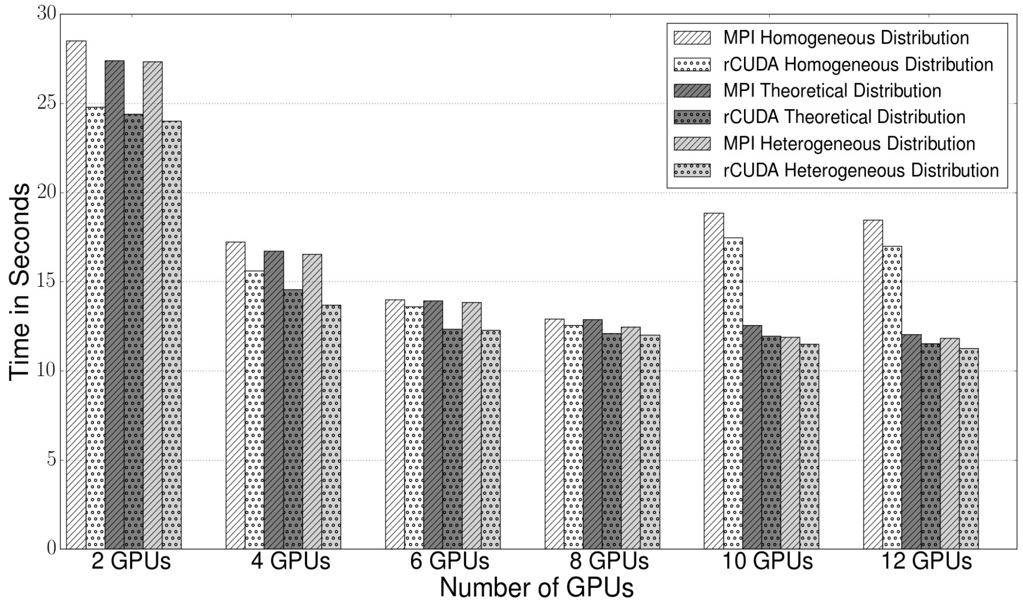

Figure 3: Execution time (in seconds) with COMT complex and M1 for different distributions with MPI + OpenMP + CUDA and OpenMP + rCUDA implementations.

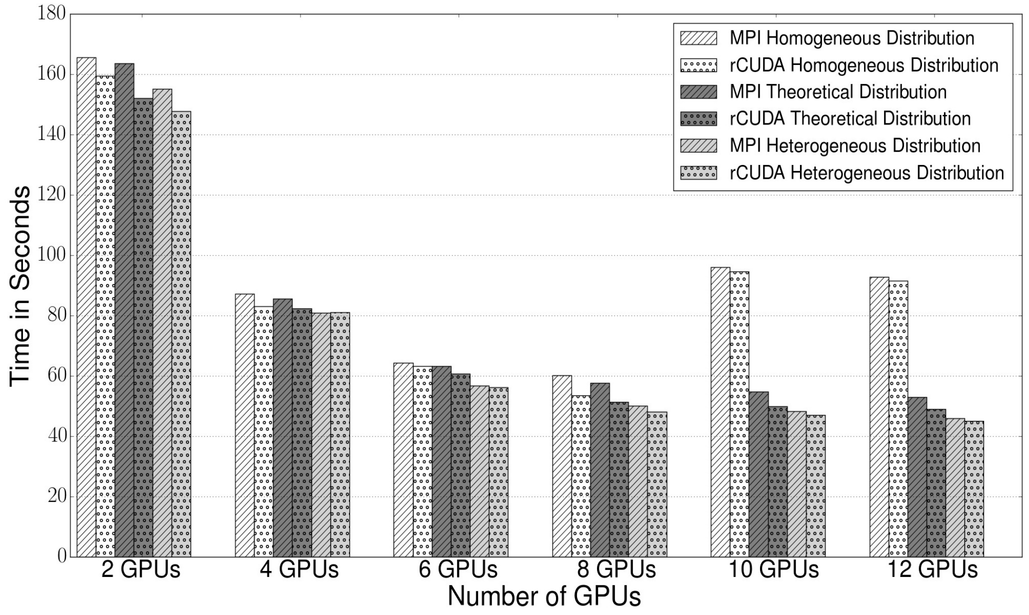

Figure 4: Execution time (in seconds) with GPB complex and M1 for different distributions with MPI + OpenMP + CUDA and OpenMP + rCUDA implementations.

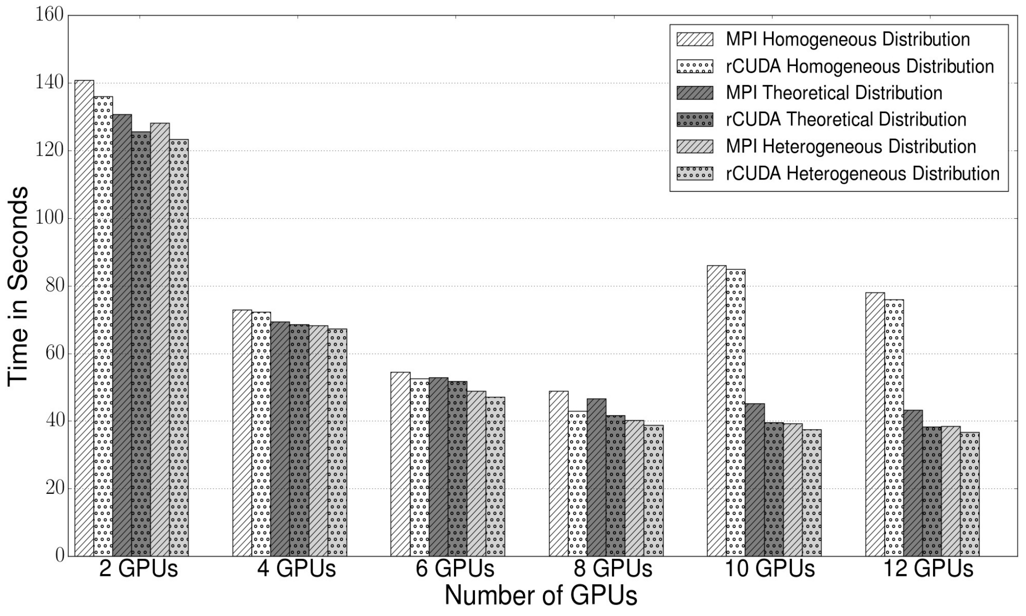

Figure 5: Execution time (in seconds) with SRC complex and M1 for different distributions with MPI + OpenMP + CUDA and OpenMP + rCUDA implementations.

Figures 3, 4 and 5 show the execution time in seconds for the load-balancing strategies considered (homogeneous, theoretical and heterogeneous), using MPI and rCUDA versions for the COMT, GPB and SRC protein-ligand complexes. We use only one metaheuristic (M1) for clarity, but the results are representative of the other metaheuristics experimented with. For low experimentation times, the results correspond to only one step of the metaheuristic, with a single execution of each function. The scalability is compared for the different complexes and implementations. In all the cases the homogeneous distribution obtains worse results, which are particularly noticeable when Fermi GPUs are introduced (10-12 GPUs case). The reason is clear, we introduce a GPU that offers a lower theoretical peak performance and the same workload is assigned to each GPU. On the contrary, the theoretical distribution has a similar behavior to the heterogeneous distribution, increasing the difference between MPI and rCUDA as the complex is bigger (see Figures 4 and 5). The heterogeneous distribution obtains the best performance. However, these execution times do not include the *warm-up* stage, which may be representative, as previously commented. For a real application where many steps are needed, the overhead would be less important. Furthermore, the application of the *warm-up* at installation stage would hide overhead.

5.4. Metaheuristic evaluation

In sections 5.2 and 5.3 we have verified that using a greater number of GPUs reduces the execution time. In this section we show that: (1) different metaheuristics have different behavior for the same compound; (2) parallelism contributes to better fitnesses, with larger number of GPUs enabling more evaluations, and consequently better results, in the same execution time; (3) for the same reason, the exploitation of heterogeneity also contributes to better fitnesses. rCUDA is used in the experiments.

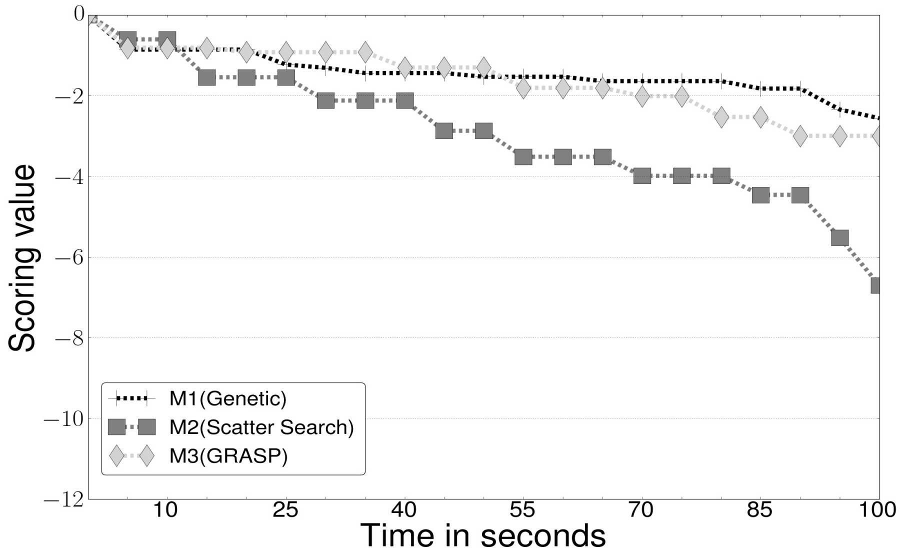

Figure 6: Comparison of fitness with the COMT complex and 12 GPUs on rCUDA for three metaheuristics at different time-steps.

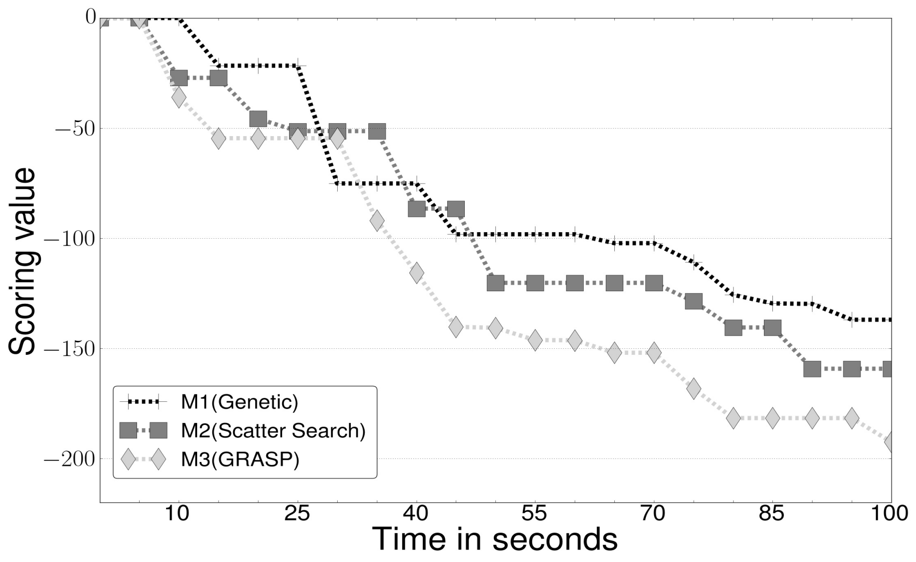

Figure 7: Comparison of fitness with the GPB complex and 12 GPUs on rCUDA for three metaheuristics at different time-steps.

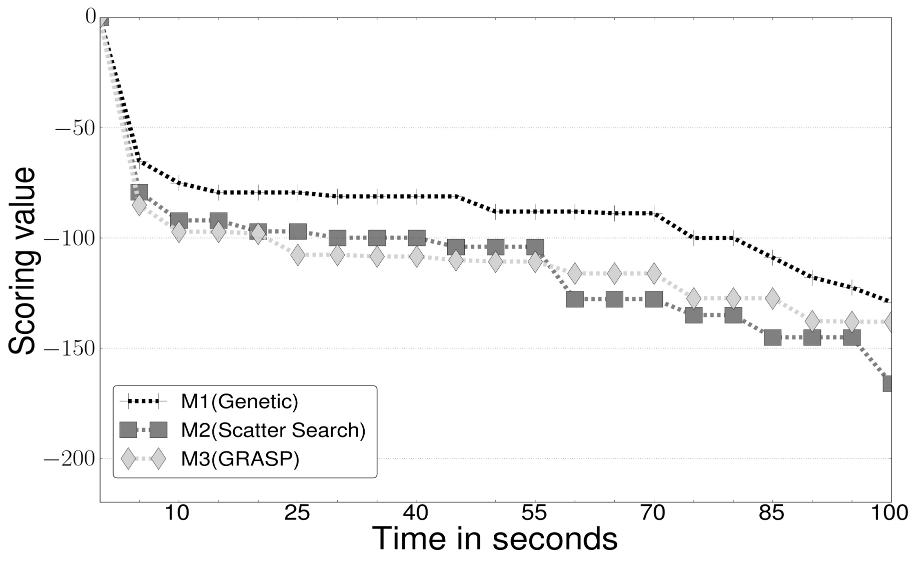

Figure 8: Comparison of fitness with the SRC complex and 12 GPUs on rCUDA for three metaheuristics at different time-steps.

Figures 6, 7 and 8 show the evolution of the fitness with the time of the different metaheuristics and complexes considered with rCUDA. As commented, the behavior of the metaheuristics is different depending on the complex and no metaheuristic offers the best results with all the complexes. The worst results are obtained with metaheuristic M1 (close to a Genetic Algorithm). With COMT and SRC complexes, M2 (close to Scatter Search) obtains the best fitness (minor value), while for GPB, with a greater number of spots to look for and more individuals per spot, the behavior of the local search metaheuristic is better. Metaheuristic M3 is used in the following experiments to analyze the influence of the workload distributions in Section 5.3 on the fitness.

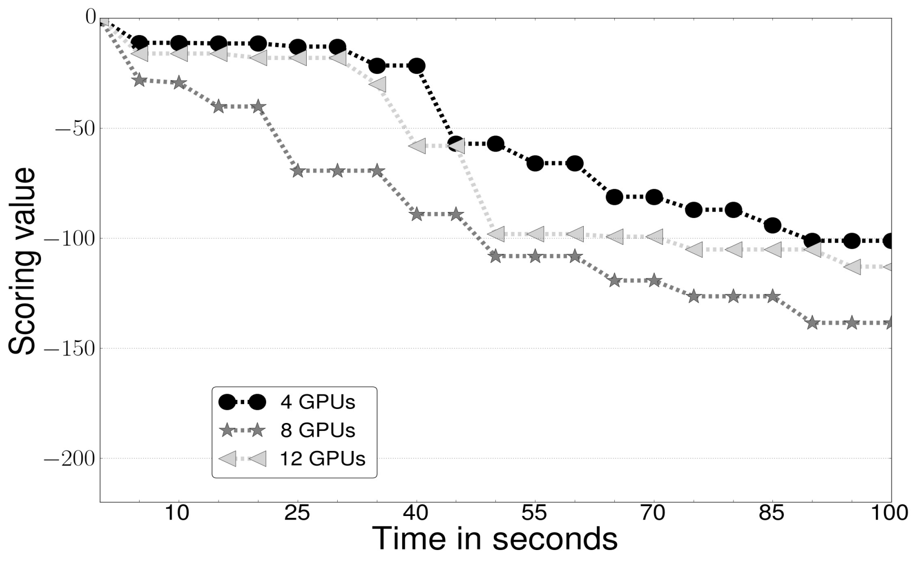

Figure 9: Comparative fitness for homogeneous distribution with GPB complex on 4, 8 and 12 GPUs on rCUDA with metaheuristic M3 and time-limit.

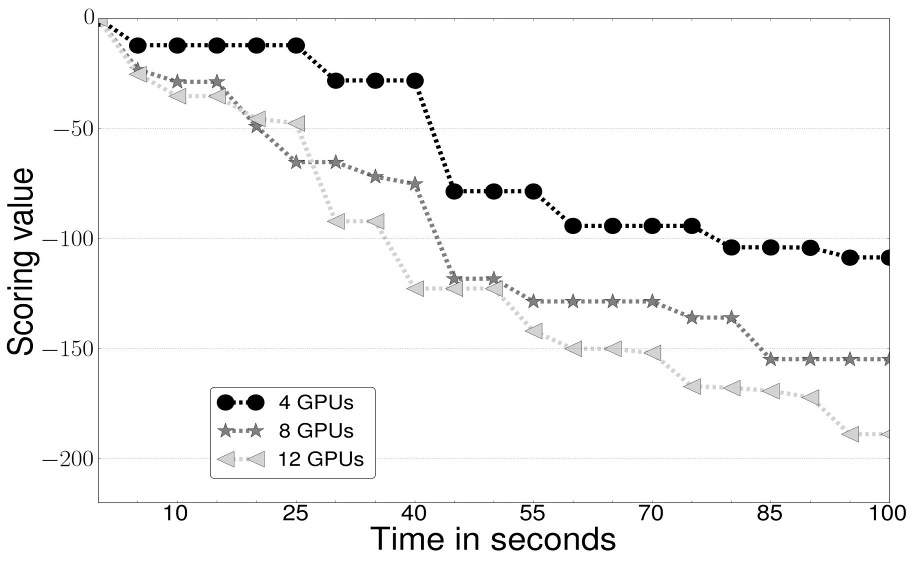

Figure 10: Comparative fitness for theoretical distribution with GPB complex on 4, 8 and 12 GPUs on rCUDA with metaheuristic M3 and time-limit.

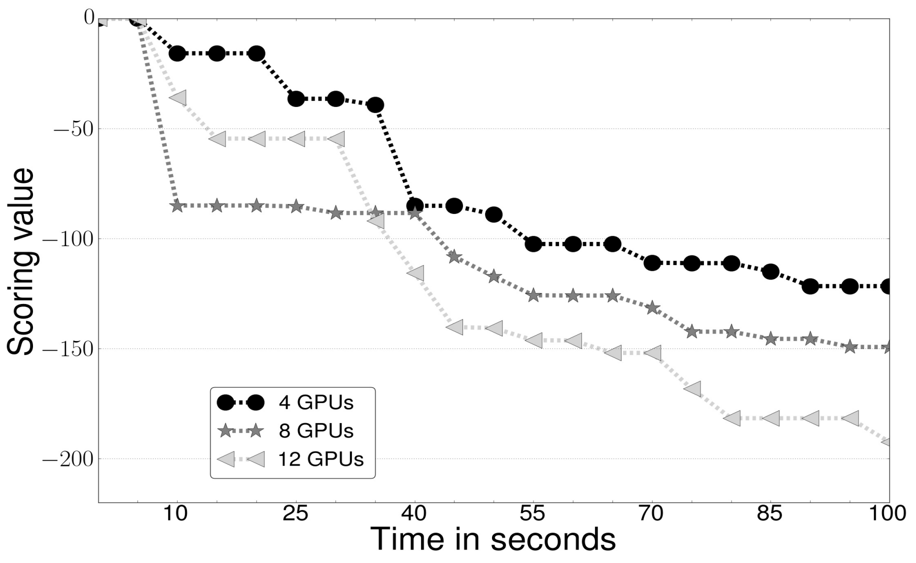

Figure 11: Comparative fitness for heterogeneous distribution with GPB complex on 4, 8 and 12 GPUs on rCUDA with metaheuristic M3 and time-limit.

Figures 9, 10 and 11 analize the quality of the solution with the number of GPUs with the three distributions (homogeneous, theoretical and heterogeneous). In Figure 9 we observe that with homogeneous distribution the fitness with 12 GPUs is worse than with 8, which is due to the same workload for Fermi and Kepler GPUs. This makes the execution slower and the number of steps is smaller than if only Kepler GPUs are used. Figures 10 and 11 show the behavior with the theoretical and heterogeneous distribution, in this case the quality of the fitness increases with the number of GPUs.

6. Related work

This section analyzes different docking techniques from a computational point of view. Widely-used docking programs like Autodock [6] and Autodock VINA [7] are CPU-based. They accelerate their computations by means of the OpenMP runtime to fully leverage multicore architectures. Other CPU-based docking programs are LeadFinder [9], Glide [8], SurFlex [10], ICM+ [11], DOCK [13] or FMD [12]. These applications use OpenMP, and some of them also use the MPI library in order to distribute their computations across the CPUs of several nodes of the cluster.

Moreover, there are other docking applications that use the GPU to offload the computational intensive parts of the computation. BUDE [15] is a software for molecular docking that leverages the heterogeneity of CPU-GPU systems. This software is programmed using OpenMP and OpenCL for portability to different architectures like NVIDIA and AMD GPUs, but it does not consider the use of the MPI technology. AMBER [16] also uses GPU acceleration. However, the possibility of using multiple GPUs requires using OpenMP or MPI, and therefore it cannot be directly compared with our environment. $BINDSURF$ [17] is fully developed in GPU, although it does not support MultiGPU or MPI to extend the computation to several nodes in a cluster.

7. Conclusions and Future work

Virtual screening methods are computational techniques that aid the drug discovery process. They are very computational demanding applications, and the use of clusters combining multicore CPUs and several GPUs contribute to accelerate their solution. However, these are heterogeneous systems, and heterogeneity may limit application performance unless programmers: (1) develop smart applications to control those features wisely on the road towards an optimal performance or (2) use middleware tools that manage these computing issues efficiently and transparently. This paper analyzes both options by developing initially an MPI, OPENMP and CUDA based Virtual Screening application and also by using rCUDA with only OpenMP and CUDA counterpart version of the same application. rCUDA offers an easy-to-use framework to develop data-intensive applications that use several remote GPUs transparently. We have not noticed any substantial pay off in terms of performance; actually some slightly improvements are reported in some cases, thanks to an efficient implementation of memory transfers through pinned buffers on rCUDA.

Virtual Screening requires the analysis of large data-bases of chemical compounds. Those compounds are independent of each other and, therefore, a load-balancing technique is necessary to distribute the workload efficiently among all GPUs, which can be from different generations. Here, three different load-balancing techniques are studied. Our baseline technique is a homogeneous distribution among GPUs which is not efficient as long as there are notable computational differences between GPUs. The theoretical distribution, based on the peak performance reported by the manufacturer, is a good option as it does not require extra computation and it determines relatively well performance differences between GPUs.

The best performance is achieved by our load balancing technique based on a warm-up strategy. The execution time of the warm-up phase largely increases as long as the number of GPUs does so. In particular, the virtualization offered by rCUDA avoids parallelization at the warm-up and therefore the execution time can increase notably. Anyway, the warm-up times reported are affordable in real applications where the number of steps of the metaheuristics for solutions of high quality is large, or when the warm-up is carried out for representative metaheuristics and sizes during the installation of the docking method for a computational system.

In summary, the main conclusions are that: (1) population-based metaheuristics hybridized with local search methods give satisfactory results for our docking problem; (2) parallelism can help to reduce both the execution time of this computationally demanding problem and the quality of the solutions; (3) a virtual system as rCUDA eases the exploitation of heterogeneous systems for the problem in hand; (4) to fully exploit this type of systems the heterogeneity should be considered for workload distribution, with a result in the improvement of the solutions as a consequence of the reduction of execution times.

For future work, and in order to deal with larger problems or for better solutions with limited execution times, it could be convenient to adapt our virtual screening method to even more complex systems, with other types of accelerators and with accelerators of various types and at different speeds in the same node. Energy efficiency should also be considered.

In this paper, GPU virtualization has been proved as a very promising technique, and, therefore, we will follow this path, including multi-tenancy at a GPU level, by running several instances of our program in the same physical GPU to increase the overall throughput. Indeed, an energy efficiency evaluation in this context will be a very interesting subject of study, which will also comprise the usage of other system architectures like 64-bit ARM-based systems. Moreover, virtual screening is still at a relatively early stage, and we acknowledge that we have tested a relatively simple variant of the algorithm. But, with many other types of scoring functions still to be explored, this field seems to offer a promising and potentially fruitful area of research.

Acknowledgments

This work is jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 18946/JLI/13, and by the Spanish MEC and European Commission FEDER under grants TIN2015-66972-C5-3-R and TIN2016-78799-P (AEI/FEDER, UE). We also thank NVIDIA for hardware donation under GPU Educational Center 2014-2016 and Research Center 2015-2016. Furthermore, researchers from Universitat Politècnica de València are supported by the Generalitat Valenciana under Grant PROMETEO/2017/077. Authors are also grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] P. J. Hajduk, J. Greer, A decade of fragment-based drug design: strategic advances and lessons learned, Nature Reviews Drug discovery 6 (2007) 211–219.

[2] W. L. Jorgensen, The Many Roles of Computation in Drug Discovery, Science 303 (2004) 1813–1818.

[3] J. M. Rollinger, H. Stuppner, T. Langer, Virtual screening for the discovery of bioactive natural products, in: Natural Compounds as Drugs Volume I, Springer, 2008, pp. 211–249.

[4] J. J. Irwin, B. K. Shoichet, ZINC–a free database of commercially available compounds for virtual screening, Journal of Chemical Information and Modeling 45 (2005) 177–182.

[5] D. B. Kitchen, H. Decornez, J. R. Furr, J. Bajorath, Docking and scoring in virtual screening for drug discovery: methods and applications, Nature Reviews Drug Discovery 3 (2004) 935–949.

[6] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, A. J. Olson, Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function, Journal of Computational Chemistry 19 (1998) 1639–1662.

[7] O. Trott, 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 31 (2010) 455–461.

[8] R. A. Friesner, et al., Glide: A New Approach For Rapid, Accurate Docking and Scoring: Method and Assessment of Docking Accuracy, Journal of Medicinal Chemistry 47 (2004) 1739–1749.

[9] O. V. Stroganov, F. N. Novikov, V. S. Stroylov, V. Kulkov, G. G. Chilov, Lead finder: an approach to improve accuracy of protein- ligand docking, binding energy estimation, and virtual screening, Journal of Chemical Information and Modeling 48 (2008) 2371–2385.

[10] A. N. Jain, Surflex: fully automatic flexible molecular docking using a molecular similarity-based search engine, Journal of Medicinal Chemistry 46 (2003) 499–511.

[11] P. Smielewski, A. Lavinio, I. Timofeev, D. Radolovich, I. Perkes, J. Pickard, M. Czosnyka, ICM+, a flexible platform for investigations of cerebrospinal dynamics in clinical practice, in: Acta Neurochirurgica Supplements, Springer, 2008, pp. 145–151.

[12] R. Dolezal, T. C. Ramalho, T. C. França, K. Kuca, Parallel flexible molecular docking in computational chemistry on high performance computing clusters, in: Computational Collective Intelligence, Springer, 2015, pp. 418–427.

[13] T. J. A. Ewing, S. Makino, A. G. Skillman, I. D. Kuntz, DOCK 4.0: Search strategies for automated molecular docking of flexible molecule databases, Journal of Computer-Aided Molecular Design 15 (2001) 411–428.

[14] Top500, Top500 supercomputer site, `http://www.top500.org/`, 2017. (accessed, April, 3th, 2017).

[15] S. McIntosh-Smith, J. Price, R. B. Sessions, A. A. Ibarra, High performance in silico virtual drug screening on many-core processors, The International Journal of High Performance Computing Applications 29 (2015) 119–134.

[16] R. Salomon-Ferrer, A. W. Gotz, D. Poole, S. Le Grand, R. C. Walker, Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. explicit solvent particle mesh ewald, Journal of Chemical Theory and Computation 9 (2013) 3878–3888.

[17] I. Sánchez-Linares, H. Pérez-Sánchez, J. M. Cecilia, J. M. García, High-throughput parallel blind virtual screening using BINDSURF, BMC Bioinformatics 13 (2012) S13.

[18] J. Carretero, et al., Optimizations to enhance sustainability of MPI applications, in: Proceedings of the 21st European MPI Users' Group Meeting, ACM, 2014, p. 145.

[19] M. Rosenblum, T. Garfinkel, Virtual machine monitors: Current technology and future trends, Computer 38 (2005) 39–47.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, ACM SIGOPS Operating Systems Review 37 (2003) 164–177.

[21] C. Reaño, F. Silla, G. Shainer, S. Schultz, Local and Remote GPUs Perform Similar with EDR 100G InfiniBand, in: Proceedings of the

16th International Middleware Conference, Middleware '15, ACM, New York, NY, USA, 2015, pp. 4:1–4:7.

[22] S. Iserte, J. Prades, C. Reaño, F. Silla, Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm, in: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), ACM, 2016, pp. 98–101.

[23] B. Imbernón, J. M. Cecilia, H. Pérez-Sánchez, D. Giménez, METADOCK: A Parallel Metaheuristic schema for Virtual Screening methods (in press), The International Journal of High Performance Computing Applications (2017).

[24] A. A. Franco, Multiscale modelling and numerical simulation of rechargeable lithium ion batteries: concepts, methods and challenges, RSC Advances 3 (2013) 13027–13058.

[25] N. Lagarde, J.-F. Zagury, M. Montes, Benchmarking data sets for the evaluation of virtual ligand screening methods: review and perspectives, Journal of Chemical Information and Modeling 55 (2015) 1297–1307.

[26] A. N. Jain, Scoring functions for protein-ligand docking, Current Protein and Peptide Science 7 (2006) 407–420.

[27] P. Csermely, T. Korcsmáros, H. J. Kiss, G. London, R. Nussinov, Structure and dynamics of molecular networks: a novel paradigm of drug discovery: a comprehensive review, Pharmacology & Therapeutics 138 (2013) 333–408.

[28] J. Wang, Y. Deng, B. Roux, Absolute Binding Free Energy Calculations Using Molecular Dynamics Simulations with Restraining Potentials, Biophys J 91 (2006) 2798–2814.

[29] L. Bianchi, M. Dorigo, L. M. Gambardella, W. J. Gutjahr, A survey on metaheuristics for stochastic combinatorial optimization, Natural Computing: An International Journal 8 (2009) 239–287.

[30] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Computing Surveys (CSUR) 35 (2003) 268–308.

[31] G. R. Raidl, A unified view on hybrid metaheuristics, in: Hybrid Metaheuristics, Springer, 2006, pp. 1–12.

[32] P. Hansen, N. Mladenović, Variable neighborhood search, in: Search Methodologies, Springer, 2014, pp. 313–337.

[33] F. Almeida, D. Giménez, J.-J. López-Espín, M. Pérez-Pérez, Parameterised schemes of metaheuristics: basic ideas and applications with Genetic algorithms, Scatter Search and GRASP, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 43 (2013) 570–586.

[34] L.-G. Cutillas-Lozano, J.-M. Cutillas-Lozano, D. Giménez, Modeling shared-memory metaheuristic schemes for electricity consumption, in: Distributed Computing and Artificial Intelligence - 2012 9th International Conference, pp. 33–40.

[35] J.-M. Cutillas-Lozano, D. Giménez, Determination of the kinetic constants of a chemical reaction in heterogeneous phase using parameterized metaheuristics, Procedia Computer Science 18 (2013) 787–796.

[36] T. Austin, Bridging the Moore's Law Performance Gap with Innovation Scaling, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ACM, 2015, p. 1.

[37] D. B. Kirk, W. H. Wen-mei, Programming massively parallel processors: a hands-on approach, Elsevier, 2013.

[38] OpenMP Architecture Review Board, The OpenMP Specification, `http://www.openmp.org`, 2017. (accessed, April, 2th, 2017).

[39] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, D. K. Panda, MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters, Computer Science-Research and Development 26 (2011) 257.

[40] D. R. Kaeli, P. Mistry, D. Schaa, D. P. Zhang, Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015.

[41] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, J. Duato, A complete and efficient CUDA-sharing solution for HPC clusters, Parallel Computing 40 (2014) 574–588.

[42] S. K. Kuntz, R. C. Murphy, M. T. Niemier, J. A. Izaguirre, P. M. Kogge, Petaflop Computing for Protein Folding, in: Proceedings of the 2001 Tenth SIAM Conference on Parallel Processing for Scientific Computing, pp. 12–14.

[43] DUD, Directory of Useful Decoys, `http://dud.docking.org/`, 2006. (accessed, April, 3th, 2016).