Parallelization of Virtual Screening in Drug Discovery on Massively Parallel Architectures

Ginés D. Guerrero, Horacio E. Pérez-Sánchez, José M. Cecilia, José M. García
Grupo de Arquitectura y Computación Paralela
Dpto. de Ingeniería y Tecnología de Computadores
Universidad de Murcia, Campus de Espinardo, 30100 Murcia, Spain
{gines.guerrero, horacio, chema, jmgarcia}@ditec.um.es

Abstract

The current trend in medical research for the discovery of new drugs is the use of Virtual Screening (VS) methods. In these methods, the calculation of the non-bonded interactions, such as electrostatics or van der Waals forces, plays an important role, representing up to 80% of the total execution time. These kernels are computational intensive and massively parallel in nature, and thus they are well suited to be accelerated on parallel architectures. In this work, we discuss the effective parallelization of the non-bonded electrostatic interactions kernel for VS on three different parallel architectures: a shared memory system, a distributed memory system, and a Graphics Processing Units (GPUs). For an efficient handling of the computational intensive and massively parallelism of this kernel, we enable different data policies on those architectures to take advantage of all computational resources offered by them. Four implementations are provided based on MPI, OpenMP, Hybrid MPI-OpenMP and CUDA programming models. The sequential implementation is defeated by a wide margin by all parallel implementations, obtaining up to 72x speed-up factor on the shared memory system through OpenMP, up to 60x and 229x speed-ups factors on the distributed memory system for the MPI implementation and the Hybrid MPI-OpenMP implementation respectively, and finally, up to 213x speed-up factor for the CUDA implementation on the GPU architecture to offer the best alternative in terms of performance/cost ratio.

1. Introduction

The discovery of new drugs can enormously benefit from the use of Virtual Screening (VS) methods [12]. The different approaches used in VS methods differ mainly by the way they model the interacting molecules but all of them have in common that they screen databases of chemical compounds containing up to millions of ligands [11]. Larger databases increase the chances of generating hits or leads, but the computational time needed for the calculations increases not only with the size of the database but also with the accuracy of the chosen VS method. Fast docking methods with atomic resolution require a few minutes per ligand [29], while more accurate molecular dynamics-based approaches still require hundreds or thousands of hours per ligand [28]. Therefore, the limitations of VS predictions are directly related to a lack of computational resources, a major bottleneck that prevents the application of detailed, high-accuracy models to VS.

In most of the VS methods the biological system is represented in terms of interacting particles. For the calculation of the interaction energies, classical potentials are commonly used, separated into bonded and non-bonded terms. The latter describe interactions between all the elements of the system. The relevant non-bonded potentials used in VS calculations are the Coulomb and the Lennard-Jones potentials, since these describe very accurately the most important short and long range interactions between protein and ligand atoms.

In VS methods the most intensive computations are spent in the calculation of non-bonded kernels. For example, in Molecular Dynamics it takes up to 80% of the total execution time [14]. Thus this part can be considered as a bottleneck, and its has been shown that its parallelization and optimization [23] permits VS methods to deal with more complex systems, simulate longer time scales or screen larger databases.

High Performance Computing (HPC) solutions [13, 24] have demonstrated they can increase considerably the performance of the different VS methods, as well as, the quality and quantity of the conclusions we can get from screening. In addition, emergent HPC platforms such as Cell BE processor [3], and more recently, Graphics Processing Units (GPUs) [15] are at the leading edge of increasing chip-level

parallelism. They have been widely applied in many different fields of applications [17, 6], and concretely in VS methods [22] [8]. Moreover, driven by the video game market, they offer this compelling solution at very low prices. All GPU platforms can be programmed using the Compute Unified Device Architecture (CUDA) programming model which makes the GPU to operate as a highly parallel computing device.

In this work, we discuss different HPC implementations for a VS method on three different parallel architectures: a shared memory system, a distributed memory system and a Graphics Processing Unit (GPU). Moreover, three different programming models are used: MPI [16], OpenMP [21], and CUDA [20]. We also mix MPI and OpenMP to provide a hybrid solution [25] that takes advantage of all resources available on a cluster.

Our results reveals that all our designs defeat by a wide margin the sequential counterpart version of the non-bounded kernel. We obtain up to 72x speed-up factor on shared memory system, using a OpenMP implementation; up to 60x and 229x speed-ups factors on the distributed memory architecture for the MPI implementation and the Hybrid MPI-OpenMP implementation respectively, and finally, up to 213x speed-up factor for the CUDA implementation on the GPU architecture. The GPU architecture provides a really compelling high performance solution, obtaining similar performance than large cluster of computers but a much lower cost.

The rest of the paper is structured as follows. In Section 2 we present the previous work and the main objectives of this research. Section 3 describes the sequential algorithm, and the parallel algorithms that have been implemented in CUDA, OpenMP, MPI programming models. We present the performance evaluation of them in the Section 4. Finally, Section 5 ends with some conclusions and ideas for future work.

2. Previous Work and Objectives

The implementation and optimization of biologically relevant simulation kernels in parallel arquitectures is a very active field. The effort in the last years has been targeted to the exploitation of the Cell Broadband Engine (CBE) and Graphics Processing Units (GPUs) for such objective. Several authors achieved speed-ups up to 200 or 300 times for some variants of non-bonded kernels and in some specific conditions [23]. More general implementations of this kernel for GPUs and CBE reached speed-ups of 260 times [8].

Regarding parallel implementations in Supercomputers, there have been previous efforts to port the FlexScreen program in DEISA environments [4], by means of the middleware UNICORE [1]. At this point we realized that this approach can be drastically improved since the original code of the program was designed for serial/sequential processors. Similar scenario can be found for another VS methods [18]. A better strategy is to isolate the most expensive computing kernels (non-bonded interactions) and to implement them in parallel. Actually we did not find an implementation of full non-bonded interactions kernels for OpenMP/MPI thus we decide to work in this direction and exploit HPC infrastructures to accelerate Virtual Screening calculations. We will study how to obtain an efficient and scalable OpenMP/MPI implementation for this kernel.

3. Data Policies Description

In this section, we describe the data policies for the calculation of the electrostatic potential kernel on the shared and distributed memory systems as well as GPUs.

CUDA [20] programming model is used for GPU architecture, where the global data is visible for all the threads, and a particular model of computation previously described is carried out.

OpenMP [21] programming model is used for the shared memory architectures, in which the shared memory space is uniformly distributed among the n processes. Finally, the distributed programming model MPI [16] is used by our designs on the distributed memory architectures, in which the master process orchestrates the data distribution, so that all the processes (including the master) perform the required calculations.

To exploit all the resources available on the supercomputer, such as vector operations and the multiple cores within a chip, we decide to develop a hybrid MPI-OpenMP implementation that enables vector operations. This implementation somehow emulates the execution of our application in the GPU, enabling two levels of parallelism in a vectorized fashion, to provide a fair comparison between the traditional parallel programming models and CUDA.

3.1. Sequential Baseline

In our study we focus on the particular case of protein-ligand docking, and concretely, in the calculation of the electrostatic potential kernel show in Algorithm 1. This is the baseline for several methodologies used in VS methods, such as Molecular Dynamics and protein-protein docking, just to name a few.

Algorithm 1 The sequential pseudocode.

```
1: for i = 0 to nrec do
2:     for j = 0 to nlig do
3:         calculus(rec[i], lig[j])
4:     end for
5: end for
```

Both receptor and ligand molecules are represented by rec and lig particles, which are specified by their positions and charges, being $nrec$ the number of atoms of rec and $nlig$ the number of atoms of lig.

3.2. Implementation on the GPU

The CUDA programming model is based on a hierarchy of abstraction layers. The **thread** is the basic execution unit that is mapped to a single SP. A **block** is a batch of threads which can cooperate together because they are assigned to the same multiprocessor, and therefore they share all the resources included in this multiprocessor, such as register file and shared memory. A **grid** is composed of several blocks which are equally distributed and scheduled among all multiprocessors. Finally, threads included in a block are divided into batches of 32 threads called **warps**. The warp is the scheduled unit, so the threads of the same block are scheduled in a given multiprocessor warp by warp. The programmer declares the number of blocks, the number of threads per block and their distribution to arrange parallelism given the program constraints (i.e., data and control dependencies), providing two-levels of parallelism [2].

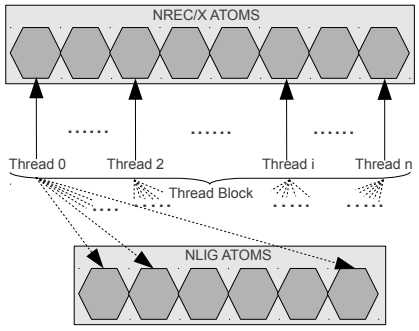


Figure 1. CUDA design for X thread blocks (with $X = 1$) with n threads layout.

Our departure point is a CUDA implementation previously presented in [8]. Figure 1 shows this design. Each atom from the receptor molecule is represented by a single thread. Then, every CUDA thread goes through all the atoms of the ligand molecule. The double parallelism within CUDA is exploited by

1. having as many thread blocks as the number of $nrec$ atoms divided by the number of threads within a block. This number is a configuration parameter of our application.

2. having as many threads as $nrec$ atoms, each thread computes the energy calculations with the entire ligand data.

We also enable a tiling technique to take advantage of the data locality, and thus to increase the memory bandwidth of our application. We group atoms of the ligand molecule in tiles, and thus threads can collaborate in order to bring that information to the *shared memory*. Insights can be found in [8].

3.3. The Shared Memory Implementation

In the shared memory architecture, the energy computation is divided among different processors. Each processor only performs the computation associated with its own part of the receptor data ($nrec$ atoms in Figure 2). Thus, each processor computes the energy interactions between its own private $nrec$ atoms and all atoms from the ligand ($nlig$ atoms in Figure 2). To obtain the final result a reduction of the partial results is performed. Notice that, both $nlig$ and $nrec$ atoms are placed in the same memory space without any data duplications.

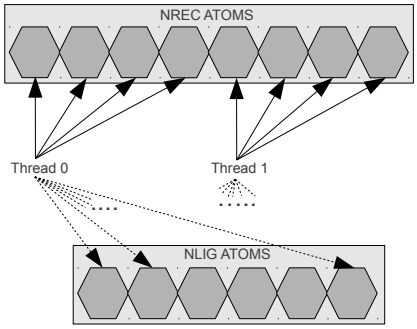


Figure 2. Design for computing the electrostatic interaction kernel with 2 threads in shared memory architectures.

3.4. The Distributed Memory Implementation

In the distributed memory architecture, the underlying design for the energy computation is quite similar than the shared memory one. The $nrec$ data is distributed among all processors, however now, the entire $nlig$ data is sent to all the processors in the cluster which are taking part of the computation.

In order to reduce the communication overhead between nodes in the cluster, all information related with one molecule (positions, charge, energies) is sent all at once in a single packet. In this way, we submit the data of both molecules through two collective sends, instead of submitting many small messages with the information of the molecules which is much more inefficient [25].

The distribution can be done with two MPI instructions: `MPI_Scatter` and `MPI_Bcast`, the first one splits the receptor data among all the MPI processes, and the second one allows MPI processes to share the entire ligand data. Both instructions perform a collective communication which is usually optimized for this architecture, minimizing the communication times [7]. Figure 3 shows an example of this issue.

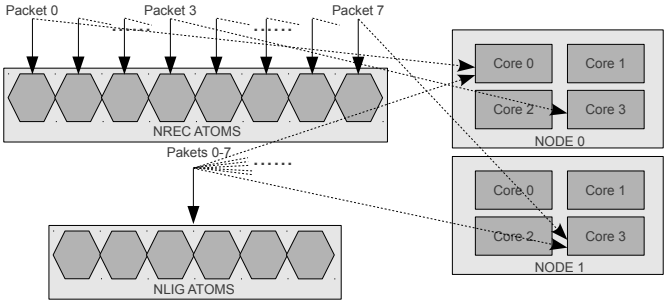


Figure 3. Design for computing the electrostatic interaction kernel with 8 processes in distributed memory architectures.

3.5. Hybrid MPI-OpenMP Implementation

A hybrid solution is implemented using both OpenMP and MPI, which becomes more important on modern multicore parallel systems, decreasing unnecessary communications between processes running on the same node, as well as, decreasing the memory consumption, and improving the load balance. With this implementation, we can emulate the two-levels of parallelism we can find in the CUDA programming model. On one hand, the block-level parallelism is matched by the parallelism between nodes in the cluster (the data will be distributed by MPI). On the other hand, the threads cooperate in parallel to perform the energy calculations within each node in a vectorized fashion like warps in CUDA.

Figure 4 shows the landscape of communications in this implementation. MPI is used to send the data of both molecules to the nodes. instead of sending all the information to each core. The communications are reduced by a ratio of `number of cores per node` with respect to the MPI implementation. Once the data has been distributed by MPI, the calculation of the energy is performed on each node with OpenMP, using its own memory and executing as many threads as the number of cores per node.

Moreover, the communication and computation can be overlapped by asynchronous send/receive instructions. To do so, as soon as a subset of $nlig$ data belonging to each processor is received, the processor starts the energy computation while it is waiting for receive another subset of $nlig$

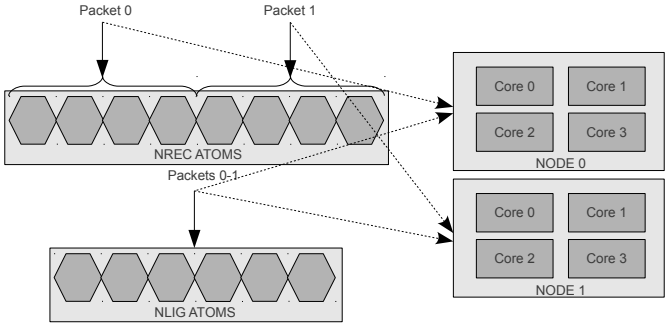


Figure 4. Design for computing the electrostatics interaction kernel in a hybrid MPI-OpenMP implementation with 2 MPI processes and 4 OpenMP threads per node.

data (see Figure 5). For this purpose we use the instructions: `MPI_Isend` and `MPI_Irecv` instead of `MPI_Bcast` to send/receive the ligand data.

The fact of overlapping communication and computation causes an unnecessary overhead whenever the $nlig$ data is received and the process is performing parallel computations with $nrec$. The parallel sections are released as long as a chunk of $nlig$ data is processed. Therefore, we decide takes the parallel sections out of the outer loop where the data packets are received [25].

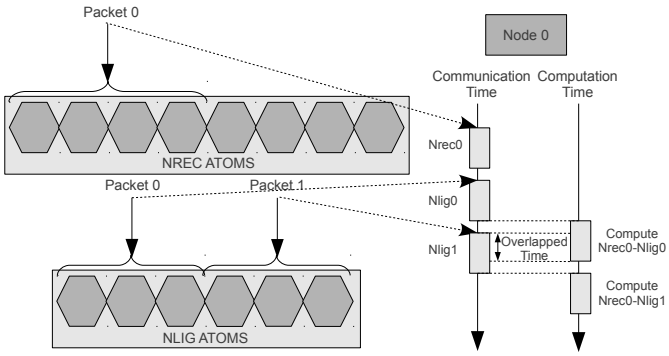


Figure 5. Packet timeline of asynchronous data communications and computations, which are overlapped along the time.

An additional gain of performance can be obtained by taking advantage of the vector instructions to enhance the energy calculation. The $nrec$ atoms are placed in a 128-bytes vector, and each element of $nlig$ is copied four times into another 128-bytes vector. The energy calculation is now vectorized by each processor. The SSE instructions of the x86 processor are used to vectorize the code.

4. Performance Evaluation

This section evaluates our energy interactions kernel implementations in three different platforms. The shared memory platform is a HP Integrity Superdome SX2000. The distributed memory system is a HP BladeSystem. Finally, our GPU-based platform is GPU Nvidia Tesla C2050 based on the Fermi architecture [19]. Hardware and software features are summarized in Table 1 and Table 2.

Table 1. Summary of hardware and software features for the platform used during our experimental survey.

	Shared memory	Distributed memory
Compute Capacity	819 GFlops	9,72 TFlops
Processor Model	Intel Itanium2 Dual-Core Montvale	Intel Xeon Quad-Core E5450
Cache	18 MB	3 MB (L1 32 KB)
Number of nodes	1	102
CPU cores	128	816
Clock Frequency	1,6 GHz	3 GHz
Main memory (DRAM)	1536 GB	1072 GB
Compiler	icc 11.1	Intel MPI 4.0

Table 2. Hardware features for C2050 GPUs.

GPU element	Feature	Tesla C2050
Streaming processors (GPU cores)	Cores per multiprocessor	32
	Number of multiprocessors	14
	Total number of cores	448
	Clock frequency	1 147 MHz
Maximum number of threads	Per multiprocessor	1 536
	Per block	1 024
	Per warp	32
SRAM memory available per multiprocessor	32-bit registers	32 K
	Shared memory	16 KB or 48 KB
	L1 cache	48 KB or 16 KB
	Total SRAM (shared + L1)	64 KB
Global (video) memory	Size	3 GB
	Speed	2x1500 MHz
	Width	384 bits
	Bandwidth	144 GB/sc
	Technology	GDDR5 DRAM

We now present a performance evaluation for all our implementations. They include all associated overheads such as communications, synchronization, load balancing, etc. The performance metric to compare our parallel implementations is the execution time. Our sequential code is the MPI version of a single process. We evaluate the scalability by varying both the $nrec$ and $nlig$ sizes. The maximum cores number used in our test is determined by the number of cores that have the shared memory system, in this case 128 cores. The CUDA implementation is only analyzed for comparison purposes. A deeper analysis can be found in [8].

4.1. The Shared Memory Platforms

Figure 6(a) shows the execution times for the OpenMP implementation. Different overheads introduced by OpenMP such as load balancing and synchronization are quite representative for the smallest benchmarks. Moreover, the scalability problem becomes an issue whenever the value $nrec \geq 2^{15}$ and a many threads are working in parallel, obtaining in some cases a similar execution time with a different number of threads for a specific benchmark configuration. However, we report up to 72x speed-up factor compared to its sequential counterpart.

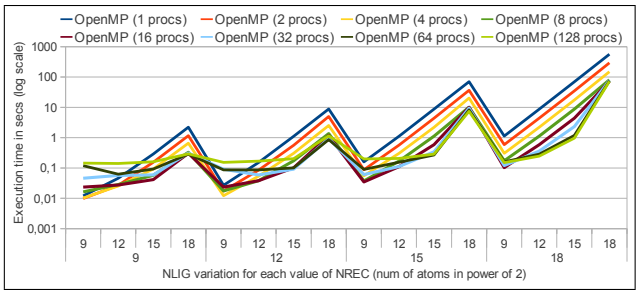
4.2. The Distributed Memory Platform

4.2.1 Only MPI

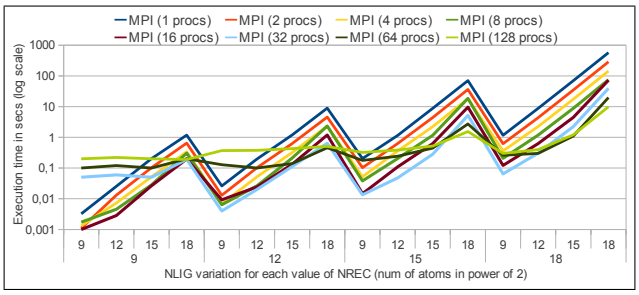
The execution times for the MPI code are shown in Figure 6(b). We also vary the benchmark size to analyze the scalability. In this case it is noteworthy to mention that the scalability improves along with the problem size, even when the value $nrec \geq 2^{15}$. The communication and initialization overheads to send data related to small molecules is actually much higher than the compute time. Moreover, the execution time is drastically affected by the interconnection network status when a large number of processors participate in the execution, producing variations in the communication time which can be hidden by larger computation times. In the MPI case, the maximum speed-up factor obtained is above 60x.

4.2.2 Hybrid MPI-OpenMP Results

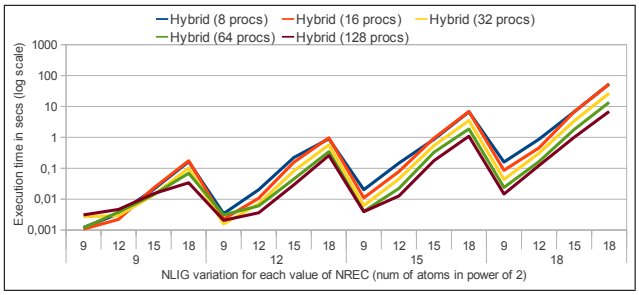
In the hybrid solution, we obtain initially worse results than in the other two implementations because each MPI process (that was mapping to a specific core of each node) was the only responsible to perform the parallel OpenMP computations. This occurs due to conflicting settings between the MPI distribution and the OpenMP runtime. When using hybrid MPI/OpenMP strategy, the OpenMP threads are created as part of the MPI process. If the affinity for the threads is not set explicitly, they all inherit the affinity mask of the process. CPU affinity allows us to specify which CPU each thread should run on. We use the `KMP_AFFINITY` [10] environment variable for the Intel C/C++ compiler to force the threads to be tied down to individual cores. The results of this implementation are shown in Figure 6(c), where the reduction in the number of communications is reflected as previously discussed. In this case, there are eight cores in each node, then the number of communication are reduced



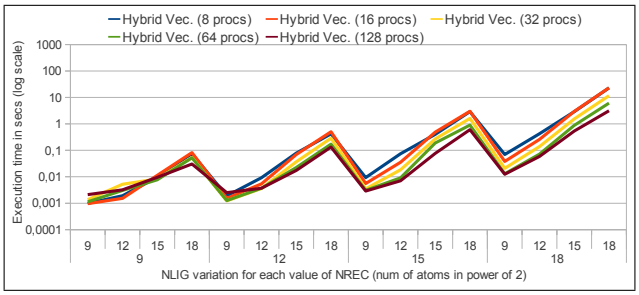
(a) OpenMP implementation.



(b) MPI implementation.



(c) Hybrid MPI-OpenMP implementation.



(d) Hybrid MPI-OpenMP vectorized version implementation.

Figure 6. The execution time obtained for the calculation of the electrostatic potential by the different implementations for different molecular size ratios, in single precision.

Table 3. The execution time obtained for the calculation of the electrostatic potential by different *NLIG* block sizes, in single precision, for 8 MPI processes (64 OpenMP threads).

Nrec	*Nlig*	512	1024	2048	4096	8192	16384	32768	65536	131072	262144
512	**32768**	0,005813	0,005716	0,036385	0,005318	0,005412	**0,004766**	0,006283			
	262144	0,041912	0,041761	0,062274	0,03616	**0,035454**	0,041171	0,043	0,042115	0,046315	0,043294
4096	**32768**	0,016155	0,014836	0,017325	0,016492	0,026334	**0,013769**	0,018485			
	262144	0,130676	0,128422	0,13464	0,187063	**0,126647**	0,176818	0,134821	0,15087	0,139603	0,143179
32768	**32768**	0,126015	0,091419	0,133345	0,108467	0,11098	**0,08681**	0,115749			
	262144	0,780044	0,739505	0,789636	0,766617	**0,724251**	0,765314	0,752469	0,767784	0,749612	0,724349
262144	**32768**	0,667324	0,700655	0,724774	0,734036	0,720468	**0,519282**	0,66324			
	262144	4,986411	4,928454	4,88287	4,896011	4,984927	**4,87261**	4,902244	4,926879	4,929046	4,998905

by a factor of eight. These eight threads are created by generating (1) a MPI process per node, and (2) seven OpenMP processes per node. This give us the total of eight processes running within a node. Figure 6(c) shows the reduction of communications and the use of the OpenMP to perform the energy computation allow in this implementation achieve up to 83x.

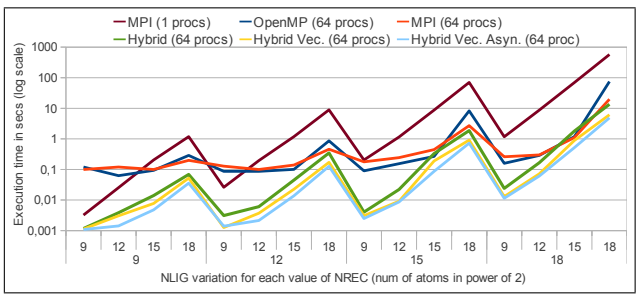
4.2.3 Code Vectorization

In all the previous implementations the Intel compiler auto-vectorize the code. This can be generated by compiling with `-vec-report1` flag [9]. However, we obtain a performance gain of 2.2x respect the Intel auto-vectorize version, given a total of 182x speed-up factor respect the sequential version (see Figure 6(d)).

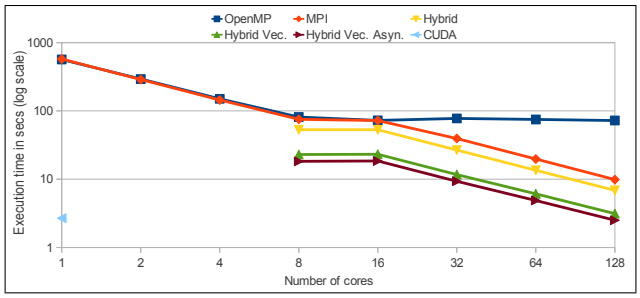
4.2.4 Overlapping Communications/Computations

As previously mentioned the communication overhead can be reduced by an asynchronous computation overlapped with the computation time. We empirically demonstrate that an improvement is achieved whenever large data is sent instead of having smaller package to be sent. This improvement depends on the chunk size of the receptor data.

Figure 7(a) shows the execution times of this technique compared to the rest of versions. This technique is rewarded with up to 1.25x gain. This improvement is not only due to overlap communications and computations, but also to take advantage of the data locality. Whenever a small ligand data



(a) Fixing the number of threads to 64.



(b) Fixing the benchmark size to $nrec = nlig = 262144$ particles.

Figure 7. Comparative times obtained in the different implementations for the calculation of the electrostatics potential.

packet is received, it can be fully stored in the highest level of cache. Table 3 shows the execution times for the asynchronous version with 64 cores by setting different block sizes of $nlig$. It reveals the optimal block size configuration is between 32KB and 64KB which is very close to the L1 cache size (32 KB).

4.3. Overall Comparison

Finally, Figure 7(b) shows an overall comparison between our shared and distributed memory designs and a GPU tiled version. Several conclusion can be extracted from this analysis: (1) The performance of the OpenMP implementation is limited by the parallel overheads whenever more than eight cores are utilized concurrently, (2) Similar execution times are obtained by eight and sixteen cores on the distributed memory architecture. The reason of that is only a node is required to execute a MPI program with eight processes, whereas two nodes are need whenever sixteen MPI processes are involved in the execution. Therefore the communication overhead hides the scalability of using twice the number of nodes, and finally (3), the Figure 7(b) shows the execution time achieved by the GPU defeats almost all implementations developed on the other two architectures, obtaining similar performance with the version that overlaps communication/computation and reduces cache misses. The maximum speed-up factor obtained by this version is 229x while the GPU obtain up to 213x.

5. Conclusions and Future Work

In this article, we have described the implementation for the calculation of non-bonded interactions applied to electrostatics interactions on three different parallel architectures based on shared memory, distributed memory and GPUs. We have also used three different programming models: OpenMP, MPI and CUDA respectively.

Two main levels of parallelism are identified in the CUDA programming model which are matched in the distributed memory by vectorizing the code and providing a hybrid MPI-OpenMP execution. In addition, we optimize this code by overlapping communication/computation and reducing cache misses.

The results obtained in the OpenMP implementation show a reasonable scalability on shared memory architecture, obtaining the lowest performance since the pressure on shared resources increases with the number of processors.

The distributed memory system exhibits good scalability with the number of processors, which is explained by the low number of communications required by our simulations in the hybrid MPI-OpenMP implementation. The hybrid optimized version reaches up to 229x speed-up factor versus its sequential counterpart.

Our previous GPU implementation for the same kernel using CUDA programming model, obtains similar gains versus the sequential code (213x speed-up factor). Therefore, GPUs provides good performance but a much lower cost. Our main conclusion here is GPUs can even outperform a supercomputer for massively parallel computation as the one targeted here. Moreover, the programming effort of optimizing a code in a supercomputer is quite similar to the one employed to do so in GPUs, as long as, the programmer wants to take advantage of all available resources in them. The GPU version of this kernel has been implemented in multiple-target [27] and fast blind VS [26] VS methodologies. In both cases, final global speedups of up to 60x are reached.

For the future, we are working on the implementation of other relevant VS kernels, using the targeted platforms of this work. Our main goal is to provide several high performance alternatives over different computational patterns to evaluate each of them, and thus look for the best solution in terms of performance, power consumption and total cost of ownership.

Acknowledgements

This research was supported by Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, and by the Spanish MEC and European Commission FEDER under grants CSD2006-00046 and TIN2009-14475-C04, and also by a postdoctoral contract from the University of Murcia (30th December 2010 resolution). We would also like to acknowledge the support of the Centro de Supercomputación de la Fundación Parque Científico de Murcia where our experiments were conducted.

References

[1] Uniform Interface to Computing Resources (UNICORE), 2011. http://www.unicore.eu/unicore/.

[2] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P Systems with Active Membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[3] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51:559–572, 2007.

[4] Distributed European Infrastructure for Supercomputing Applications. High Throughput in-silico screening in HPC architectures for new inhibitors for treatment of blood diseases. http://www.deisa.eu/science/deci/projects2010-2011/BLOODINH. (accessed, July, 2011).

[5] B. Fischer, S. Basili, H. Merlitz, and W. Wenzel. Accuracy of binding mode prediction with a cascadic stochastic tunneling method. *Proteins*, Apr. 2007.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008.

[7] J. P. Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2), 2007.

[8] G. Guerrero, H. Pérez-Sánchez, W. Wenzel, J. Cecilia, and J. García. Effective Parallelization of Non-bonded Interactions Kernel for Virtual Screening on GPUs. In *5th International Conference on Practical Applications of Computational Biology and Bioinformatics (PACBB 2011)*, volume 93 of *Advances in Intelligent and Soft Computing*, pages 63–69. Springer Berlin / Heidelberg, 2011.

[9] Intel Corporation. A Guide to Vectorization with Intel C++ Compilers, 2010.

[10] Intel Corporation. Intel(R) Math Kernel Library for Linux* OS User's Guide, 2010.

[11] J. J. Irwin and B. K. Shoichet. ZINC – A Free Database of Commercially Available Compounds for Virtual Screening. *Journal of Chemical Information and Modeling*, 45(1):177–182, 2005.

[12] W. L. Jorgensen. The Many Roles of Computation in Drug Discovery. *Science*, 303:1813–1818, 2004.

[13] K. Kadau, T. C. Germann, and P. S. Lomdahl. Molecular Dynamics Comes of Age: 320 Billion Atom Simulation on BlueGene/L. *International Journal of Modern Physics C*, 17(12):1755–1761, 2006.

[14] S. K. Kuntz, R. C. Murphy, M. T. Niemier, J. Izaguirre, and P. M. Kogge. Petaflop computing for protein folding. In *In Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, pages 12–14.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Ieee Micro*, 28(2):39–55, 2008.

[16] Message Passing Interface Forum. Message Passing Interface (MPI). http://www.mcs.anl.gov/mpi. (accessed, July, 2011).

[17] M. Morgan Kaufmann, Boston. *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*. Morgan Kaufmann, 1 edition, Feb. 2011.

[18] G. M. Morris, D. S. Goodsell, R. Huey, and A. J. Olson. Distributed automated docking of flexible ligands to proteins: Parallel applications of AutoDock 2.4. *Journal of Computer-Aided Molecular Design*, 10(4):293–304, Aug. 1996.

[19] NVIDIA Corporation. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[20] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide 4.0*. 2011.

[21] OpenMP Architecture Review Board. The OpenMP Specification. http://www.openmp.org. (accessed, July, 2011).

[22] H. Pérez-Sánchez and W. Wenzel. Implementation of an effective non-bonded interactions kernel for biomolecular simulations on the Cell processor. In *GI Jahrestagung*, pages 721–729, 2009.

[23] H. Pérez-Sánchez and W. Wenzel. Optimization methods for virtual screening on novel computational architectures. *Curr Comput Aided Drug Des*, 7(1):44–52, 2011.

[24] N. D. Prakhov, A. L. Chernorudskiy, and M. R. Gaiin. VS-Docker: a tool for parallel high-throughput virtual screening using AutoDock on Windows-based computer clusters. *Bioinformatics*, 26(10):1374–1375, 2010.

[25] A. Rane and D. Stanzione. *Experiences in Tuning Performance of Hybrid MPI/OpenMP Applications on Quad-Core Systems*. 2009.

[26] I. Sánchez-Linares, H. Pérez-Sánchez, J. M. Cecilia, and J. M. García. Bindsurf: a fast blind virtual screening methodology on gpus. In *Network Tools and Applications in Biology (NETTAB 2011), Clinical Bioinformatics*, pages 95–97, 2011.

[27] I. Sánchez-Linares, H. Pérez-Sánchez, G. D. Guerrero, J. M. Cecilia, and J. M. García. Accelerating multiple target drug screening on gpus. In *Proce. of the 9th International Conference on Computational Methods in Systems Biology*, CMSB '11, pages 95–102, New York, NY, USA, 2011. ACM.

[28] J. Wang, Y. Deng, and B. Roux. Absolute Binding Free Energy Calculations Using Molecular Dynamics Simulations with Restraining Potentials. *Biophys. J.*, 91(8):2798–2814, Oct. 2006.

[29] Z. Zhou, A. K. Felts, R. A. Friesner, and R. M. Levy. Comparative performance of several flexible docking programs and scoring functions: enrichment studies for a diverse set of pharmaceutically relevant targets. *Journal of Chemical Information and Modeling*, 47(4):1599–1608, 2007.