






Dynamic Load Balancing Algorithm for Heterogeneous Clusters

Tiago Marques do Nascimento, Rodrigo Weber dos Santos ,
and Marcelo Lobosco  

Graduate Program on Computational Modeling, Federal University of Juiz de Fora,
Juiz de Fora, Brazil

tiago.nascimento@uab.ufjf.br, {rodrigo.weber,marcelo.lobosco}@ufjf.edu.br

Abstract. Half of the ten fastest supercomputers in the world use multiprocessors and accelerators. This hybrid environment, also present in personal computers and clusters, imposes new challenges to the programmer that wants to use all the processing power available on the hardware. OpenCL, OpenACC and other standards can help in the task of writing parallel code for heterogeneous platforms. However, some issues are not eliminated by such standards. Since multiprocessors and accelerators are different architectures and for this reason present distinct performance, data parallel applications have to find a data division that distributes the same amount of work to all devices, i.e., they have to finish their work in approximately the same time. This work proposes a dynamic load balancing algorithm that can be used in small-scale heterogeneous environments. A simulator of the Human Immune System (HIS) was used to evaluate the proposed algorithm. The results have shown that the dynamic load balancing algorithm was very effective in its purpose.

Keywords: Load balancing · Heterogeneous cluster · GPUs
Multiprocessors

1 Introduction

Heterogeneous clusters environments are becoming popular parallel platforms. These environments are composed by distinct processors and accelerators, such as GPUs. From a programmer perspective, it is not an easy task to write a parallel program to take advantage of all the computing resources, CPUs¹ and GPUs, present in such environment. This happens not only due to the fact that the computing resources have distinct computational power, but also because of the distinct types of parallelism they were designed to exploit. There are basically two types of parallelism in applications: Data-Level Parallelism (DLP) and Task-Level Parallelism (TLP). The first one arises due to the multiple data

The authors would like to thank UFJF and the Brazilian agencies FAPEMIG, CAPES, and CNPq.

¹ The term CPU in this work refers to multicore processors.

items that must be computed by an application, whereas the second one due to the multiple tasks that must be executed. CPUs were designed to deal with TLP and small amounts of DLP, whereas GPUs were designed to explore large amounts of DLP [6]. Programmers that want to explore DLP in all devices of a heterogeneous platform must take these differences into account, since they can impact the way the code is written and executed.

Some tools, such as those based on OpenCL (Open Computing Language) [15] and OpenACC (Open Accelerators) [1] standards, can help programmers to write code to execute on heterogeneous architectures. Some issues, however, remain open, such as the data division between GPUs and CPUs that balance the amount of work each one will execute. Since GPUs were designed to explore large amounts of DLP, they must receive more data than CPUs to compute, but how much more? Also, depending on the type of instruction that is executed (e.g., float point or integer instruction), the amount of data each computing resource must receive changes. A load balancing (LB) algorithm can help in this task. In this work we use the term LB in the sense of the data division that makes all devices in a heterogeneous cluster composed by CPUs and GPUs finish their computing in approximately the same time.

In previous works [16, 17] we have proposed two distinct algorithms to deal with LB in Accelerated Processing Units (APUs) [3]. APUs merge, in a single silicon chip, the functionality of GPUs with the traditional multicore CPUs. In this paper two new contributions are presented. The first one is the extension done on the dynamic algorithm in order to execute it on a distinct architecture, a heterogeneous cluster. Some modifications in the original algorithm were introduced in order to deal with the new environment. The last one is the evaluation of the impact of the dynamic LB algorithm in performance, using for this purpose the HIS (Human Immune System) simulator.

The remaining of this work is organized as follows. Section 2 presents related works. Section 3 presents a brief overview of OpenCL. OpenCL and MPI were used in the implementation of HIS. In Sect. 4 we present the dynamic LB algorithm. Section 5 presents the performance evaluation. Finally, Sect. 6 presents our conclusions and plans for future works.

2 Related Work

A significant amount of research has been done on heterogeneous computing techniques [14]. Harmony [4] is a runtime supported programming and execution model that uses a data dependency graph to schedule and run independent kernels in parallel heterogeneous architectures. This approach is distinct from ours because we focus on data parallelism, while Harmony focus on task parallelism. Merge [11] is a library system that deals with map-reduce applications on heterogeneous system. Qilin [12] is an API that automatically partitions threads to one CPU and one GPU. SKMD [9] is a framework that transparently distributes the work of a single parallel kernel across CPUs and GPUs. SOCL [7] is an OpenCL implementation that allows users to dynamically dispatch kernels over devices.

StarPU [2] is a task programming library for hybrid architectures that provides support for heterogeneous scheduling. Our approach is distinct because we are not proposing a new library, API, framework or OpenCL implementation, nor we limit the number of CPUs or GPUs that can be used as Qilin does. Also, StarPU does not perform inter-node load-balancing as our approach does. Since the proposed dynamic scheduling approach is implemented in the application code, we do not have to pay the overheads imposed by the frameworks, runtime systems or APIs.

3 OpenCL

OpenCL (Open Computing Language) [15] is an open standard framework that was created by the industry (Khronos Group) in order to help the development of parallel applications in heterogeneous systems. An OpenCL platform includes a single host, which submits work to devices. OpenCL devices, such as CPUs, GPUs, and so on, are divided into compute units, which can further be divided into processing elements (PEs). An OpenCL application consists of two parts, the host program and one or more kernels. PEs execute the kernels, while the host program is executed by the host. The host sends commands to devices through a command-queue. There are three types of commands that can be issued: kernel execution, memory and synchronization commands. The commands issued to a specific queue can be executed in the same order they appear in the command-queue (in-order execution), or can be executed out-of-order. The programmer can use explicit synchronization mechanisms to enforce an order constrain to the execution of commands in a queue. An automatic LB scheme, based on the master-worker parallel pattern [13,15], can be implemented using command-queues, specially those that implements out-of-order execution. However, this parallel pattern is particularly suited for problems based on TLP [13]. In previous works [16,17] we proposed distinct solutions based on an in-order execution for problems based on DLP for an APU architecture.

4 Dynamic Load Balancing Scheme

Heterogeneous computers represent a big challenge to the development of applications that explore DLP. The use of all distinct PEs available to simultaneously operate in all data items is not easy due to the distinct hardware characteristics. In fact, heterogeneous computing on CPUs and GPUs using architectures like CUDA [8] has fixed the roles for each device: GPUs have been used to handle data parallel work while CPUs handle all the rest. The use of this fixed role has impacts on performance, since CPUs are idle while GPUs are handling the data parallel work. Actually CPUs could handle part of the work submitted to the GPU. In this scenario, OpenCL [15] represents an interesting alternative, since it is easy to program parallel codes that use all devices to operate in data items. The point is that the programmer is responsible for the data division between CPUs and GPUs. A good data division would give to each PE a distinct amount

of data proportional to its relative performance. So if device A is 1.5 times faster than device B , it should receive 1.5 times more data to compute than device B .

In previous works we have presented two distinct LB algorithms [16, 17] to be used with data parallel OpenCL codes running on an APU. The key idea behind the two algorithms is similar: data is split into two parts, one of which will be computed by the CPU, while the other one will be computed by the GPU. The amount of data that will be assigned to the CPU and GPU depends on their relative computing capabilities, which is measured in both LB algorithms during the execution of the application.

This work further extends our previous LB algorithms to be used in a different hardware platform: a heterogeneous cluster. Since an APU merges GPUs and CPUs cores on a single silicon chip, some modifications have to be done in the algorithm to deal with multiple GPUs and CPUs available in distinct nodes of a cluster. Also, the algorithm does not assume that all machines in a cluster have the same configuration, i.e., the same number and types of CPUs and GPUs.

The dynamic LB algorithm can be used in a wide variety of applications that explore DLP. Usually these applications have at least two aligned loops, in which the inner loop performs the same operations on distinct data items, as Algorithm 1 shows. Each step of the inner loop (or a collection of loops, if a multidimensional data structure is used) could be executed in any order, since no data dependency occurs between two distinct loop iterations. The number of steps the outer loop iterates is determined by the nature of the problem, but usually a dependency exists between two consecutive steps: a new step cannot proceed without the result of a previous one, since their results will be used during the computation of the new step. In many applications the outer loop is related to the progress of a simulation over time, and for this reason will be referred in this work as time-steps. The dynamic LB algorithm will decide the amount of data each PE will receive to compute in the inner loop.

During the computation of each data item, some applications require also access to its neighbors data, which can be located at distinct memory spaces due to data splitting between CPUs and GPUs. These data, called boundaries, must be updated between two consecutive iteration of the outer loop. This update requires the introduction of synchronization operations and the explicit copy of data. In the case of a heterogeneous cluster, this copy may occur inside a machine (e.g., copying data between two distinct GPUs or between the memory space of a CPU and a GPU, and vice-versa) or between machines, which imposes the use of communication primitives. Both data copy and synchronization operations are expensive, deteriorating performance, and for this reason should be avoided.

The dynamic LB algorithm is presented in Algorithm 2 and works as follows. For a single time-step, all GPUs and CPUs receive an equal amount of data to compute (data size divided by the total number of PEs) and the time required to compute them is recorded. This information is then used to compute the relative computing power of each PE and consequently determine the amount of data it will receive for the next time-steps. Equation 1 is used for this purpose.

```

1 for all time-steps do
2   for each data item do
3     | call cpus/gpus devices to compute a piece of data;
4   end
5   send/receive boundaries;
6   synchronize devices;
7 end

```

Algorithm 1. Data parallel algorithm

```

1 initialize MPI and OpenCL;
2 allocate memory in each device's memory space;
3 divide data equally among all devices;
4 start clock;
5 for a single time-step do
6   | call cpus/gpus to compute their data;
7   | synchronize;
8 end
9 finish clock;
10 compute  $P_i^{(t)}$  and transfer data accordingly;
11 for all remaining time-steps do
12   if time-step % LB interval == 0 then
13     | start clock;
14     | call cpus/gpus to compute their data;
15     | synchronize;
16     | finish clock;
17     | compute  $P_i^{(t)}$ ;
18     if  $|P_i^{(t)} - P_i^{(t-1)}| > LB\ threshold$  then
19       | transfer data accordingly;
20       | synchronize devices;
21     end
22     else
23       |  $P_i^{(t)} = P_i^{(t-1)}$  (keeps data distribution);
24     end
25   end
26   else
27     | call cpus/gpus to compute interior points and transfer border points in
28     | parallel;
29     | synchronize;
30     | call cpus/gpus to compute border points;
31     | synchronize;
32 end

```

Algorithm 2. The dynamic LB algorithm

$$P_i^{(t)} = \frac{P_i^{(t-1)} \times T_r^{(t-1)}}{T_i^{(t-1)} \times \sum_{k=1}^n \frac{P_k^{(t-1)} \times T_r^{(t-1)}}{T_k^{(t-1)}}}, \quad (1)$$

where $P_i^{(t)}$ is the percentage of data the PE i will receive to compute in the next time-step, $P_i^{(t-1)}$ is the percentage of data the PE i received in the previous time-step, $T_i^{(t-1)}$ is the time in which PE i executed the previous time-step, $T_r^{(t-1)}$ is the time in which an arbitrary reference PE r executed the previous time-step and k is the total number of PEs available in the heterogeneous cluster. In the first time-step ($t = 0$), the percentage of data each PE will receive to compute is divided equally among all PEs.

After the computation of the amount of data each PE will compute in the next time-step, memory should be reallocated and data copied from its last owner to the new one. However, in order to avoid memory reallocations, the dynamic LB algorithm allocates, at each PE, an additional amount of memory to avoid memory reallocations, and only data copies are required.

After the first time-step has finished, the LB algorithm will be executed from time to time to adjust the amount of data each PE will receive till the end of the computation. This occur because some applications exhibit an irregular behavior during computation, while other applications that seems to be regular parallel applications, such as the one that will be used in the performance evaluation, suffer from irregular execution time phases during their execution. This happens due to hardware optimizations done in the CPU, which would impact a static LB algorithm, i.e., an algorithm that keeps the percentage found in the first time-step until the end of computation [18].

The LB step is a time consuming task, specially due to data transfers between PEs located in distinct machines. If the change in the amount of data each PE must compute is minimal, the eventual performance gain is not compensated by the overhead of moving data. So a parameter, called *LB threshold*, was added to avoid this situation. If the difference between $P_i^{(t)}$ and $P_i^{(t-1)}$ is lower than this threshold, the PEs remain with their previous loads until another LB step is reached.

A final optimization is done in order to reduce the communication costs. Each PE divides its data into two subsets: borders and interior points. The border points are composed by the points that must be exchanged with the neighbors, whereas the interior points are not exchanged. The PE compute first the border points. While computing the interior points, the PE exchange borders with its neighbors, so computation and communication overlap.

5 Performance Evaluation

This section evaluates the performance of the LB algorithm presented in this work using for this purpose a simulator of the HIS [19,20]. This simulator was

chosen because it is a representative of data parallel algorithm: the same set of operations must be executed on a large amount of data.

All tests were executed on a small cluster composed by 3 machines. The machines have two AMD 6272 processors (each machine has 32 cores), 32 GB of main memory, two Tesla M2075 GPUs, each one with 448 CUDA cores and 6 GB of global memory. Linux 2.6.32, OpenMPI version 1.6.2 and gcc version 4.4.7 were used to run and compile all codes. The machines are connected by a Gigabit Ethernet network. Although the AMD machines have a total of 96 cores, one Float-Point Unit (FPU) is shared by two cores, so only 48 FPUs are available in the machines.

5.1 Benchmark

A three dimensional simulator of the HIS [19, 20] was used to evaluate the performance of the two load-balancing algorithms. The simulator implements a mathematical model that uses a set of eight Partial Differential Equations (PDEs) to describe how some cells and molecules involved in the innate immune response, such as neutrophils, macrophages, protein granules, pro- and anti-inflammatory cytokines, react to an antigen, which is represented by lipopolysaccharides. The diffusion of some cells and molecules are described by the mathematical model, as well as the process of chemotaxis. Chemotaxis is the movement of immune cells in response to chemical stimuli by pro-inflammatory cytokine. Neutrophils and macrophages move towards the gradient of pro-inflammatory cytokine concentration. A detailed discussion about the model can be found in [19, 20].

The numerical method used in the computational implementation of the mathematical model was the Finite Difference Method [10], a method commonly used in the discretization of PDEs. The computation of the convective term (the chemotaxis term) is a complex part in the resolution of the PDEs. Our implementation is based on the finite difference method for the spatial discretization and the explicit Euler method for the time evolution. First-Order Upwind scheme [5] is used in the discretization of the chemotaxis term. More details about the numerical implementation, specially how the Laplace operator, that simulates the diffusion phenomenon, is implemented in 3D, can be found in a previous work [20]. This previous work used C and CUDA in the implementation, using only GPUs in the computation, while this work uses C and OpenCL, using all resources (CPUs and GPUs) available in the cluster.

There are two ways to divide the data mesh: division by planes and division by individual elements. The division by individual elements allows the algorithm to use of a fine-grain data partition in the LB. In a previous work [18], we have found that the division by individual elements performs better and, for this reason, this division will be used in this work. A mesh of size $50 \times 50 \times 3200$ was used in the experiments. The values used to set the initial conditions and parameters of HIS are the same used in our previous work [16]. A total of 10,000 time-steps were executed. The LB interval is equal to 10% of the time-steps and the LB threshold is equal to 50 elements.

Three versions of the HIS were executed: a sequential one, a version that used the dynamic LB algorithm and one that did not use LB. In the version that did not use the LB, the mesh size was divided equally among all PEs that were used to execute the code. Each HIS version was executed at least 3 times, and all standard deviations of the execution time were below 1%.

5.2 Results

Table 1 presents the results. As one can observe, the sequential version of the code executes in more than 36 h. A typical simulation requires 1,000,000 time-steps, which represents more than 151 days of computation. The parallel version of the simulator that does not use the LB algorithm executes up to 435 times faster. But the dynamic LB algorithm improved the performance even more: using the same configuration, the application executed 916 times faster than the sequential one and 2.1 times faster than the version that does not use the LB algorithm.

Table 1. Experimental results for the parallel version of the code in a small cluster. Average execution time(s) and gains relative to the version without LB and the sequential one.

Platform	w/o LB	LB	Gain
32 CPUs + 2 GPUs	531.3	283.1	1.9
64 CPUs + 4 GPUs	308.3	182.1	1.7
96 CPUs + 6 GPUs	300.5	142.7	2.1
Sequential	130,694.33	916	

Table 2 presents the HIS parallel execution time in a single machine, considering the use of each computational resource at a time, as well as using all of them simultaneously, with and without the use of the LB algorithm proposed in this work. The best result obtained with the use of a single computational

Table 2. Using all resources types available in a single machine \times using one at a time. Times in seconds. Gains compared to the version that uses 2 GPUs to execute the code.

Platform	Execution time	Gain
32 CPUs	1,688	-
1 GPU	627	-
2 GPUs	317	-
32 CPUs + 2 GPUs (w/o LB)	531.3	0.6
32 CPUs + 2 GPUs (LB)	283.1	1.12

resource was 317s, when 2 GPUs are used to execute the code. The simultaneous use of all resources does not guarantee a performance gain: if all CPUs and GPUs are included in the computation, the parallel execution time increases to 531 s. However, the same configuration can obtain a performance gain if the LB algorithm is used: the execution time reduces to 283 s.

6 Conclusions and Future Works

This paper presented the implementation of a dynamic LB algorithm in a heterogeneous cluster environment. Its key idea is to split data items of an application that explore DLP into multiple parts that will be computed simultaneously by CPUs and GPUs. The amount of data that will be assigned to CPUs and GPUs depends on their relative computing capabilities, which is measured and updated during all the execution of the application.

A performance evaluation of the dynamic LB algorithm was executed, using for this purpose the Human Immune System simulator. The results have shown that the algorithm was very effective in its purpose, resulting in gains up to 916-fold in execution time compared to the sequential one. Compared to the version that did not use the LB, the gains in performance were 2.1 times. We have also shown that performance gains could only be obtained using all resources in a single machine if the LB algorithm was used.

As future works, we plan: (a) to measure the overheads imposed by the algorithm, specially the time spent with communication due to a new data division; (b) to develop a static version of the algorithm, and compare it to the dynamic one; (c) to evaluate the proposed LB algorithm using other benchmarks; and (d) evaluate the impacts of the algorithm in the scalability of applications.

References

1. The OpenACC application programming interface - version 2.5. Technical report, OpenAcc.org (2015)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.: Pract. Exp.* **23**(2), 187–198 (2011)
3. Branover, A., Foley, D., Steinman, M.: AMD fusion APU: Llano. *IEEE Micro* **32**(2), 28–37 (2012)
4. Damos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC 2008*, pp. 197–200. ACM, New York(2008)
5. Hafez, M.M., Chattot, J.J.: *Innovative Methods for Numerical Solution of Partial Differential Equations*. World Scientific Publishing Company, Singapore (2002)
6. Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)

7. Henry, S., Denis, A., Barthou, D., Counilh, M.-C., Namyst, R.: Toward OpenCL automatic multi-device support. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 776–787. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_65
8. Kirk, D.B., Wen-Mei, W.H.: Programming Massively Parallel Processors: A Hands-on Approach, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
9. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT 2013, pp. 245–256. IEEE Press, Piscataway (2013)
10. LeVeque, R.: Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics). Society for Industrial and Applied Mathematics, Philadelphia (2007)
11. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 287–296. ACM, New York (2008)
12. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multi-processors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM, New York (2009)
13. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Boston (2004)
14. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.* **47**(4), 69:1–69:35 (2015)
15. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide, 1st edn. Addison-Wesley Professional, Boston (2011)
16. do Nascimento, T.M., de Oliveira, J.M., Xavier, M.P., Pigozzo, A.B., dos Santos, R.W., Lobosco, M.: On the use of multiple heterogeneous devices to speedup the execution of a computational model of the human immune system. *Appl. Math. Comput.* **267**, 304–313 (2015)
17. do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: On a dynamic scheduling approach to execute OpenCL jobs on APUs. In: Osthoff, C., Navaux, P.O.A., Barrios Hernandez, C.J., Silva Dias, P.L. (eds.) CARLA 2015. CCIS, vol. 565, pp. 118–128. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26928-3_9
18. do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: Performance evaluation of two load balancing algorithms on a hybrid parallel architecture. In: Malyshev, V. (ed.) PaCT 2017. LNCS, vol. 10421, pp. 58–69. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62932-2_5
19. Pigozzo, A.B., Macedo, G.C., Santos, R.W., Lobosco, M.: On the computational modeling of the innate immune system. *BMC Bioinform.* **14**(6), S7 (2013)
20. Rocha, P.A.F., Xavier, M.P., Pigozzo, A.B., de M. Quintela, B., Macedo, G.C., dos Santos, R.W., Lobosco, M.: A three-dimensional computational model of the innate immune system. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012. LNCS, vol. 7333, pp. 691–706. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31125-3_52