A Dynamic Load-Balancing Algorithm for Heterogeneous GPU Clusters

Luis Sant'Ana, *CMCC-UFABC* Daniel Cordeiro, *DCC-USP* and Raphael Camargo, *CMCC-UFABC*

***Abstract*—The use of GPU clusters for scientific applications is becoming more widespread, with applications in areas such as physics, chemistry and bioinformatics. As the number of different GPU models increases, these clusters may become more heterogeneous, with different types of GPUs and CPUs. To use these machines in an efficient manner, it is necessary to perform load-balancing among the GPUs and CPUs, minimizing the execution time of the application. We propose an algorithm for dynamic load balancing in heterogeneous GPU cluster. We implemented the algorithm in the StarPU framework and compared with existing load-balancing algorithms.**

***Keywords*—parallel computing, distribuited systems, GPU cluster, GPGPU.**

I. Introduction

The use of GPUs (Graphics Processing Units) has become popular among developers of high-performance applications that can benefit from a large degree of parallelism [1]. Modern GPUs can have thousands of simple floating point units (FPUs) that, combined, can be several times faster than traditional CPUs. The development of efficient GPU applications is, however, more difficult mainly due to difference on the amount of available cache memory and on the control logic of the application.

Many applications already use the processing power of GPUs. For example, applications in fluid mechanics [2], visualization science [3], machine learning [4], bioinformatics [5] and neural networks [6]. However, many complex problems require the usage of multiple GPUs hosted on different machines (GPU clusters [7], [8]).

The development of distributed application running on a GPU cluster must also take care of the management of the multiple memory spaces of all GPUs on all nodes of the cluster. This memory spaces are memories that have different characteristics like, speed access and capacity.

This includes transferring data between these spaces of memories and ensure the consistency of data. A combination of CUDA (Compute Unified Device Architecture) and MPI (Message Passing Interface) is a common choice to develop applications for GPU clusters. There are several efforts by the scientific community for the creation of new programming models [9], [10] for the development of efficient applications on clusters GPUs [11], [12], [13].

GPU clusters are still typically *homogeneous*. Homogeneity facilitates the development of applications, since they can be optimized just for a single architecture and is more easy to achieve a good load balance among GPUs. But homogeneity is difficult to maintain in a context where a new generation of hardware is launched every couple of years.

Developing a load-balancing mechanism that works efficiently for all kinds of applications is difficult. With two or more GPUs (or even CPUs), this problem is strictly equivalent to the classic problem of minimizing the maximum completion time of all tasks (makespan), which is known to be NP-hard [14]. An efficient load-balancing scheme must be considered on case by case basis.

For example, there are several studies on data-parallel applications for that can be divided using domain decomposition [15] on GPUs. In this case, the data can be easily distributed among the available GPUs. The main task of the load-balancing mechanism is to devise the best data division amont the GPUs. Several scientific applications fit into this group, including applications in bioinformatics [5], neural networks [6], chemistry, physics and materials science.

However, with heterogeneous groups of GPUs, this distribution is more difficult. There are currently several types of GPUs with different architectures, such as *Tesla*, *Fermi*, *Kepler* and *Maxwell*. These architectures have different organizations of FPUs (Floating-point Unit), caches, shared memory and memory speeds. A division of the load based on simple heuristics, such as a the number of cores in the GPU, is not effective and can be worse than a simple division [7]. Also, different architectures require different low-level optimizations and a code can be better optimized for one architecture than the other. Finally, GPUs clusters normally have high-end CPUs in addition to the GPUs, and these CPUs may be used by the applications; for instance, to execute parts of the code which cannot be coded efficiently using the Single Instruction Multiple Thread (SIMT) model of GPUs.

Another frequent approach is to devise execution time profiles for each GPU type and application task and use it to determine the amount of work given to each GPU. This profiling can be done statically [7] or dynamically [16], [17]. Another solution is to use simple algorithm for task dispatching, such as the greedy algorithm from StarPU [13], where tasks are dispatched to the devices as soon as they become available.

In this work we propose a novel adaptive algorithm for dynamic load balancing of data-parallel applications in heterogeneous GPU clusters. Based on run-time measures, the

Luis Sant'Ana is with the Center Mathematics, Computation and Cognition, Federal Uniersity of ABC, Santo André, SP, Brazil e-mail: luis.ana@ufabc.edu.br

Daniel Cordeiro is with the Department of Computer Science, University of São Paulo, São Paulo, SP, Brazil e-mail: danielc@ime.usp.br

Raphael Carmargo is with the Center Mathematics, Computation and Cognition, Federal Uniersity of ABC, Santo André, SP, Brazil e-mail: raphael.camargo@ufabc.edu.br

algorithm creates a execution model to dynamically adjust the size of data blocks depending on the characteristics of the processing units, both CPUs and GPUs. We have compared the algorithm with a static approach, which greedily distributes the tasks to every available device, with a dynamic approach and with the HDSS [17] algorithm.

II. Related Work

A common solution for load balancing in distributed systems is to divide the tasks according to a weight factor representing the processing speed of each processor. Early approaches used fixed weight factors determined at compile time with limited success [18]. But these weight factors are difficult to determine in heterogeneous GPUs, due to the architectural differences that affect the performance.

The problem of load balancing in heterogeneous GPUs, began to be studied recently. Acosta *et al.* [16] proposed a dynamic load balancing algorithm that interactively try finds a good distribution of work between GPUs during the execution of the application. A library monitors differences during run-time in each processor. It is a decentralized scheme in which synchronizations are done at each iteration to determine whether there is a need to rebalance the load. Each processor performs a redistribution of workload according to the size of the assigned task and the capabilities of the assigned processor. Load balancing is obtained by comparing the execution time of the current task for each processor with the redistribution of the subsequent task. It this time difference reaches a threshold, a load redistribution is triggered. Our algorithm is different because is centralized and it creates curves for each processing unit, and based on these curves performs the load balancing.

A static algorithm that determines the distribution before starting the execution of the application, using the static profiles from previous executions, was also proposed [7]. The algorithm was evaluated using a large-scale neural network simulation. The algorithm finds the distribution of data that minimizes the execution time of the application, based on profiles from previous executions, ensuring that all GPUs to spend same amount of time performing the processing of kernels. This approach is static and does not allow dynamic changes in the data distribution.

The Heterogeneous Dynamic Self-Scheduler (HDSS) [17] provides a dynamic load-balancing algorithm, divided in two phases. The first is the adaptive phase, where it determines weights that reflect the speed of each processor. These weights are determined based on fits of logarithmic curves on processing speed graphs. These weights are used during the completion phase, where it divides the remaining iterations among the GPUs based on the relative weights of each GPU. The main differences to our algorithm is that we do not restrict the execution models to logarithm curves, which are appropriate for GPUs but not to CPUs in the range of interest. Also, our algorithm can perform adjustments until the end of the execution.

III. Proposed Algorithm

In a typical data-parallel application, the application data is divided among the threads in a process called domain decomposition [15]. The threads then simultaneously process their part of the data. After finishing, the threads merge the processed results, and the application terminates or continues to the next phase of computing. The task of our load-balancing algorithm is determining the size of the data block assigned to each GPU and CPU in the system. We will use the term processor to mean a single CPU or GPU.

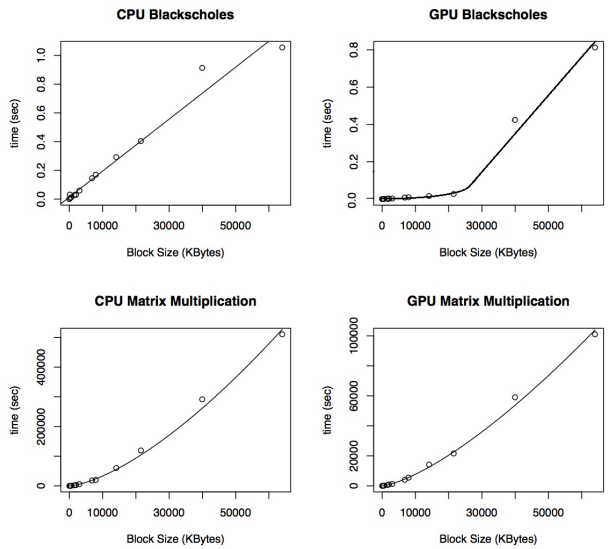

Fig. 1. GPU and CPU processing times for different block sizes.

Overview: The algorithm has three parts, which are: (1) processor performance model, where an execution model for each processor is determined during application execution; (2) optimal block size selection, where, based on the performance model, the algorithm selects the best distribution of block size among the processors; and (3) re-balancing, where the algorithm recalculates the optimal block sizes when the difference in execution time by different processors is larger than a threshold.

Processor performance model: We initially determine a function $P_p[x]$, which provides the processing speed for a block of data of size x on processor p. To construct these curves, a block of size $initialBlockSize$, defined by the user, is allocated to each processor. We select the processor with the earliest finish time t_f and double the block size for this processor. For other processors, with finish times t_i, we select blocks of size proportional to the ratio t_f/t_i.

Figure 1 shows sample processing time measurements for a GPU and a CPU for different block sizes. We can see that the curves can be approximated by linear functions. We find best fit curves by the method of *least squares*, using a function of the form $f(x) = ax - b$. The curve is fitted after generating a few points, for example, four, becoming a model for the processor execution times. The curve can be taken from the time that the determination coefficient is greater than 0.7.

The Figure 2 shows a schematic simplified of the execution algorithm . Boxes are numbers indicating the order of

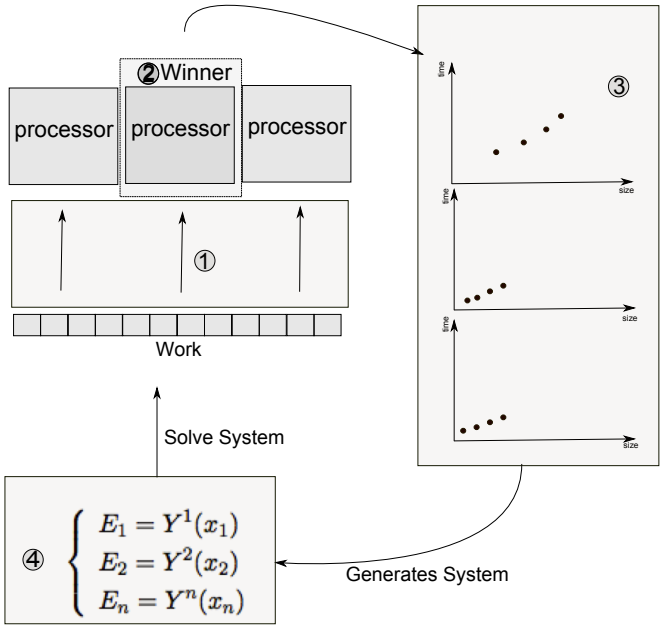

Fig. 2. Summary of the algorithm

execution of the algorithm. At number 1 the work is equally divided among the processors that make up the cluster. This step is elected faster machine indicated by the number 2, it receives o double of size block initial, while the other machines receive size block proportional to their performance in previous step. The values time are recorded for each processor in 3, generating a curve of the block size as a function of time. These curves are approximated by functions, the most common functions. For each curve, a function is generated, which comprise a system of equations showed in 4. Using a search method, a solution is found, with the restriction time. The time requires to be the same for all processors. The result is block of optimum size. If the cluster undergoes a change in behavior, and the time difference among the processors exceeds a threshold, the process and the calculation is redone.

Algorithm 1 Processor performance model

```
function determineModel()
blockSizeList ← initialBlockSize;
while fitValues.error ≥ 0.5 do
    finishTimes = executeTasks(blockSizeList);
    synchronize();
    blockSizeList = evaluateNextBlockSizes(finishTimes);
    fitValues = determineCurveProcessor();
end while
return fitValues;
```

These steps are shown in Algorithm 1. Variable $blockListSize$ contains the size of the blocks assigned to each processor and is initialized with $initialBlockSize$, which is defined by the user. Variable $fitValues$ contains results from the last least square fitting, including the $error$ in the fitting, which is initialized as $+\infty$.

In the main loop, while the error is larger than a predefined value, the function send a chunk of data for each processor and obtain the finish time for each processor. After waiting for all processor to finish, it determine the block sizes for the next iteration, based on their finish times. Finally, it tries to fit model curves to each processor and receives the fitting results.

Optimal block size selection: Our algorithm determines the optimal block size for each processor with the objective of minimizing the total time of the application. Consider we have n processors and a input data size Z. The algorithm assigns a data chunk of size x^g for each processor $g = 1, ..., n$, corresponding a fraction of input data, such that $\sum_{g=1}^{n} x^g = Z$. We denote as $E^g(x^g)$ the execution time of task E in the processor g, for input of size x^g. To distribute the work among the processors, we find a set of values

$$X = \{x^g \in \mathbb{R} : [0, 1] / \sum_{g=1}^{n} x^g = Z\} \tag{1}$$

that minimizes $E_1(x_1)$ while satisfying the constraint

$$E_1 = E_2 = ... = E_n \tag{2}$$

which represents that all processors should spend the same amount of time performing the processing. To determine the set of values x, we solve the system of fitted curves for all processors, given by:

$$\begin{cases} E_1 = f(x_1) \\ E_2 = f(x_2) \\ E_n = f(x_n) \end{cases} \tag{3}$$

The equations system is solved applying the interior point line search filter method [19]. The algorithm seeks to minimize the functions in a search space. Determining the value of x such that the time is minimal.

Rebalancing: After solving the system of equations the scheduler keeps sending tasks of size x_i for each processor i, as soon as the processor finishes the previous task. It also monitors the finish time of each task. If the difference in finishing times x_i and x_j of any two tasks i and j goes above a threshold, the balancing process is re-executed. In this case, the algorithm applies the processor performance model and optimal block size selection routines again. The scheduler then synchronizes the tasks and start using the new x_i values. The threshold needs to be determined empirically through some tests. If the threshold is a value too small, happen unnecessary balancing, which will increase the total time of the application. If the threshold is too large imbalances may arise which will cause threads are idle.

Figure 3 shows a diagram of execution of four threads and the unbalance of a thread. The boxes represent the threshold, the lines represent the threads and the circle represents the unbalance. The threads start synchronized, receive load, but in the second step one thread takes longer than the threshold value, this causes an unbalance. Detected unbalance, all threads

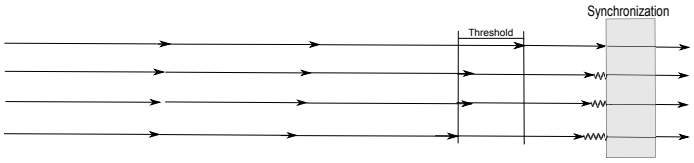

Fig. 3. Threads execution with threshold

are synchronized, and the partitions are recalculated for each thread, causing the threads back to stay synchronized.

Complete algorithm: Algorithm 2 shows the pseudo-code of the scheduling algorithm. The $determineModel$ function, shown in Algorithm 1 returns the performance model for each processor. It then solve the system of equations to determine the best distribution X of chunk sizes for each processor.

Algorithm 2 Complete dynamic algorithm

```
function dynamic()
fitValues = determineModel()
X = solveEquationSystem(fitValues);
while there is data do
    finishTimes = executeTasks(X);
    if maxDifference(finishTimes)≥ threshold then
        fitValues = determineCurveProcessor();
        X = solveEquationSystem(fitValues);
        synchronize();
    end if
end while
```

The main loop repeats while there is still data for processing. It distributed chunks of data for each processor in the system, obtaining the finish times for each task execution. It then checks if the maximum difference between the finish tasks is above a threshold, obtained empirically. If threshold is reached, the algorithm fits new model curves and solve the system of equations for these new curves to determine a new distribution of chunk sizes.

In the figure 4 we compare the behavior in relation to submission of blocks among machines. It analyzed the block size for the HDSS and our algorithm. In this case three machines were used, it is possible to notice the difference between runs. Our algorithm in step 400 suffered a re-balancing, causing it to change the size of the block. In HDSS, a decrease of the block size remains over the iterations.

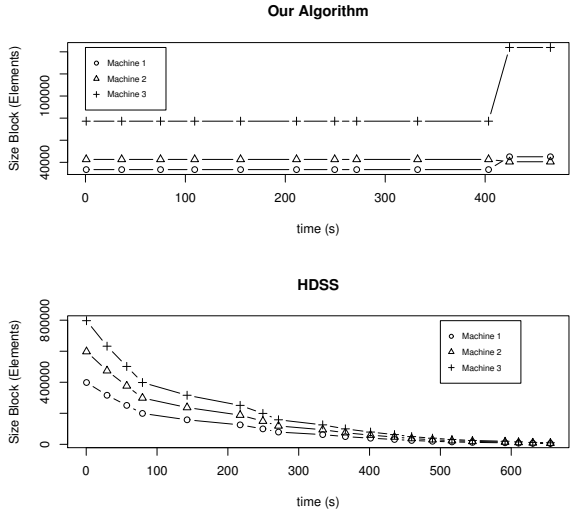

Fig. 4. Difference in runtime with different numbers of genes for gene regulatory network inference

IV. Implementation

The implementation was done in the C language with the framework StarPU. StarPU [13] is a tool for parallel programming that supports hybrid architectures like multicore CPUs and accelerators. The StarPU proposes an approach of independent tasks based architecture. Codelets are defined as an abstraction of a task that can be performed on one core of a multicore CPU or subjected to an accelerator. Each codelet may have multiple implementations, one for each architecture in which codelet can be performed using specific languages and libraries for the target architecture. A StarPU application is described as a set of codelets with data dependencies.

The tool has a set of scheduling policies implemented that the programmer can choose according to the characteristics of the application. The main one is the use of static scheduling algorithm HEFT (Heterogeneous Earliest Finish Time) to schedule tasks based on cost models of task execution.

For each device one codelet has been programmed with the characteristics of the devices. A codelet is a structure that represents a computational kernel. Such a codelet may contain an implementation of the same kernel on different architectures (e.g. CUDA and x86). The applications were implemented by dividing the data set into tasks, implemented as codelets. The tasks are independent, with each task receiving a part of the input set proportional to the processor weight. Two codelets were implemented one for GPU/CUDA and one for CPU architecture.

To evaluate our load-balancing algorithm, we implemented it by modify the default StarPU balancing algorithm. The modification of the load balancing algorithm is realized by changing the STARPU_SCHED variable. The STARPU framework has an API that allows modifying the scheduling policies. There are data structures and functions that speed up the process of development. For example the function ”double starpu_timing_now (void)” return the current date in micro seconds, which makes it easier for the determination of measures runtime. In StarPU there is a data structure called ”starpu_sched_policy” This structure contains all the methods que Implement a scheduling policy.

Three other algorithms were implemented for comparison: the greedy, static and HDSS. The greedy consisted in dividing the input set in pieces and assigning each piece of input to any idle processor, without any priority assignment. The static [7], measures processing speeds before the execution and set static

block sizes per processor at the beginning of the execution, with the block size proportional to the processor speed. Finally, The HDSS [17] was implemented using minimum square estimation to estimate the weights and divided into two phases: adaptation phase and completion phase.

The library used for solve the equation system like 3 was the IPOPT [19]. IPOPT (Interior Point Optimize) is an open source software package for large-scale nonlinear optimization. It can be used to solve general nonlinear programming problems.

A. Applications

For evaluating our algorithm, we adapted two applications from the CUDA SDK [20] to execute using the StarPU framework, the blackscholes and matrix multiplication applications. And we adapted a application to gene regulatory networks (GRN) inference [21].

A copy of the matrix A was distributed to all processing units and matrix B was divided according to the load-balancing scheme, the matrix multiplication version use the shared memory. Matrix multiplication has complexity $O(n^{3/2})$.

Blackscholes is a popular financial analysis algorithm for calculating prices for European style options. The Black-Scholes equation is a differential equation that describes how, under a certain set of assumptions, the value of an option changes as the price of the underlying asset changes. More precisely, it is a stochastic differential equation that includes a random walk term, which models the random fluctuation of the price of the underlying asset over time. The Black-Scholes equation implies that the value of a European call option.

The cumulative normal distribution function, gives us the probability that a normally distributed random variable will have a value less than x. There is no closed-form expression for this function, and as such it must be evaluated numerically. It is typically approximated using a polynomial function. The idea is to calculate the Black-Scholes the greatest amount of possible options. Thus, the input is a vector of data that the options should be calculated by applying a differential equation. The division of the task is given a position of providing input vector to each thread. The complexity of the algorithm is $O(n)$.

Gene regulatory networks (GRN) inference is an important bioinformatics problem in which the gene interactions need to be deduced from gene expression data, such as $microarray$ data. Feature selection methods can be applied to this problem. A feature selection technique is composed by two parts: a search algorithm and a criterion function. Among the search algorithms already proposed, there is the exhaustive search where the best feature subset is returned, although its computational complexity is unfeasible in almost all situations. The objective of work is the development of a low cost parallel solution based on GPU architectures for exhaustive search with a viable cost-benefit. The complexity of the algorithm is $O(n^3)$. The division of labor consisted in distributing genes between processors, and certain steps to synchronize the information search in the solution space, more details of the algorithm in [21].

V. Results

A. System Configuration

We used three different machines to evaluate our algorithm, presented in table **??**. We performed the experiments with four settings, with one machine (A), with two machines (A, B), with three machines (A, B, C) and four machines (A, B, C and D). The machine A has a CPU with 4 cores and two GPUs with 280 cores. The machine B has 1 CPU with 6 cores and two GPUs with 1536 cores. The machine C has 1 CPU with 6 cores and 1 GPU with 2688 cores and finally the machine D has 1 CPU with 14 cores and 2 GPUs with X cores.

To use all the n multiprocessors from a GPU, it is necessary to create at least n blocks. Moreover, each multiprocessor simultaneously executes groups (called warps) of m threads from a single block, and several warps should be present on each GPU for efficient usage of its cores.

In all tests, we used all the SM(Stream Multiprocessor) of the GPUs, launching kernels with k blocks per with 1024 threads for each block, where k is the number of processor in the GPU. For the used GPUs, k is **14, 8 and 30** in GTX Titan, GTX 680 and GTX 295 respectively. For the CPUs, we used all the CPU cores, launching one task per core.

The parameters used for each algorithm were obtained via experiments. A lot of values were tested. The best parameters were obtained specifically for each application.

B. Matrix Multiplication

We evaluated the execution time of the matrix multiplication application using one, two, three and four machines and four different scheduling algorithms: (1) our algorithm, (2) static, (3) HDSS, and (4) Greedy.

Figure 5 shows the results for matrices with sizes from 4096 x 4096 to 65536 x 65536 elements. In all scenarios, our scheduling algorithms obtained the best results, with the HDSS algorithm in second. The static and greedy algorithm were clearly inferior.

With one machine the difference was smaller because there are few types of devices for the scheduler to select. With two machines our algorithm starts to perform better, especially for larger matrices, which is explained by the fact that the execution time of the matrix multiplication increases quickly as we increase the matrix size. As in the other scenarios, increasing the matrix size also increases the performance gain with our algorithm. For matrices 4096 elements and four machines, the proposed algorithm spent 23.11 seconds while the HDSS spent 29.91, which results in 22.73% faster. And for matrices 65536, the algorithm proposed spent 383.28 seconds while the HDSS 443.37, which results in 15.67% faster, the summary is table II. The total cost of calculating the distribution of the blocks is 664.87 milli seconds and standard deviation of 98.3 mili seconds.

The results obtained shows that in more heterogeneous environment the advantage using our proposed algorithm is largest than the other. This advantage is due to a better distribution of the data along the execution. Figure 6 shows the time difference between the earliest and latest finishing threads, in the scenario with three machines.

TABLE I. CONFIGURATION OF THE MACHINES

Machine	Description						
A	**CPU Info**	intel i7 a20	4 cores	2.67 GHz	8192 MB cache	8 GB RAM	1 processor
	GPU Info	GTX 295	280 cores	223.8 GB/s Memory Bandwidth	896 MB	999 MHz Memory Clock	2 processors
B	**CPU Info**	intel i7 4930K	6 cores	3.4 GHz	12,288 KB	32 GB RAM	1 processor
	GPU Info	GTX 680	1536 cores	192.2 GB/s Memory Bandwidth	2 GB	6 GHz Memory Clock	2 processors
C	**CPU Info**	intel i7 3939K	6 cores	3.2 GHz	12,288 KB cache	32 GB RAM	1 processor
	GPU Info	GTX Titan	2688 cores	223.8 GB/s Memory Bandwidth	6 GB	6 GHz Memory Clock	1 processor
D	**CPU Info**	intel Xeon E5-2695V3	14 cores	2.3 GHz	35 MB	32 GB RAM	1 processor
	GPU Info	?	?	?	?	?	

TABLE II. COMPARATIVE: HDSS X OUR ALGORITHM

	4096 elements			65536 elements		
Num. Machines	HDSS (s)	Our algorithm (s)	Diff. (%)	HDSS (s)	Our Algorithm (s)	Diff. (%)
1	233	231	0.86	696	689	1.01
2	129	128	0.78	587	502	16.93
3	36	28	28.57	543	497	9.25
4	29	23	26.09	488	428	14.02

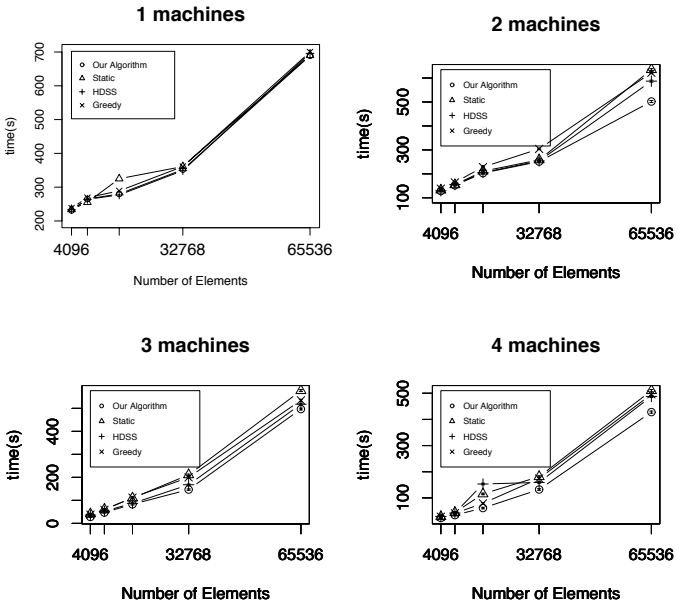

Fig. 5. Difference in runtime with different sizes of matrices for matrices multiplication

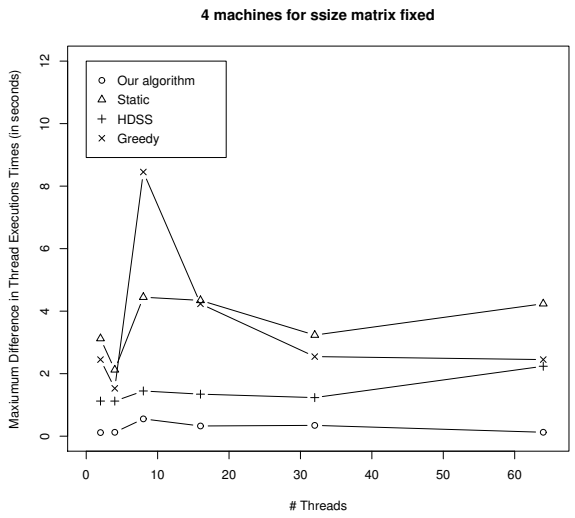

Fig. 6. Time differences between the earliest and latest finishing threads

The greedy algorithm performed well considering that it do not directly use information on the processing speeds of the devices. But it uses this information indirectly, since faster devices finish their tasks earlier and, consequently, receive more tasks. The static algorithm had the worst performance, because it leaves the thread idle for a long time see 6. There are big differences among end of threads.

HDSS uses the beginning of the execution to estimate the best block size distribution and uses this distribution through the complete execution. Moreover, larger blocks are used in the beginning of the execution, causing a larger delay due to dis-balance when use different types of devices, such as GPUs and CPUs. Finally, HDSS uses a crude approximation to the device capabilities.

Our algorithm estimate fairly accurately the amount of data that should be provided to each processing unit, solving an optimization problem with the restriction that all units should finish the execution of the tasks at the same time. When the difference between finishing the execution of threads for certain data partition exceeds a certain threshold, the algorithm re-balances the data distribution.

C. Blackscholes

We evaluate the execution time of the Blackscholes application using different machine configurations. Figure 7 shows the execution times, where we varied the number of options on each execution and obtained the runtimes. Similarly to the matrix multiplication experiments, the performance gains were the largest with bigger problem sizes (number of options)

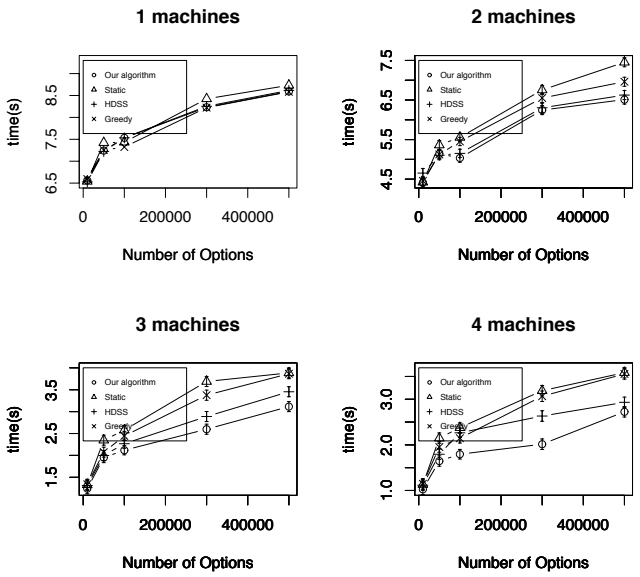

Fig. 7. Difference in runtime with different number options

and more heterogeneous environments, and the results can be explained using the same arguments. Interestingly, the Blackscholes application has linear complexity with the number of options, showing that our scheduling algorithm is also useful with this class of application. The table III shows the extreme values of 10,000 and 500,000 options, the best result was with 4 machines, and with the greatest number of options.

Also, considering that the application finishes in less than 4 seconds, for the scenario with 4 machines, we also see that the cost of solving the optimization problem to determine the best data distribution is small and the obtained gains outweighs the cost of the calculations. For applications tested in a few iterations around 6, it was possible to obtain the solution of the system in a few milliseconds. The total cost of calculating the distribution of the blocks is 201.32 milli seconds and standard deviation of 0.34 mili seconds

Figure 8 confirms this result, showing that the time difference among the earliest and latest finishing threads, in the scenario with four machines is always smaller for our proposed algorithm.

D. Gene regulatory networks inference

As in previous applications, we compare the execution time in four different environments: with one machine, two machines, three machines and four machines. The results are shown in 9. We can notice that our algorithm outperformed in four cases. Especially in the case that three and four machines were used.

The table IV shows the extreme values, for 100,000 and 2,000,000 genes. To the environment with 3 and 4 machines the performance were similar due to limitation in data transfer, in the case of four machines the communication cost limited

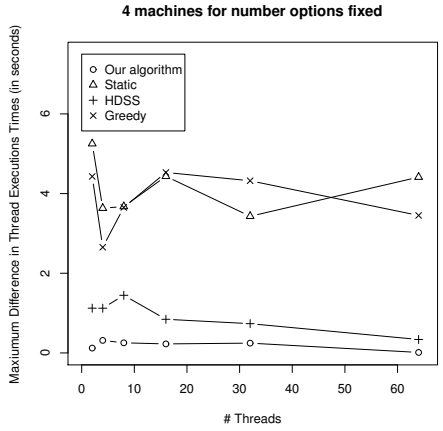

Fig. 8. Time differences between the earliest and latest finishing threads

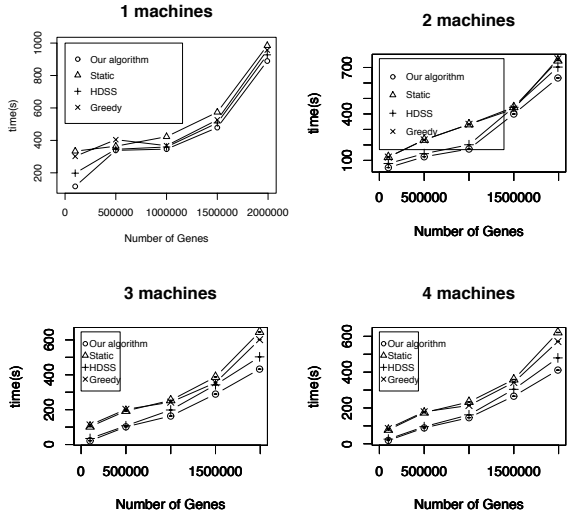

Fig. 9. Difference in runtime with different numbers of genes for gene regulatory network inference

the gain. The total cost of calculating the distribution of the blocks is 821.54 milli seconds and standard deviation of 142.23 mili seconds.

Like the other experiments, the maximum difference among the threads followed the same principle. Our algorithm has the smallest difference among the threads, figure 10.

VI. Conclusions and Future Work

In this paper we propose an algorithm for scheduling tasks in domain decomposition problems executing on clusters of heterogeneous CPUs and GPUs. It outperforms other similar existing algorithms, due to the online estimation of the performance curve for each processing unit and the selection of the best data distribution for the devices. We showed for two

TABLE III. COMPARATIVE: HDSS X OUR ALGORITHM

	10,000 Options			500,000 Options		
Num. Machines	HDSS (s)	Our algorithm (s)	Diff. (s)	HDSS (%)	Our algorithm (s)	Diff. (%)
1	6.52	6.56	-0.61	8.61	8.59	0.23
2	4.65	4.41	5.44	6.62	6.51	1.68
3	1.31	1.24	5.56	3.45	3.11	10.93
4	1.11	1.02	8.82	2.93	2.72	7.72

TABLE IV. COMPARATIVE: HDSS X OUR ALGORITHM

	100,000 Genes			2,000,000 Genes		
Num. Machines	HDSS (s)	Algorithm (s)	Diff. (%)	HDSS (s)	Algorithm (s)	Diff. (%)
1	197.94	116.43	70.00	927.43	889.13	4.31
2	79.76	53.43	49.28	702.65	632.73	11.05
3	34.76	21.39	62.50	502.65	432.43	16.24
4	25.91	18.54	39.75	479.54	411.46	16.54

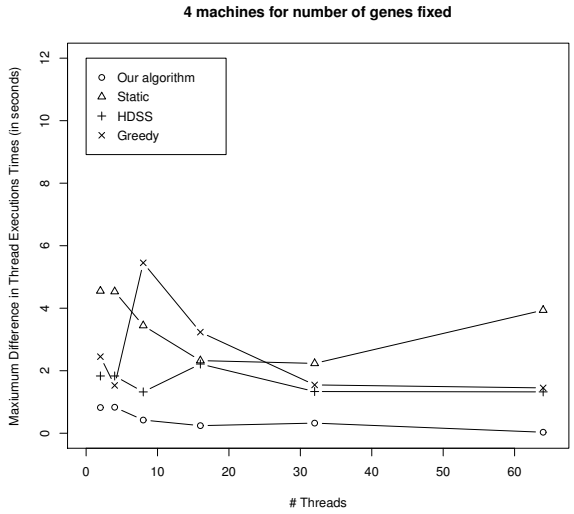

Fig. 10. Difference in runtime with different numbers of genes for gene regulatory network inference

applications that our algorithm provides the highest gains for more heterogeneous and larger problems sizes.

Although we used dedicated clusters, we can also consider the usage of public clouds, where the user can request a number of resources allocated in virtual machines from shared machines. In this case, the quality of service may change during execution, and the addition of the execution time difference threshold permits readjustments in data distributions. We can also consider the scenario with added fault-tolerance, where machines may become unavailable during execution. In this scenario, a simple redistribution of the data among the remaining devices would permit the application to readapt to this scenario.

As ongoing work, we are including the cost of communication in the scheduling algorithm, which is essential for applications where the time spent with information exchange among the tasks cannot be ignored.

ACKNOWLEDGMENT

The authors would like to thank FAPESP (Proc. n. 2012/03778-0 and Proc. n. 2013/14603-9) for the financial support.

REFERENCES

[1] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: ACM, 2014, pp. 1:1–1:8.

[2] G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva, "Direct numerical simulation of incompressible flows on unstructured meshes using hybrid cpu/gpu supercomputers," *Procedia Engineering*, vol. 61, no. 0, pp. 87 – 93, 2013, 25th International Conference on Parallel Computational Fluid Dynamics.

[3] H. K. Raghavan and S. S. Vadhiyar, "Efficient asynchronous executions of amr computations and visualization on a gpu system," *Journal of Parallel and Distributed Computing*, vol. 73, no. 6, pp. 866 – 875, 2013.

[4] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "Parallel multitask cross validation for support vector machine using gpu," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 293 – 302, 2013, models and Algorithms for High-Performance Distributed Data Mining.

[5] Y. Jeon, E. Jung, H. Min, E.-Y. Chung, and S. Yoon, "Gpu-based acceleration of an rna tertiary structure prediction algorithm," *Computers in Biology and Medicine*, vol. 43, no. 8, pp. 1011 – 1022, 2013.

[6] J. Liu and L. Guo, "Implementation of neural network backpropagation in cuda," in *Intelligence Computation and Evolutionary Computation*, ser. Advances in Intelligent Systems and Computing, Z. Du, Ed. Springer Berlin Heidelberg, 2013, vol. 180, pp. 1021–1027.

[7] R. de Camargo, "A load distribution algorithm based on profiling for heterogeneous gpu clusters," in *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, 2012, pp. 1–6.

[8] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 47–.

[9] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/gpu nodes," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10.

[10] L. Wang, W. Jia, X. Chi, Y. Wu, W. Gao, and L.-W. Wang, "Large scale plane wave pseudopotential density functional theory calculations on gpu clusters," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–10.

[11] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352.

[12] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, "Flat: a gpu programming framework to provide embedded mpi," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 20–29.

[13] C. Augonnet, S. Thibault, and R. Namyst, "Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines," Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, Rapport de recherche RR-7240, Mar 2010.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[15] W. D. Gropp, "Parallel computing and domain decomposition," in *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[16] A. Acosta, V. Blanco, and F. Almeida, "Towards the dynamic load balancing on heterogeneous multi-gpu systems." in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, 2012, pp. 646–653.

[17] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 57:1–57:20, 2013.

[18] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 318–328.

[19] J. Nocedal, A. Wächter, and R. Waltz, "Adaptive barrier update strategies for nonlinear interior methods," *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1674–1693, 2009. [Online]. Available: http://dx.doi.org/10.1137/060649513

[20] CUDA Development Core Team, "CUDA 4.1 Programming Guide," Santa Clara CA, EUA, 2012. [Online]. Available: http://www.nvidia.com/cuda

[21] F. F. Borelli, R. Y. de Camargo, D. C. Martins Jr, and L. C. Rozante, "Gene regulatory networks inference using a multi-gpu exhaustive search algorithm," *BMC bioinformatics*, vol. 14, no. 18, pp. 1–12, 2013.