PLB-HeC: A Profile-based Load-Balancing Algorithm for Heterogeneous CPU-GPU Clusters

Luis Sant'Ana and Raphael Camargo
Center for Mathematics, Computation and Cognition
Federal University of ABC
Santo André, Brazil
{luis.ana,raphael.camargo}@ufabc.edu.br

Daniel Cordeiro
Department of Computer Science
University of São Paulo
São Paulo, Brazil
danielc@ime.usp.br

***Abstract*—The use of GPU clusters for scientific applications in areas such as physics, chemistry and bioinformatics is becoming more widespread. These clusters frequently have different types of processing devices, such as CPUs and GPUs, which can themselves be heterogeneous. To use these devices in an efficient manner, it is crucial to find the right amount of work for each processor that balances the computational load among them. This problem is not only NP-hard on its essence, but also tricky due to the variety of architectures of those devices.**

We present PLB-HeC, a Profile-based Load-Balancing algorithm for Heterogeneous CPU-GPU Clusters that performs an online estimation of performance curve models for each GPU and CPU processor. Its main difference to existing algorithms is the generation of a non-linear system of equations representing the models and its solution using a interior point method, improving the accuracy of block distribution among processing units. We implemented the algorithm in the StarPU framework and compared its performance with existing load-balancing algorithms using applications from linear algebra, stock markets and bioinformatics. We show that it reduces the application execution times in almost all scenarios, when using heterogeneous clusters with two or more machine configurations.

Keywords*—*GPU clusters; GPGPU; load-balancing; parallel computing;

I. Introduction

The use of GPUs (Graphics Processing Units) is becoming increasingly popular among developers and HPC practitioners. Modern GPUs are composed of thousands of simple floating point units (FPUs) that, combined, can provide a computational power several times superior to traditional CPUs. For increased parallelism, multiple GPUs are usually offered on GPU clusters [1], with the GPUs distributed on different machines. The use of GPUs can benefit applications that have high computational demands and large degrees of parallelism [2]. Among these applications, we can include fluid mechanics [3], visualization science [4], machine learning [5], bioinformatics [6] and neural networks [7].

The main drawback is the difficulty in optimizing GPU code, specially considering the architectural differences among different GPUs — they may differ on the distribution of cores, shared memory, presence of cache, etc. There is a clear effort (by both academy and industry) to ease the development of optimized code on such hardware [8], [9]. The development of applications for GPU clusters is even more complex, since it requires the management of the multiple memory spaces, one for each GPU in the cluster, in addition to the main memory of the each node. This management includes transferring data between these memory spaces and ensuring the consistency of data. There are several efforts by the scientific community for the creation of new programming models [10], [11] and frameworks [12], [13], [14] to simplify the development of GPU cluster applications. However, the combination of CUDA (Compute Unified Device Architecture) and MPI (Message Passing Interface) is still the standard choice to develop applications for GPU clusters.

High-performance GPU clusters are typically *homogeneous*, containing nodes and GPUs with the same configuration. Homogeneity facilitates the development of applications, since they can be optimized just for a single architecture.

Homogeneity, however, can be hard to maintain in the current scenario; a new generation of hardware is launched every couple of years. It is also difficult to be guaranteed on distributed cooperative platforms, composed of commodity hardware from different research groups (in a grid-like fashion). Thus, combining heterogeneous machines can increase the available computational power to cluster users.

Developing a load-balancing mechanism that works efficiently for all kinds of applications is difficult. With two or more GPUs (or 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 [15]. An efficient load-balancing scheme must be considered on case-by-case basis. For example, there are several types of data-parallel applications [16] that could be divided using domain decomposition. Several scientific applications fit into this group, including applications in bioinformatics [6], neural networks [7], chemistry, physics and materials science.

When using homogeneous clusters, data from a data-parallel application can be distributed among the available GPUs using tasks of the same size, i.e., that take approximately the same amount of time to be executed. With heterogeneous GPUs, however, this distribution is more difficult. At the time of writing, the major GPU vendor offers GPU processors with more than four different microarchitectures: *Tesla*, *Fermi*, *Kepler* and *Maxwell*. These architectures have different organizations of FPUs (Floating-point Unit), caches, shared memory and memory speeds. Even for GPUs with the same architecture, characteristics can vary considerably among models. These differences increase the complexity of determining the division of tasks among the available GPUs that will result in the best performance. Finally, GPU clusters normally have high-end CPUs in addition to the GPUs, and these CPUs can used by the applications.

The main task of the load-balancing mechanism is to devise the best data division among the GPUs. A division of the load based on simple heuristics, such as the number of cores in the GPU, may be ineffective and could result in worse performance if compared to using a simple homogeneous division [17]. A possible approach is to determine the performance 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 statically computed, before the execution of the application [17], or dynamically computed, at runtime [18], [19].

Another solution is to use simple algorithms for task dispatching — such as the greedy algorithm used by several frameworks including StarPU [14] — where tasks are dispatched to the devices as soon as they become available. Such algorithms use simple (but fast) heuristics to determine the distribution of tasks among the processors, but can result on suboptimal distributions. A more elaborate and precise load-balancing algorithm causes a higher overhead, but can potentially compensated by a better task distribution.

In this work we present PLB-HeC, a novel profile-based load-balancing algorithm for data-parallel applications in heterogeneous CPU-GPU clusters. The algorithm uses performance information gathered at runtime in order to devise a performance model customized for each device (CPU or GPU). Differently from other existing algorithms, the key idea is generate and solve a non-linear system of equations representing the performance model for each device, and then use a interior point method that can generate a better task distribution among processors. The algorithm is implemented inside the StarPU framework, easing its use both on legacy applications and the new ones. We compared the proposed algorithm with Acosta [18] and dynamic (HDSS) [19] algorithms and with the standard StarPU greedy algorithm.

II. Related Work

The standard approach for load balancing in distributed systems is to divide the tasks among CPUs 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 [20]. The main problem is that these weight factors are difficult to determine, specially in heterogeneous scenarios.

The problem of load balancing in heterogeneous GPUs, began to be studied recently. One proposal was the usage of a static algorithm that determines the distribution before the execution of the application, using profiles from previous executions [17]. The algorithm uses these profiles to find the distribution of data that minimizes the execution time of the application, ensuring that all GPUs to spend same amount of time performing the processing of kernels. The algorithm was evaluated using a large-scale neural network simulation. Its main drawback is that since it is static, an initial unbalanced distribution cannot be adjusted in runtime. Another problem is that it requires previous executions of the applications in the target devices to determine its execution profiles. Finally, it does not consider the case where application behavior changes with the parameters. A dynamic algorithm, like the one that we propose, does not have these limitations.

The problem was also studied using classic Scheduling Theory. On hybrid architectures with m homogeneous CPUs and k homogeneous (i.e., the heterogeneity comes only from the different types of processors available) the problem is already NP-hard. Bleuse *et al.* [21] proposed an approximation algorithm which achieved a ratio of $\frac{4}{3} + \frac{1}{3k} + \epsilon$ using dual approximation with a dynamic programming scheme. They have proved that the algorithm takes $O(n^2k^3m^2)$ per step of the dual approximation to schedule n tasks. Besides their performance guarantees, the running time of such solutions is often dominated by the cost of scheduling itself.

In [22], the authors use the concept of logical processors to model GPU-CPU hybrid systems, where the processor represents an independent group of tightly coupled devices such as cores on the same socket or a GPU and its host core. The authors propose a way to measure the impact of having multiple cores sharing the same computational resources and show that using this information in the load-balancing process can reduce the total application execution time.

In [23] the authors proposed two algorithms. The first, called *naive algorithm*, is based on an online profiling scheme. The algorithm executes in two phases - profiling phase and execution phase. In the profiling phase the algorithm determines the distribution of items among devices based on the processing rate Gr of GPUs and Cr of CPUs. These rates are used for data distribution in the execution phase. The second algorithm, called *asymmetric algorithm*, reduces the overhead of initial phase by sharing a pool of work between CPU and GPU that avoids the need of synchronization. The overhead of the execution phase is reduced by repeating the profiling until a certain termination condition is reached or the convergence to a certain size of the job is achieved. Similarly to other models, its performance model is based on single numbers, which limits the accuracy of the block distribution. Also, it does not allow further improvements in the load-balancing during the execution phase.

In [24], the authors propose a framework, based on Charm++, called G-Charm. It schedules chare objects among CPUs and GPUs at runtime, based on the current loads and the estimated execution times, which depend on the average time taken by the task in previous time steps. The memory layer of G-Charm keeps track of chare buffers in order to reduce memory transfer, but the scheduling strategy, however, seems to not consider data transfer costs.

Acosta *et al.* [18] proposed a dynamic load balancing algorithm that interactively searches for a good distribution of work among the available GPUs. It uses a decentralized scheme in which synchronizations are made at each iteration to determine if there is need to rebalance the load. The idea is to use a shared vector where each processor records the time they spent on the last task. If the difference between these times is larger than a threshold defined by user, the algorithm computes a vector called RP (Relative Power), which relates the time spent to process a certain load versus the load processed. The processors then calculate the SRP (Sum Relative Power), which is the sum of all RP vectors, and their next attributed load. The disadvantage of this algorithm is that the convergence of the load balance is asymptotic, as the definition of the load on each processor is based on a simple weighted average of the measured relative power (RP) in the last iteration. This may cause suboptimal load distributing during several iterations, resulting in poor performance and the need for several rebalancing processes. PLB-HeC generates a curve of execution times for each processor and is able to determine the distribution of the load faster and more accurately, preventing unnecessary rebalancing.

The Heterogeneous Dynamic Self-Scheduler (HDSS) [19] is a dynamic load-balancing algorithm for heterogeneous GPU clusters. It is divided in two phases. The first is the adaptive phase, where it determines weights that reflect the speed of each GPU — similarly to Hummel *et al.* [20] — but performed dynamically and using profiling data. A performance curve with the FLOPs per second for each task size is created for each GPU. The weights are determined based on logarithmic fits on the curves. These weights are used during the second phase, called completion phase, where it divides the remaining iterations among the GPUs based on their relative weights. It starts allocating larger block sizes to the GPUs, and decrease their size as the execution progresses. The weight model based on logarithmic curves is more suitable for GPUs, where the number of FLOPs per second stabilizes with larger tasks. A drawback of HDSS is that using of a single number to model each processor can limit the accuracy of the load-balancing. Moreover, once determined, these weights are not changed throughout the execution. Our proposed algorithm uses a curve for modeling each processor, resulting in better block size distributions. Finally, our algorithm performs a progressive refinement of the performance models for the processors during execution, allowing the balancing during execution.

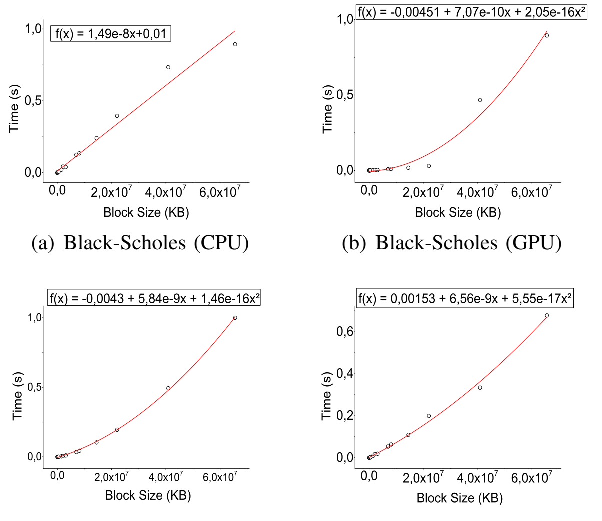
III. Proposed Algorithm

In a typical data-parallel application, data is divided in blocks, which can be concurrently processed by multiple threads, in a process called domain decomposition [16]. After finishing, the threads merge the processed results and the application proceeds to its next phase.

The task of our load-balancing algorithm is determining the size of data blocks assigned to threads located on each GPU and CPU in the system. We consider that application data can be decomposed into small data blocks, allowing the algorithm to find a near optimal distribution of block sizes. We will use the term “processing units” to refer to both CPUs and GPUs.

A. Overview

The proposed algorithm is composed of three phases: (i) processing unit performance modeling, where a performance model for each processing unit is devised during application execution, based on the tasks’ execution and data transfer times; (ii) block size selection, where the algorithm determines the best distribution of block size among the processing units, based on the computed performance model; and (iii) execution and rebalancing, where the algorithm provides the processing units with blocks of the appropriate size until the end of application execution, or until the difference between the execution



(a) Black-Scholes (CPU) (b) Black-Scholes (GPU)

(c) Matrix multiplication (CPU) (d) Matrix multiplication (GPU)

Fig. 1. Execution times and performance models for the GPU and CPU implementations of the Black-Scholes and matrix multiplication applications.

times of different processing units becomes larger than a given threshold. In this case, it generates a new performance model and recalculates the block sizes for each processing unit.

B. Processing unit performance modeling

In this phase, the algorithm devises a performance model for each processing unit based on execution time measurements. The algorithm constructs two functions $F_p[x]$ and $G_p[x]$. $F_p[x]$ represents the time a processing unit p spends processing a block of size x. $G_p[x]$r represents the time spent to send a block of size x to processing unit p. In order to generate these functions, there is a training phase, where the algorithm assigns blocks of different sizes for each processing unit and records the time spent to process and transfer them. For each processing unit, we interpolate a curve that best represents the measured times, allowing the extrapolation of execution times for other block sizes.

The curve is initially fitted using four points, representing four different block sizes allocated to each processing unit. The first block has size $initialBlockSize$, defined by the user. For the second point, the size of the block is doubled and then adjusted to the performance of the processing unit, based on the execution of the first block. The processing unit with the earliest finish time, t_f, receives a block of size $2 * initialBlockSize$. The other processing units, with finish times t_k, receive blocks of size $2 * initialBlockSize * t_f/t_k$. For the third block, the multiplier is changed from 2 to 4 and in the fourth one, we used a multiplier of 8. This process is illustrated in the upper part of Figure 2, which shows a schematic view of the complete load-balancing algorithm.

The above procedure has the advantage that in the first step of the performance modeling phase, a performance preview (t_f/t_k) of the processing units is already obtained, using a small block size. The use of this preview to select the following block sizes, which increase exponentially to cover a large spectrum of block sizes, can significantly reduce the amount

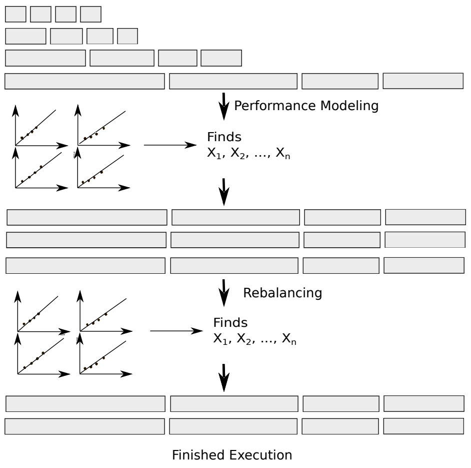


Fig. 2. Load-balancing algorithm, with the performance modeling, block size selection and execution phases.

of idle processing unit times during the performance modeling phase.

The algorithm then follows to next step, where it searches for a curve that fits the existing points using the least squares fitting method for the functions $F_p[x]$ and $G_p[x]$, representing the processing and data transfer times, respectively. If the coefficient of determination of the least squares method is greater than 0.7 for all processing units, the algorithm and finishes this phase. Otherwise, it generates more points in the curve, until reaching a coefficient larger than 0.7 or when 20% of the application data is processed. A value of 0.7 provides a good approximation for the curve and prevents overfitting.

To determine the execution time model ($F_p[x]$), we find the best fit curves using the *least squares* method, using a function of the form:

$$F_p[x] = a_1 f_1(x) + a_2 f_2(x) + ... + a_n f_n(x) \qquad (1)$$

where $f_i(x)$ is one function of the set $\ln x$, x, x^2, x^3, e^x, x and the combinations $x \cdot e^x$ and $x \cdot \ln x$. This set should contemplate the vast majority of applications, but other functions can be included if necessary. Figure 1 shows sample processing time measurements for a GPU and a CPU for different block sizes on two different applications. We can see that the curves can be approximated by different types of functions.

For the $G_p[x]$ function, we used an equation of the form:

$$G_p[x] = a_1 x + a_2 \qquad (2)$$

where the linear coefficient a_1 represents the network and PCIe bandwidths, and a_2 the network and system latencies. These values are also adjusted from profiling data using the *least squares* method, capturing all transfer overheads. We consider that the data transfer delay increases linearly with data size, which is a valid approximation for compute-bound applications.

The pseudocode for this phase is shown in Algorithm 1. This code is executed in a single node, called *master*

Algorithm 1 Processing units performance model

```
function determineModel()
$blockSizeList \leftarrow initialBlockSize$;
$i \leftarrow 1$;
while $i \leq 4$ do
    $finishTimes \leftarrow$ executeTasks($blockSizeList$);
    $blockSizeList \leftarrow$ setNextBlockSizes($finishTimes$);
    $i \leftarrow i + 1$;
end while
$fitValues \leftarrow$ determineCurveProcessor();
while $fitValues.error \geq 0.7$ do
    $finishTimes \leftarrow$ executeTasks($blockSizeList$);
    $blockSizeList \leftarrow$ setNextBlockSizes($finishTimes$);
    $fitValues \leftarrow$ determineCurveProcessor();
end while
return $fitValues$;
```

node. Variable $blockListSize$ contains the size of the blocks assigned to each processing unit and is initialized with $initialBlockSize$. The algorithm iteratively sends chunks of data to each processing unit and measures the time that each one spent to process them and the time spent to transfer the data. These results are used to determine the block size for each processing unit in the next iteration. After processing four blocks it fits a performance curve model using the function `determineCurveProcessor`. Variable $fitValues$ receives the curve fitting results, including the $error$ in the fitting. If the fit error is above 0.7, it keeps sending new data chunks to the nodes until the error is bounded to 0.7 or when the submitted block sizes reaches 20% of the total application data.

C. Block size selection

In this phase, the PLB-HeC algorithm determines the block size assigned to each processing unit with the objective of providing blocks with the same execution time on each processing unit.

Consider that we have n processing units and an input data of size normalized to 1. The algorithm assigns a data chunk of size x_g for each processing unit $g = 1, ..., n$, corresponding to a fraction of the input data, such that $\sum_{g=1}^{n} x_g = 1$. We denote as $E_g(x_g)$ the execution time of task E in the processing unit g, for input of size x_g. To distribute the work among the processing units, we find a set of values

$$X = \{x_g \in \mathbb{R} : [0, 1] \mid \sum_{g=1}^{n} x_g = 1\} \qquad (3)$$

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

$$E_1(x_1) = E_2(x_2) = \ldots = E_n(x_n) \qquad (4)$$

which indicates that all processing units should spend the same amount of time to process their tasks. In order to determine this set of values x, we solve the system of fitted curves for all processing units, determined in the performance modeling phase, and given by:

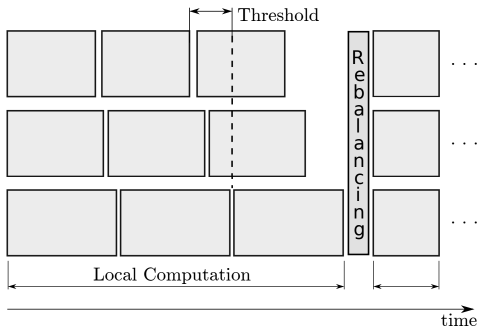


Fig. 3. Gantt chart of the execution of tasks on three processing units. When the difference among processing units is more than a threshold, the system re-balances the tasks.

$$\begin{cases} E_1(x_1) = F_1(x_1) + G_1(x_1) \\ E_2(x_2) = F_2(x_2) + G_2(x_2) \\ \dots \\ E_n(x_n) = F_n(x_n) + G_n(x_n) \end{cases} \quad (5)$$

The equations system is solved applying an interior point line search filter method [25], which finds the minimum solution of a convex equation set, subject to a set of constraints, by traversing the interior of its feasible region. The system of equations is solved subject to the restrictions (3) and (4).

D. Execution and Rebalancing

After determining the task's sizes, the scheduler sends a block of the selected size x_g for each processing unit g. The block size x_g is a floating-point number, which is rounded to the closest valid application data block size. When a processing unit finishes executing a task, it requests another task of the same size. The processing units then continue the execution asynchronously, until completing all the tasks.

The scheduler also monitors the finish time of each task. If the difference in finishing times t_i and t_j between any two tasks of processing units i and j goes above a threshold, the rebalancing process is executed. Small thresholds may cause excessive rebalancing while large thresholds may tolerate larger imbalances that will cause more processing unit idleness. The threshold must be determined empirically; in practice, values of about 10% of the execution time of a single block results in a good trade-off.

During the rebalancing, the scheduler synchronizes the processing units and apply the performance modeling algorithm to fit the best curves for the functions $F_p[x]$ and $G_p[x]$, updated with the execution times of the tasks from the execution phase. The block size selection routine is then applied to determine the new block sizes x_g for each processing unit g.

The rebalancing process for the scenario with three processing units is illustrated in Figure 3. Note that the synchronization does not occur immediately after the threshold detection, since it is necessary to wait for the other processing units to finish processing their tasks. The processing units that detected the threshold also receives a new task, otherwise it would remain idle for a long period waiting for the other processing units.

E. Complete algorithm

Algorithm 2 shows the pseudocode of the PLB-HeC algorithm. The `determineModel` function, shown in Algorithm 1 returns the performance model for each processing unit into variable $fitValues$. The algorithm solves the system of equations contained in $fitValues$ to determine the best distribution X of task sizes for each processing unit. It then calls function `distributeTasks` for each processing unit, passing as parameters the identification of the processing unit and the task size.

Algorithm 2 PLB-HeC algorithm

```
function PLB-HeC()
$fitValues \leftarrow$ determineModel()
$X \leftarrow$ solveEquationSystem($fitValues$);
for each $proc$ from $processorList$ do
   distributeTasks($X$, $proc$);
end for

function FinishedTaskExecution($proc$, $finishTime$)
if $rebalance$ = true then
   $fitValues \leftarrow$ determineCurveProcessor();
   $X \leftarrow$ solveEquationSystem($fitValues$);
   synchronize();
   $rebalance \leftarrow$ false;
end if
if there is data then
   if maxDifference($finishTime$)$\geq$ $threshold$ then
      $rebalance \leftarrow$ true;
   end if
   distributeTask($X$, $proc$);
end if
```

When a processing unit finishes a task it calls the function `FinishedExecution`, passing as parameters its finish time $finishTime$ and its identifier $proc$. The function first verifies if the $rebalance$ flag was previously activated due to a threshold detection. If it is active, the function determines new curves for each processing unit's performance model, solves the system of equations to determine a new distribution of tasks sizes and synchronizes the processes.

The function `FinishedExecution` then checks if there is more data to process and, if true, send a new task to the processing unit that called the function.

IV. Implementation

The algorithms were implemented in the C language, using the StarPU framework. StarPU [14] is a framework for parallel programming that supports hybrid architectures like multicore CPUs and accelerators. The goal of StarPU is to act like a runtime layer that provides an interface unifying execution on accelerator technologies as well as multicore processors. StarPU propose the use of *codelets*, defined as an abstraction for 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, which may use specific languages and libraries for the target architecture. A StarPU application is described as a set of codelets and their data dependencies. New applications can be ported to StarPU by implementing these codelets.

We implemented our load-balancing algorithm in the StarPU framework. The StarPU framework offers a rich API that allows one to modify and develop new scheduling policies. For comparison sake, we also implemented three other algorithms: *greedy*, *Acosta* and *HDSS*. The greedy consisted in dividing the input set in pieces and assigning each piece of input to any idle processing unit, without any priority assignment. Acosta algorithm [18], described in Section II, iteratively finds a good distribution of work between CPUs and GPUs based on the execution of the previous task. Finally, The HDSS [19] was implemented using minimum square estimation to estimate the weights and divided into two phases: adaptation phase and completion phase. StarPU permits the selection of the scheduling algorithm to use for different applications and hardware architectures.

Finally, we used the IPOPT [25] (Interior Point OPTimizer) library to solve the equations systems produced by Equation 5. IPOPT is an open-source software package for large-scale nonlinear optimization for solving nonlinear programming problems.

A. Applications

We used three applications to evaluate the PLB-HeC algorithm: a *matrix multiplication* application, a gene regulatory networks (GRN) inference [26] application, and a financial analysis algorithm (Black-Scholes). Each application was implemented as a pair of codelets, one containing the GPU implementation and the other containing the CPU implementation.

The matrix multiplication application distributes a copy of the matrix A to all processing units and divides matrix B among the processing units according to the load-balancing scheme. We used an optimized version of the matrix multiplication, available from the CUBLAS 4.0 library. Multiplication of two $n \times n$ matrices has complexity $O(n^3)$.

Gene regulatory networks (GRN) inference [26] is an important bioinformatics problem in which gene interactions must to be deduced from gene expression data, such as microarray data. Feature selection methods can be applied to this problem and are composed by two parts: a search algorithm and a criterion function. This application was developed as a parallel solution based on GPU architectures for performing an exhaustive search of the gene subset with a given cardinality that best predict a target gene. The division of work consisted in distributing the gene sets that are evaluated by each processor. The complexity of the algorithm is known to be $O(n^3)$, where n is the number of genes.

Black-Scholes is a popular financial analysis algorithm, based on a stochastic 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. It includes a random walk term, which models random fluctuations of prices over time. The input is a vector of data, from which options should be calculated. The division of the task consists in giving a range of the input vector to each thread. The complexity of the algorithm is $O(n)$, where n is the number of options.

V. Results

A. System Configuration

We used four machines with different processor configurations (shown in Table I) to evaluate our algorithm. The GPUs differ in their clock speed, number of cores and architecture. We performed the experiments using four scenarios: using only one machine (A); two machines (A, B); three machines (A, B, C); and four machines (A, B, C and D). Note that some boards, such as GTX 295 and GTX 680 have two GPU processors.

To use the k Stream Multiprocessors (SMs) from each GPU and the cores of the each SM efficiently, we launched kernels with k blocks and 1024 threads per block. The number of available Stream Multiprocessors k is 13, 14, 8 and 30 for the Tesla K20c, GTX Titan, GTX 680 and GTX 295, respectively. For the CPUs, we created one thread per virtual core, maximizing the CPU usage for hyper-threaded processors.

We used the Matrix Multiplication (MM), Gene Regulatory Networks (GRN) inference, and Black-Scholes applications to evaluate the execution time for different input sizes and number of machines, with StarPU configured to use the following scheduling algorithms: (i) PLB-HeC, (ii) Acosta algorithm, (iii) HDSS, and (iv) Greedy. For all applications, our algorithm was configured with a threshold of 10% of the execution time for the rebalancing process. We determined the initial block size for each application empirically, so that the initial phase of the algorithm would take about 10% of the application execution time, and used the same initial block size for all algorithms. The block size values generated by the scheduling algorithm were rounded to the closest valid block sizes for each application: one line for the matrix multiplication, one gene for the GRN, and one option for the Black-Scholes application.

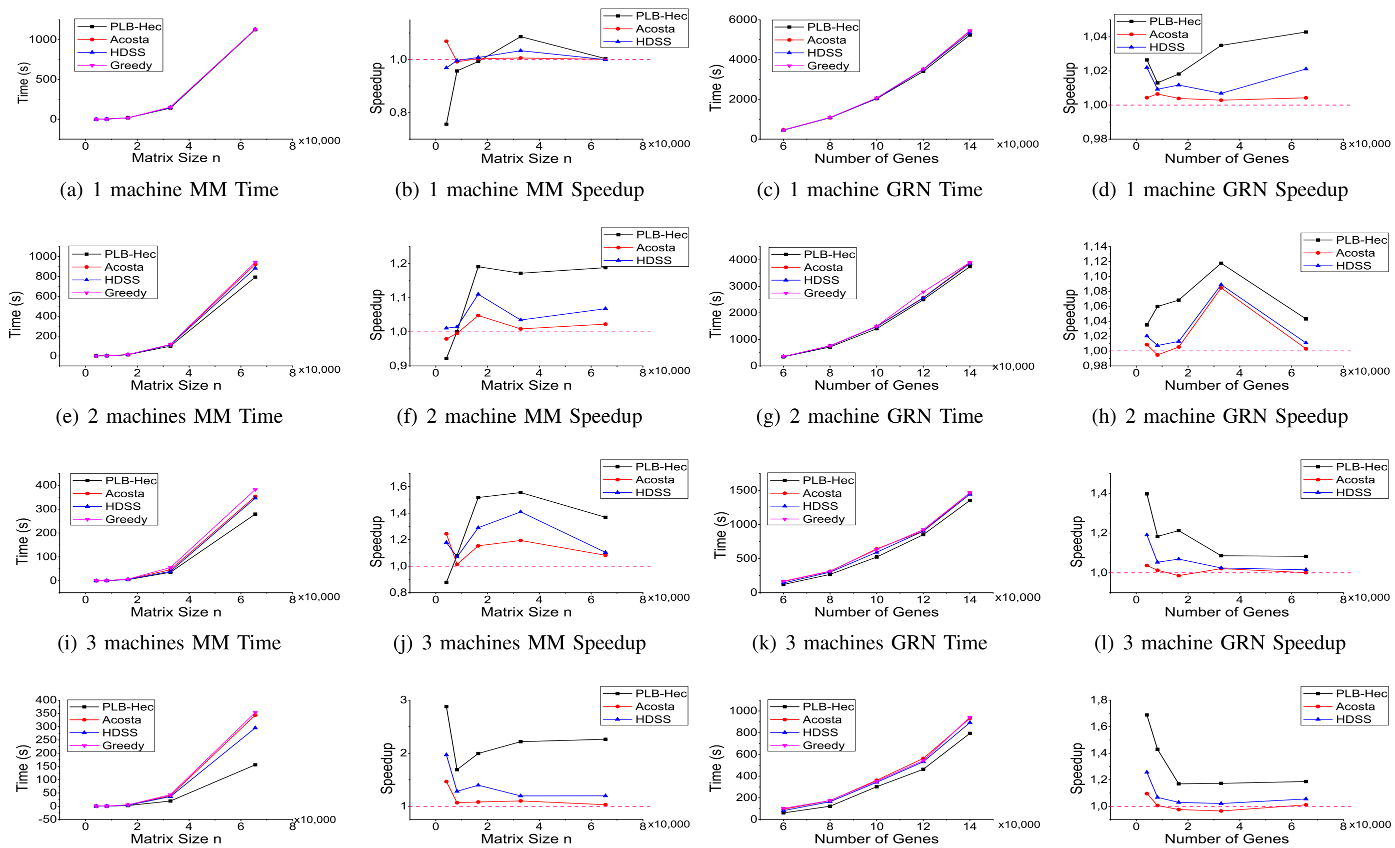
a) Application execution times: Figure 4 shows the execution times using the four load-balancing algorithms, using one to four machines, for the matrix multiplication and GRN applications. We also show the speedup relative to the Greedy algorithm, to facilitate the comparison. We performed experiments using matrices with sizes from 4096×4096 to 65536×65536 elements, and 60,000 to 140,000 genes. Similarly, Figure 5 shows the results for the Black-Scholes application, with 10,000 to 500,000 options. We present the average execution time of 10 experiments. The standard deviations, using dedicated resources, were small and are not shown in the graphs.

The execution time increased with the input size, as expected, but the speedup graphs shows a large fluctuation in the performance of the load-balancing for small inputs. With matrix multiplication, matrices of size 4096×4096 had a limited number of block divisions, and the small total application execution times were not sufficient to compensate the larger overheads of more complex load-balancing algorithms, causing the greedy algorithm to perform better.

We will focus our analysis on larger input sizes, which are the ones which normally demand the usage of GPU clusters. With one machine, the influence of the scheduling algorithm was small, with speedups close to 1, as it is only necessary to divide the application data between a single CPU and GPU. With more machines the differences between the algorithms become more evident.

TABLE I. MACHINE CONFIGURATIONS

Machine	Description				
A	**CPU Info**	Intel Xeon E5-2690V2	10 cores @ 3.0 GHz	25 MB cache	256 GB RAM
	GPU Info	Tesla K20c	2496 / 13 SMs	205 GB/s	6 GB
B	**CPU Info**	Intel i7 a20	4 cores @ 2.67 GHz	8 MB cache	8 GB RAM
	GPU Info	GTX 295	2 x 240 cores / 30 SMs	223.8 GB/s	896 MB
C	**CPU Info**	Intel i7 4930K	6 cores @ 3.4 GHz	12 MB cache	32 GB RAM
	GPU Info	GTX 680	2 x 1536 cores / 8 SMs	192.2 GB/s	2 GB
D	**CPU Info**	Intel i7 3939K	6 cores @ 3.2 GHz	12 MB cache	32 GB RAM
	GPU Info	GTX Titan	2688 cores / 14 SMs	223.8 GB/s	6 GB



(a) 1 machine MM Time (b) 1 machine MM Speedup (c) 1 machine GRN Time (d) 1 machine GRN Speedup

(e) 2 machines MM Time (f) 2 machine MM Speedup (g) 2 machine GRN Time (h) 2 machine GRN Speedup

(i) 3 machines MM Time (j) 3 machines MM Speedup (k) 3 machines GRN Time (l) 3 machine GRN Speedup

(m) 4 machines MM Time (n) 4 machines MM Speedup (o) 4 machines GRN Time (p) 4 machines GRN Speedup

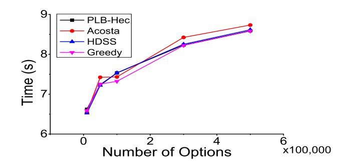
Fig. 4. Execution time and speedup (compared to the Greedy algorithm) for the Matrix Multiplication (MM) and Gene Regulatory Network (GRN) inference applications, using different number of machines and input sizes.

We verified that PLB-HeC obtained the highest speedups in the more heterogeneous scenarios, as expected, in which case the more precise block size selection algorithm becomes crucial. We emphasize that the presented execution times include the time spent calculating the size of the task sizes for each processing unit using the interior point method. The mean time spent on this calculation was 170 ms, for the scenario with 4 machines and matrices of order 65536, with standard deviation of 32.3 ms. Although this time is not negligible, the gains obtained with the better distribution of tasks sizes largely compensates the overhead caused by these calculations.

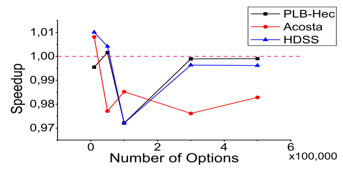
The highest gains occurred with the matrix multiplication application and four machines, with a speedup of 2.2 using our algorithm in comparison with the Greedy ones for matrices with 65536×65536 elements. The other dynamic algorithms, HDSS and Acosta, obtained speedups of 1.2 and 1.04, respectively. For other machine configurations and other applications the speedup was lower, but for configurations with 3 or more machines was consistently above 1.2, except for the GRN with three machines. Also, for larger inputs, our algorithm always resulted in lower execution times in comparison to the other three load-balancing algorithms.

To understand the origin of this difference in application execution, we evaluated the block distributions generated by each load-balancing algorithm and the resulting idleness time of the processing unit.

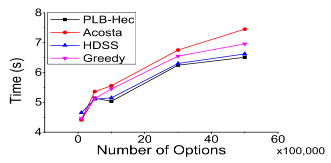
b) Block size distribution: Figure 6 shows the distribution of blocks among the machines for the three algorithms that uses block size estimation for the distribution: Acosta, HDSS and PLB-HeC. We present the results for two matrix sizes, two GRN sizes and two number of options. We used the machines A, B, C and D with one GPU per machine and used one thread per CPU core and 1 thread per GPU core. The



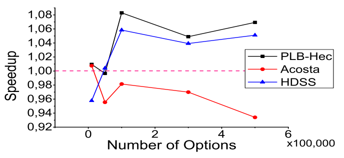
(a) 1 machine BlackScholes Time



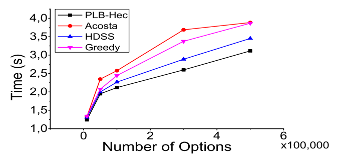
(b) 1 machine BlackScholes Speedup



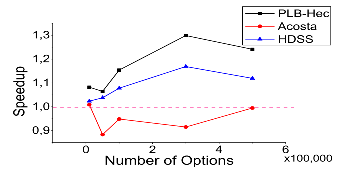
(c) 2 machine BlackScholes Time



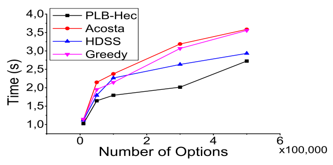
(d) 2 machine BlackScholes Speedup



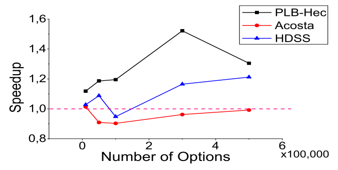
(e) 3 machine BlackScholes Time



(f) 3 machine BlackScholes Speedup



(g) 4 machine BlackScholes Time



(h) 4 machine BlackScholes Speedup

Fig. 5. Execution time and speedup (compared to the Greedy algorithm) for the BlackScholes application, using different number of machines and input sizes.

values represent the ratio of total data allocated on a single step for to each CPU/GPU processor. We considered the block sizes generated at the end of the performance modeling phase for the algorithm PLB-HeC, of phase 1 for the HDSS algorithm, and of the application execution for Acosta algorithm. The values are normalized so the total size is equal to 1. We performed 10 executions and present the average values and standard deviations. The standard deviation values are small, showing that all algorithms are stable through different executions.

When comparing the block size distributions generated by the three load-balancing algorithms, we note that Acosta and HDSS algorithms produce similar distributions. The PLB-HeC algorithm produces a qualitatively different distribution, with proportionally smaller blocks allocated to CPUs and larger blocks to GPUs, especially for the machines C and D, which contain the largest number of cores. We note that both HDSS and Acosta algorithm use simple linear weighted means from a set of performance coefficients, which resulted in similar distribution. Nevertheless, the HDSS algorithm resulted in lower execution times due to the faster convergence. PLB-HeC uses a performance curve model for each processing unit and solves the resulting system of equations, probably resulting in a more precise estimation of the best block partition among processing units.

c) Processing unit idleness: We also measure the percentage of time that each CPU and GPU was idle during application execution. Figure 7 shows the results for the PLB-HeC and HDSS algorithm. We used the same experimental setup from the block size distribution experiment. At each task submission round we recorded the time that each processing unit was idle. We executed each application with each algorithm 10 times.

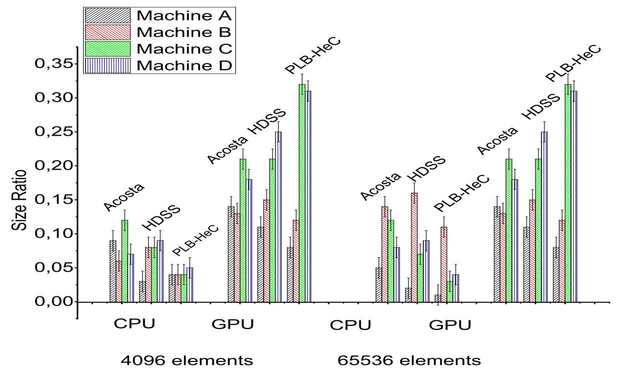
The HDSS algorithm produced larger processing unit idleness than PLB-HeC in all scenarios. This idleness occurred mainly in the first phase of the HDSS algorithm, where non-optimal block sizes are used to estimate the computational capabilities of each processing unit. PLB-Hec prevents this idleness periods in the initial phase by starting to adapt the block sizes after the submission of the first block, significantly reducing the idleness generated on this phase. Acosta algorithm, not shown, produced significantly larger idleness, since it synchronizes the block executions multiple times during execution until it reaches an equilibrium, which is obtained asymptotically.

Another measured effect is that with larger input sizes, which are the most important when considering GPU clusters, the percentage of idleness time was smaller. This occurred mainly because the time spent in the initial phase, where most of the idleness time occurs, was proportionally smaller when compared to the total execution time. This effect is evident when comparing the idle times of the matrix multiplication application with 4096 and 65536 elements for the PLB-Hec algorithm.

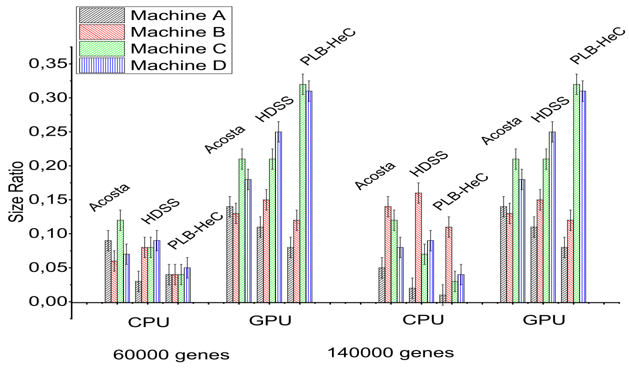
Incorrect block size estimations also produces idleness in the execution phase of the algorithms, specially in the final part, since some processing units may finish their tasks earlier than others. HDSS prevents part of this idleness using decreasing block size values during the execution. If some processing unit is idle at the end, it can receive extra small blocks to process. PLB-Hec uses another approach, by performing a rebalancing process when the difference in execution time among processing units is above a threshold. Interestingly, this rebalancing was not executed, since the generated block size distributions were always within the 10% threshold range. But this mechanism can be useful in scenarios where the execution conditions on the machines change or in applications where the block size estimates are less accurate.

VI. Conclusions and Future Work

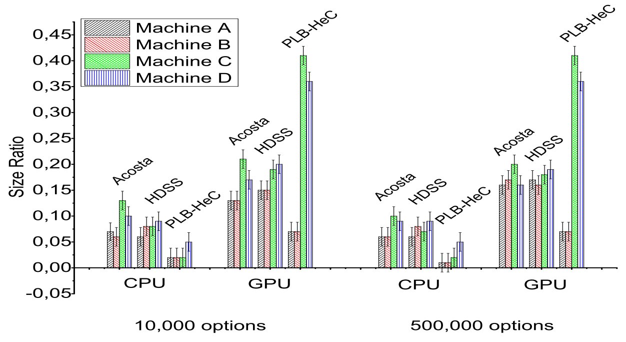
In this paper we present PLB-Hec, a novel algorithm for dynamic load-balancing of domain decomposition applications executing on clusters of heterogeneous CPUs and GPUs. It performs a profile-based online estimation of the performance curve for each processing unit and selects the block size distribution among processing units solving a non-linear system of equations subject to restrictions. We used three real-world applications in the fields of linear algebra, bioinformatics and stock markets and showed that our approach decreased the application execution time, when compared to other dynamic algorithms. PLB-Hec obtained the highest performance gains with more heterogeneous clusters and larger problems sizes, where a more refined load-distribution is required. The PLB-Hec was implemented on top of the well-known StarPU



(a) Matrix Multiplication



(b) Gene Regulatory Network



(c) Blackscholes

Fig. 6. Block size distribution among the processing units (CPU and GPU) from the four machines, for the Matrix Multiplication, GRN and BlackScholes applications, using two different input sizes for each.

framework, what allows its immediate usage for a several existing applications and a easier development cycle for new application.

Although we used dedicated clusters, we can also envisage the usage of our load-balancing algorithm on public clouds, where the user can request a number of resources allocated in virtual 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. Another interesting scenario would be to consider 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 re-adapt to this scenario.

Acknowledgment

The authors would like to thank UFABC, FAPESP (Proc. n. 2012/03778-0 and Proc. n. 2013/26644-1) and NAPSoL-PRP-USP for the financial support and Fabrizio Borelli for providing the gene regulatory network application.

References

[1] 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, p. 47.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

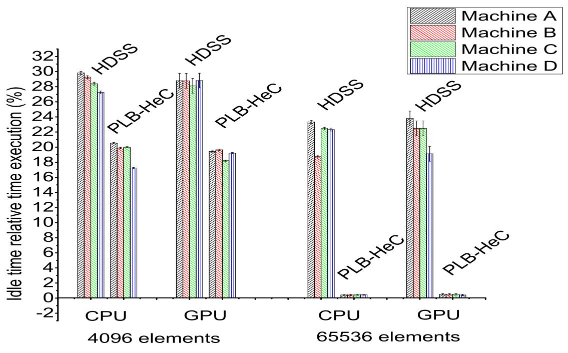
[6] Y. Jeon, E. Jung, H. Min, E.-Y. Chung, and S. Yoon, "GPU-based acceleration of an RNA tertiary structure prediction algorithm," *Computers in Biology and Medicine*, vol. 43, no. 8, pp. 1011 – 1022, 2013.

[7] 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.

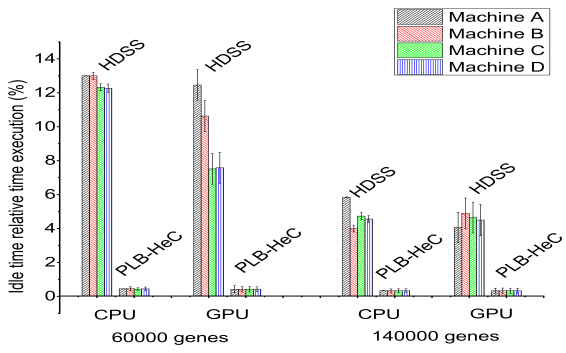
[8] NVIDIA, "Nvidia cuDNN – GPU accelerated machine learning," Available at: https://developer.nvidia.com/cuDNN, Oct. 2014.

[9] ——, "CUDA math library," Available at: https://developer.nvidia.com/cuda-math-library, Oct. 2014.

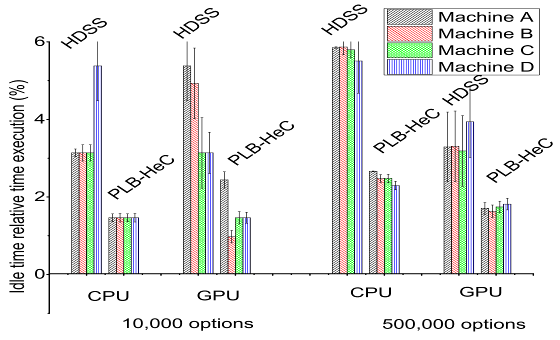
[10] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/GPU nodes," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10.



(a) Matrix Multiplication



(b) Gene Regulatory Network



(c) Blackscholes

Fig. 7. Processing unit idle time in relation to the total execution time for the Matrix Multiplication, GRN and BlackScholes applications, using two different input sizes for each.

[11] 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.

[12] 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.

[13] 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.

[14] 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, INRIA Bordeaux - Sud-Ouest, Rapport de recherche RR-7240, Mar 2010.

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

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

[17] 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.

[18] 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.

[19] 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, Jan. 2013.

[20] 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.

[21] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with gpu accelerators," *Concurrency and Computation: Practice and Experience*, 2014.

[22] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012, pp. 191–199.

[23] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628088

[24] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé, "G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 349–358. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465444

[25] 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

[26] 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.