PLB-HAC: Dynamic Load-Balancing for Heterogeneous Accelerator Clusters

Luis Sant'Ana[1], Daniel Cordeiro[2], and Raphael Y. de Camargo[1](✉)

[1] Federal University of ABC, Santo André, Brazil
[2] University of São Paulo, São Paulo, Brazil
raphael.camargo@ufabc.edu.br

Abstract. Efficient usage of Heterogeneous clusters containing combinations of CPUs and accelerators, such as GPUs and Xeon Phi boards requires balancing the computational load among them. Their relative processing speed for each target application is not available in advance and must be computed at runtime. Also, dynamic changes in the environment may cause these processing speeds to change during execution. We propose a Profile-based Load-Balancing algorithm for Heterogeneous Accelerator Clusters (PLB-HAC), which constructs a performance curve model for each resource at runtime and continuously adapt it to changing conditions. It dispatches execution blocks asynchronously, preventing synchronization overheads and other idleness periods due to imbalances. We evaluated the algorithm using data clustering, matrix multiplication, and bioinformatics applications and compared with existing load-balancing algorithms. PLB-HAC obtained the highest performance gains with more heterogeneous clusters and larger problems sizes, where a more refined load-distribution is required.

1 Introduction

Heterogeneous clusters, containing different combinations of CPUs and accelerators, such as GPUs and Intel MIC boards, are becoming increasingly widespread. In order to achieve the best performance offered by these clusters, scientific applications must take into account the relative processing speed of each processor unit and balance the computational load accordingly.

For data-parallel applications, it is necessary to determine an appropriate data (task) division among the CPUs and accelerators. A division of the load based on simple heuristics, such as the number of cores in the GPU is usually ineffective [5]. Another solution is to use simple algorithms for task dispatching, such as greedy algorithms, where tasks are dispatched to the devices as soon as the devices become available. Such heuristics are fast and straightforward, but result in suboptimal distributions.

A more elaborate and precise load-balancing algorithm causes a higher overhead, but a better task distribution can compensate for the overhead. For instance, it is possible to determine the performance profiles for each GPU type

© Springer Nature Switzerland AG 2019
R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 197–209, 2019.
https://doi.org/10.1007/978-3-030-29400-7_15

raphael.camargo@ufabc.edu.br

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 [5], or dynamically computed at runtime [1,3,10].

We present a Profile-based Load-Balancing algorithm for Heterogeneous Accelerator Clusters (PLB-HAC), which improves PLB-HeC [10] by removing synchronization phases, enhancing the rebalancing mechanism, and including support for Xeon Phi accelerators. The algorithm uses performance information gathered at runtime in order to devise a performance model customized for each processing device. The algorithm is implemented inside the StarPU framework [2], easing its use both on legacy applications and novel ones.

2 Related Work

In this work, we focus on the development of a dynamic algorithm with adaptability, through the use of performance models based on the processing capacity. Acosta *et al.* [1] proposed an algorithm for GPUs where processors record their individual execution time on periodical synchronization, to asymptotically generate the RP (Relative Power) of the processors. The main drawbacks are that asymptotic convergence causes suboptimal load distributions during several iterations and frequent synchronizations further slow down application execution.

In another work, Zhong *et al.* [11] use the concept of logical processors to model GPU-CPU hybrid systems. The workload is split using an FPM (Functional Performance Model) that provides a detailed performance model. The approach is limited because it requires prior information about the problem to set up the model parameters.

Heterogeneous Dynamic Self-Scheduler (HDSS) [3] is a dynamic load-balancing algorithm for heterogeneous GPU clusters. In an adaptive phase, it determines weights that reflect the speed of each GPU, which it uses to divide the work among GPUs in the remaining iterations. The performance model is specific to GPUs and the use of a simple weight per GPU limits the data distribution. Finally, it does not adjust the data distribution during the execution phase and synchronizations in the adaptive phase slow down application execution.

Kaleen *et al.* [8] proposed the *naive algorithm*, which executes in two phases: profiling and execution. In the profiling phase, the algorithm determines the processing rate Gr of GPUs and Cr of CPUs, which are used for data distribution in the execution phase. A second algorithm, called *asymmetric algorithm*, reduces the overhead of the initial phase by sharing a pool of work between CPU and GPU. Their approach is suited for CPUs and GPUs and reduces synchronizations, but the obtained performance can degrade in case of changes during the execution phase.

PLB-HeC [10] performs dynamic load-balancing in two phases for clusters with CPUs and GPUs. The first phase constructs the performance model using profiling, while in the second blocks of the selected sizes are dispatched to the processing units. It differs from previous approaches in that it models the processors (CPUs and GPUs) using a system of nonlinear equations to improve the

raphael.camargo@ufabc.edu.br

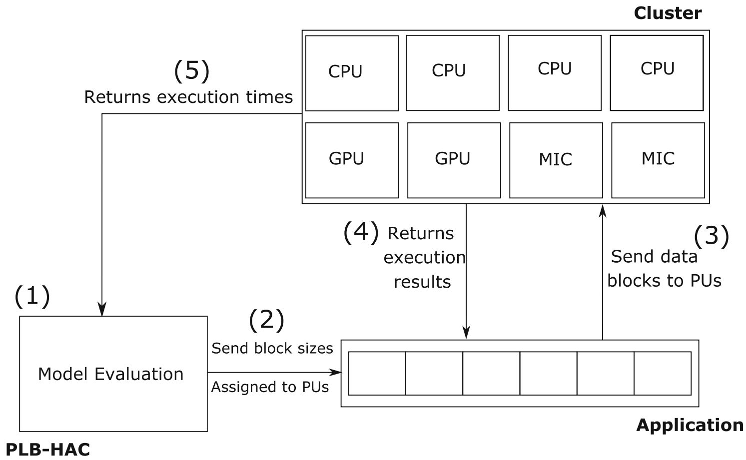

Fig. 1. Overview of the load-balancing algorithm, including the performance model evaluation, block distribution, and task execution.

accuracy of block size distributions. However, it still contains several synchronization steps, which slowdowns application execution.

Our proposed algorithm addresses the limitations of previous dynamic load-balancing algorithms. It uses the same approach of solving a system of nonlinear equations from PLB-HeC, but it has no explicit or implicit synchronization between processors within the training and execution phases. Moreover, it performs a progressive refinement of the performance models for the processors during the entire execution, which allows it to adapt to changes in the execution environment. Idle periods that could still result from imperfect load-balancing are filled with smaller blocks of the correct size. Finally, it supports several classes of processing devices, including CPUs, GPUs, and Xeon Phi boards.

3 Proposed Algorithm

In a typical data-parallel application, data is divided into blocks that can be concurrently processed by multiple threads, in a process called domain decomposition [6]. The results are then merged, and the application can proceed to its next phase. The goal of the PLB-HAC algorithm is to find a near-optimal distribution of the size of data blocks assigned to threads located on each CPU and accelerator in the system. We use the term Processing Unit (PU) to refer to both CPUs, GPUs and Xeon Phi coprocessors.

PLB-HAC generates and evaluates performance models of PUs and determine the optimal block size distribution, which is shown in the "Model Evaluation" box in Fig. 1. The list of block sizes is sent (2) to the application, which sends the data blocks for execution in the assigned PUs (3), together with the execution code for that PU. The PUs process the data blocks and return the results to the

raphael.camargo@ufabc.edu.br

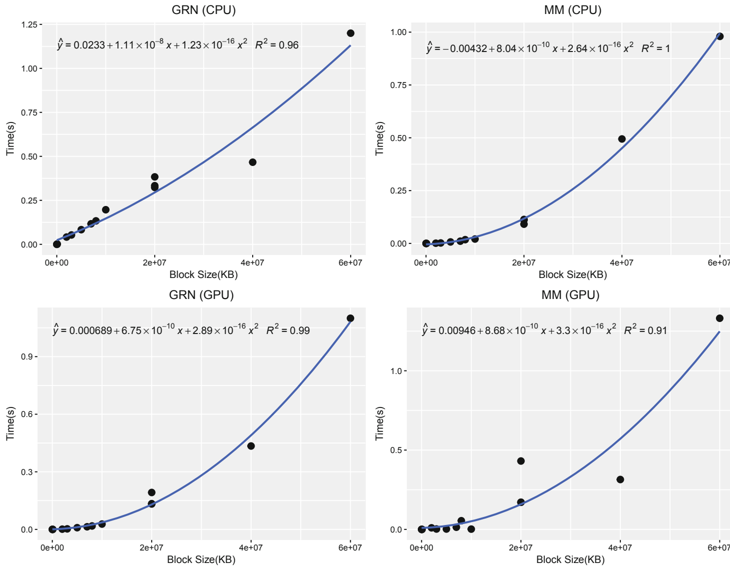

Fig. 2. Execution times and performance models for the GPU and CPU implementations of the k-means and matrix multiplication applications.

application (4) and the execution time of the block to PLB-HAC (5), which are used to improve the performance model.

The mechanics of data and code migration can be managed by a framework such as StarPU and Charm++, where a user is presumed to implement only the task code. The remaining of this section discuss the implementation of the "Model Evaluation" from Fig. 1 in PLB-HAC in a framework-agnostic way and the next section show how the algorithm can be integrated to StarPU.

3.1 Processing Unit Performance Modeling

The algorithm devises a performance model for each processing unit based on execution time measurements. The algorithm constructs two functions $F_p[x]$ and $G_p[x]$, representing the amount of time a processing unit p spends processing and transmitting a block of size x, respectively. These functions are generated in a training phase, where the algorithm first assigns a block of size x_{init}—initially defined by the user—to be processed by each PU. The unit that finishes first receives a second block of size $2 * x_{init}$, while each other PU p receives a block of size equal to $2 * x_{init} * R$, where R is the ratio between the time spent by the fastest unit and the time spent by unit p. The idea is to balance the load better, preventing a long delay between the finish times of the different processing units.

The measured execution and data transfer times for each new block and PU are used to create the performance model. The algorithm performs a linear regression to determine the execution time functions $F_p[x]$ that better fit the

raphael.camargo@ufabc.edu.br

existing pairs (x, t_x) using the least squares method. The same is done for $G_p[x]$, but using the data transfer times. The curve is initially fitted using two points and after each new iteration, another point is added to the model, resulting in better models. This calculation is done in the first CPU that finishes the execution of its assigned block. The algorithm performs a linear regression of the form:

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

where $f_i(x)$ are functions from the set x, x^2, x^3, e^x, $\ln x$, 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 2 shows sample processing time measurements and model fittings for a GPU and a CPU for different block sizes on two different applications. For the $G_p[x]$ function, we used an equation of the form:

$$G_p[x] = a_1 x + a_2 \tag{2}$$

where the linear coefficient a_1 represents the network and PCIe bandwidths, and a_2 the network and system latency. We assume that the data transfer delay increases linearly with data size, which should be a valid approximation for compute-bound applications.

3.2 Block Size Selection

The proposed algorithm determines the block size assigned to each processing unit with the objective that all PUs have the same execution time. Consider that we have n processing units and input data of size normalized to 1. The algorithm assigns a data chunk of size $x_p \in [0, 1]$ for each processing unit $p = 1, ..., n$, corresponding to a fraction of the input data, such that $\sum_{p=1}^{n} x_p = 1$. We denote as $E_p(x_p)$ the execution time of task E in the processing unit p, for input of size x_p. To distribute the work among the processing units, we find a set of values $(x_p)_{i=1}^{n}$ that minimizes the system of fitted curves for all processing units, determined in the training phase, while keeping the same execution time for all units. The full set of equations are given by:

$$\begin{cases} E_1(x_1) = F_1(x_1) + G_1(x_1) \\ E_2(x_2) = F_2(x_2) + G_2(x_2) \\ \dots \\ E_n(x_n) = F_n(x_n) + G_n(x_n) \\ E_1(x_1) = E_2(x_2) = \cdots = E_n(x_n) \\ \sum_{p=1}^{n} x_p = 1 \end{cases} \tag{3}$$

The equation system is solved applying an interior point line search filter method [9], 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 solution of the equation system results in a block size x_p for each processing unit p. The block size is rounded to the closest valid application data block size, so that all units will spend approximately the same processing time executing their blocks. We also use a minimum block size that does not underutilize the GPUs.

raphael.camargo@ufabc.edu.br

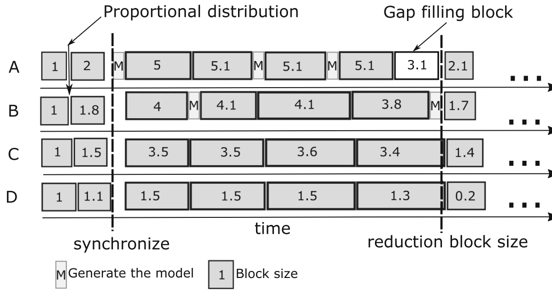

Fig. 3. PLB-HAC execution example. The number inside boxes represents the block sizes assigned to units, M represents model evaluation to determine block sizes. Shaded boxes are regular assigned blocks and white boxes are gap filling blocks.

3.3 Execution Phase

The execution phase is asynchronous. Tasks of size x_p, determined at the training phase, are sent to each PU p. When a PU notifies the scheduler that it finished executing a task, the measured performance is added to the set of points of the performance model and another task of size x_p is sent to the PU.

The task size x_p is updated at the end of each "virtual step", which occurs when all PUs finish the execution of their assigned blocks. During the update, the first CPU to become idle solves the equation system (3)—using all the execution time measurements collected from each unit—to determine the block size x_p for each processing unit p. No synchronization is required since the scheduler uses the most recent already available x_p when assigning a block size to PU p.

Two other mechanisms improve the load-balancing process. The first one is a gap-filling mechanism, used if a processing unit finishes the execution of its assigned block earlier than expected. The algorithm provides a new block to fill the gap between the predicted and actual execution time that a unit needed to process its last block. We used a default threshold of 400 ms, which can be changed by the user. The second mechanism is a gradual decrease in block sizes after 70% of the application data was processed, with the goal of reducing possible unbalances at the end of the execution. The decrease is by a constant factor $\alpha = 0.1$, and the user can adjust this factor to suit the application better.

Figure 3 shows an execution example of four heterogeneous PUs (A, B, C, and D). At the left of the first vertical dashed line is the training phase, where blocks of size 1 are sent to each PU. Machine A finishes processing its block first and receives a new block of size 2, while others receive smaller blocks. A synchronization then takes place and the first complete model is generated. This is the only synchronization step in the entire execution. Between the dashed lines is the execution phase, with several virtual steps. At the beginning of each virtual step, the first unit to finish the last step evaluates a new model. Unit A also receives a gap-filling block. Near the end of the execution, the blocks are progressively reduced in size until there is no more data to process.

raphael.camargo@ufabc.edu.br

3.4 Complete Algorithm

Algorithm 1 shows the pseudocode of the PLB-HAC algorithm. The function `FinishedTaskExecution` is a callback function, invoked when a processing unit finishes a task. It receives the finish time ($finishTime$) and the processing unit identifier (*proc*). If there is still data left to be processed, it first checks if the finish time of the application was smaller than the *gapThreshold*, in which case it sends a new block to fill the gap between the predicted and finished time. Otherwise, it calls `PLB-HAC` to determine the next block size.

Algorithm 1. Compute the performance model for each processing unit

X : global vector
function PLB-HAC(*proc*)
$firstProc \leftarrow$ firstProcessVirtualStep(*proc*);
if $proc == firstProc$ **then**
  $fitValues \leftarrow$ determineModel()
  $X \leftarrow$ solveEquationSystem($fitValues$);
  if $assignedData \geq 0.7 * totalData$ **then**
    $X \leftarrow X * k$;
  end if
  $assignedData \leftarrow assignedData + \text{sum}(X)$;
end if
distributeTask($X[proc]$, $proc$);

function FinishedTaskExecution(*proc*, $finishTime$)
if $assignedData \leq totalData$ **then**
  if $finishedTime$ / $predictedTime \leq gapThreshold$ **then**
    $assignedData \leftarrow assignedData +$ determineGapBlockSize();
    distributeTask(X, $proc$);
  else
    PLB-HAC();
  end if
end if

Function `PLB-HAC` first calls `firstProcessVirtualStep`, which keeps track of the PU's current virtual step and returns the identifier $firstProc$ of the first unit to enter the current virtual step. If the calling unit is $firstProc$, it first determines a new performance model for all units using the `determineModel` function, which returns a structure $fitValues$ containing the model. Function `solveEquationSystem` then solves the system of equations (3) and returns the best distribution of block sizes for each PU in X. Finally, if 70% of all data was already processed, the algorithm decreases the block size by multiplying it by k, a constant that defines the rate of block size reduction. At the end of the function, function `distributeTask` is called, which sends the data block of the determined size to the unit.

raphael.camargo@ufabc.edu.br

4 Implementation

The PLB-HAC algorithm was implemented in C++ over StarPU [2], a framework for parallel programming that supports hybrid architectures. StarPU is based on the concept of codelets, which describe computational kernels that can be implemented on multiple architectures.

For comparison sake, we also implemented two other load-balancing algorithms: *greedy* and HDSS [3]. The greedy algorithm divides the input set in pieces and assigns each piece to the first available processing unit. HDSS was implemented using minimum square estimation to determine the weights and divided into two phases: adaptation and completion phase. We used the IPOPT [9] (Interior Point OPTimizer) library to solve the equations systems produced by Eq. (3).

4.1 Applications

We used three applications to evaluate the PLB-HAC algorithm: a *matrix multiplication* (MM) application, a gene regulatory network (GRN) inference [4] application, and a clustering algorithm, the K-Means. Each application was implemented as a pair of *codelets*, containing optimized GPU and CPU implementations.

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

Gene Regulatory Network (GRN) inference [4] is a bioinformatics problem in which gene interactions must be deduced from gene expression data. It depends on an exhaustive search of the gene subset with a given cardinality that best predicts a target gene. The division of work consisted of 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.

K-means clustering is a popular method for cluster analysis which partitions n observations into k clusters. The problem can be exactly solved in time $O(n^{dk+1})$, where n is the number of entities to be clustered and d is the input dimensionality [7].

5 Results

We used five machines with different PU configurations (Table 1). We considered five scenarios: one machine (A); two machines (A, B); three machines (A, B, and C); four machines (A, B, C, and D) and five machines (A, B, C, D, and E). For GPUs, we launched kernels with 1 thread block per Stream Multiprocessors (SMs). For the CPUs and Xeon Phi, we created one thread per (virtual) core.

raphael.camargo@ufabc.edu.br

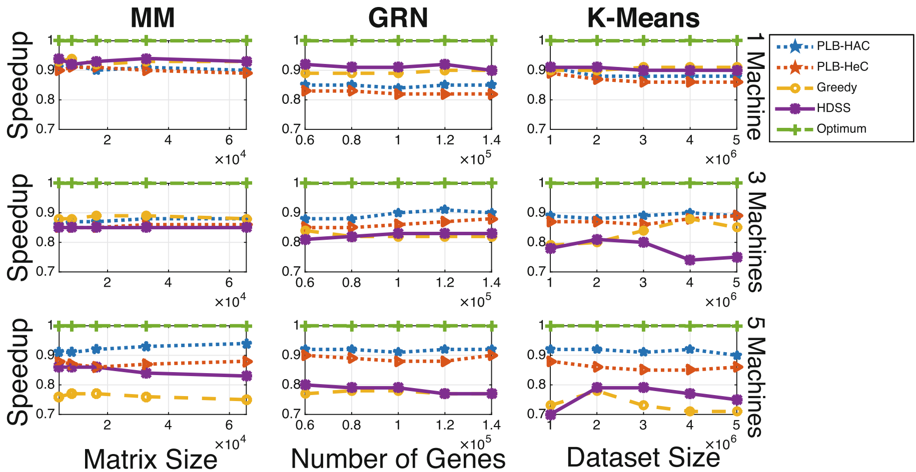

Fig. 4. Speedup (compared to the optimum execution) for the Matrix Multiplication (MM), Gene Regulatory Network (GRN) inference application and K-Means algorithm, using different number of machines and input sizes.

Table 1. Machine configurations

Machines	Description				
	PU type	Model	Core count	Cache/throughput	Memory
A	CPU	Intel i7 - 5930K	6 cores @ 3.5 GHz	15 MB cache	32 GB
	GPU	Quadro K5200	2304 cores	192 GB/s	8 GB
B	CPU	Intel i7 - 5930K	6 cores @ 3.5 GHz	15 MB cache	32 GB
	GPU	GTX 970	1667 cores	224 GB/s	4 GB
	Xeon Phi	3120 series	57 cores	240 GB/s	6 GB
C	CPU	Intel Xeon E-2620	6 cores @ 2.10 GHz	15 MB cache	32 GB
	GPU	Quadro K620	384 cores	29 GB/s	2 GB
D	CPU	Intel i7-4930k	6 cores @ 3.40 GHz	12 MB cache	32 GB
	GPU	GPU Titan	2688 cores	288.4 GB/s	6 GB
E	CPU	Intel Xeon E-2620	6 cores @ 2.10 GHz	15 MB cache	32 GB
	GPU	Quadro K620	384 cores	29 GB/s	2 GB
	Xeon Phi	3120 series	57 cores	240 GB/s	6 GB

5.1 Application Speedup

Figure 4 shows the average of 10 runs of each algorithm. Lines labeled "Optimum" show results for the optimal load balancing, obtained empirically by brute force searching. We used the same initial block size for both PLB-HAC, PLB-HeC, and HDSS, with 1024 elements.

raphael.camargo@ufabc.edu.br

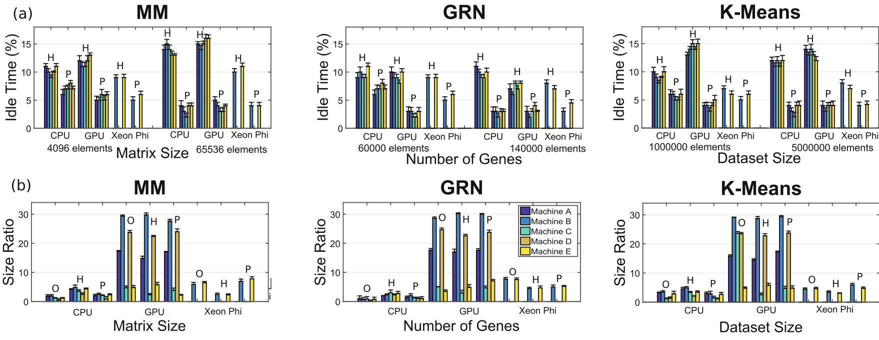

Fig. 5. (a) Percentage of idle time for each PU class (CPU, GPU, and Xeon Phi) in the five machines (colored bars), when using the HDSS (H) and PLB-HAC (P) algorithms for each application. (b) The block size ratio distributed to each PU in the five machines for the Optimal (O), HDSS (H) and PLB-HAC (P) distributions.

With three or more machines, PLB-HAC algorithm approximates the optimal curve and exceeds the performance of the compared algorithms. The cost involved in the calculation of the block size distribution (about 100 ms per iteration), is mitigated by the better distribution of blocks. The behavior is similar for all three applications evaluated, with PLB-HAC performing better in more heterogeneous environments.

With one machine, PLB-HAC exhibited lower performance than HDSS and Greedy. For HDSS, the overhead from the model generation occurs only once, at the end of the adaptive phase. Note that PLB-HAC has a better performance than PLB-HeC due to the removal of the synchronization steps.

5.2 Block Size Distribution

We compared the distribution of block sizes among the PUs. Figure 5b shows the results for a matrix of 65,536 elements for matrix multiplication, 140,000 genes for GRN and 500,000 points for k-means. We used the five machines A, B, C, D, and E. The values represent the ratio of the total data allocated on a single step to each CPU/GPU processor, normalized so that total size is equal to 100. We considered the block sizes generated at the end of the training phase for the algorithm PLB-HAC, and at the end of phase 1 for the HDSS algorithm. 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.

The PLB-HAC algorithm produced a distribution that was qualitatively more similar to the optimum algorithm than HDSS, with proportionally smaller blocks allocated to CPUs and larger blocks to GPUs. We attribute this difference to the use of a performance curve model by PLB-HAC, in contrast to the use of

raphael.camargo@ufabc.edu.br

the simpler linear weighted means from a set of performance coefficients done by HDSS.

5.3 Processing Unit Idleness

We also measured the percentage of time that each CPU and GPU was idle during application execution, using the same experimental setup from the block size distribution experiment. At each task submission round, we recorded the time intervals where each processing unit remained idle. We executed each application with each load-balancing algorithm 10 times.

Figure 5a shows that HDSS produced larger processing unit idleness than PLB-HAC in all scenarios. This idleness occurred mainly in the first phase of HDSS, where non-optimal block sizes are used to estimate the computational capabilities of each processing unit. PLB-HAC prevents these idleness periods in the initial phase by starting to adjust the block sizes after the submission of the first block, significantly reducing the idleness generated on this phase.

Another measured effect is that with larger input sizes—which are the most representative 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 4,096 and 65,536 elements for the PLB-HAC algorithm.

Incorrect block size estimations also produce idleness in the execution phase of the algorithms, especially in the final part, since some processing units may finish their tasks earlier than others. HDSS and PLB-HAC prevent part of this idleness using decreasing block size values during the execution.

5.4 Adaptability

We evaluated the adaptability of PLB-HAC to situations where the resource state changes during application execution. For instance, an external application could be started in some of the machines where the PLB-HAC managed application is executing.

We used two machines (A and B), with one CPU and GPU on each. They are initially idle, and we start the execution of the MM application. After 142 s, we start the execution of a CUDA-based GRN application at machine A, which competes for GPU resources. PLB-HAC detects that executions at GPU A are taking longer and reduces the block size for this GPU, as shown in Fig. 6a. Conversely, the block size for GPU B is increased, compensating the reduction in GPU A. Figure 6b shows a scenario where we start a render application in GPU A after 142 s. Note that the PLB-HAC reduces the block size to GPU A to near zero while increasing the block size of GPU B.

It is important to note that the adaptation was fast, with the block size falling from 2500 KB to 1188 KB within 38 s in the first case (a) and from 2500 KB to

raphael.camargo@ufabc.edu.br

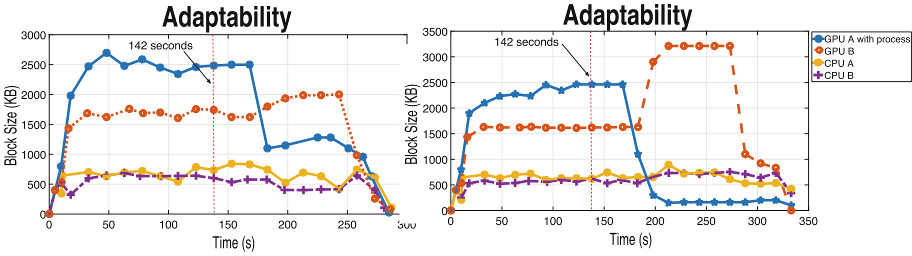

Fig. 6. Evolution of the block size distribution for two machines (A and B) in the presence of a competing process, which is started at GPU A at instant 142 s, denoted by the vertical line.

320 KB in 43 s in the second case (b). Also note the decrease of block sizes at the end of the execution, which is a result of PLB-HAC policy of distributing smaller blocks at the end of the execution, avoiding possible load unbalances that could occur at this phase.

6 Conclusions

In this paper, we presented PLB-HAC, a novel algorithm for dynamic load-balancing of domain decomposition applications executing on clusters of heterogeneous CPUs, GPUs and Xeon-Phi. 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. We used three real-world applications in the fields of linear algebra, bioinformatics, and data clustering and showed that our approach decreased the application execution time when compared to other dynamic algorithms.

Experiments showed that PLB-HAC performed better for higher degrees of heterogeneity and larger problem sizes, where a more refined load-distribution is required. The PLB-HAC was implemented on top of the well-known StarPU framework, which allows its immediate use for several existing applications and an easier development cycle for new applications.

As future work, we need to evaluate the scalability of PLB-HAC by executing experiments with applications that require hundreds or thousands of processing units. Another point is to extend the method to work with applications that have multiple kernels.

Acknowledgment. The authors would like to thank UFABC and FAPESP (Proc. n. 2013/26644-1) for the financial support, Fabrizio Borelli for providing the GRN application and Samuel Thibalt for helping with StarPU. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

raphael.camargo@ufabc.edu.br

References

1. Acosta, A., Blanco, V., Almeida, F.: Towards the dynamic load balancing on heterogeneous multi-GPU systems. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 646–653 (2012)
2. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. Technical report RR-7240, INRIA, March 2010
3. Belviranli, M.E., Bhuyan, L.N., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Arch. Code Optim. **9**(4), 57:1–57:20 (2013)
4. Borelli, F.F., de Camargo, R.Y., Martins Jr., D.C., Rozante, L.C.: Gene regulatory networks inference using a multi-GPU exhaustive search algorithm. BMC Bioinform. **14**(18), 1–12 (2013)
5. de Camargo, R.: A load distribution algorithm based on profiling for heterogeneous GPU clusters. In: 2012 Third Workshop on Applications for Multi-Core Architectures (WAMCA), pp. 1–6 (2012)
6. Gropp, W.D.: Parallel computing and domain decomposition. In: Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA (1992)
7. Inaba, M., Katoh, N., Imai, H.: Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. In: Proceedings of the Tenth Annual Symposium on Computational Geometry, pp. 332–339. ACM (1994)
8. Kaleem, R., Barik, R., Shpeisman, T., Lewis, B.T., Hu, C., Pingali, K.: Adaptive heterogeneous scheduling for integrated GPUs. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT 2014, pp. 151–162. ACM, New York (2014). https://doi.org/10.1145/2628071.2628088
9. Nocedal, J., Wächter, A., Waltz, R.: Adaptive barrier update strategies for nonlinear interior methods. SIAM J. Optim. **19**(4), 1674–1693 (2009). https://doi.org/10.1137/060649513
10. Sant'Ana, L., Cordeiro, D., Camargo, R.: PLB-HeC: a profile-based load-balancing algorithm for heterogeneous CPU-GPU clusters. In: 2015 IEEE International Conference on Cluster Computing, pp. 96–105, September 2015. https://doi.org/10.1109/CLUSTER.2015.24
11. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications. In: 2012 IEEE International Conference on Cluster Computing (CLUSTER), pp. 191–199, September 2012. https://doi.org/10.1109/CLUSTER.2012.34

raphael.camargo@ufabc.edu.br