Simultaneous Multi-Level Divisible Load Balancing for Heterogeneous Desktop Systems

Aleksandar Ilic and Leonel Sousa
INESC-ID, IST/TULisbon
Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
Email: {ilic,las}@sips.inesc-id.pt

***Abstract*—In this paper, we propose an algorithm for efficient divisible load balancing across all processing devices available in a heterogeneous desktop system. The proposed algorithm allows to achieve simultaneous load balancing at different execution levels, namely between execution subdomains defined with several processing devices, and between devices in each subdomain. Moreover, the algorithm builds partial performance models for each execution subdomain, using the minimal set of approximation points determined during the algorithm run, while converging towards the optimal multi-level load distributions. The proposed approach was experimentally evaluated in a real desktop system with a quad core CPU and two GPUs, for matrix multiplication. Experimental results show the ability of the algorithm to provide significant performance improvements with very low scheduling overhead when compared to similar scheduling approaches.**

***Keywords*-divisible load scheduling; load balancing; heterogenous systems**

I. Introduction

Modern desktop systems are complex heterogeneous platforms capable of providing high computing power by merging the execution space of general purpose processors (CPUs) and specialized accelerators, such as graphics processing units (GPUs). Advances in multi-core CPU designs and accelerator architectures allow to unlock desktop performance which was obtainable for computing clusters only a while ago. However, approaching to the peak performance of a single desktop system is not an easy task due to the significant architectural differences between processing devices that yield highly disproportional device performances.

Common practice in heterogenous desktop computing is to employ an accelerator by devoting a single CPU core to control its execution. The benefits of underusing the core's computation potentials are usually justified by the fact that the accelerator might provide the performance of an order of magnitude higher than the performance of a core. Hence, the devoted core just issues the commands to commence on-accelerator execution, but most of the time it is idle, waiting for the accelerator to finish with processing. During this idle period, the core might be successfully employed to perform certain portions of the computation that can be overlapped with the execution on the accelerator. However, the amount of computation to be given to the core and accelerator must be carefully chosen to achieve load balancing between both devices, such that they finish their computations at the same time. For modern desktop environments with several accelerators, it must be further guaranteed that not only the execution within each core-accelerator pair is balanced, but also the execution between the execution pairs. This example serves as the major motivation for the research conducted herein, however the methods proposed in this paper are applicable to a wider range of scheduling problems and environments. To that extent, we refer to an execution subdomain which consists of a subset of available compute devices, as a processing group.

In this paper, we consider the problem of multi-level scheduling discretely divisible load (DL) applications on heterogeneous desktop platforms that allows to efficiently use all available computing power. The DL model [1] represents parallel computations that can be divided into pieces of arbitrary sizes, which can be processed independently with no precedence constraints. The algorithm proposed herein allows to achieve simultaneous load balancing at each execution level, i.e. between the processing groups, and between the devices in each processing group, such that the overall application execution time is minimal. Moreover, it automatically discovers the minimal set of points required to build the partial estimations of the groups' performance, which are used to decide upon the multi-level load distributions. The time taken to determine the load distributions impose very low scheduling overhead, thus qualifies the proposed algorithm as very suitable for online scheduling.

Current state of the art approaches in DL scheduling mainly consider the problems in environments with a single-level disposition of processing elements [2]–[5]. Several authors have only partially addressed the problems considered herein by pointing out potential advantages of exploiting CPU and GPU execution overlap for specific applications [6]–[9]. The authors in [10] derive metrics to allow execution overlap for a single core-GPU pair for matrix multiplication. However, to the best of our knowledge this is one of the first works that targets simultaneous multi-level divisible load scheduling problems in heterogeneous desktop environments.

The rest of the paper is organized as follows. In Section

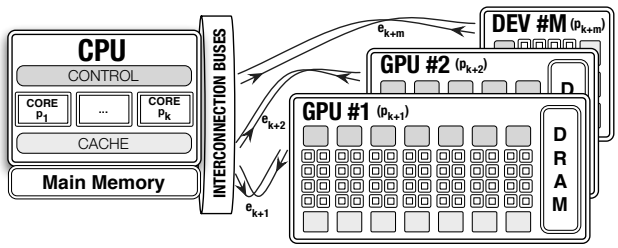


Figure 1. Heterogeneous desktop system.

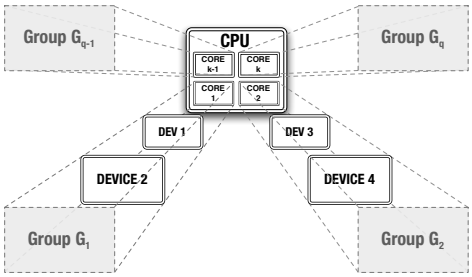


Figure 2. Abstraction of processing groups.

2 we discuss single-level DL balancing algorithm on which our approach is based upon, followed by a description of the system model and problem formulation in Section 3. In Section 4 we propose the algorithm for simultaneous multi-level DL balancing. The experimental results are presented in Section 5 and concluding remarks are given in Section 6.

II. Single-Level Divisible Load Balancing Algorithm

The Single-Level Divisible Load Balancing Algorithm (SLBA) is a two-step algorithm, initially proposed in [3], which relies on functional performance models to find the optimal distribution of N total loads between processors in a single level hierarchy for a heterogeneous distributed system. SLBA algorithm takes as *inputs* the total problem size (N), the number of processors (p), and their computation performance models ($s_i(x)$), and it *outputs* per-processor load distributions x_i ($1 \leq i \leq p$). This algorithm is based on the observation that the optimal distribution x_i lies on a straight line that passes through the origin of the coordinate system and intersects the processors' absolute performance models $s_i(x)$, such that:

$$\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \cdots = \frac{x_p}{s_p(x_p)}; \ \sum_{i=1}^{p} x_i = \mathrm{N} \tag{1}$$

The first step of the algorithm commences by defining the upper and lower bounds of the solution search space, and it converges towards the optimal distribution by bisecting the angle between these two boundaries and by assigning the bisection as one of the search limits in each iteration. Once the distribution that satisfies Eq. 1 is found, the algorithm proceeds to the second step (refinement). The second step is needed due to the rounding procedure in the first step, as per-processor loads must be discreet values. The refinement procedure iteratively increments the processor loads until assigning the total problem size (N) to all processors. In each iteration only one processor load is incremented such that minimal parallel application execution time is achieved. The algorithm also makes realistic assumptions on the shape of performance functions and is known for its relatively low complexity $O(p \times \log_2 N)$.

III. System Model and Problem Formulation

Let $\mathcal{D} = (\mathcal{A}, (P, E), \Gamma, \psi_t, \psi_g)$ be a DLS system, where the divisible load $\mathcal{A}$ is distributed and processed on a heterogeneous desktop system (P, E), depicted in Fig. 1. A set of $k+m$ processing devices is defined as $P = P_M \cup P_W$, where $P_M = \{p_1, ..., p_k\}$ is a set of k cores on the CPU master (positioned at the center of the star), and $P_W = \{p_{k+1}, ..., p_{k+m}\}$ is a set of m "distant workers". The term "distant worker" is used herein to designate a processing device, such as GPU, connected to the master via a communication link. $E = \{e_{k+1}, ..., e_{k+m}\}$ is a set of m links that connect the master to the P_W distant workers.

Let Γ be a set of q processing groups $G_i = (P_i, E_i) \subset (P, E)$ ($1 \leq i \leq q$) representing different execution subdomains. Each G_i processing group comprises a $P_i = P_{M_i} \cup P_{W_i}$ subset of k_i cores and m_i distant workers from the P set, and a E_i set of m_i communication links for each distant worker in P_{W_i} set. Figure 2 depicts one possible arrangement of processing elements in a typical heterogeneous desktop system, forming q different processing groups. For example, one can read that *Group* G_1 deals with the the execution subdomain defined with *Core 1*, *Device 1* and *Device 2* (due to presentation simplicity, communication lines are not shown in Fig. 2).

Initially, the total load N of the application $\mathcal{A}$ is stored at the master, which can be split into load fractions of an arbitrary size x. In contrast to the usual DL practice where computation/communication time is modeled with linear or affine functions of the load size x [2], we use more realistic, dynamically built performance functions to describe these relations. In detail, for each load fraction x that is processed/transferred in a certain time t, we calculate the performance as x/t in order to construct $f_d : \mathbb{N} \to \mathbb{R}_+$ function which is continuously extended by piece-wise linear approximation to a performance function $f_c : \mathbb{R}_+ \to \mathbb{R}_+$ ($f_d(x) = f_c(x), \forall x \in \mathbb{N}$), such as in [4]. Hence, $\psi_t(x)$ function is used to describe the total performance of a device which is calculated as ratio between the load size x and the total time taken to distribute and process the load and to return the results. $\psi_g(x, \beta)$ functions refer to the processing group performance models and depend on two parameters:

(a) the total load x processed on all devices in parallel within a processing group; and (b) β load distribution between devices in a group for which this performance is obtained. Group performance in a single point (x, β) is usually calculated according to the maximum time taken to process in parallel the loads from β on all devices in a group. It is worth to emphasize that these models encapsulate both the time taken to process and the time taken to communicate the loads across different devices in a processing group.

The problem that we tackle herein is how to simultaneously find the load distributions at each level of processing hierarchy, i.e. within devices in each G_i processing group, and between G_i groups in a Γ set, such that the overall parallel application execution time is minimal. This load balancing strategy should, at the same time, partition the total application load N into fractions $\alpha = \{\alpha_1, ..., \alpha_q\}$ to be processed in parallel in each processing group $G_1, ..., G_q$, and to find the cross-device β load distributions for each G_i group, such that parallel executions across different execution subdomains are as balanced as possible. Within a group, distribution $\beta_i = \{\beta_{i,1}, ..., \beta_{i,k_i}, ...\beta_{i,k_i+m_i}\}$ should be determined in order to hold the loads to be assigned to each device in a G_i group, such that $\sum_{j=1}^{k_i+m_i} \beta_{i,j} = \alpha_i$.

IV. Multi-Level Divisible Load Balancing Algorithms

Due to the lack of the current state of the art approaches, we present herein three methods for multi-level DL balancing in heterogeneous desktop environments. The first two approaches are intuitive, non-simultaneous two-step procedures that rely on fully built performance models for each processing group in the system. The third approach, which is the major contribution of this paper, allows to achieve simultaneous multi-level load balancing across processing groups and between devices within each group. The presented methods rely on SLBA procedure, thus adhere to the assumptions on the shape of performance function from [3]. The performance function should have one intersection point with the line that passes through the origin of coordinate system, and it has to be monotonically increasing and concave until a certain point, after which it should be monotonically decreasing.

A. Multi-Level Load Balancing Algorithms using Full Group Performance Models (MLFBA)

This class of algorithms follows a two-step procedure to determine multi-level load distributions. In the first step, the optimal β distributions are decided for devices in each processing group and the full group performance models are built. The second step deals with the load balancing between the processing groups by determining the α distribution according to the group performance models.

In the first step, it is needed to determine the cross-device β load distributions that allow to achieve the best group performance for a certain problem size. This practically means that for a single point x_t in G_i group performance model $\psi_{g_i}(x, \beta)$, it is needed to conduct a series of tests over the different combinations of β_t load distributions (with loads that sum up to the value of x_t), and to permute these load values across the devices in the processing group. From this set of distributions, only one β_t distribution contributes to $\psi_{g_i}(x, \beta)$ model. The selected β_t distribution should allow to achieve the maximum group performance in point x_t, and it is the one that provides minimum execution time from the maximum parallel cross-device execution times for each β_t distribution run for the size of x_t. This exhaustive search procedure, referred here as MLFBA1, requires $\sum_{i=1}^{|\Gamma|} \sum_{l=k_i+m_i}^{N} \frac{l!}{(l-k_i-m_i)!}$ tests to obtain $N \times |\Gamma|$ points to fully model all processing groups, where the tests are run for each group and for a range of problem sizes.

MLFBA2 is the another approach that might be used to construct the full group performance models in significantly less time. It relies on SLBA procedure and total device performance models $\psi_t(x)$. In order to obtain a single x_t point in the group performance model $\psi_{g_i}(x, \beta)$, it is needed to run SLBA for a problem size of x_t over the total performance models $\psi_t(x)$ of devices in a G_i group. The SLBA will provide the optimal β_t distribution that contributes to a single x_t point in a $\psi_{g_i}(x, \beta)$ model. To build the full models for each group, this procedure is needed to be repeated over a range of problem sizes (up to N), and for each group in the system. This gives the total of $\sum_{i=1}^{|\Gamma|}(k_i + m_i) \log N!$ steps to obtain $N \times |\Gamma|$ points to fully model the processing groups.

After the full group performance models are constructed, both approaches proceed to the second step. In this step, SLBA is run for a problem size of N over the group performance models $\psi_{g_i}(x, \beta)$ $(1 \leq i \leq q)$ in order to obtain the α_i load distributions. During the execution, the devices from each group are assigned with loads from β_i distributions found in point $\psi_{g_i}(\alpha_i, \beta_i)$.

The rationale behind constructing the full group models lies in the fact that for each x_t point in a group model, the optimum β_t distribution must be known prior to proceeding to the second step of the algorithm. This requirement is still valid even when partially built models are used to describe device performance, due to the fact that highly disproportional performance of devices makes predicting the group performance almost impossible.

B. Multi-Level Simultaneous Load Balancing Algorithm (MLSBA)

The MLSBA algorithm simultaneously achieves the multi-level load balancing across processing groups and between devices in each group by dynamically building the partial estimations of the group performance with the minimal number of points needed to provide the load distributions. The algorithm consists of two phases: (a) *Phase 1* of

the algorithm finds the multi-level distributions, and (b) *Phase 2* provides the refinement procedure for the obtained distributions. These phases correspond to SLBA algorithms in [3], and need to be serially performed to achieve multi-level load balancing.

Phase 1. Simultaneous determination of load distributions at multiple execution levels:

1) For each processing group G_i $(1 \le i \le q)$ with the number of devices equal to one, assign the total device performance model $\psi_t(x)$ as the group performance model $\psi_{g_i}(x,\beta)$, such that $\forall(x_j \in \psi_t(x) \wedge (x_l,\beta_l) \in \psi_{g_i}(x,\beta) \wedge j = l)\ x_l = \beta_l = x_j$. x_j and x_l represent the approximation points for the total performance of a device and a group, respectively;
2) Find performance $\psi_{g_i}(\frac{N}{P},\beta_{N/P})$ in point $\frac{N}{P}$, for each processing group G_i $(1 \le i \le q)$.
   If $(\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \frac{N}{P})$ there is no exact approximation point $x_i = \frac{N}{P}$ in $\psi_{g_i}(x,\beta)$ group performance model **then** insert and update the current approximation of the group performance $\psi_{g_i}(x,\beta)$ in an integer point $\frac{N}{P}$ using *Procedure 1*; **else** directly obtain group performance $\psi_{g_i}(\frac{N}{P},\beta_{N/P})$;
3) Draw the upper line U through the points $(0,0)$ and $(\frac{N}{P},\max_i\{\psi_{g_i}(\frac{N}{P})\})$, and the lower line L through the points $(0,0)$ and $(\frac{N}{P},\min_i\{\psi_{g_i}(\frac{N}{P})\})$ $(1 \le i \le q)$.
4) Find $x_i^{(U)}$ and $x_i^{(L)}$ intersections of the upper U and the lower L lines with current partial performance estimations $\psi_{g_i}(x)$ for each G_i group.
   If group G_i contains more than one device:
   a) **If** $((\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lfloor x_i^{(U)}\rfloor) \wedge (\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lceil x_i^{(U)}\rceil))$ points $\lfloor x_i^{(U)}\rfloor$ and $\lceil x_i^{(U)}\rceil$ are not exact approximation points in a group $\psi_{g_i}(x,\beta)$ performance model **then** update group performance model $\psi_{g_i}(x,\beta)$ for $x_i^{(U)}$ with *Procedure 2*, recalculate intersection $x_i^{(U)}$ according to updated group model and repeat step 4a; **else** go to step 4b;
   b) **If** $((\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lfloor x_i^{(L)}\rfloor) \wedge (\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lceil x_i^{(L)}\rceil))$ points $\lfloor x_i^{(L)}\rfloor$ and $\lceil x_i^{(L)}\rceil$ are not exact approximation points in a group $\psi_{g_i}(x,\beta)$ performance model **then** call *Procedure 2* to update $\psi_{g_i}(x,\beta)$ for $x_i^{(L)}$, recalculate intersection $x_i^{(L)}$ according to updated group model and repeat step 4b;
   else calculate intersections for groups with one device;
5) **If** $x_i^{(L)} - x_i^{(U)} \ge 1$ **then** go to step 6; **else** to step 8;
6) Bisect the angle between U and L lines with the line B. Calculate coordinates $x_i^{(B)}$ of the intersection points of line B with current partial estimations $\psi_{g_i}(x,\beta)$ for each group.
   If group G_i contains more than one processing device:
   a) **If** $((\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lfloor x_i^{(B)}\rfloor) \wedge (\neg\exists x_i \in \psi_{g_i}(x,\beta)\ x_i = \lceil x_i^{(B)}\rceil))$ points $\lfloor x_i^{(B)}\rfloor$ and $\lceil x_i^{(B)}\rceil$ are not exact approximation points in $\psi_{g_i}(x,\beta)$ performance model; **then** update group performance model $\psi_{g_i}(x,\beta)$ for $x_i^{(B)}$ with *Procedure 2*, recalculate intersection $x_i^{(B)}$ with the updated model and repeat step 6a;
   else calculate intersections for groups with one device;
7) **If** sum $\sum_{i=1}^{|\Gamma|} x_i^{(B)} \le N$ **then** U = B **else** L = B; Repeat step 4;
8) Approximate α_i such that $\alpha_i = \lceil x_i^{(U)}\rceil$;

Procedure 1. *Update group model from an integer point*. Let $x_c \in \mathbb{N}$ be an integer point for which the current partial estimation of group performance $\psi_g(x,\beta)$ needs to be updated, and let κ be the number of approximation points used for piece-wise linear approximation of $\psi_g(x,\beta)$. The update procedure is as follows:

1) Find the per-device $\beta_{x_c} = \{\beta_{x_c,i}\}_{i=1}^{|\beta_{x_c}|}$ distribution with SLBA algorithm for the problem size of x_c using the total device performance models $\psi_{t_i}(x)$ for each device i in the processing group;
2) Update $\psi_g(x,\beta)$ model in point (x_c,β_{x_c}) with the total performance of $\psi_c = x_c/\max_i\{\frac{\beta_{x_c,i}}{\psi_{t_i}(\beta_{x_c,i})}\}$:
   If $\kappa == 0$ **then** insert the first point in $\psi_g(x,\beta)$, such that $\psi_g(x_c,\beta_{x_c}) = \psi_c$ and approximate the group performance as ψ_c in interval $(0,\infty)$;
   else find first j for which $x_j \ge x_c$ $(x_j \in \psi_g(x,\beta))$ and insert point $\psi_g(x_c,\beta_c) = \psi_c$ such that:
   a) **if** $j < \kappa$ **then** update the current $\psi_g(x,\beta)$ model by piece-wise approximations in intervals $(x_{j-1},x_c]$ and $(x_c,x_j]$ in respect to the performance values $\psi_g(x_{j-1},\beta_{j-1})$, ψ_c and $\psi_g(x_j,\beta_j)$;
   b) **if** $j == \kappa$ **then** update $\psi_g(x,\beta)$ performance approximation in interval $(x_\kappa,x_c]$ in respect to the performances in points $\psi_g(x_\kappa,\beta_\kappa)$ and ψ_c; whereas the performance in interval (x_c,∞) is modeled as ψ_c;

Procedure 2. *Update group model from a real point*
Let $\psi_g(x,\beta)$ be a current partial estimation of the group performance, and let point $t \in \mathbb{R}_+$ be a real point value for which the performance model $\psi_g(x,\beta)$ needs to be updated. The update procedure goes as follows:

1) Find values $\lfloor t\rfloor$ and $\lceil t\rceil$, such that $\lfloor t\rfloor, \lceil t\rceil \in \mathbb{N}$.
2) **If** $(\exists x \in \psi_g(x,\beta)\ x = \lfloor t\rfloor)$ there is an exact approximation point in the group performance model $\psi_g(x,\beta)$ with the value of $\lfloor t\rfloor$ **then** go to step 3; **else** update the current performance approximation $\psi_g(x,\beta)$ in integer point $\lfloor t\rfloor$ using *Procedure 1*;
3) **If** $(\exists x \in \psi_g(x,\beta)\ x = \lceil t\rceil)$ there is an exact approximation point in $\psi_g(x,\beta)$ with the value of $\lceil t\rceil$ **then** stop; **else** update model $\psi_g(x,\beta)$ in integer point $\lceil t\rceil$ with *Procedure 1*;

Phase 2. *Refinement procedure*. Iterative incrementing of some α_i until $\alpha_1 + ... + \alpha_q = N$:

1) **If** $\alpha_1 + ... + \alpha_q < N$; **then** go to step 2; **else** stop the algorithm;
2) For each group G_l find performance Ψ_{l+1} in point $\alpha_l + 1$ according to the current group performance approximations $\psi_{g_l}(x)$ $(1 \leq l \leq q)$.
   If group G_l contains more than one device and there is no exact approximation point with value $\alpha_l + 1$ in $\psi_{g_l}(x, \beta)$ performance model $(\neg \exists x_l \in \psi_{g_l}(x, \beta)\ x_l = \alpha_l + 1)$; **then** call *Procedure 1* to insert and update current performance approximation $\psi_{g_l}(x, \beta)$ in integer point $\alpha_l + 1$;
   Obtain Ψ_{l+1} from the group performance such that $\Psi_{l+1} = \psi_{g_l}(\alpha_l + 1, \beta_{l,\alpha_l+1})$;
3) Find $l \in \{1, ..., q\}$ such that $\frac{\alpha_l+1}{\Psi_{l+1}} = \min_{i=1}^{q} \{\frac{\alpha_i+1}{\Psi_{i+1}}\}$;
4) $\alpha_l = \alpha_l + 1$. Repeat step 1;

It is worth to emphasize that MLSBA encapsulate all the functionality of SLBA algorithm and can be considered as its generalization, where SLBA is a special case when each processing group consists of a single device.

Estimation of complexity for MLSBA is beyond the scope of this paper due to a very complex and lengthy derivation procedure. However, it preserves the logarithmic complexity which can be clearly evidenced in Section V.

V. Experimental Results

In order to validate the efficiency of the proposed MLSBA algorithm, we have conducted a series of tests in a real heterogeneous desktop environment consisting of a quad core Intel Core i7 950 processor (CPU) and two NVIDIA GeForce GPUs from different architectures, i.e. GTX580 (GPU0) and GTX285 (GPU1). The execution is setup in four processing groups, where CPU Core0 and GPU0 form processing Group 0, Group 1 comprises CPU Core1 and GPU1, whereas the last two groups contain one of each remaining CPU cores.

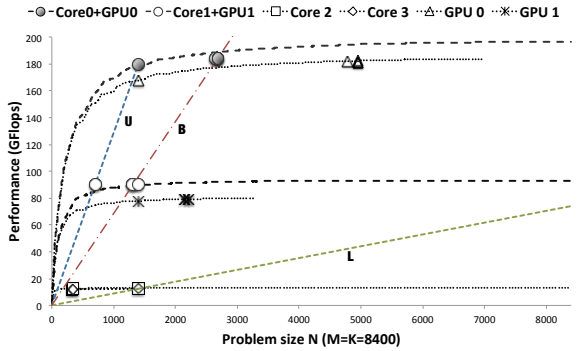
Due to its usual adoption for expressing system/device performance, we selected matrix multiplication ($A_{M\times K} \times B_{K\times N} = C_{M\times N}$) as the testing application. The column-based one dimensional parallelization method was applied, which requires to transfer to/from GPU devices matrix A and portions of B and C matrices [11]. We used optimal, vendor-provided high performance libraries as matrix multiplication kernels, i.e., Intel MKL 10.3 for CPU and NVIDIA CUBLAS 4.1 for GPU devices. In all tested cases no a priori knowledge was used to ease performance modeling of any system resource.

A. MLSBA efficiency for fixed problem size

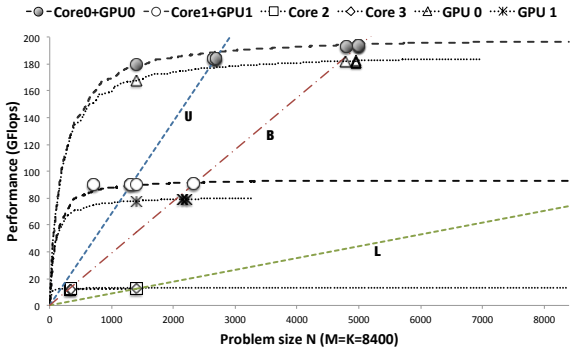
In order to show the process of achieving simultaneous multi-level load balancing and efficient partial group performance modeling with proposed MLSBA algorithm, we performed matrix multiplication of a size $M = N = K = 8400$ in the above-mentioned four processing group execution setup. Due to the fact that the performance models of neither groups nor devices were known a priori, we conducted an iterative routine which perfoms MLSBA in each iteration to decide on load distributions at each execution level. The iterative routine is initialized by assigning P devices with $\frac{N}{P}$ load size to execute for initial device performance modeling. In each subsequent iteration, MLSBA is used to construct partial group performance models according to the current partial estimates of device performance and to decide on load distributions across different execution subdomains. The iterative routine improves the device models with one point per iteration and the complete load balancing procedure stops after achieving desired accuracy at both levels, i.e., between the groups and between devices in each group. It is worth to mention that regardless of the type of algorithm used within the iterative routine (MLSBA of MLFBA2), group performance models are needed to be flushed between iterations due to update of partial device models.

By relying on MLSBA algorithm, a total of four iterations were required to achieve multi-level load balancing. Figure 3 depicts several stages of MLSBA while it converges towards the optimal per-group and per-device distributions and builds the partial group performance approximations in the final, fourth iteration. The dotted lines represent the full models of devices which are obtained using the exhaustive search over the range of problem sizes up to 8400 for each device in the system. The dashed lines represent the full models of processing groups which are obtained by applying the first step of MBFLA2 procedure to the full device models. Those full performance models are depicted in Fig. 3 to show how close MLSBA group approximations are to the real values from the full models. Thus, the load balancing decisions in MLSBA were taken according to the piece-wise linear approximations of the performance models which are built from the approximation points shown in Fig. 3. The GPU performance models take into account the time to transfer matrices to/from GPU devices and it can be noticed that none of the GPU devices was capable of executing a complete problem size of 8400 due to its limited memory.

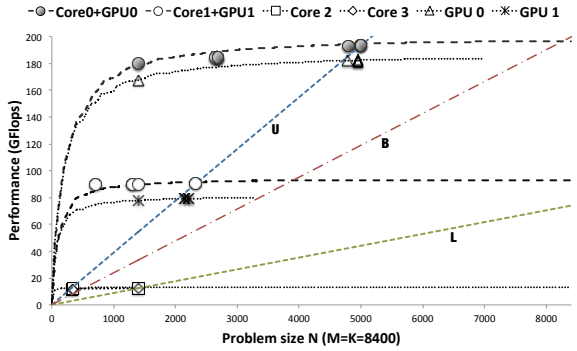
Figure 3(a) shows the initial search space for optimal solution (between U and L lines), which is determined in the first MLSBA stage. The bisection line B halves the search region between U and L lines in Fig. 3(a), but different scaling factors of x and y axes prevents its clear visibility. The group approximation points obtained with *Procedure 2* can be noticed in intersections of line B and group performance models. Figure 3(b) depicts MLSBA in the second stage, where solution search space is reduced by assigning $U = B$, then new B line is drawn, and the number of group approximation points is further improved (in intersections of B line and the group models). Figures 3(c), 3(d) and 3(e) present the next 3 stages of MLSBA where further reductions of the search space and



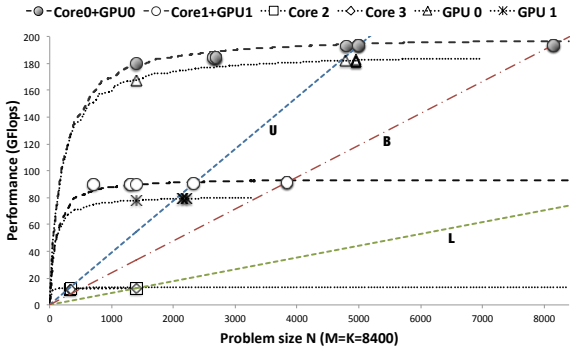
(a) Initial solution search space and partial modeling



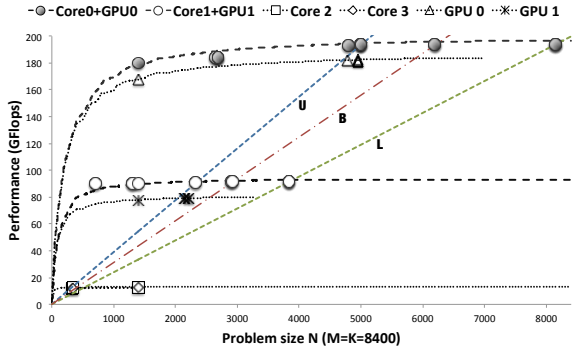
(b) Second stage of MLSBA



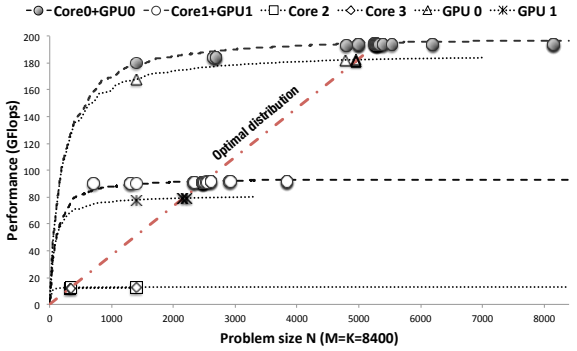
(c) MLSBA in the third stage



(d) Search space and models in fourth stage



(e) Towards optimal distribution in fifth stage



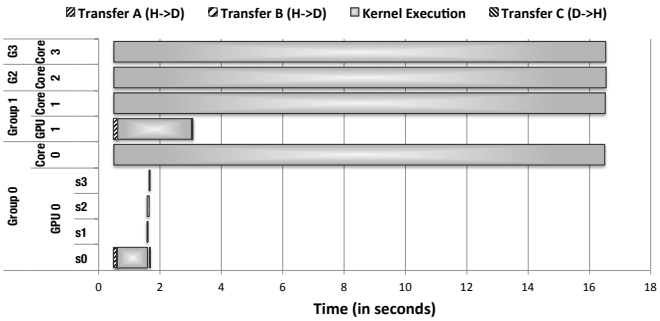
(f) Optimal distribution and models in the final stage

Figure 3. Process of simultaneous multi-level load balancing and partial performance modeling with MLSBA algorithm for parallel matrix multiplication ($M = N = K = 8400$).

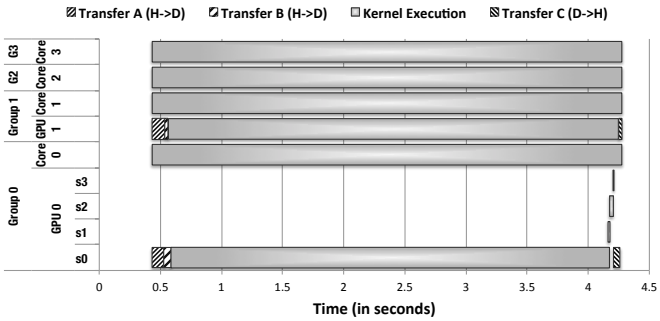
logarithmic growth of number of approximation points for group performance modeling can be seen.

Figure 3(f) shows the final partial group performance models obtained when the optimal distribution is found across each execution subdomain, i.e. between groups and devices within each group. In total 111 points were obtained for group performance modeling: 58 points to model performance of Group 0, Group 1 was modeled with 45 points, and Groups 2 and 3 used 4 points each (due to the fact that these groups contain only a single CPU Core). In contrast, MLFBA procedures need to determine a total of 16808 approximation points to converge towards the same load distributions (Groups 0 and 1 need 8400 points each). In terms of scheduling overhead, MBFLA2 required around 0.8 seconds per iteration, whereas the proposed MLSBA required 0.0009 seconds per iteration (0.9 ms).

Figures 4(a) and 4(b) present the load balancing achieved in the first and the final iteration of iterative routine, respec-



(a) Homogeneous load distribution between devices $\beta_i = \frac{N}{P}$



(b) Multi-level load balancing achieved with MLSBA algorithm

Figure 4. Load balancing analysis when performing parallel matrix multiplication ($M = N = K = 8400$).

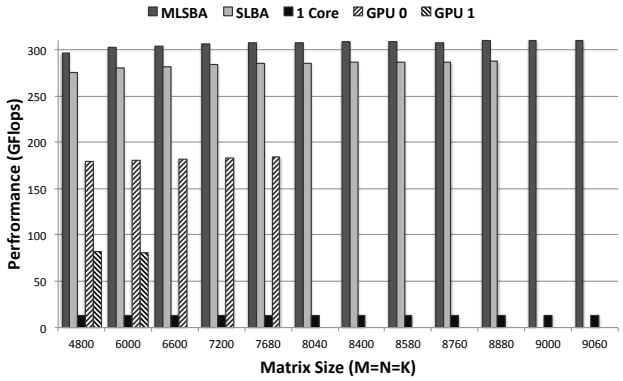


Figure 5. Scalability analysis for matrix multiplication ($M = N = K$).

Table I
SCHEDULING OVERHEAD PER ITERATION FOR MATRIX MULTIPLICATION

Matrix size ($M = N = K$)	**MLSBA** [s]	**MLFBA2** [s]
4800	0.0011	0.29
6000	0.0007	0.45
6600	0.0010	0.54
7200	0.0008	0.64
7680	0.0008	0.73
8040	0.0008	0.76
8400	0.0008	0.80
8580	0.0009	0.91
8760	0.0011	0.95
8880	0.0009	0.97
9000	0.0010	1.01
9060	0.0011	1.02

tively. As expected, homogeneous load distribution between devices allows to achieve load balancing between the groups due to the similar performance of CPU Cores. However, this distribution was not capable of providing adequate load balancing between devices in the groups, i.e. GPU-CPU Core pairs (see Fig. 4(a)). Figure 4(b) depicts the load balancing achieved when running the load distributions obtained with MLSBA in the final, fourth iteration. As it can be seen the execution is balanced at each execution level, and results in an overall reduction of the application execution time of almost 4 times when compared to the homogeneous distribution. Moreover, the proposed algorithm was capable of dealing with the complex behavior of CUBLAS library for GPU0 (NVIDIA Fermi architecture), where a single problem run is automatically split into several sequential kernel executions (presented in Fig. 4(b) as s0, s1, s2 and s3). In terms of performance, the current state of the art approaches with a single-level disposition of processing elements (SLBA) are capable of discovering a total of 285 Gflops, whereas with the proposed MLSBA approach we achieved around 309 Gflops for the total system performance for the same matrix multiplication problem size.

B. Scalability Results and Scheduling Overhead Analysis

In order to demonstrate the scalability of the proposed approach, we ran series of matrix multiplication tests for different problem sizes. The obtained overall system performance with MLSBA approach is depicted in Fig. 5 and compared with state of the art SLBA approach and with the performance obtained at the single device level, i.e. GPU0, GPU1 and CPU Core. In detail, MLSBA succeeded in providing the performance improvements for all problem sizes, which results in an average improvement of 22 Gflops comparing to SLBA approach. This improvement is mainly due to the possibility of MLSBA to employ all CPU Cores for execution, while in SLBA a single CPU Core must be devoted to control the execution on a GPU device. Moreover, with MLSBA it was possible to successfully execute the problem sizes which are not executable neither on a single GPU device (above 7680 for GPU0 and 6000 for GPU1) nor with SLBA approach (above 8880) due to the limited memory of GPU devices.

Furthermore, achievable performance improvement is not limited by the efficiency of MLSBA method, but by the performance of CPU cores. The results presented in Fig. 5, where the maximum obtainable performance for a CPU core

was about 12 Gflops, show that MLSBA was really close to achieve the system peak performance for all problem sizes. However, there are several limiting factors that do not allow to achieve this performance, such as problem size granularity for discrete divisible load scheduling, execution time fluctuations and memory hierarchies, optimization strategies in vendor's libraries, and finally optimal load distribution line. The optimal distribution lies on a straight line that passes through the origin of the coordinate system and intersects the performance models, which means that for highly disproportional device performances (such in case of GPUs and CPU cores) the devices with lower performance can be assigned with the loads for which it might not be possible to operate on a per device peak performance (see Fig. 3). Moreover, with further advances in the CPU core design even higher performance improvements are expected. For example, by relying on Intel AVX instructions for the new generations of processors it is possible to speed up the matrix multiplication for about 2 times on a single core using the MKL library.

Due to the fact that performance comparisons between MLSBA and MLFBA2 approaches are not valuable (as they converge towards the very same, optimal distributions), we conducted series of tests to validate those two approaches in terms of induced scheduling overhead. As shown in Table I, scheduling overhead of MBFLA2 grows with the problem size as it needs to construct group performance models for each point in a range up to the total problem size. However, MLSBA algorithm is capable of providing the same distributions in significantly less time (in average around 0.9 ms per iteration), which qualifies it for online scheduling. On the other hand, for certain problem sizes (such as 8400) the overall scheduling time of MLFBA2 algorithm, for all four iterations of iterative routine, might take the same amount of time as the total application run in the final iteration. In terms of the number of approximation points for group performance modeling, MLSBA was capable of determining the minimum set of points required for simultaneous optimal multi-level load balancing that range from 53 to 62 points for Group 0 and from 46 to 51 points for Group 1 for different problem sizes considered in Table I. In contrast, MLFBA2 required to detect a full range of approximation points for group models that varies from 4800 to 9000. For example, for a problem size of 9000, MLSBA needed in total 114 points to model all four groups, while MLFBA2 used 18008 approximation points to achieve the optimal distribution.

VI. Conclusions

This paper proposes, for the first time, an algorithm for simultaneous multi-level balancing of discretely divisible load applications in heterogeneous desktop systems. The proposed algorithm does not only efficiently distributes the loads between different execution subdomains defined with several processing devices, but it also achieves load balancing between the devices within each subdomain. During the algorithm run, the partial models for subdomains are built using the minimal set of points required to converge towards the optimal load distributions. The efficiency of the proposed approach was validated in a real desktop system with four CPU cores and two GPUs with matrix multiplication as the testing application. The experimental results show that the presented algorithm was capable of delivering significant performance improvements when compared to the similar scheduling strategies. Moreover, the proposed approach induces very low scheduling overhead, this it is perfectly suitable for online scheduling.

Acknowledgment

This work was supported by FCT through the PIDDAC Program funds (INESC-ID multiannual funding) and a fellowship SFRH/BD/44568/2008.

References

[1] B. Veeravalli, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Computing*, vol. 6, pp. 7–17, 2003.

[2] O. Beaumont *et al.*, "Scheduling divisible loads on star and tree networks: results and open problems," *IEEE Trans. Parallel Distributed Systems*, vol. 16, p. 2005, 2003.

[3] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 76–90, 2007.

[4] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity," in *HeteroPar 2010*, 2010.

[5] A. Ilic and L. Sousa, "Scheduling divisible loads on heterogeneous desktop systems with limited memory," in *HeteroPar*, 2011.

[6] S. Venkatasubramanian, R. W. Vuduc, and n. none, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *Supercomputing*. ACM, 2009, pp. 244–255.

[7] J. White and J. Dongarra, "Overlapping computation and communication for advection on hybrid parallel computers," in *IPDPS*, 2011, pp. 59 –67.

[8] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortı, and A. Remón, "A mixed-precision algorithm for the solution of lyapunov equations on hybrid cpu-gpu platforms," *Parallel Comput.*, vol. 37, pp. 439–450, August 2011.

[9] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid cpugpu architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 1316, pp. 1490 – 1508, 2011.

[10] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *GPGPU*, vol. 383. ACM, 2009, pp. 46–51.

[11] A. Ilic and L. Sousa, "CHPS: an environment for collaborative execution on heterogeneous desktop systems," *Int. J. of Networking and Computing*, vol. 1, no. 1, 2011.