DOI: 10.14529/jsfi140304

Heterogeneous Parallel Computing: from Clusters of Workstations to Hierarchical Hybrid Platforms

A.L. Lastovetsky[1]

The paper overviews the state of the art in design and implementation of data parallel scientific applications on heterogeneous platforms. It covers both traditional approaches originally designed for clusters of heterogeneous workstations and the most recent methods developed in the context of modern multicore and multi-accelerator heterogeneous platforms.

Keywords: parallel computing, heterogeneous computing, data partitioning.

Introduction

High performance computing systems become increasingly heterogeneous and hierarchical. A typical compute node integrates multiple (possibly heterogeneous) cores as well as hardware accelerators such as Graphics Processing Units. The integration is often hierarchical. The motivation behind such complicated architecture is to make these systems more energy efficient. The energy consideration is paramount as future large-scale cluster infrastructures will have to have hundreds of thousands of compute nodes to solve Exascale problems and would not be energy sustainable if nodes of traditional architecture were used. Future large-scale systems will exhibit multiple forms of architectural and non-architectural heterogeneity as well as mean-time-to-failure of minutes. How to develop parallel applications and software that efficiently utilize highly heterogeneous and hierarchical computing and communication resources, while scaling them towards Exascale, maintaining a sustainable energy footprint, and preserving correctness are highly challenging and open questions.

Heterogeneous parallel computing is the area that emerged in 1990s to address the challenges posed by ever increasing heterogeneity and complexity of the HPC platforms. This paper overviews the development of heterogeneous parallel computing technologies as they followed the evolution of heterogeneous HPC platforms from simple single-switched heterogeneous clusters of (uniprocessor) workstations to modern hierarchical clusters of heterogeneous hybrid nodes. It mainly focuses on the design of fundamental data partitioning algorithms supporting the development of data parallel applications able to automatically tune to the executing heterogeneous platform achieving optimal performance (and energy) efficiency. Data parallel applications are the main target of parallel computing technologies because they dominate the scientific and engineering computing domain, as well as the emerging domain of large-scale ("Big") data analytics.

Optimization of data parallel applications on heterogeneous platforms is typically achieved by balancing the load of the heterogeneous processors and minimizing the cost of moving data between the processors. Data partitioning algorithms solve this problem by finding the optimal distribution of data between the processors. They typically require a priori information about the parallel application and platform. Data partitioning is not the only technique used for load balancing. Dynamic load balancing, such as task queue scheduling and work stealing [5, 9, 26, 39–41] balance the load by moving fine-grained tasks between processors during the calculation. Dynamic algorithms do not require a priori information about execution but may incur significant

[1]University College Dublin

communication overhead on distributed-memory platforms due to data migration. At the same time, dynamic algorithms often use static data partitioning for their initial step to minimize the amount of data redistributions needed. For example, in the state-of-the-art load balancing techniques for multi-node, multicore, and multi-GPU platforms, the performance gain is mainly due to better initial data partitioning. It was shown that even the static distribution based on simplistic performance models (single values specifying the maximum performance of a dominant computational kernel on CPUs and GPUs) improves the performance of traditional dynamic scheduling techniques by up to 250% [44]. In this overview we focus on parallel scientific applications, where computational workload is directly proportional to the size of data, and dedicated HPC platforms, where: (i) the performance of the application is stable in time and is not affected by varying system load; (ii) there is a significant overhead associated with data migration between computing devices; (iii) optimized architecture-specific libraries implementing the same kernels may be available for different computing devices. On these platforms, for most scientific applications, static load balancing algorithms outperform dynamic ones because they do not involve data migration. Therefore, for the type of applications and platforms we focus on, data partitioning is the most appropriate optimization technique.

One very important aspect of optimization of parallel applications on distributed-memory heterogeneous platforms – optimization of their communication cost, is not covered in this paper. A recent analytical overview of methods for optimization of collective communication operations in heterogeneous networks can be found in [21].

1. Optimization of parallel applications on heterogeneous clusters of workstations

1.1. Data partitioning algorithms based on constant performance models

Since the late 1990s, when the first pioneering works in the field were published, the design of heterogeneous parallel algorithms has made a significant progress. At that time, the main target platform for the heterogeneous parallel algorithms being developed was a heterogeneous cluster of workstations, and the simplest possible performance model of this platform was used in the algorithm design. Namely, it was seen as a set of independent heterogeneous (uni)processors, each characterized by a single positive number representing its speed. The speed of the processors can be absolute or relative. The absolute speed of the processors is understood as the number of computational units performed by the processor per one time unit. The relative speed of the processor can be obtained by the normalization of its absolute speed. While this performance model has no communication-related parameters, it still allows for optimization of the communication cost through the minimization of the amount of data moved between processors. This model is also known as Constant Performance Model, or CPM.

Using the CPM, a fundamental problem of optimal distribution of independent equal units of computation over a set of heterogeneous processors was formulated and solved in [7]. The algorithm [7] solving this problem is of complexity $O(p^2)$ and only needs relative speeds. This algorithm is a basic building block in many heterogeneous parallel and distributed algorithms.

This is typical in the design of heterogeneous parallel algorithms that the problem of distribution of computations in proportion to the speed of processors is reduced to the problem of partitioning of some mathematical objects, such as sets, matrices, graphs, etc. Most of the CPM-based algorithms designed so far have been aimed at numerical linear algebra. For exam-

ple, the problem of LU factorization of a dense matrix A was reduced to the problem of optimal mapping of its column panels $a_1, \ldots, a_n$ to p heterogeneous processors, and the latter problem was further reduced to the problem of partitioning of a well-ordered set (whose elements represent the column panels). Two efficient algorithms solving this partitioning problem have been proposed — the Dynamic Programming (DP) algorithm [7, 10] and the Reverse algorithm [34]. The latter is more suitable for extension to more complex heterogeneous performance models. Other algorithms of partitioning of well-ordered sets, e.g. [6], do not guarantee the return of an optimal solution.

As matrices are probably the most widely used mathematical objects in scientific computing, most of data-partitioning studies deal with them. Matrix partitioning problems occur during the design of parallel linear algebra algorithms for heterogeneous platforms. A typical heterogeneous linear-algebra algorithm is designed as a modification of its homogeneous prototype, and its design is eventually reduced to the problem of optimally partitioning a matrix over heterogeneous processors. From the partitioning point of view, a dense matrix is an integer-valued rectangular. Therefore, if we are only interested in an asymptotically optimal solution (which is typically the case), the problem of its partitioning can be reduced to a problem of the partitioning of a real-valued rectangle.

In a general form, the related geometrical problem has been formulated as follows [8]: Given a set of p processors P_1, P_2, ..., P_p, the relative speed of each of which is characterized by a positive constant, s_i, partition a unit square into p rectangles so that:

- there is a one-to-one mapping between the rectangles and the processors;
- the area of the rectangle allocated to processor P_i is equal to s_i;
- the partitioning minimizes the sum of half-perimeters of the rectangles.

This formulation is motivated by the SUMMA matrix multiplication algorithm [23] and aimed at balancing the load of the processors and minimization of the total volume of data communicated between the processors. Fig. 1 shows one iteration of the heterogeneous SUMMA algorithm assuming that matrices A, B and C are identically partitioned into rectangular submatrices. At each iteration of the main loop, pivot block column of matrix A and pivot block row of matrix B are broadcast horizontally and vertically, then all processors update their own parts of matrix C in parallel. The blocking factor b is a parameter used to adjust the granularity of communications and computations [13], whose optimal value can be found experimentally.

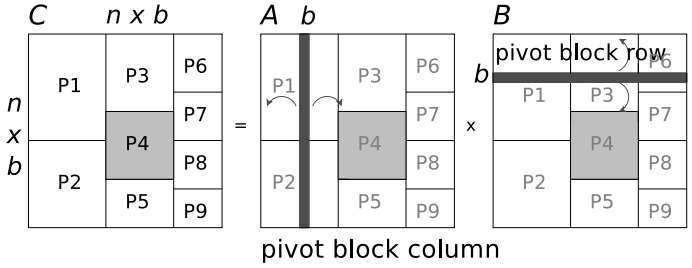


Figure 1. Heterogeneous parallel matrix multiplication

This geometrical partitioning problem is NP-complete [8], but many restricted and practically important versions of this problem have been efficiently solved. The least restrictive is probably the column-based problem looking for an optimal partitioning, the rectangles of which make up columns as illustrated in Fig. 2. An algorithm of the complexity $O(p^3)$ was proposed in [8]. More restricted forms of the column-based geometrical partitioning problem have also

been addressed. The pioneering result in the field was a linear algorithm [27] additionally assuming that the number of columns c in the partitioning and the number of rectangles in each column are given. A column-based partitioning with the same number of rectangles in each column is known as a grid-based partitioning. An algorithm of the complexity $O(p^{3/2})$ solving the grid-based partitioning problem was proposed in [29].

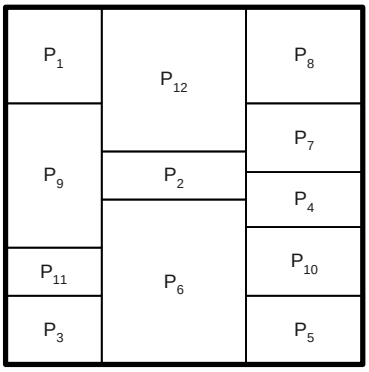


Figure 2. Column-based partitioning of the unit square into 12 rectangles. The rectangles of the partitioning form three columns

A partitioning whose rectangles make both columns and rows is known as a Cartesian partitioning. It is attractive from the implementation point of view because of its very simple and scalable communication pattern. However, the related partitioning problems are very difficult and very little has been achieved in addressing them so far [7].

More recent research [19, 20] challenged the optimality of the rectangular matrix partitioning. Using a specially developed mathematical technique and five different parallel matrix multiplication algorithms, it was proved that the optimal partition shape can be non-rectangular, and the full list of optimal shapes for the cases of two and three processors was identified. Fig. 3 shows these for the case of three processors. The performance model used in this work combined the CPM and the Hockney communication model [24]. These results have a potential to significantly improve the performance of matrix computations on platforms that can be modeled by a small number of interconnected heterogeneous abstract processors, such as hybrid CPU/GPU nodes and clusters of clusters.

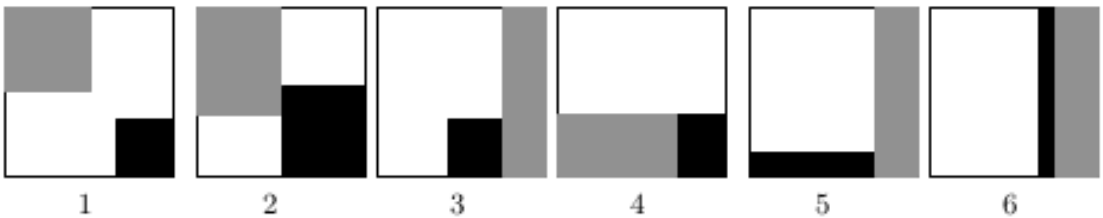


Figure 3. The candidate partition shapes previously identified as potentially optimal three processor shapes. Processors P, R, and S are in white, grey, and black, respectively. (1) Square Corner (2) Rectangle Corner (3) Square Rectangle (4) Block 2D Rectangular (5) L Rectangular (6) Traditional 1D Rectangular

Significant work has been done in partitioning algorithms for graphs, which are then applied to sparse matrices and meshes, the mathematical objects widely used in many scientific applications, e.g. computational fluid dynamics. Algorithms implemented in ParMetis [28],

SCOTCH [12], JOSTLE [45] reduce the number of edges between the target subdomains, aiming to minimize the total communication cost of the parallel application. Algorithms implemented in Zoltan [11], PaGrid [4] try to minimize the execution time of the application. All these graph partitioning libraries use performance models combining the CPM and the Hockney model. The models have to be provided by the users.

1.2. Data partitioning algorithms based on functional performance models

The CPM can be a sufficiently accurate approximation of the performance of heterogeneous processors executing a data parallel application if: (i) the processors are general-purpose and execute the same code, (ii) the local tasks are small enough to fit in the main memory but large enough not to fully fit in the processor cache. However, if we consider essentially heterogeneous processors using different code to solve the same task locally, or allow the tasks to span different levels of memory hierarchy on different processors, then the relative speed of the processors can significantly differ for different task sizes. In these situations, the CPM becomes inaccurate, and its use can lead to highly imbalanced load distribution [16]. To address this challenge, a functional performance model (FPM) [35, 37, 38] was proposed. The FPM represents the speed of a processor by a function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application. The speed is defined as the number of computation units processed per second. The computation unit can be defined differently for different applications. The important requirement is that its size (in terms of arithmetic operations) should not vary during the execution of the application. One FLOP is a simplest example of computation unit.

The fundamental problem of optimal distribution of n independent equal units of computation between p heterogeneous processors represented by their speed functions was formulated, and very efficient geometrical algorithms (of complexities $O(p^2 \log_2 n)$ and $O(p \log_2 n)$) solving this problem under different assumptions about the shape of the speed functions were proposed [31, 35]. These algorithms are based on the following observation. Let the speed of processor P_i be represented by continuous function $s_i(d) = \frac{d}{t_i(d)}$, where $t_i(d)$ is the execution time for processing of d computation units on the processor P_i. Then the optimal solution of this problem, which balances the load of the processors, will be achieved when all processors execute their work within the same time: $t_1(d_1) = ... = t_p(d_p)$. This can be expressed as:

$$\frac{d_1}{s_1(d_1)} = ... = \frac{d_p}{s_p(d_p)} \text{ , where } d_1 + d_2 + ... + d_p = n \tag{1}$$

The solution to these equations, $d_1, ..., d_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system as illustrated in Fig. 4

The geometrical algorithms proceed as follows. As any line passing through the origin and intersecting the speed functions represents an optimum distribution for a particular problem size, the space of solutions of the problem (1) consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line represents the optimal data distribution $x_1^u, ..., x_p^u$ for some problem size $n_u < n$, $n_u = x_1^u + ... + x_p^u$, while the lower line gives the solution $x_1^l, ..., x_p^l$ for $n_l > n$, $n_l = x_1^l + ... + x_p^l$. The region between two lines is iteratively bisected as shown in Fig. 5.

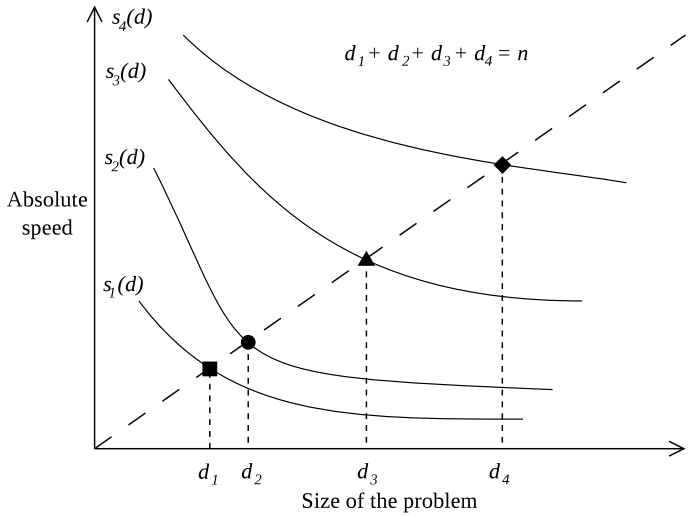


Figure 4. Optimal distribution of computational units showing the geometric proportionality of the number of computation units to the speeds of the processors

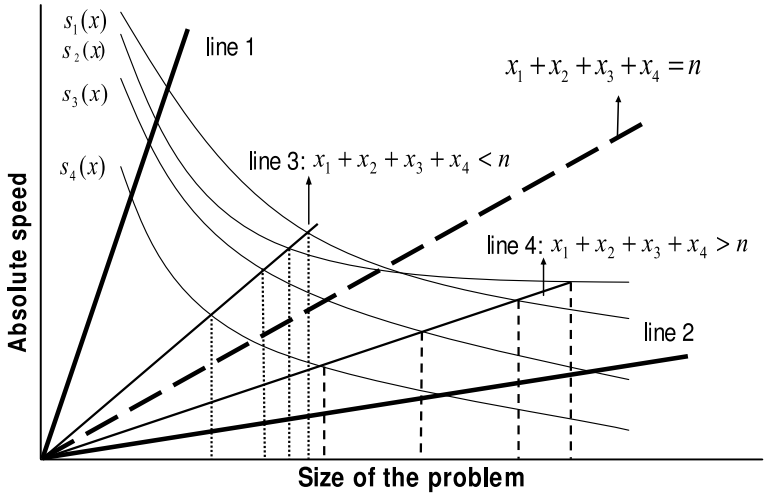


Figure 5. Geometrical data partitioning algorithm. Line 1 (the upper line) and line 2 (the lower line) represent the two initial outer bounds of the solution space. Line 3 represents the first bisection. Line 4 represents the second one. The dashed line represents the optimal solution

At the iteration k, the problem size corresponding to the new line intersecting the speed functions at the points $x_1^k, ..., x_p^k$ is calculated as $n_k = x_1^k + ... + x_p^k$. Depending on whether n_k is less than or greater than n, this line becomes a new upper or lower bound. Making n_k close to n, this algorithm finds the optimal partition of the given problem $x_1, ..., x_p$: $x_1 + ... + x_p = n$. The geometrical algorithms will always find a unique optimal solution if the speed functions satisfy the following assumptions:

1. On the interval $[0, X]$, the function is monotonically increasing and concave.
2. On the interval $[X, \infty]$, the function is monotonically decreasing.

Extensive experiments with many scientific kernels on different workstations have demonstrated that, in general, processor speed can be approximated, within some acceptable degree of accuracy, by a function satisfying these assumptions.

Another algorithm [43] significantly relaxes the restrictions on the shape of speed functions but does not always guarantee the globally optimal solution. This algorithm assumes that the

Akima spline interpolation [1] is used to approximate the speed function. Then it formulates the problem of optimal data partitioning in the form of a system of non-linear equations and applies multidimensional solvers to numerical solution of this system. The algorithm is iterative and always converges in a finite number of iterations returning a solution that balances the load of the processors. The number of iterations depends on the shape of the functions. In practice, the number can be as little as 2 iterations for very smooth speed functions and up to 30 iterations when partitioning in regions of rapidly changing speed functions. For illustration, Fig. 6 shows speed function approximations used in the geometrical algorithms and in the algorithm based on the multidimensional solvers.

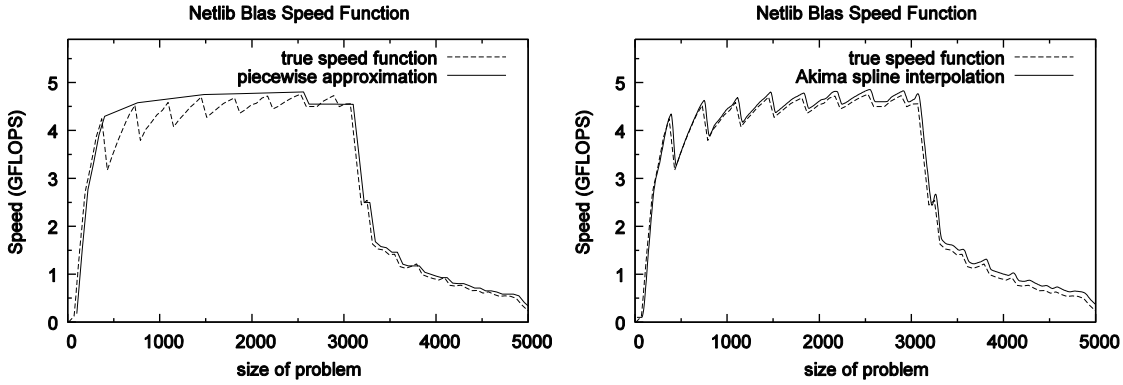


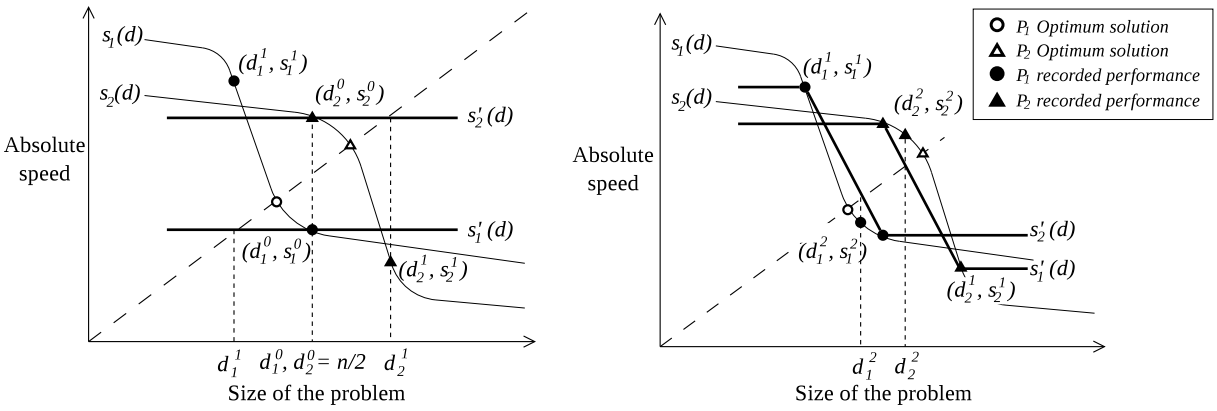
Figure 6. Speed function for non-optimized Netlib BLAS: the piecewise approximation satisfying the restriction of monotonicity (left), and the Akima spline interpolation (right)

These algorithms have been successfully employed in different data-parallel kernels and applications and significantly outperformed their CPM-based counterparts [2, 15, 16, 18, 25, 34].

Algorithms that require full FPMs as input to find the optimal partitioning can be used in applications developed for execution on the same stable platform multiple times. In this case, the cost of building the FPMs for the full range of problem sizes will be insignificant in comparison with the accumulated gains due to the optimal parallelization. However, these algorithms cannot be employed in self-adaptable applications that are supposed to discover the performance characteristics of the executing heterogeneous platform at run-time. To address that type of application, a new class of partitioning algorithms was proposed [36]. They do not need the FPMs as input. Instead, they run on the processors executing the application and iteratively build partial approximations of their speed functions until they become sufficiently accurate to partition the task of the given size with the required precision. For example, if we want to distribute n units of computation between p heterogeneous processors using the geometrical data partitioning, but the speed functions $s_i(x)$ of the processors are not known a priori, we will proceed as follows. The first approximations of the partial speed functions, $\bar{s}_i(x)$, are created as constants $\bar{s}_i(x) = s_i^0 = s_i(n/p)$ as illustrated in Fig. 7(a). At the iteration k, the piecewise linear approximations $\bar{s}_i(x)$ are improved by adding the points (d_i^k, s_i^k), Fig. 7(b). Namely, let $\{(d_i^{(j)}, s_i^{(j)})\}_{j=1}^m$, $d_i^{(1)} < \ldots < d_i^{(m)}$, be the experimentally obtained points of $\bar{s}_i(x)$ used to build its current piecewise linear approximation, then

1. If $d_i^k < d_i^{(1)}$, then the line segment $(0, s_i^{(1)}) \to (d_i^{(1)}, s_i^{(1)})$ of the $\bar{s}_i(x)$ approximation will be replaced by two connected line segments $(0, s_i^k)) \to (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \to (d_i^{(1)}, s_i^{(1)})$;
2. If $d_i^k > d_i^{(m)}$, then the line $(d_i^{(m)}, s_i^{(m)}) \to (\infty, s_i^{(m)})$ of this approximation will be replaced by the line segment $(d_i^{(m)}, s_i^{(m)}) \to (d_i^k, s_i^k)$ and the line $(d_i^k, s_i^k) \to (\infty, s_i^k)$;

3. If $d_i^{(j)} < d_i^k < d_i^{(j+1)}$, the line segment $(d_i^{(j)}, s_i^{(j)}) \to (d_i^{(j+1)}, s_i^{(j+1)})$ of $\bar{s}_i(d)$ will be replaced by two connected line segments $(d_i^{(j)}, s_i^{(j)}) \to (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \to (d_i^{(j+1)}, s_i^{(j+1)})$.



(a) (b)

Figure 7. Construction of partial speed functions using linear interpolation.

After adding the new data point (d_i^j, s_i^j) to the partial speed function $\bar{s}_i(x)$, we verify that the shape of the resulting piecewise linear approximation satisfies the above assumptions, and update the value of s_i^j when required. Namely, to keep the partial speed function increasing and convex on the interval $[0, X]$, we ensure that $s_i^{j-1} \le s_i^j \le s_i^{j+1}$ and $\frac{s_i^{j-1}-s_i^{j-2}}{d_i^{j-1}-d_i^{j-2}} \ge \frac{s_i^j-s_i^{j-1}}{d_i^j-d_i^{j-1}} \ge \frac{s_i^{j+1}-s_i^j}{d_i^{j+1}-d_i^j}$. The latter expression represents non-increasing tangent of the pieces, which is required for the convex shape of the piecewise linear approximation. On the interval $[X, \infty]$, we ensure that $s_i^{j-1} \ge s_i^j \ge s_i^{j+1}$ for monotonously decreasing speed function.

This approach has proved to be very efficient in practice, typically converging to the optimal solution after a very few iterations [16].

While some other non-constant performance models of heterogeneous processors such as the unit-step functional model [22], the functional model with limits on task size [32] and the band model [30] have been proposed and used for the design of heterogeneous algorithms, they did not go beyond some preliminary studies as they appeared to be not suitable for practical use in high-performance heterogeneous scientific computing due to a variety of reasons.

1.3. Implementation of heterogeneous data partitioning algorithms

It is important to note that the effectiveness of the data partitioning algorithms presented in this section strongly depends on how accurately the performance models employed in these algorithms are reflecting the real performance of the data parallel applications on the executing platforms. Unfortunately many algorithms, especially CPM–based, come without a method for estimation of the employed performance model, leaving this task to the user. Therefore the use of these algorithms as well as tools straightforwardly employing these algorithms is a challenging task. The graph partitioning libraries [4, 11, 12, 28, 45] give us examples of such tools.

At the same time, some algorithm designers include the method of construction of the employed performance model in the definition of the algorithm. Such algorithms are easy to use and compare. The estimation method helps to understand: (i) the meaning of the model parameters leaving no room for interpretation, and (ii) the assumptions made about the application and the target platform better. According to this approach, model-based algorithms will be different even if they only differ in the method of model construction. Such algorithms can be found

in [15, 16, 35, 43]. For example, [15] proposes a two-dimensional matrix partitioning algorithm designed for heterogeneous SUMMA (see Fig. 1). The definition of this algorithm specifically stipulates that the FPMs of the processors will be built using the computational kernel performing one update of the submatrix C_i with the portions of pivot block column A_i and pivot block row B_i: $C_i+ = A_i \times B_i$ as shown in Fig. 8.

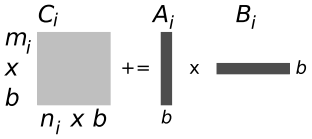


Figure 8. The computational kernel

Moreover, it proposes to use one-dimensional FPMs by combining the height m_i and width n_i parameters into one parameter, area $d_i = m_i \times n_i$, measured in $b \times b$ blocks, and to only use square areas in benchmarking, $m = n = \sqrt{d}$, for $0 < d \leq M \times N$. Then it is partitioned using a one-dimensional FPM-based algorithm to determine the areas of the rectangles that should be partitioned to each processor. The CPM-based algorithm [8] is then applied to calculate the optimum shape and ordering of the rectangles so that the total volume of communication is minimized.

The algorithm described above makes the assumption that a benchmark of a square area will give an accurate prediction of computation time of any rectangle of the same area, namely $s(x, x) = s(x/c, c.x)$. However, in general this does not hold true for all c (Fig. 9(a)). Fortunately, in order to minimise the total volume of communication the algorithm [8] arranges the rectangles so that they are as square as possible. It has been verified experimentally [15] by partitioning a medium sized square dense matrix using the new algorithm for 1 to 1000 nodes from the Grid'5000 platform (incorporating 20 unique nodes), and plotted the frequency of the ratio $m : n$ in Fig. 9(c). Fig. 9(b), showing a detail of Fig. 9(a), illustrates that if the rectangle is approximately square the assumption holds.

The efficiency of the FPM-based data-parallel applications strongly depends on the accuracy of the evaluation of the speed function of each heterogeneous processor. It is a challenging problem that requires: (i) carefully designed experiments to accurately and efficiently measure the speed of the processor for each problem size; (ii) appropriate interpolation and approximation methods which use the experimental points to construct an accurate speed function of the given shape. A software tool, FuPerMod, helping the application programmer solve these problems has been recently developed and released [17]. FuPerMod also provides a number of heterogeneous data partitioning algorithms for sets, ordered sets and matrices, both CPM-based and FPM-based. It does not provide graph-partitioning algorithms though. Graph-partitioning algorithms are provided by a number of libraries such as ParMetis [28], SCOTCH [12], JOSTLE [45], Zoltan [11], PaGrid [4]. While the partitioning algorithms implemented in these libraries use performance models, the libraries provide no support for their construction.

2. Optimization of parallel applications on hybrid multicore and multi-accelerator heterogeneous platforms

Thus, the traditional heterogeneous performance models and data partitioning algorithms and applications are designed for platforms whose processing elements are independent of each

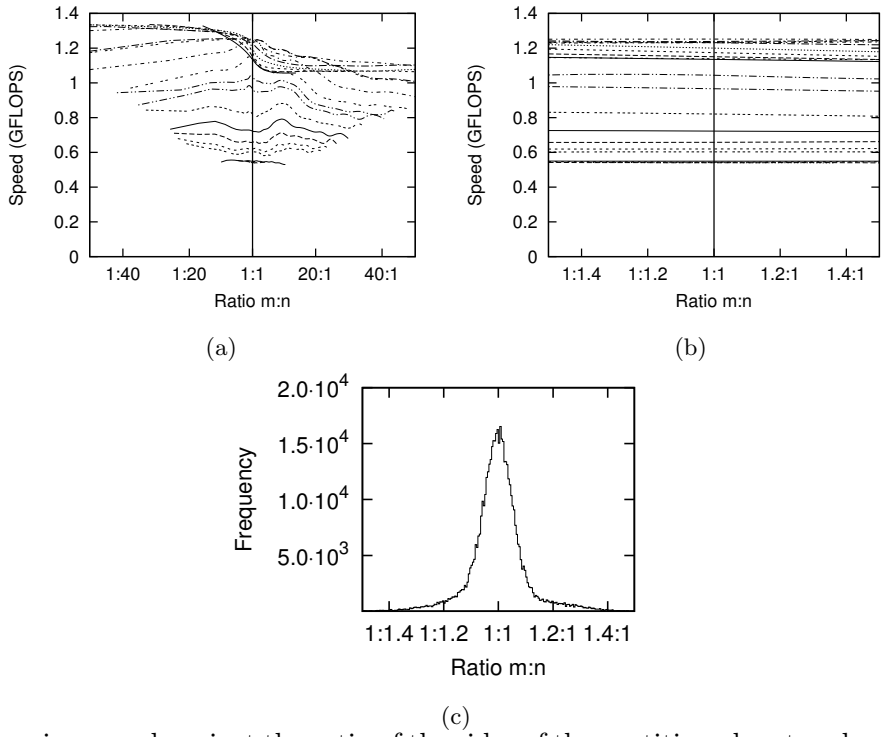


Figure 9. Showing speed against the ratio of the sides of the partitioned rectangles. Lines connect rectangles of equal area. The centerline at 1 : 1 represents square shape. In general speed is not constant with area (a). However when the ratio is close to 1 : 1, speed is approximately constant (b). (c) Shows the frequency distribution of the ratio of $m : n$ using the new partitioning algorithm for 1 to 1000 machines (incorporating 20 unique hardware configurations)

other. In modern heterogeneous multicore and multi-accelerator compute nodes, however, processing elements are coupled and share system resources. In such platforms, the speed of one processing element often depends on the load of others due to resource contention. Therefore, they cannot be considered independent, and hence their associated performance models cannot be considered and built independently. This makes the traditional models, methods of their evaluation and algorithms no longer applicable to the new platforms.

This problem was recently addressed in [46] [47] [48]. In this work, the authors do not study how to develop computational kernels for individual computing devices used in hybrid heterogeneous platforms, such as multicore CPUs or GPUs. They assume that such kernels are available for the use in parallel applications on these platforms. While being very challenging and important, this problem has attracted significant attention of the HPC research community and many important kernels have been ported to modern multicores and GPUs. Instead, they focus on a wide open problem of optimal data distribution between kernels of the data-parallel application assuming that the configuration of the application is fixed. Finding the optimal configuration of the application is another challenge to be addressed, which is out of the scope of this work. The authors however give few basic empirical rules that, they believe, lead to optimal configurations. For example, never run a NUMA-unaware multi-threaded computational kernel across multiple NUMA nodes. Use instead multiple instances of this kernel, one per NUMA node.

A multicore and multi-GPU system, which the main target architecture in this work, is modeled by a set of heterogeneous abstract processors determined by the configuration of the

parallel application. Namely, a group of processing elements executing one computational kernel of the application will make a combined processing unit and will be represented in the model by one abstract processor. For example, if a single-threaded computational kernel is used, then each CPU core executing this kernel will be represented in the model by a separate abstract processor. If a multi-threaded computational kernel is used, then each group of CPU cores executing the kernel will make a combined processing unit represented in the model by one abstract processor. A GPU is usually controlled by a host process running on a dedicated CPU core. This process instructs the GPU to perform computations and handles data transfers between the host and device memory. In the case of a single-GPU computational kernel, the GPU and its dedicated CPU core will make a combined processing unit represented by an abstract processor. If a multi-GPU computational kernel is used in the application, the GPUs and their dedicated CPU core will make a combined processing unit represented by one abstract processor.

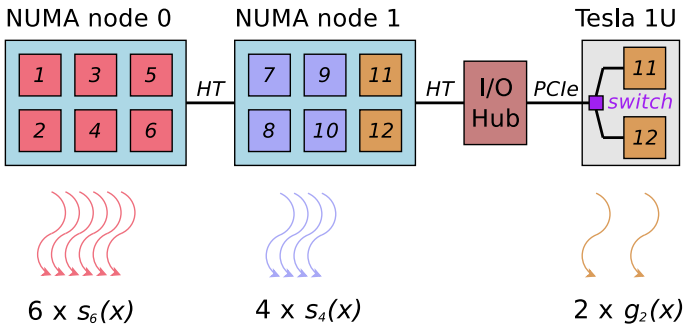


Figure 10. Performance modeling on a GPU-accelerated multicore server of NUMA architecture: single-threaded and single-GPU computational kernels executed

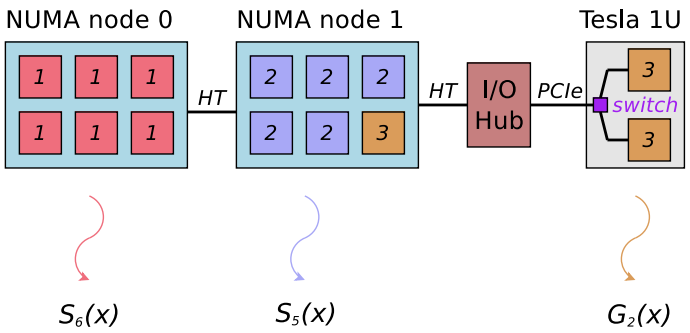


Figure 11. Performance modeling on a GPU-accelerated multicore server of NUMA architecture: multi-threaded and multi-GPU computational kernels executed; two GPUs handled by a single dedicated CPU core

Figures 10 and 11 illustrate this approach showing a GPU-accelerated multicore server of NUMA architecture executing a parallel application in two different configurations. The configuration shown in Fig. 10 is based on the single-threaded and single-GPU computational kernels. It consists of ten processes running the CPU kernels on ten cores of both NUMA nodes, and two processes running the GPU kernels on accelerators and their dedicated cores on the second NUMA node. The configuration in Fig. 11 is based on the multi-threaded and multi-GPU computational kernels. It consists of one process running the 6-thread CPU kernel on one NUMA node, one process running the 5-thread CPU kernel on another NUMA node, and one process running the GPU kernel on the GPUs and their single dedicated core. All processing elements in these diagrams are enumerated. Each number indicates the combined processing unit to which

the processing element belongs. For example, in the first configuration, the cores in NUMA node 0 make six processing units, and each GPU with its dedicated CPU core in NUMA node 1 make a combined processing unit.

In the first configuration, the cores in NUMA node 0 execute six identical processes and are modeled by six abstract processors. These cores are tightly coupled and share memory, therefore, they cannot be considered independent. On the other hand, this group of processing elements is relatively independent of other processing elements of the server. Therefore, their performance should be measured simultaneously in a group but can be measured separately from the others. In the second configuration, these six cores execute one process and modeled as one combined processing unit. Its performance can be measured separately from other processing elements of the server.

Next steps are to build functional performance models of the abstract processors and perform model-based data partitioning in order to balance the workload between the combined processing units represented by these abstract processors.

In order to build the performance models of the abstract processors, the performance of the processing units representing these processors has to be measured. To measure the performance of the processing units accurately, they are grouped by the shared system resources, so that the resources be shared within each group but not shared between groups. The performance of processing units in a group is measured when all processing units in the group are executing some workload simultaneously, thereby taking into account the influence of resource contention. To prevent the operating system from migrating processes excessively, processes are bound to CPU cores. Processes are synchronized to minimize the idle computational cycles, aiming at the highest floating point rate for the application. Synchronization also ensures that the resources will be shared between the maximum number of processes. To ensure the reliability of the results, measurements are repeated multiple times, and average execution times are used.

One important empirical rule used in this work is that when looking for the optimal distribution of the workload, only the solutions that evenly distribute the workload between identical CPU processing units are considered. This simplification significantly reduces the complexity of the data partitioning problem. It is based both on the authors' extensive experiments that have shown no evidence that uneven distribution between identical processing units could speed up applications, and on the absence of such evidence in literature. Therefore, identical processing units that share system resources will be always given the same amount of workload during performance measurements.

To account for different configurations of the application, three types of functional performance models for CPU cores are defined:

1. $s(x)$ approximates the speed of a uniprocessor executing a single-threaded computational kernel. The speed $s(x) = x/t$, where x is the number of computation units, and t is the execution time.
2. $s_c(x)$ approximates the speed of one of c CPU cores all executing the same single-threaded computational kernel simultaneously. The speed $s_c(x) = x/t$, where x is the number of computation units executed by each CPU core, and t is the execution time.
3. $S_c(x)$ approximates the collective speed of c CPU cores executing a multi-threaded computational kernel. The speed $S_c(x) = x/t$, where x is the total number of computation units executed by all c CPU cores, and t is the execution time. $S_c(cx)/c$ is used to approximate the average speed of a CPU core.

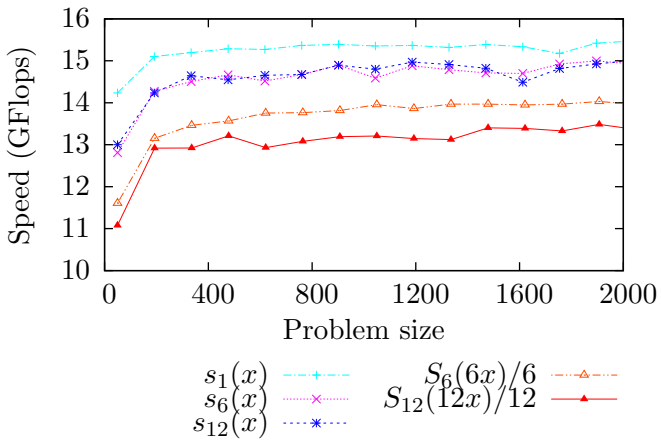


Figure 12. Speed functions of a CPU core built in different configurations

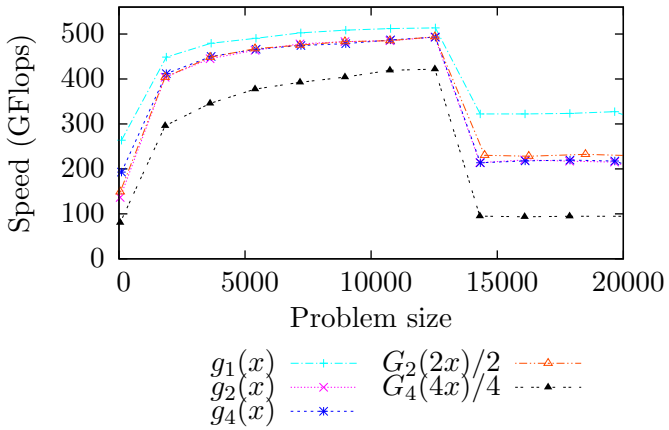


Figure 13. Speed functions of a GPU processing unit built in different configurations

Fig. 12 shows speed functions of a CPU core built in different configurations on a server, consisting of eight NUMA nodes connected by AMD HyperTransport(HT) links, with 6 cores and 16 GB local memory each. The server is equipped with a NVIDIA Tesla S2050 server, which consists of two pairs of GPUs. Each pair is connected by a PCIe switch and linked to a separate NUMA node by a PCIe bus.

Similarly, three types of functional performance models for GPUs are defined as follows:

1. $g(x)$ approximates the speed of a combined processing unit made of a GPU and its dedicated CPU core that execute a single-GPU computational kernel, exclusively using a PCIe link. The speed $g(x) = x/t$, where x is the number of computation units, and t is the execution time.
2. $g_d(x)$ approximates the speed of one of d combined processing units, each made of a GPU and its dedicated CPU core. All processing units execute identical single-GPU computational kernels simultaneously. The speed $g_d(x) = x/t$, where x is the number of computation units executed by each GPU processing unit, and t is the execution time.
3. $G_d(x)$ approximates the speed of a combined processing unit made of d GPUs and their dedicated CPU core that collectively execute a multi-GPU computational kernel. The speed $G_d(x) = x/t$, where x is the total number of computation units processed by all d GPUs, and t is the execution time. $G_d(dx)/d$ is used to approximate the average speed of a GPU.

Fig. 13 shows the speed functions of a combined GPU processing unit built in different configurations on the same server.

From these experiments we can see that depending on the configuration of the application the speed of individual cores and GPUs can vary significantly. Therefore, to achieve optimal distribution of computations it is very important to build and use speed functions which accurately reflect their performance during the execution of the application. This work also reveals that the speed of GPU can depend on the load of CPU cores, which should be also taken into account during the partitioning step. Experiments with linear algebra kernels and a CFD application validated the efficiency of the proposed approach.

At the same time, this work has demonstrated the importance of proper configuration of the application. For example, Fig. 14 demonstrates the impact of NUMA mapping on the performance of a GPU processing unit, comprised of a CPU core and a GPU of Tesla S2050 deployed in the experimental server. $g_1(x)$ is built by executing one single-GPU *gemm* kernel, which uses exclusively the data link and the memory of a local or remote NUMA node. $g_2(x)$ is built by executing two single-GPU kernels simultaneously on two GPU units that share the PCIe link and the memory of the same NUMA node, local or remote. In the remote configuration, the GPU units also share an extra HT link to the remote NUMA node. Speed function $g_2(x)$ is also built in the configuration when two dedicated CPU cores are located on different NUMA nodes, which is denoted as *local + remote*. In this case, the processing units share PCIe but do not share memory.

The difference between speed functions $g_1(x)$ and $g_2(x)$ reflects the performance degradation due to the contention for PCIe, HT and memory. Significant difference is observed for large problem sizes when many data transfers are required. Communication overhead between NUMA nodes can be estimated by the difference between $g_1(x)$ in *local* and *remote* configurations. The combined effect of both phenomena is reflected by the $g_2(x)$ functions in different configurations.

Multilevel hierarchy in modern heterogeneous clusters represents another challenge to be addressed in the design of data partitioning algorithms. One solution, a hierarchical matrix partitioning algorithm based on realistic performance models at each level of hierarchy, was recently proposed in [14]. To minimize the total execution time of the application it iteratively partitions a matrix between nodes and partitions these sub-matrices between the devices in a node. This is a self-adaptive algorithm that dynamically builds the performance models at runtime and it employs an algorithm to minimize the total volume of communication. This algorithm

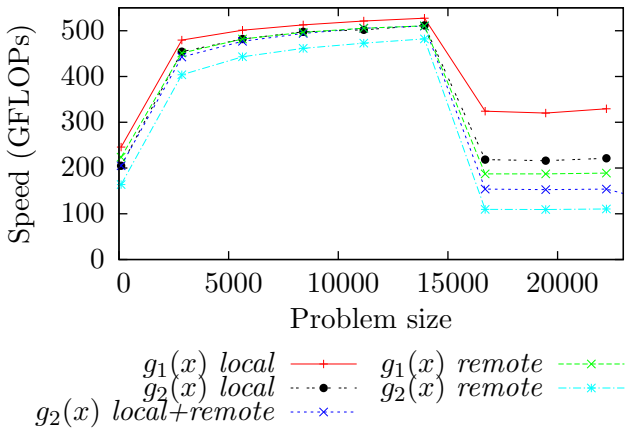


Figure 14. Speed functions of a GPU processing unit built in different configurations

allows scientific applications to perform load balanced matrix operations with nested parallelism on hierarchical heterogeneous platforms. Large scale experiments on a heterogeneous multi-cluster site incorporating multicore CPUs and GPU nodes have shown that this hierarchical algorithm outperforms all other state–of–the–art approaches and successfully load balance very large problems.

3. Programming tools

In the past, the main platform used for non-trivial heterogeneous parallel computing (as opposed to volunteer computing, such as the seti@home project) has been a heterogeneous cluster of workstations. MPI is a standard programming model for this platform. However, the implementation of real-world heterogeneous parallel algorithms in an efficient and portable form requires much more than just the code implementing the algorithm for each legal combination of its input parameters. Extra code should be written to find optimal values of some parameters (say, the number of processes and their arrangement in a multi-dimensional shape) or to accurately estimate the others (such as relative speeds of the processors). This extra code may account for at least 95% of all code in common cases. Therefore, for the implementation of heterogeneous parallel algorithms on this platform, a small number of programming tools was developed. mpC [3] is the first programming language designed for heterogeneous parallel computing. It facilitates the implementation of heterogeneous parallel algorithms by automating the development of the routine code, which comes in two forms: (i) application specific code generated by a compiler from the specification of the implemented algorithm provided by the application programmer; (ii) universal code in the form of run-time support system and libraries. HeteroMPI [33] is an extension of MPI inspired by mpC. It allows the programmer to re-use the available MPI code when developing applications for heterogeneous clusters of workstations. Both mpC and HeteroMPI have been used for development of a wide range of real-life applications. HeteroMPI was also the instrumental tool for implementation of Heterogeneous ScaLAPACK [42], a version of ScaLAPACK optimized for heterogeneous clusters of workstations.

Modern and future heterogeneous HPC systems necessitate the synthesis of multiple programming models in the same code. This will be a result of the use of multiple heterogeneous many-core devices for accelerating code, as well as the use of both shared- and distributed-address spaces in the same code to cope with heterogeneous memory hierarchies and forms of communication. Synthesizing multiple programming models in the same code in a way that would provide a good balance of performance, portability and programmability, is far from trivial. Despite long-standing efforts to program parallel applications with hybrid programming models (e.g. MPI/OpenMP) and some recent developments in programming models for hybrid architectures (e.g. OpenCL), it is still a long way towards solutions that would satisfy the HPC community.

This work was conducted with the financial support of Science Foundation Ireland, Grant 08/IN.1/I2054.

References

1. H. Akima. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of the ACM*, 17:589–602, 1970.
2. A. Alonazi, D. Keyes, A. Lastovetsky, and V. Rychkov. Design and optimization of openfoam-based cfd applications for hybrid and heterogeneous hpc platforms. 26th International Conference on Parallel Computational Fluid Dynamics (ParCFD 2014), Trondheim, Norway, 2014.
3. D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis. A programming environment for heterogenous distributed memory machines. In *6th Heterogeneous Computing Workshop (HCW 1997)*, pages 32–45. IEEE, 1997.
4. E. Aubanel and X. Wu. Incorporating latency in heterogeneous graph partitioning. In *IPDPS 2007*, pages 1–8, 2007.
5. C. Augonnet et al. Automatic calibration of performance models on heterogeneous multicore architectures. In *EuroPar*, 2009.
6. J. Barbosa, J. Tavares, and A. J. Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *9th Heterogeneous Computing Workshop (HCW 2000)*, pages 147–159, 2000.
7. O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *IEEE Transactions on Computers*, 50(10):1052–1070, 2001.
8. O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.
9. R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.
10. P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2):197–213, 1999.
11. U. Catalyurek, E. Boman, K. Devine, et al. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IPDPS 2007*, pages 1–11, 2007.
12. C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6–8):318–331, 2008.
13. J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *HPC Asia*, pages 224 –229, 1997.
14. D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa. Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu+ gpu clusters. In *Euro-Par 2012 Parallel Processing*, pages 489–501. Springer, 2012.
15. D. Clarke, A. Lastovetsky, and V. Rychkov. Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In *HeteroPar 2011*, pages 450–459. Springer, 2011.
16. D. Clarke, A. Lastovetsky, and V. Rychkov. Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms. *Parallel Processing Letters*, 21(2):195–217, 2011.

17. D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky. Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms. In *PaCT 2013*, volume 7979 of *LNCS*, pages 182–196. Springer, 2013.

18. J. Colaco, A. Matoga, et al. Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems. In *PPAM 2013, Part I*, pages 693–703, 2014.

19. A. DeFlumere and A. Lastovetsky. Searching for the optimal data partitioning shape for parallel matrix matrix multiplication on 3 heterogeneous processors. In *23rd Heterogeneity in Computing Workshop (HCW 2014)*, pages 1–12, 2014.

20. A. DeFlumere, A. Lastovetsky, and B. Becker. Partitioning for parallel matrix multiplication with heterogeneous processors: The optimal solution. In *21st Heterogeneity in Computing Workshop (HCW 2012)*, pages 1–15, 2012.

21. K. Dichev and A. Lastovetsky. Optimization of collective communication for heterogeneous hpc platforms. *High-Performance Computing on Complex Environments*, pages 95–114, 2014.

22. M. Drozdowski and P. Wolniewicz. Out-of-core divisible load processing. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1048–1056, 2003.

23. R. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

24. R. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

25. A. Ilic, F. Pratas, P. Trancoso, and L. Sousa. High-performance computing on heterogeneous systems: Database queries on cpu and gpu. In *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*. IOS Press, 2011.

26. A. Ilic and L. Sousa. On realistic divisible load scheduling in highly heterogeneous distributed systems. In *PDP 2012*, pages 426–433. IEEE, 2012.

27. A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In *7th International Conference on High Performance Computing and Networking Europe (HPCN'99)*, pages 191–200, 1999.

28. G. Karypis and K. Schloegel. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 4.0.* University of Minnesota, MN, USA, 2013.

29. A. Lastovetsky. On grid-based matrix partitioning for heterogeneous processors. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 383–390. IEEE, 2007.

30. A. Lastovetsky and R. Higgins. Scheduling for heterogeneous networks of computers with persistent fluctuation of load. In *13th International Conference on Parallel Computing (ParCo 2005)*, pages 383–390, 2005.

31. A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *IPDPS 2004*, pages 1–15, 2004.

32. A. Lastovetsky and R. Reddy. Data partitioning for multiprocessors with memory heterogeneity and memory constraints. *Scientific Programming*, 13(2):93–112, 2005.

33. A. Lastovetsky and R. Reddy. Heterompi: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

34. A. Lastovetsky and R. Reddy. Data partitioning for dense factorization on computers with memory heterogeneity. *Parallel Computing*, 33(12):757–779, 2007.

35. A. Lastovetsky and R. Reddy. Data partitioning with a functional performance model of heterogeneous processors. *International Journal of High Performance Computing Applications*, 21:76–90, 2007.

36. A. Lastovetsky and R. Reddy. Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models. In *Euro-Par'09*, pages 91–101, 2009.

37. A. Lastovetsky, R. Reddy, and R. Higgins. Building the functional performance model of a processor. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, pages 746–753. ACM, 2006.

38. A. Lastovetsky and J. Twamley. Towards a realistic performance model for networks of heterogeneous computers. In *High Performance Computational Science and Engineering*, pages 39–57. Springer, 2005.

39. M. Linderman, J. Collins, H. Wang, et al. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, 2008.

40. G. Quintana-Ortí et al. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44:121–130, 2009.

41. J.N. Quintin and F. Wagner. Hierarchical work-stealing. *Euro-Par 2010-Parallel Processing*, pages 217–229, 2010.

42. R. Reddy and A. Lastovetsky. Heterompi+ scalapack: towards a scalapack (dense linear solvers) on heterogeneous networks of computers. In *13th IEEE International Conference on High Performance Computing (HiPC 2006)*, pages 242–253. 2006.

43. V. Rychkov, D. Clarke, and A. Lastovetsky. Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous hpc platforms. In *PaCT 2011*, pages 332–346. Springer-Verlag, 2011.

44. F. Song et al. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *ICS*, 2012.

45. C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems*, 17(5):601–623, 2001.

46. Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore platforms. In *Cluster 2011*, pages 580–584. IEEE, 2011.

47. 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 2012*, pages 191–199. IEEE, 2012.

48. Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu platforms using functional performance models. *IEEE Transactions on Computers*, pages 1–14, 2014.