Hierarchical Partitioning Algorithm for Scientific Computing on Highly Heterogeneous CPU + GPU Clusters

David Clarke[1], Aleksandar Ilic[2], Alexey Lastovetsky[1], and Leonel Sousa[2]

[1] School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland
[2] INESC-ID, IST/Technical University of Lisbon, Rua Alves Redol, 9, 1000-029 Lisbon, Portugal

Abstract. Hierarchical level of heterogeneity exists in many modern high performance clusters in the form of heterogeneity between computing nodes, and within a node with the addition of specialized accelerators, such as GPUs. To achieve high performance of scientific applications on these platforms it is necessary to perform load balancing. In this paper we present a hierarchical matrix partitioning algorithm based on realistic performance models at each level of hierarchy. To minimise 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 run-time and it employs an algorithm to minimise the total volume of communication. This algorithm allows scientific applications to perform load balanced matrix operations with nested parallelism on hierarchical heterogeneous platforms. To show the effectiveness of the algorithm we applied it to a fundamental operation in scientific parallel computing, matrix multiplication. Large scale experiments on a heterogeneous multi-cluster site incorporating multicore CPUs and GPU nodes show that the presented algorithm outperforms current state of the art approaches and successfully load balance very large problems.

Keywords: parallel applications; heterogeneous platforms; GPU; data partitioning algorithms; functional performance models; matrix multiplication

1 Introduction

In this paper we present a matrix partitioning algorithm for load balancing parallel applications running on highly heterogeneous hierarchical platforms. The target platform is a dedicated heterogeneous distributed memory platform with multi level hierarchy. More specifically, we focus on a platform with two levels of hierarchy. At the top level is a distributed memory cluster of heterogeneous nodes, and at the lower level, each node consists of a number of devices which may be a combination of multicore CPUs and specialized accelerators/co-processors (GPUs). We refer to both nodes and devices collectively as processing

elements. The applications we target perform matrix operations and are characterised by discretely divisible computational workloads where the computations can be split into independent units, such that each computational unit requires the same amount of computational work. In addition, computational workload is directly proportional to the size of data and dependent on data locality. High performance of these applications can be achieved on heterogeneous platforms by performing load balancing at each level of hierarchy. Load balancing ensures that all processors complete their work within the same time. This requirement is satisfied by partitioning the computational workload unevenly between processing elements, at each level of hierarchy, with respect to the performance of that element.

In order to achieve load balancing on our target platform, the partitioning algorithm must be designed to take into account both the hierarchy and the high level of heterogeneity of the platform. In contrast to the traditional, CPU-only distributed memory systems, highly heterogeneous environments employ devices which have fundamental architectural differences. The ratio of performance differences between devices may be orders of magnitude more than the ratio between traditional heterogeneous platforms; moreover this ratio can vary greatly with a change in problem size. For example, accelerators need to physically load and offload portions of data on which computations are performed in order to ensure high performance and full execution control, and the executable problem size is limited by the available device memory. Finally, architectural differences impose new collaborative programming challenges, where it becomes necessary to use different programming models, vendor-specific tools and libraries in order to approach the per-device peak performance. However, even if some of the already existing collaborative execution environments are used (such as OpenCL, StarPU [1] or CHPS [11]), the problem of efficient cross-device problem partitioning and load balancing still remains.

The work proposed herein takes into account this complex heterogeneity by using realistic performance models of the employed devices and nodes. The model of each device or node is constructed by measuring the real performance of the application when it runs on that device or node. Thus, they are capable of intrinsically encapsulating all the above-mentioned architectural and performance diversities. Traditional partitioning algorithms define the performance of each processor by a single number. We refer to this simplistic model of processor performance as a constant performance model (CPM). The functional performance model (FPM), proposed in [16], is a more realistic model of processor performance, where processor speed is a function of problem size. Partitioning algorithms which use these FPMs always achieve better load balancing than traditional CPM-based algorithms.

The main contribution of this work is a new hierarchical matrix partitioning algorithm, based on functional performance models, which performs load balancing at both the node and device levels. This algorithm performs a one to one mapping of computational workload and data to nodes and a one to one mapping of workload to devices. The device level partitioning is performed on

each node by sub-partitioning workload assigned to that node. In contrast to the some state of the art approaches, this algorithm does not require any a priori information about the platform, instead all required performance information is found by performing real benchmarks of the core computational kernel of an application.

To the best of our knowledge this is the first work that targets large scale partitioning problems for hierarchical and highly heterogeneous distributed systems. To show the effectiveness of the proposed algorithm we applied it to parallel matrix multiplication, which is representative of the class of computationally intensive parallel scientific applications that we target. Experiments on 3 interconnected computing clusters, using a total of 90 CPU+GPU heterogeneous nodes, showed that, for a wide range of problem sizes, the application based on FPM-based partitioning outperformed applications based on CPM algorithms.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we propose hierarchical partitioning algorithm for highly heterogeneous CPU+GPU clusters. The experimental results are presented in Section 4. Finally, concluding remarks are given in Section 5.

2 Related work

Divisible load theory(DLT), surveyed in [21], defines a class of applications characterised by workload that can be divided into discrete parts for parallel computation. The applications we target belong to this class. Scheduling and work stealing algorithms [7, 3, 20], often used in DLT, move workload between processing elements, during execution of the application, to achieve load balancing. However, on distributed memory platforms, such an approach can incur a high cost of data migration with applications where data locality is important. Moreover we are not aware of any dynamic-scheduling/work-stealing matrix multiplication application for highly heterogeneous distributed memory platforms.

A different class of load balancing algorithms are *partitioning algorithms*, also known as *predicting-the-future*; so called because they rely on performance models as input to predict the future execution characteristics of the application. The global workload is partitioned between the available processing elements. Traditional partitioning algorithms [2, 8, 10, 14, 18, 19] model processor performance by a single positive number and partition workload proportionally. We refer to these simplistic models as constant performance models (CPM).

The partitioning algorithm proposed in this paper predicts future performance by using more realistic functional performance models (FPM) [16]. This algorithm is designed to be self-adaptable [17], making it suitable for applications for which each run is considered to be unique because of a change of input parameters or execution on a unique subset of hardware. This is achieved by dynamically building partial estimates of the full speed functions to the required degree of accuracy. It has been shown in [5] that applications using partitioning based on FPMs can outperform applications based on CPMs. In [12], we investigated the potentials of hierarchical divisible load scheduling on our tar-

get platform using the master-worker paradigm. Experiments on a network of off-the-shelf heterogeneous desktops (CPU + GPU), shows the benefit of using realistic performance models to load balance and efficiently overlap computations and communications at the GPU device level. In this paper, we focus on load balancing with respect to computational performance of processing elements, and to this end, we do not measure the interconnect speed between each pair of processing elements; instead we arrange elements such that the communication volume is minimised [2].

Several scientific studies have already dealt with the problems investigated herein, but only partially. For example, MAGMA [9] is a library for matrix algebra for GPU and multicore which uses scheduling for load balancing, but only on a single node. In terms of the target platform, [15, 13] consider homogeneous multi-GPU cluster systems without CPUs, whereas [6] is designed for a homogeneous hierarchical platform.

3 Hierarchical Matrix Partitioning Algorithm

A typical computationally intensive parallel scientific application performs the same iterative core computation on a set of data. The general scheme of such an application can be summarised as follows: (i) all data is partitioned over processing elements, (ii) some independent calculations are carried out in parallel, and (iii) some synchronisation takes place. High performance on a distributed memory, hierarchical heterogeneous platform, for such an application, is achieved by partitioning workload in proportion to the speed of the processing elements. The speed of a processing element is best represented by a continuous function of problem size [5]. These FPMs are built empirically for each application on each processing element.

Building these speed functions for the full range of potential problem sizes can be expensive. To reduce this cost and allow the parallel application to be self adaptable to new platforms we make two optimisations: (i) many computationally intensive scientific applications repeat the same core computational kernel many times on different data; to find the performance of this application for a given problem size it is only necessary to benchmark one representative iteration of the kernel; (ii) partial estimates of the speed functions may be built at application run-time to a sufficient level of accuracy to achieve load balancing [17].

Our target platform is a two level hierarchical distributed platform with q nodes, $Q_1, \ldots, Q_q$, where a node Q_i has p_i devices, $P_{i1}, \ldots, P_{ip_i}$. The problem to be solved by this algorithm is to partition a matrix between these nodes and devices with respect to the performance of each of these processing elements. The proposed partitioning algorithm is iterative and converges towards an optimum distribution which balances the workload. It consists of two iterative algorithms, *inter-node partitioning algorithm (INPA)* and *inter-device partitioning algorithm (IDPA)*. The IDPA algorithm is nested inside the INPA algorithm.

Without loss of generality we will work with square $N \times N$ matrices. We introduce a blocking factor b to allow optimised libraries to achieve their peak

performance as well as reducing the number of communications. For simplicity we assume N to be a multiple of b, hence there is a total of W computational units to be distributed, where $W = (N/b) \times (N/b)$.

The INPA partitions the total matrix into q sub-matrices to be processed on each heterogeneous computing node. The sub-matrix owned by node Q_i has an area equal to $w_i \times b \times b$, where $w_1 + \ldots + w_q = W$. The *Geometric partitioning algorithm* (GPA) uses experimentally built speed functions to calculate a load balanced distribution $w_1, \ldots, w_q$. The shape and ordering of these sub-matrices is calculated by the *communication minimising algorithm* (CMA). The CMA uses column-based 2D arrangement of nodes and outputs the heights bm_i and widths bn_i for each of the q nodes, such that $m_i \times n_i = w_i$, $bm = b \times m$ and $bn = b \times n$ (Fig. 1(a)). This two dimensional partitioning algorithm uses a column-based arrangement of processors. The values of m_i and n_i are chosen so that the column widths sum up to N and heights of sub-matrices in a column sum to N.

The IDPA iteratively measures, on each device, the time of execution of the application specific core computational kernel with a given size while converging to a load balanced inter-device partitioning. It returns the kernel execution time of the last iteration to the INPA. IDPA calls the GPA to partition the sub-matrix owned by Q_i into vertical slices of width d_{ij}, such that $d_{i1} + \ldots + d_{ip} = bn_i$ (Fig. 1(b)) to be processed on each device within a Q_i node. Device P_{ij} will be responsible for doing matrix operations on $bm_i \times d_{ij}$ matrix elements.

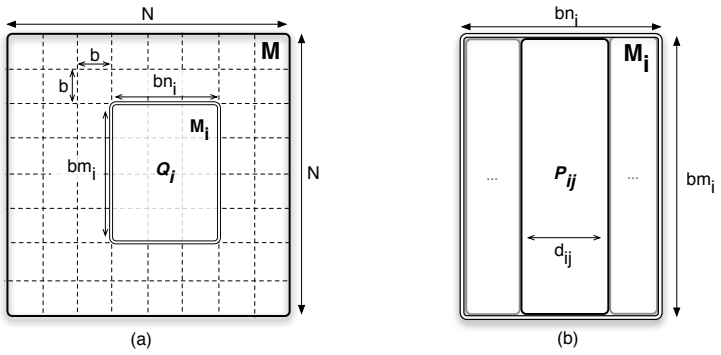

Fig. 1. Two level matrix partitioning scheme: (a) two dimensional partitioning between the nodes; (b) one dimensional partitioning between devices in a node

We now present an outline of a parallel application using the proposed hierarchical partitioning algorithm. The partitioning is executed immediately before execution of the parallel algorithm. The outline is followed by a detailed description of the individual algorithms.

INPA$\big($IN: $N, b, q, p_1, \ldots, p_q$ OUT: $\{m_i, n_i, d_{i1}, \ldots, d_{ip}\}_{i=1}^{q}\big)$ {

WHILE inter-node imbalance

CMA$\Big($IN: $w_1, \ldots, w_q$ OUT: $(m_1, n_1), \ldots, (m_q, n_q)\Big)$;

On each node i (IDPA):

WHILE inter-device imbalance

On each device j: **kernel**$\Big($IN: bm_i, bn_i, d_{ij} OUT: $t_{ij}\Big)$;

GPA$\Big($IN: p_i, bn_i, p_iFPMs OUT: $d_{i1}, \ldots, d_{iq})\Big)$;

END WHILE

GPA$\Big($IN: q, W, qFPMs OUT: $w_1, \ldots, w_q\Big)$;

END WHILE

}

Parallel application$\Big($IN: $\{m_i, n_i, d_{i1}, \ldots, d_{ip}\}_{i=1}^{q}$, $\ldots\Big)$

Inter-Node Partitioning Algorithm (INPA)
Run in parallel on all nodes with distributed memory. Inputs: square matrix size N, number of nodes q, number devices in each node $p_1, \ldots, p_q$ and block size b.

1. To add initial small point to the model, each node, in parallel, invokes the IDPA with an input (p_i, $bm_i = 1$, $bn_i = 1$). This algorithm returns a time which is sent to the head node.
2. The head node calculates speeds from these times as $s_i(1) = 1/t_i(1)$ and adds the first point, $(1, s(1))$, to the model of each node.
3. The head node then computes the initial homogeneous distribution by dividing the total number of blocks, W, between processors $w_i = W/q$.
4. The CMA is passed $w_1, \ldots, w_q$ and returns the inter-node distributions $(m_1, n_1), \ldots, (m_q, n_q)$ which are scattered to all nodes.
5. On each node, the IDPA is invoked with the input (p_i, bm_i, bn_i) and the returned time t_i is sent to the head node.
6. IF $\max\limits_{1 \le i,j \le q} \left| \frac{t_i(w_i) - t_j(w_i)}{t_i(w_i)} \right| \le \varepsilon_1$ THEN the current inter-node distribution solves the problem. All inter-device and inter-node distributions are saved and the algorithm stops;
   ELSE the head node calculates the speeds of the nodes as $s_i(w_i) = w_i/t_i(w_i)$ and adds the point $(w_i, s_i(w_i))$ to each node-FPM.
7. On the head node, the GPA is given the node-FPMs as input and returns a new distribution $w_1, \ldots, w_q$
8. GOTO 4

Inter-Device Partitioning Algorithm (IDPA)
This algorithm is run on a node with p devices. The input parameters are p and the sub-matrix sizes bm, bn. It computes the device distribution $d_1, \cdots, d_p$ and returns the time of last benchmark.

1. To add an initial small point to each device model, the *kernel* with parameters $(bm, bn, 1)$ is run in parallel on each device and its execution time is measured. The speed is computed as $s_j(1) = 1/t_j(1)$ and the point $(1, s_j(1))$ is added to each device model.
2. The initial homogeneous distribution $d_j = bn/p$, for all $1 \leq j \leq p$ is set.
3. In parallel on each device, the time $t_j(d_j)$ to execute the kernel with parameters (bm, bn, d_j) is measured.
4. IF $\max_{1 \leq i,j \leq p} \left| \frac{t_i(d_i) - t_j(d_j)}{t_i(d_i)} \right| \leq \varepsilon_2$ THEN the current distribution of computations over devices solves the problem. This distribution $d_1, \cdots, d_p$ is saved and $\max_{1 \leq j \leq p} t_j(d_j)$ is returned;
   ELSE the speeds $s_j(d_j) = d_j/t_j(d_j)$ are computed and the point $(d_j, s_j(d_j))$ is added to each device-FPM.
5. The GPA takes bn and device-FPMs as input and returns a new distribution $d_1, \ldots, d_p$.
6. GOTO 3

Geometric Partitioning Algorithm (GPA)
The geometric partitioning algorithm presented in [16] can be summarised as follows. To distribute n computational units between p processing elements, load balancing is achieved when all elements execute their work within the same time: $t_1(x_1) \approx t_2(x_2) \approx \ldots \approx t_p(x_p)$. This can be expressed as:

$$\begin{cases} \frac{x_1}{s_1(x_1)} \approx \frac{x_2}{s_2(x_2)} \approx \ldots \approx \frac{x_p}{s_p(x_p)} \\ x_1 + x_2 + \ldots + x_p = n \end{cases} \quad (1)$$

The solution of these equations, $x_1, \ldots, x_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system. Any such line represents an optimum distribution for a particular problem size. Therefore, the space of solutions of the partitioning problem 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 distribution for some problem size $n_u < n$, while the lower line gives the solution for $n_l > n$. The region between two lines is iteratively bisected. The bisection line gives the optimum distribution for the problem size n_m. If $n_m < n$, then bisection line becomes the new upper bound, else it becomes the new lower bound. The algorithm iteratively progresses until converging to an integer solution to the problem.

Communication Minimising Algorithm (CMA)
This algorithm is specific to communication pattern of application and the topology of the communication network. It takes as input the number of computational units, w_i, to assign to each processing element and arranges them in such away, (m_i, n_i), as to minimise the communication cost. For example, for matrix multiplication, $\mathbf{A} \times \mathbf{B} = \mathbf{C}$, the total volume of data exchange is minimised by minimising the sum of the half perimeters $H = \sum_{i=1}^{q} (m_i + n_i)$. A column-based restriction of this problem is solved by an algorithm presented in [2].

4 Experimental Results

To demonstrate the effectiveness of the proposed algorithm we used parallel matrix multiplication as the application. This application is hierarchical and uses nested parallelism. At the inter-node level it uses a heterogeneous modification of the two-dimensional blocked matrix multiplication [4], upon which ScaLAPACK is based. At the inter-device level it uses one-dimensional sliced matrix multiplication. It can be summarised as follows: to perform the matrix multiplication $C = A \times B$, square dense matrices A, B and C are partitioned into sub-matrices A', B', C' (Fig. 2(a)), according to the output of the INPA. The algorithm has N/b iterations, within each iteration, nodes with sub-matrix A' that forms part of the pivot column will send their part horizontally and nodes with sub-matrix B' that forms part of the pivot blocks from the pivot row will broadcast their part vertically. All nodes will receive into a buffer $A_{(b)}$ of size $bm_i \times b$ and $B_{(b)}$ of size $b \times bn_i$. Then on each node Q_i with devices P_{ij}, for $0 \leq j < p_i$, device P_{ij} will do the matrix operation $C'_j = C'_j + A_{(b)} \times B_{(b)j}$ where sub-matrix C'_j is of size $bm_i \times d_{ij}$ and sub-matrix B'_j is of size $b \times d_{ij}$ (Fig. 2(b)). Therefore the kernel that is benchmarked for this application is the dgemm operation $C'_j = C'_j + A_{(b)} \times B_{(b)j}$.

The Grid'5000 experimental testbed proved to be an ideal platform to test our application. We used 90 dedicated nodes from 3 clusters from the Grenoble site. 12 of these nodes from the Adonis cluster included NVIDIA Tesla GPUs. The remaining nodes where approximately homogeneous. In order to increase the impact of our experiments we chose to utilise only some of the CPU cores on some machines (Table 1). Such an approach is not unrealistic since it is possible to book individual CPU cores on this platform. For the local *dgemm* routine we used high performance vendor-provided BLAS libraries, namely Intel MKL for CPU and CUBLAS for GPU devices. Open MPI was used for inter-node communication and OpenMP for inter-device parallelism. The GPU execution time includes the time to transfer data to the GPU. For these experiments, an out of core algorithm is not used when the GPU memory is exhausted. All nodes are interconnected by a high speed InfiniBand network which reduces the

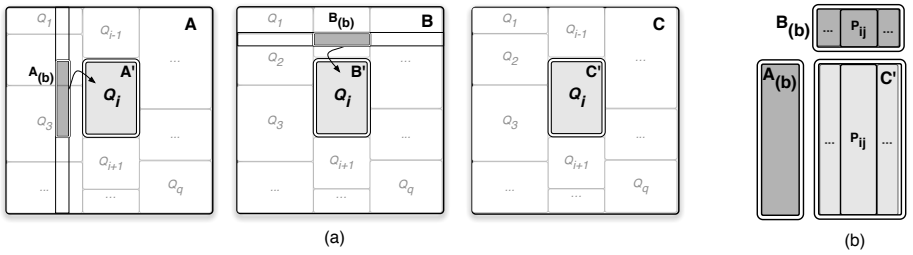

Fig. 2. Parallel matrix multiplication algorithm: (a) two-dimensional blocked matrix multiplication between the nodes; (b) one-dimensional matrix multiplication within a node

Table 1. Experimental hardware setup using 90 nodes from three clusters of the Grenoble site from Grid'5000. All nodes have 8 CPU cores, however, to increase heterogeneity only some of the CPU cores are utilised as tabulated below. One GPU was used with each node from the Adonis cluster, 10 nodes have Tesla T10 GPU and 2 nodes have Tesla C2050 GPU, and an CPU core was devoted to control execution on the GPU. As an example, we can read from the table that two Adonis nodes used only 1 GPU and 6 Edel nodes used just 1 CPU core. All nodes are connected with InfiniBand 20G & 40G.

Cores:	0	1	2	3	4	5	6	7	8	Nodes	CPU Cores	GPUs	Hardware
Adonis	2	1	1	1	1	1	2	3	0	12	48	12	2.27/2.4GHz Xeon, 24GB
Edel	0	6	4	4	4	8	8	8	8	50	250	0	2.27GHz Xeon, 24GB
Genepi	0	3	3	3	3	4	4	4	4	28	134	0	2.5GHz Xeon, 8GB
Total										90	432	12	

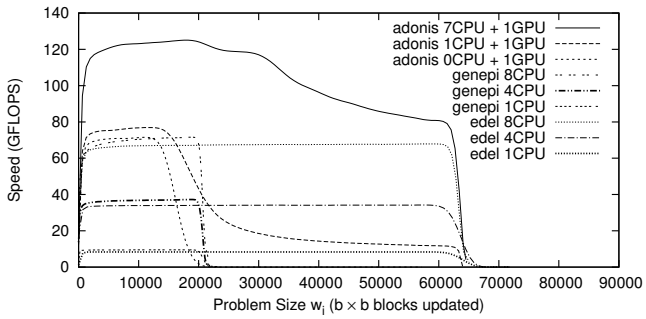

Fig. 3. Full functional performance models for a number of nodes from Grid'5000 Grenoble site. Problem size is in number of $b \times b$ blocks of matrix C updated by a node. For each data point in the node model it was necessary to build device models, find the optimum inter-device distribution and then measure the execution time of the kernel with this distribution.

impact of communication on the total execution time, for $N = 1.5 \times 10^5$ all communications (including wait time due to any load imbalance) took 6% of total execution time. The full functional performance models of nodes, Fig. 3, illustrate the range of heterogeneity of our platform.

Before commencing full scale experiments it was necessary to find an appropriate block size b. A large value of b allows the optimised BLAS libraries to achieve their peak performance as well as reducing the number of communications, while a small value of b allows fine grained load balancing between nodes. We conducted a serious of experiments, using one Adonis node with 7 CPU cores + 1GPU, for a range of problem sizes and a range of values of b. The IDPA was used to find the optimum distribution between CPU cores and GPU. As shown in Fig. 4, a value of $b = 128$ achieves near-peak performance,

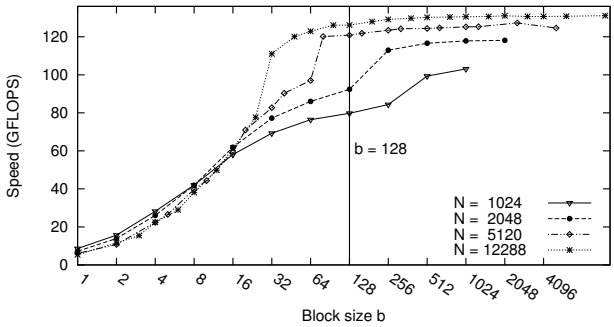

Fig. 4. Overall node performance obtained for different ranges of block and problem sizes when running optimal distribution between 7 CPU cores and a GPU

especially as N increases, while still allowing reasonably fine grained inter-node load balancing. For all subsequent experiments we used $b = 128$.

In order to demonstrate the effectiveness of the proposed FPM-based partitioning algorithm we compare it against 3 other partitioning algorithms. All four algorithms invoke the *communication minimisation algorithm* and are applied to an identical parallel matrix multiplication application. They differ on how load balancing decisions are made.

- **Multiple-CPM Partitioning** uses the same algorithm as proposed above, with step 7 of the INPA and step 5 of the IDPA replaced with $w_i = W \times \frac{s_i}{\sum_q s_i}$ and $d_j = bn \times \frac{s_j}{\sum_p s_j}$ respectively, where s_i and s_j are constants. This is equivalent to the approach used in [8, 19, 18].
- **Single-CPM Partitioning** does one iteration of the above multiple-CPM partitioning algorithm. This is equivalent to the approach used in [10, 2].
- **Homogeneous Partitioning** uses an even distribution between all nodes: $w_1 = w_2 = \cdots = w_q$ and between devices in a node: $d_{i1} = d_{i2} = \cdots = d_{ip_i}$.

Fig. 5 shows the speed achieved by the parallel matrix multiplication application when the four different algorithms are applied. It is worth emphasizeing that the performance results related to the execution on GPU devices take into account the time to transfer the workload to/from the GPU. The speed of the application with the *homogeneous distribution* is governed by the speed of the slowest processor (a node from Edel cluster with 1CPU core). The *Single-CPM* and *multiple-CPM* partitioning algorithms are able to load balance for N up to 60000 and 75000 respectivly, however this is only because the speed functions in these regions are horozontal. In general, for a full range of problem sizes, the simplistic algorithms are unable to converge to a balanced solution. By chance, for $N = 124032$, the multiple-CPM algorithm found a reasonably good partitioning after many iterations, but in general this is not the case. Meanwhile the *FPM-based partitioning* algorithm reliably found good partitioning for matrix multiplication involving in excess of 0.5TB of data.

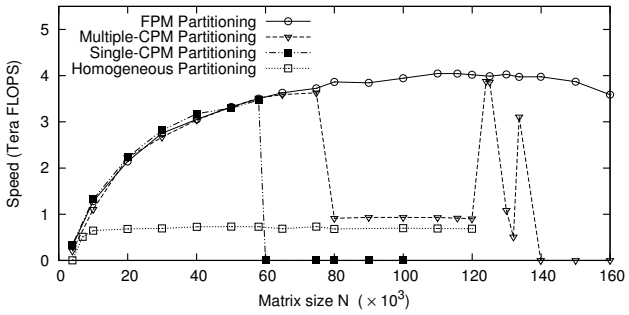

Fig. 5. Absolute speed for a parallel matrix multiplication application based on four partitioning algorithms. Using 90 heterogeneous nodes consisting of 432 CPU cores and 12 GPUs from 3 dedicated clusters.

5 Conclusions

In this paper a novel hierarchical partitioning algorithm for highly heterogeneous (CPU+GPU) clusters was presented. The algorithm load balances an application run on a hierarchical platform by optimally partitioning the workloads at both levels of hierarchy, i.e. nodes and processing devices. The presented approach is based on realistic functional performance models of processing elements which are obtained empirically in order to capture the high level of platform's heterogeneity. The efficiency of the proposed algorithm was tested in a real system consisting of 90 highly heterogeneous nodes in 3 computing clusters and compared to similar approaches for a parallel matrix multiplication case. The results show that the presented algorithm was not only capable of minimising the overall communication volume in such a complex environment, but it was also capable of providing efficient load balancing decisions for very large problem sizes where similar approaches were not able to find the adequate balancing solutions. Future work will include an out of core device kernel for when the memory limit of a device is reached; a communication efficient inter-device partitioning and multi-GPU experimental results.

Acknowledgments. This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054. This work was supported by FCT through the PIDDAC Program funds (INESC-ID multiannual funding) and a fellowship SFRH/BD/44568/2008. Experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This work was also partially supported by the STSM COST Action IC0805.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Euro-Par 2009

Parallel Processing pp. 863–874 (2009)
2. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix Multiplication on Heterogeneous Platforms. IEEE Trans. Parallel Distrib. Syst. 12(10), 1033–1051 (2001)
3. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. JACM 46(5), 720–748 (1999)
4. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. Concurrency: Practice and Experience 10(8), 655–670 (1998)
5. Clarke, D., Lastovetsky, A., Rychkov, V.: Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In: Euro-Par/HeteroPar 2011. Bordeaux, France (August 2011)
6. Dongarra, J., Faverge, M., Herault, T., Langou, J., Robert, Y.: Hierarchical qr factorization algorithms for multi-core cluster systems. Arxiv preprint arXiv:1110.1553 (2011)
7. Drozdowski, M., Lawenda, M.: On optimum multi-installment divisible load processing in heterogeneous distributed systems. In: Euro-Par. pp. 231–240 (2005)
8. Galindo, I., Almeida, F., Bada-Contelles, J.: Dynamic Load Balancing on Dedicated Heterogeneous Systems. In: EuroPVM/MPI 2008. pp. 64–74. Springer (2008)
9. Horton, M., Tomov, S., Dongarra, J.: A class of hybrid lapack algorithms for multicore and gpu architectures. In: SAAHPC. pp. 150–158 (2011)
10. Hummel, S., Schmidt, J., Uma, R.N., Wein, J.: Load-sharing in heterogeneous systems via weighted factoring. In: SPAA96. pp. 318–328. ACM (1996)
11. Ilic, A., Sousa, L.: Collaborative execution environment for heterogeneous parallel systems. In: IPDPS Workshops and Phd Forum (IPDPSW). pp. 1 –8 (2010)
12. Ilic, A., Sousa, L.: On realistic divisible load scheduling in highly heterogeneous distributed systems. In: PDP 2012. Garching, Germany (2012)
13. Jacobsen, D.A., Thibault, J.C., Senocak, I.: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In: AIAA Aerospace Sciences Meeting proceedings (2010)
14. Kalinov, A., Lastovetsky, A.: Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In: HPCN Europe 1999, LNCS 1593. pp. 191–200. Springer Verlag (1999)
15. Kindratenko, V.V., otheres: GPU clusters for high-performance computing. In: CLUSTER. pp. 1–8 (2009)
16. Lastovetsky, A., Reddy, R.: Data Partitioning with a Functional Performance Model of Heterogeneous Processors. Int. J. High Perform. Comput. Appl. 21(1), 76–90 (2007)
17. Lastovetsky, A., Reddy, R., Rychkov, V., Clarke, D.: Design and implementation of self-adaptable parallel algorithms for scientific computing on highly heterogeneous HPC platforms. Arxiv preprint arXiv:1109.3074 (2011)
18. Legrand, A., Renard, H., Robert, Y., Vivien, F.: Mapping and load-balancing iterative computations. Parallel and Distributed Systems, IEEE Transactions on 15(6), 546–558 (2004)
19. Martínez, J., Garzón, E., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. J. Supercomput. (2009)
20. Quintin, J., Wagner, F.: Hierarchical work-stealing. Euro-Par 2010-Parallel Processing pp. 217–229 (2010)
21. Veeravalli, B., Ghose, D., Robertazzi, T.G.: Divisible load theory: A new paradigm for load scheduling in distributed systems. Cluster Computing 6, 7–17 (2003)