A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms

Hamidreza Khaleghzadeh, Ravi Reddy, and Alexey Lastovetsky, *Member, IEEE*

Abstract—Modern HPC platforms have become highly heterogeneous owing to tight integration of multicore CPUs and accelerators (such as Graphics Processing Units, Intel Xeon Phis, or Field-Programmable Gate Arrays) empowering them to maximize the dominant objectives of performance and energy efficiency. Due to this inherent characteristic, processing elements contend for shared on-chip resources such as Last Level Cache (LLC), interconnect, etc. and shared nodal resources such as DRAM, PCI-E links, etc. This has resulted in severe resource contention and Non-Uniform Memory Access (NUMA) that have posed serious challenges to model and algorithm developers. Moreover, the accelerators feature limited main memory compared to the multicore CPU host and are connected to it via limited bandwidth PCI-E links thereby requiring support for efficient out-of-card execution.
To summarize, the complexities (resource contention, NUMA, accelerator-specific limitations, etc.) have introduced new challenges to optimization of data-parallel applications on these platforms for performance. Due to these complexities, the performance profiles of data-parallel applications executing on these platforms are not smooth and deviate significantly from the shapes that allowed state-of-the-art load-balancing algorithms to find optimal solutions.
In this paper, we formulate the problem of optimization of data-parallel applications on modern heterogeneous HPC platforms for performance. We then propose a new model-based data partitioning algorithm, which minimizes the execution time of computations in the parallel execution of the application. This algorithm takes as input a set of p discrete speed functions corresponding to p available heterogeneous processors. It does not make any assumptions about the shapes of these functions. We prove the correctness of the algorithm and its complexity of $O(m^3 \times p^3)$, where m is the cardinality of the input discrete speed functions.
We experimentally demonstrate the optimality and efficiency of our algorithm using two data-parallel applications, matrix multiplication and fast Fourier transform, on a heterogeneous cluster of nodes where each node contains an Intel multicore Haswell CPU, an Nvidia K40c GPU, and an Intel Xeon Phi co-processor.

Index Terms—heterogeneous HPC platforms, data-parallel applications, data partitioning, load balancing, multicore CPU, GPU, Intel Xeon Phi, performance optimization

1 Introduction

Modern HPC platforms have become highly heterogeneous owing to tight integration of multicore CPUs and accelerators (such as GPUs, Intel Xeon Phis, or FPGAs) to maximize the dominant objectives of performance and energy efficiency. The current Top500 list [1] includes sixty systems with Intel/AMD multi-core CPUs and Nvidia GPU accelerators, and twenty one systems with Intel Xeon Phi accelerators. Furthermore, there are three homogeneous clusters with hybrid nodes consisting of Intel Xeon Phi and Nvidia Kepler accelerators.

The inherent characteristic of tight integration, however, has resulted in processing elements contending for shared on-chip resources such as Last Level Cache (LLC), interconnect, etc. and shared nodal resources such as DRAM, PCI-E links, etc. This has caused severe resource contention and Non-uniform Memory Access (NUMA) that pose serious challenges to model and algorithm developers. Moreover, the accelerators feature limited main memory compared to the multicore CPU host and are connected to it via limited bandwidth PCI-E links thereby necessitating support for efficient out-of-card execution. To summarize, the newly introduced complexities (resource contention, NUMA, accelerator-specific limitations, etc.) have created new challenges to optimization of data-parallel applications on these platforms for performance.

Before we present use cases that elucidate the challenges especially for clusters of heterogeneous processors, we briefly study the evolution of performance models and data partitioning algorithms that have attempted to realistically capture the real-life behaviour of applications executing on these platforms for performance maximization.

The simplest models used positive constant numbers and different notions such as normalized processor speed, normalized cycle time, task computation time, average execution time, etc., to characterize the speed of an application [2], [3], [4]. The common aspect of these models is that the performance of a processor is assumed to have no dependence on the size of the workload. We call them the constant performance models (CPMs).

The most advanced load balancing algorithms use functional performance models (FPMs), which are application-specific. The FPMs represent the speed of a processor by continuous function of problem size but satisfying some assumptions on its shape [5],[6]. The assumptions require

• *H. Khaleghzadeh, R. Reddy and A. Lastovetsky are with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: hamidreza.khaleghzadeh@ucdconnect.ie, ravi.manumachu@ucd.ie, alexey.lastovetsky@ucd.ie*

them to be smooth enough in order to guarantee that optimal solutions minimizing the computation time are always load balanced. The FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

However, modern HPC platforms have complex nodal architectures with highly hierarchical arrangement and tight integration of processors where resource contention and NUMA are inherent. On such platforms, the performance profiles of real-life scientific applications are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load balancing algorithms to find optimal solutions.

Lastovetsky et al. [7] study the drastic deviations in the performance profiles for a real-life scientific application, Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), in a Xeon Phi co-processor. MPDATA is a core component of the EULAG (Eulerian/semi-Lagrangian fluid solver) geophysical model [8], which is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. The authors propose an optimization technique reusing an advanced performance model of computation (FPM) but using novel load distribution to minimize the computation time of the application. Lastovetsky et al. [9] illustrate in depth these variations in performance and energy profiles of two widely known and highly optimized scientific routines, OpenBLAS DGEMM [10] and FFTW [11], on a modern multicore Intel Haswell CPU platform. They explain the limitations of the FPM-based load balancing algorithms proposed in [12], [13], [14], [15], [16], [17], [18], [19], [20]. They propose novel model-based methods and algorithms for minimization of time and energy of computations for the most general performance and energy profiles of data parallel applications executing on homogeneous multicore clusters. Unlike load balancing algorithms, optimal solutions found by these algorithms may not load-balance an application.

The new model-based methods proposed in [7], [9], however, can not be used for optimization of data-parallel applications on HPC platforms with hybrid nodes for maximization of performance since they are designed for homogeneous clusters, i.e., cluster of identical processors.

We now present two motivational use cases that elucidate the additional challenges that arise in HPC platforms with hybrid nodes.

The hybrid node, *HCLServer*, that we use for the experiments, is a multi-accelerator NUMA platform. It contains an Intel Haswell multicore CPU consisting of 24 physical cores with 64 GB main memory, whose specification is shown in the Table 1. In addition to the multicore CPU, the node integrates two accelerators, Nvidia K40c GPU and Intel Xeon Phi 3120P, whose specifications are shown in Tables 2 and 3 respectively. Each accelerator is connected to a dedicated host core via a separate PCI-E link.

In each use case, we study the performance profiles of a data-parallel application executing in the hybrid node and modelled by three abstract processors. A processing unit made of one or a group of CPU cores executing one (generally speaking, multi-threaded) computational kernel of the data parallel application is modelled by an abstract processor [21]. To build the performance models of the abstract processors, the performance of the processing units representing these processors must be measured accurately. To ensure this, the processing units are grouped by shared system resources. Each group becomes an abstract processor. The performance of processing units in a group is measured when all the processing units in the group are executing a workload simultaneously, thereby taking into account the influence of resource contention. We represent a performance model by a discrete function of speed versus the problem size.

The first abstract processor contains 22 CPU cores executing the multi-threaded CPU kernel. The second abstract processor comprises the Nvidia K40c GPU along with its dedicated host CPU core executing the GPU kernel. And finally, the third abstract processor consists of Intel Xeon Phi 3120P co-processor along with its dedicated host CPU

TABLE 1
Specification of the Intel Haswell multicore CPU.

Technical Specifications	Intel Haswell E5-2670V3
Thread(s) per core	2
No. of cores per socket	12
Socket(s)	2
NUMA node(s)	2
CPU MHz	1200.402
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	30720 KB
NUMA node0 CPU(s)	0-11,24-35
NUMA node1 CPU(s)	12-23,36-47
Processor base frequency	2.30 GHz
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
TDP	240 W
Idle Power	61 W

TABLE 2
Specification of the Nvidia K40c GPU.

Technical Specifications	Nvidia K40c
No. of processor cores	2880
Base clock	745 MHz
Boost clock(s)	810 MHz, 875 MHz
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Memory I/O	384-bit GDDR5
Memory clock	3.0 GHz
TDP	235 W
Idle Power	16 W
Idle Power (Persistence mode)	68 W

TABLE 3
Specification of the Intel Xeon Phi 3120P.

Technical Specifications	Intel Xeon Phi 3120P
No. of processor cores	57
Base frequency	1.10 GHz
Total main memory	6 GB GDDR5
L2 cache size	28.5 MB
Memory bandwidth	240 GB/sec
Memory clock	3.0 GHz
TDP	300 W
Idle Power	91 W

core executing the Xeon Phi kernel. The dedicated host CPU core is responsible for sending data from host to accelerator, kernel invocations on the accelerator and then copying results back from accelerator to host. Therefore, the pair consisting of an accelerator and its dedicated host core executing one accelerator kernel is modelled by an abstract processor. The kernel executing on the accelerator uses all the cores of the accelerator. The execution time of a kernel in the GPU and Xeon Phi abstract processors includes the times of data transfer between the accelerators and their host cores.

Since the abstract processors contain CPU cores that share main memory, they cannot be considered independent. Therefore, the performance of these abstract processors must be measured simultaneously. That is, the data points for a problem size in their performance models are experimentally built simultaneously. We explain how to do this in our experimental methodology presented in the supplemental. It should be noted that while performance models are built where the data points for the same problem size are obtained simultaneously, during the actual execution of the data-parallel application using the workload distribution determined by our data partitioning algorithm, the problem sizes executed by the abstract processors can be different. This is because different processors can be allocated different problem sizes by our heterogeneous data partitioning algorithm. However, the speeds of execution of these problem sizes simultaneously would not differ significantly from those present in the performance models; the marginal differences do not imply significantly different execution times. We confirm this to be the case through exhaustive experimentation; synopsis of this is presented in the supplemental.

In our first use case, we study the performance profile of a matrix multiplication application. The application executes a highly optimized native kernel for CPU and highly optimized out-of-card kernels for the accelerators. The out-of-card kernels allow the GPU and Xeon Phi abstract processors to execute tasks of arbitrary size, not just the ones that fully fit in the accelerator memories [22]. For the multicore CPU, Intel MKL DGEMM is used. For GPU, ZZGEMMOOC out-of-card package [23] is used that reuses CUBLAS for in-card DGEMM calls. For Xeon Phi, XeonPhiOOC out-of-card package [24] is used that reuses MKL BLAS for in-card DGEMM calls. The Intel MKL and CUDA versions used are 2017.0.2 and 7.5 respectively. Since the number of threads per core in Intel Haswell is equal to 2, the Intel MKL DGEMM kernel for the multicore CPU uses 44 threads executing on 22 out of 24 physical cores.

Figure 1 shows DGEMM speed functions of CPU, GPU and Xeon Phi abstract processors along with their zoomed functions between two data points 3712^2 and 6272^2. One can observe significant fluctuations in the performance profile, which we call variations. The variation is related to the difference of speed between two subsequent local minima (s_1) and maxima (s_2) and is defined below:

$$variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100 \quad (1)$$

To make sure that our experimental results are reliable and it is not noise that is the underlying cause behind these variations, the experiments for each data point in speed functions are repeated until sample means of all the three kernels executing on the abstract processors simultaneously fall in the interval with the confidence level, 95 percent. The statistical methodology is explained in detail in the supplemental. Briefly, the methodology contains following main steps: 1) We make sure the platform is fully reserved and dedicated to our experiments and exhibits clean behaviour by monitoring its load continuously for a week. 2) For each data point in the speed functions of an application, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by checking the density plots of the observations.

From the figure, we can observe the following:

- Xeon Phi speed function is almost smooth between 64^2 to 13760^2. However, the variations increase for larger problem sizes (13824^2 and beyond) where DGEMM out-of-card computations are invoked. Unlike Xeon Phi, the variations decrease for CPU and GPU as problem size increases. The maximum variations for CPU, GPU and Xeon Phi are 700%, 50% and 150%, respectively. The maximum variations for Xeon Phi occur for problem sizes in the range $[12800^2, 19200^2]$.
- The shapes violate the assumptions of FPMs. Therefore, load balancing data partitioning algorithms based on FPMs may not return optimal solutions.
- The new model-based methods proposed in [7], [9] can not be used since they consider all the available processors to be identical and therefore take a single speed function as an input.

In our second use case, we study the performance profiles of a 2D discrete Fourier transform (DFT) application and modelled by the same three abstract processors. For the multicore CPU and Xeon Phi, Intel MKL FFT is used. For the Nvidia GPU, CUFFT is used. Unlike the matrix multiplication application, all computations for FFT are in-core (or in-card for the accelerators). Figure 2 shows the speed functions of abstract processors along with their zoomed functions between two data points 5456^2 and 6176^2. The Intel MKL FFT kernel for the multicore CPU uses 44 threads executing on 22 out of 24 physical cores.

From the figure, we can observe that Xeon Phi is markedly slower than CPU and GPU. It is because the execution time of communications between Xeon Phi and host CPU dominates the execution time of computations performed by Xeon Phi. However, GPU uses optimized data transfers by deploying two data engines (for transfers from CPU host to GPU and from GPU to CPU host) and does not suffer from this problem. The maximum variations for CPU, GPU, and Xeon Phi are almost 350%, 560% and 200%, respectively. The maximum variations for Xeon Phi occur for problem sizes in the range of $[16^2, 800^2]$. It can be seen again that the shapes violate the assumptions on shape of FPMs. Therefore, load balancing data partitioning algo-

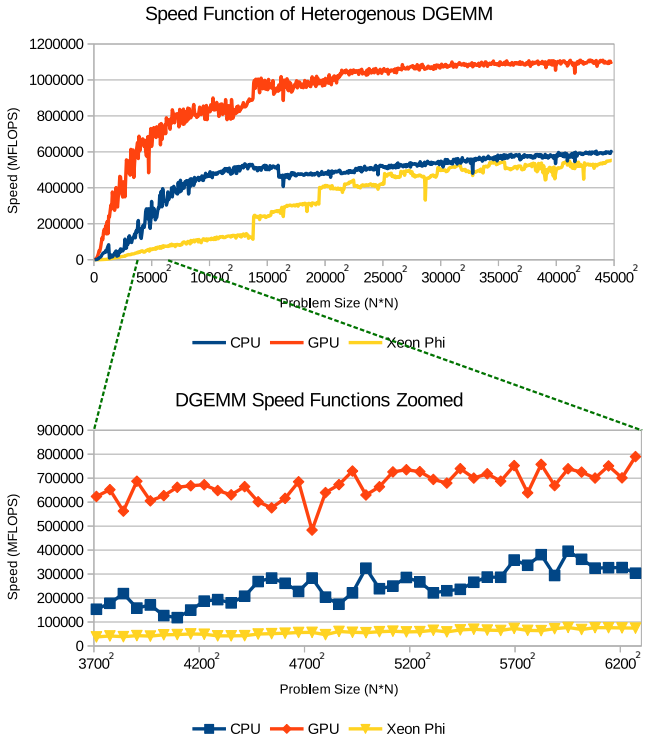

Fig. 1. Speed functions of heterogeneous DGEMM application for the multicore CPU, GPU, and Xeon Phi in the *HCLServer*. Also shown is the zoomed speed function between two points 3712^2 and 6272^2.

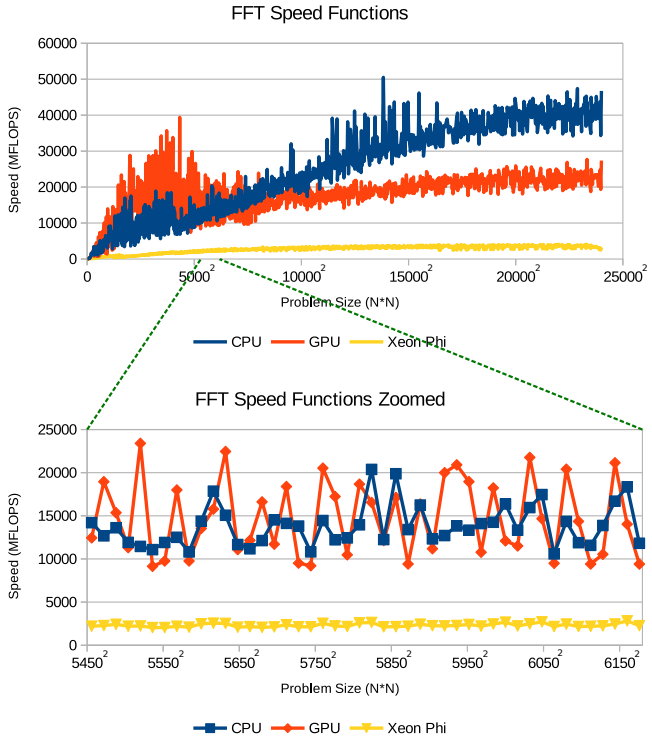

Fig. 2. Speed functions of heterogeneous 2D DFT application for the multicore CPU, GPU, and Xeon Phi in the *HCLServer*. Also shown is the zoomed speed function between two points 5456^2 and 6176^2.

rithms based on FPMs may not return optimal solutions. Also the new model-based methods proposed in [7], [9] can not be used for this case.

In the supplemental, we present one more use case demonstrating the variations in the performance profiles for a matrix-vector multiplication application executing the highly optimized multi-threaded Intel MKL DGEMV routine in a homogeneous cluster of six nodes where each node contains two Intel Xeon Phi accelerators.

Thus, the three presented use cases illustrate the dramatic variations observed in performance profiles of highly optimized scientific applications executing on heterogeneous HPC platforms. These variations are not singular and will become common because chip manufacturers are increasingly featuring tighter integration of processor cores, memory, and interconnect in their products. It is these variations that have now made the optimization problem for performance on such platforms difficult to solve. Moreover, the state-of-the-art load balancing algorithms based on FPMs and the novel model-based methods proposed in [7], [9] are not equipped to deal with such cases where different processors exhibit different shapes of speed functions.

In this paper, we propose a new model-based data partitioning algorithm, which minimizes parallel execution time for the most general shapes of performance profiles for data parallel applications executing on heterogeneous clusters of hybrid nodes. First, we formulate the problem of optimization of data-parallel applications on such platforms for performance. We then present the data-partitioning algorithm called *HPOPTA* with complexity $O(m^3 \times p^3)$ where m is the number of data points in the speed functions and p is the number of available processors. Optimal solutions found by *HPOPTA* may not be balanced in terms of execution time.

It should be mentioned that *HPOPTA* is a 1D data-partitioning algorithm. However, *HPOPTA* can be directly applied to 2D or 3D problems where the dimensionality can be reduced to 1D. Consider two examples. Our first example is the execution of *MPDATA* on Intel multicore CPUs and Intel Xeon Phis [7]. The input data structure to *MPDATA* is a dense 3D object with dimensions (m,n,l) and size $m \times n \times l$. The dimension l is fixed in real-life simulations. From our experiments, we observed that the speed of *MPDATA* varies very little with n for constant m. Therefore, *HPOPTA* can be applied directly to performance optimization of *MPDATA* where the parameter m is partitioned between the processors. In our second example, we consider the application of *HPOPTA* for optimization of *2D DFT* for performance. A sequential *2D DFT* is computed using row-column or the separable method based on 1D FFTs. Briefly, the row-column method consists of 1D FFTs on rows followed by transpose matrix and then 1D FFTs on rows followed by restoration using transpose matrix. The *1D FFT* computation is optimized using direct application of *HPOPTA*. In our future work, we will develop extensions to *HPOPTA* for optimization of 2D applications (for example, matrix multiplication) for performance.

To summarize, our main contributions in this paper are:

- Studying the behaviour of data parallel applications on modern heterogeneous clusters of hybrid nodes, and the challenges introduced to model and algorithm design because of resource contention and

NUMA for their performance optimization.
- Efficient algorithm *HPOPTA* for optimization of data-parallel applications on heterogeneous HPC platforms for performance.
- Experimental validation of *HPOPTA* on a hybrid heterogeneous server integrating a multicore CPU, a GPU and a Xeon Phi.
- Simulations demonstrating the efficiency of *HPOPTA* for large-scale parallel platforms.
- A hierarchical two-level workload distribution algorithm using *HPOPTA* on a cluster of identical hybrid nodes.

The rest of the paper is organized as follows. Section 2 surveys load-balancing techniques and novel model-based algorithms, which minimize the execution time of parallel applications. Section 3 formulates the problem of performance optimization for clusters of heterogeneous processors. In Section 4, a data partitioning algorithm called *HPOPTA* is presented to solve the performance optimization problem. Section 5 contains the formal description of the proposed algorithm. Section 6 presents experimental analysis of the algorithm. Finally, section 7 concludes the paper.

2 Related Work

In this section, we overview state-of-the-art load-balancing algorithms since they have been a dominant class of algorithms for performance optimization on parallel platforms. We then discuss the latest research works that look at the dramatic variations in performance caused by the inherent complexities of resource contention and NUMA in homogeneous multicore clusters.

There are several classifications of load-balancing algorithms: *static* or *dynamic*, *non-centralized* or *centralized*, *task queue* or *predicting-the-future*. One can find edifying descriptions of these classifications in [14], [21], [7].

Static algorithms, such as those based on data partitioning [25], [26], [27], [6], [28], use *a priori* information about the parallel application and platform. These algorithms are also known as *predicting-the-future* because they rely on accurate performance models as input to predict the future execution of the application. They are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms may be unsuitable for non-dedicated platforms, where load changes with time.

Dynamic algorithms, such as task scheduling and work stealing [29], [30], [31], [32], [33] balance the load by moving fine-grained tasks between processors during the execution. They do not require *a priori* information about execution but may incur large communication overhead due to data migration. They can use static partitioning for the initial step due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead. Dynamic load balancers based on graph partitioners are proposed by [34], [35] for adaptive scientific computations where two objectives, interprocessor communication and data migration costs, are considered.

In non-centralised algorithms [36], [37], load is migrated locally between neighbouring processors, while in centralised ones [38], [39], [40], load is distributed based on global load information. Non-centralised algorithms are slower to converge. At the same time, centralised algorithms typically have higher overhead. The centralised algorithms can be further subdivided into two groups: task queue [39] and predicting-the-future [38], [40].

Over the years, load balancing algorithms developed for performance optimization on parallel platforms have attempted to take into consideration the real-life behaviour of applications executing on these platforms. This can be discerned from the evolution of performance models for computation used in these algorithms. The simplest models used positive constant numbers and different notions such as normalized processor speed, normalized cycle time, task computation time, average execution time, etc. to characterize the speed of an application [41] [2], [3], [4] [42] [43] [44]. A singular feature of these efforts is that the performance of a processor is assumed to have no dependence on the size of the workload. The most advanced load balancing algorithms use functional performance models (FPMs), which are application-specific and represent the speed of a processor by continuous function of problem size but satisfying some assumptions on its shape [5],[45][6]. These FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

Performance profiles of modern HPC systems involve lots of variations and violate the conditions assumed by the proposed FPM-based algorithms proposed in [12], [13], [14], [15], [16], [17], [18], [19], [20]. To deal with this challenge, novel model-based algorithms have been proposed which are able to find optimal workload distribution on state-of-art homogeneous systems. The proposed approaches make no assumptions about the shapes of performance profiles.

Lastovetsky et al. [7] propose an optimization technique reusing an advanced performance model of computation (FPM) by using novel load distribution to minimize the computation time. First, they experimentally build the speed function of the application using a wide range of problem sizes separated by minimum granularity. They then use this function and its connected visual picture to distribute computations unevenly between homogeneous groups of cores of the Xeon Phi co-processor, therefore load imbalancing the application, to achieve performance optimization. This is the first work where the load-imbalancing technique is applied to partition the workload minimizing the computation time of its parallel execution. However, no general partitioning algorithm is proposed in this work. Lastovetsky et al. [9] propose such general model-based methods and algorithms for minimization of not only the time but also the energy of computations for the most general performance and energy profiles of data parallel applications executing on *homogeneous* multicore clusters. They formulate the performance and energy optimization problems and present efficient algorithms of complexity $O(m^2 \times p^2)$ solving these problems where m is the cardinality of the discrete sets representing the speed/energy functions and p is the number of available processors. The memory complexity of the algorithms is $O(n \times p^2)$. Unlike load balancing algorithms, optimal solutions found by these algorithms may not load-balance an application.

Apart from [7], [9] that have explored in-depth the deter-

ministic and reproducible performance variations for bound applications, Zhang et al. [46] also report significant non-deterministic variations for applications that are not bound to the cores of the executing multicore platform. Their approach is to try to reduce the non-deterministic variations by using different execution patterns. In our work, we use the deterministic variations in order to find the optimal parallel configuration of a data-parallel application that minimizes the computation time of its parallel execution.

3 Formulation of Performance Optimization Problem

Consider a workload of size n executed using p heterogeneous processors, whose speed functions are represented by $S = \{s_0(x), ..., s_{p-1}(x)\}$ where $s_i(x)$, $i \in \{0, 1, \cdots, p-1\}$, is a discrete speed function of cardinality m of processor P_i. The speed $s_i(x)$ for a problem size x for processor i is calculated as $\frac{x}{t_i(x)}$, where $t_i(x)$ is the time of execution of the problem size. Without loss of generality, we assume $x \in \{1, 2, \cdots, m\}$. The performance optimization problem can be then formulated as follows:

Performance Optimization Problem, *HPOPT*(n, p, m, S, D_{opt}, t_{opt}): The problem is to find a partitioning, $D_{opt} = \{x_0, ..., x_{p-1}\}$, of the workload of size n using p available heterogeneous processors so as to minimize the computation time of parallel execution of the workload. The parameters (n, p, m, S) are the inputs to the problem. The outputs are D_{opt}, which is the workload distribution, and t_{opt}, which is the optimal execution time. This problem can be formulated as an integer nonlinear programming problem (INLP) as follows:

$$
\begin{aligned}
t_{opt} &= \min_{D} \max_{i=0}^{p-1} \frac{x_i}{s_i(x_i)} \\
\text{Subject to} \quad & x_0 + x_1 + ... + x_{p-1} = n \\
& 0 \leq x_i \leq m, \qquad i = 0, \cdots, p-1 \\
\text{where} \quad & p, m, n \in \mathbb{Z}_{>0} \quad \text{and} \quad x_i \in \mathbb{Z}_{\geq 0} \quad \text{and} \\
& s_i(x) \in \mathbb{R}_{>0}
\end{aligned}
\tag{2}
$$

It should be noted that the execution time $t_i(x)$ for a problem size x for processor i is calculated as $\frac{x}{s_i(x)}$, where $s_i(x)$ is the speed of execution of the problem size.

The objective function in the formulated optimization problem is a function of workload distribution D, $D = \{x_0, ..., x_{p-1}\}$, of a given workload n between the p processors. For each given D, it returns the time of its parallel execution, which is calculated as the time taken by the longest running processor to execute its workload. Any distribution that minimizes this function is considered optimal as its execution time of workload n by the p processors cannot be improved.

4 *HPOPTA*: Algorithm Solving HPOPT

In this section, we present our algorithm, *HPOPTA* (**H**eterogeneous **P**erformance **OPT**imization **A**lgorithm), that solves HPOPT.

First, we informally describe the algorithm using an example. The input to the algorithm are discrete time functions, which are derived from discrete speed functions.

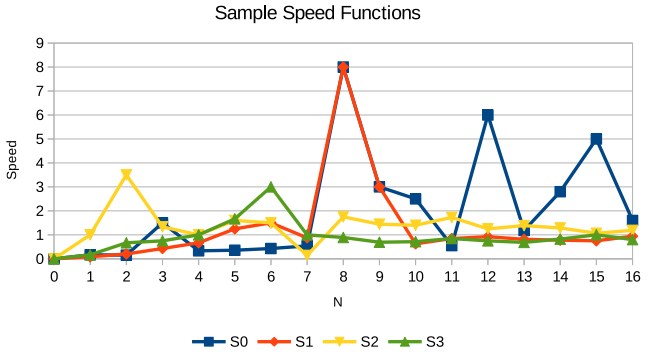

Fig. 3. Speed functions of a sample application executing on an assumed parallel machine which consists of 4 processors.

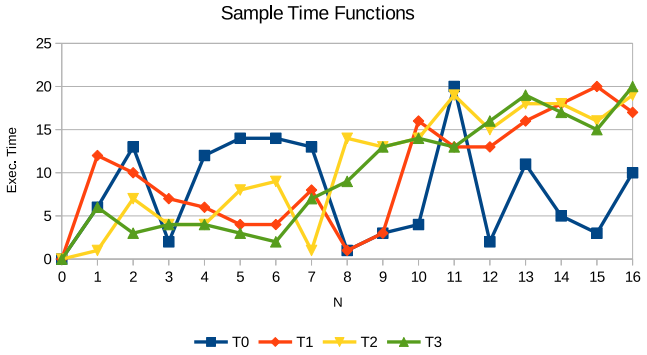

Fig. 4. The equivalent time functions for the sample speed functions in Fig. 3.

In the example, consider four heterogeneous processors ($p = 4$), which are available for execution of a workload of size $n = 16$. Figures 3 and 4 respectively show the sample speed functions, $S = \{s_0(x), \cdots, s_3(x)\}$, and the equivalent time functions, $T = \{t_0(x), \cdots, t_3(x)\}$, of the processors ($m = n = 16$ in our example for simplicity). It should be noted that these time functions are samples, which are representative of real-life data-parallel applications.

Figure 5 shows the discrete time functions, stored as arrays in non-decreasing order of execution times.

To find the optimal workload distribution, a straightforward approach would be to examine all combinations and select a workload distribution with the minimum computation time of parallel execution of the workload. Figure 6 shows the tree, which is constructed by such a naive algorithm and contains all the combinations. Due to the lack of space, we only show the tree partially.

The solution tree is constructed from the root, which is the only node at level L_0 of the tree. The value 16, which labels the root node, represents the whole workload size to be distributed between 4 processors $\{P_0, P_1, P_2, P_3\}$. Then, 17 workload sizes, including a zero workload size along with all workload sizes existing in the time function ($t_0(x)$), are assigned to the processor P_0 one by one. Although the workload sizes can be given to the processor in any order, we assign them in a non-decreasing order of their execution time by the processor. As shown in Figure 6, problem sizes $\{0, 8, 3, 12, 9, 15, 10, 14, 1, 16, 13, 4, 2, 7, 5, 6, 11\}$,

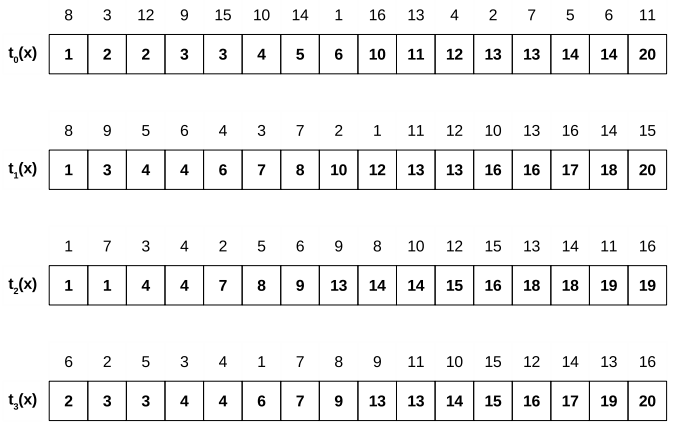

Fig. 5. Example: The sample time functions, shown in Figure 4, which are stored in array data structures. Each array is sorted in non-decreasing of execution times.

which have been sorted in non-decreasing order of execution times, are assigned to P_0 one-by-one at level L_0. Therefore, the root node is expanded into 17 children. The value, which labels an internal node at level L_1 (the root's child), represents the remaining workload size to be distributed between processors $\{P_1, P_2, P_3\}$.

In its turn, each internal node at level L_1 becomes a root of a sub-tree, which is a solution tree for distribution of the remaining workload between three processors $\{P_1, P_2, P_3\}$. Each edge connecting the root and its child is labelled by the workload size assigned to P_0 and its execution time. For example, the blue edge in Figure 6 is labelled by $(8, 1)$, which indicates that a workload of size 8 is given to P_0 and it takes 1 time unit to execute this workload by P_0. The child node connected by this edge is labelled by 8, which is the remaining workload $(= 16 - 8)$ to be distributed between processors $\{P_1, P_2, P_3\}$.

In Figure 6, the leaf node at level L_1 labelled by 0 represents a solution leaf. In general, any leaf node labelled by 0 represents one of the possible solutions, and the execution time of the corresponding solution is calculated as the maximum of the execution times labelling the edges in the path connecting the root and the solution leaf. For example, the execution time of the solution represented by the leaf labelled by red 0, which is connected to the root by two edges $\{(8, 1), (8, 1)\}$, will be equal to $max\{1, 1\} = 1$. The execution time of the solution represented by the solution leaf at level L_1 will be equal to 10 as it is connected to the root by just one edge $(16, 10)$.

The leaf node at level L_2 labelled by $\varnothing$ is a *no-solution* leaf. The path connecting this node to the root consists of two edges $\{(8, 1), (9, 1)\}$. The corresponding workload distribution results in no-solution because the sum of the workloads assigned to P_0 and P_1 will be equal to 17 $(= 8 + 9)$, which would exceed the total workload of 16.

In this example, each internal node in the solution tree has either 17 children (or $m + 1$ in general case) or just one child. The child is always a leaf. There are two types of leaves: *solution* leaves, labelled by 0, and *no-solution* leaves, labelled by $\varnothing$. Each internal node at level L_i, labelled by positive number w, becomes a root of a solution tree for distribution of the workload of size w between processors $\{P_i, \cdots, P_{p-1}\}$ and is therefore constructed recursively.

Finally, a distribution minimizing the parallel execution time will be returned as the optimal solution. In this example, the workload distribution $(8, 8, 0, 0)$, represented by the red *solution* leaf and resulting in the execution time of 1, will be returned as optimal.

It is apparent that the complexity of the presented straightforward algorithm is exponential.

We propose an efficient recursive sequential algorithm, *HPOPTA*, of polynomial complexity. *HPOPTA* employs a number of optimizations to avoid examining all the possible solutions and therefore does not explore all the paths in the tree.

The first step is to sort the discrete time functions, stored as arrays, in non-decreasing order of execution time as shown in Figure 5. We then determine the load-equal distribution and its parallel execution time, which is stored in variable τ called the *time threshold*. The load-equal distribution is the distribution where each processor is allocated the same workload of $\frac{n}{p}$ (assuming n is divisible by p). *HPOPTA* will not examine solutions with execution times greater than or equal to the time threshold. In the example, τ will be initialized by 12 $(\max_{i=0}^{3} t_i(\frac{16}{4}) = \max\{12, 6, 4, 4\} = 12)$. Therefore, only data points with execution times less than 12 will be considered and form the reduced search space. These data points are shown in gray cells in Figure 7. During the execution of *HPOPTA*, the time threshold τ will be updated every time a faster solution is found representing thus the execution time of the currently fastest solution.

HPOPTA then starts examining the solutions in the tree in the left-to-right depth-first order as shown in Figure 8. First, processors P_0 and P_1 are allocated zero workload each, making the workload to be distributed between processors P_2 and P_3 equal to 16. However, this workload exceeds the maximum workload, 15, that can be distributed between these two processors and executed in parallel in less than $\tau = 12$ time units. This maximum workload is associated with level L_2 of the solution tree and called the *size threshold* of this level, σ_2. In general, size threshold σ_i depends on the time threshold, τ, and is defined as the maximum workload that can be executed in parallel by processors $P_i, \cdots, P_{p-1}$ faster than in τ time units. The vector of size thresholds $\sigma = (\sigma_0, \sigma_1, \sigma_2, \sigma_3)$ can be determined using the time arrays and the current time threshold as follows. The maximum workloads, the execution time of which are less than $\tau = 12$ in the time arrays for processors P_0, P_1, P_2, and P_3, will be 16, 9, 7 and 8 respectively. Therefore, the size threshold of the last level (L_3) will be $\sigma_3 = 8$. The size threshold of level L_2 will be 15 $(= 7 + \sigma_3 = 7 + 8)$. Similarly, the size thresholds σ_1 and σ_0 for levels L_1 and L_0 will be 24 $(= 9 + \sigma_2 = 9 + 15)$ and 40 $(= 16 + \sigma_1 = 16 + 24)$ respectively. Thus, in Figure 8, the node labelled by 16 in L_2 cannot lead to solutions, which would be faster than the currently best (load-equal) solution with parallel execution time $\tau = 12$, and therefore this node will not be expanded. So, the red subtree in Figure 8 is cut and not explored. We call this key optimization operation *Cut*.

In general, as the algorithm progresses the vector of size thresholds, σ, changes every time the time threshold, τ, decreases. To illustrate how σ changes, we show its value

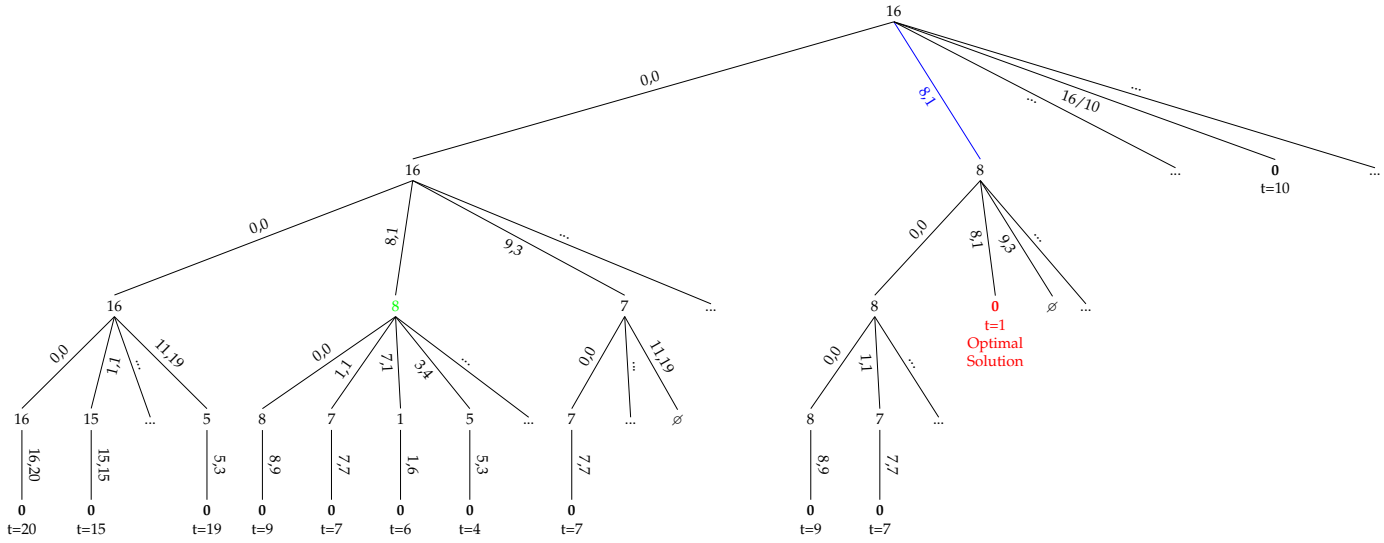

Fig. 6. Applying naive approach to examine all combinations and select a workload distribution with the minimum computation time of parallel execution of the workload

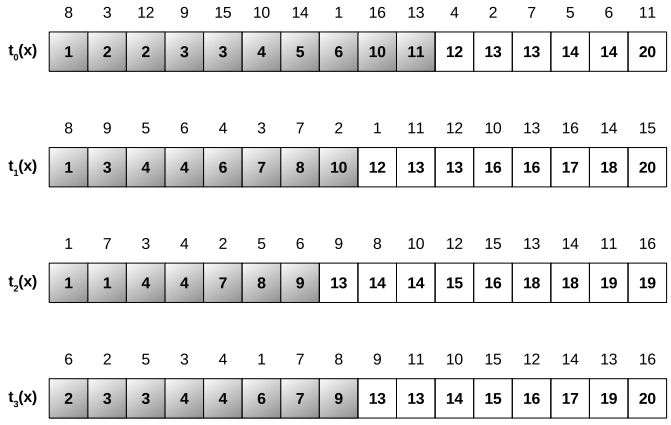

Fig. 7. Example: Applying load-equal time threshold and removing some data points from the search space.

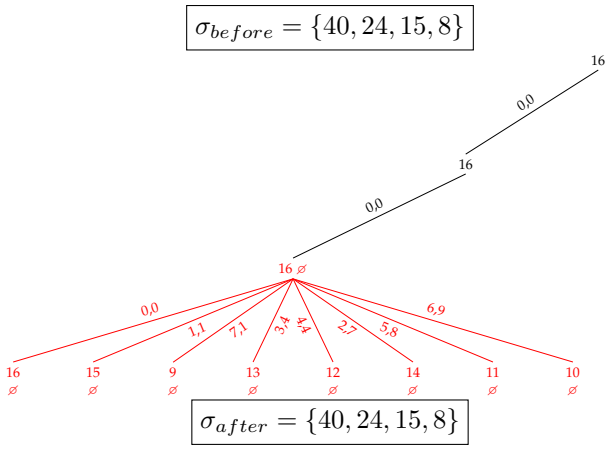

Fig. 8. Example: Applying size threshold which results in cutting some subtrees, which do not give any solution, from the search tree.

before and after each discussed step of the algorithm. As the *Cut* operation does not change τ, it also will not change σ, as illustrated in Figure 8,

Following the left-to-right depth-first order, next node to examine will be node 8 at level L_2 as shown in Figure 9. Proceeding from this node, the algorithm will generate and process solutions (leaves in the tree labelled by 0) in the left-to-right order. For each generated solution, the following operations will be performed:

- The time threshold τ is updated.
- If τ decreases, the data points in the time functions, whose time is equal to or greater than the updated time threshold, are removed from the search space, and the vector σ of size thresholds is updated.
- The solution is saved.
- Backtracking to an ancestor node of the solution is performed. We will explain in detail later how this ancestor node is chosen.

As an example, consider the solution with distribution $\{(0,0),(8,1),(3,4),(5,3)\}$ and execution time 4 (see Figure 9). The time threshold, τ, is updated to 4. Based on the new time threshold, the number of data points to be examined in the time functions is reduced. This is illustrated in the Figure 10, where one can see that fewer data points need to be examined compared to Figure 7. The vector of size thresholds, σ, is updated to $\{37,22,13,6\}$. The solution is saved, which includes memorization of the information pertaining to all the levels except for the first and the last. Thus, the information that is memorized is level-specific. For L_1, the saved information includes the problem size assigned to P_1, which is 8, the index of current element in the corresponding time function, which we call the last examined index and which is equal to 0, and the parallel execution time of the solution for processors $\{P_1,P_2,P_3\}$, which is 4. The same is done for L_2. The saved information includes the problem size assigned to P_2, which is 3, the index of current element in the corresponding time function, which is equal to 2, and the parallel execution time of the solution for processors $\{P_2,P_3\}$, which is equal to 4. We call

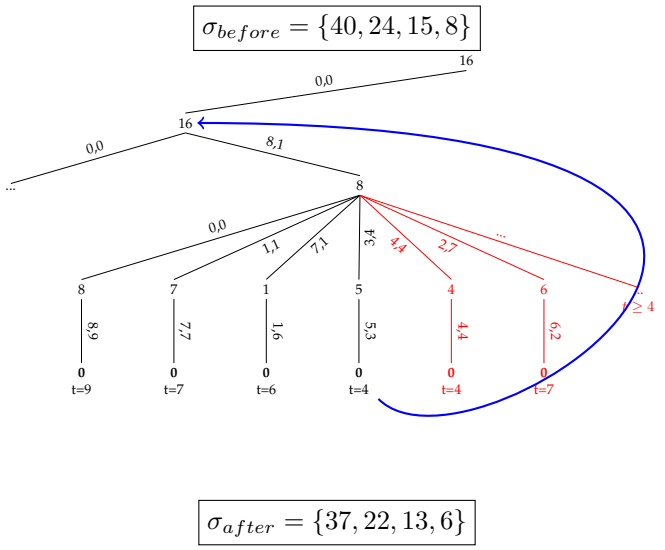

Fig. 9. Example: Backtracking to the ancestor of the node with maximum execution time, and cutting branches which do not result in any solution better than the solution have found so far.

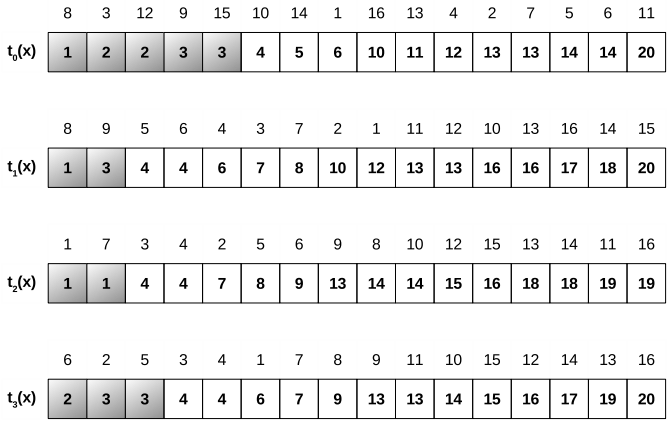

Fig. 10. Example: Applying the updated time threshold and removing more data points from the search space.

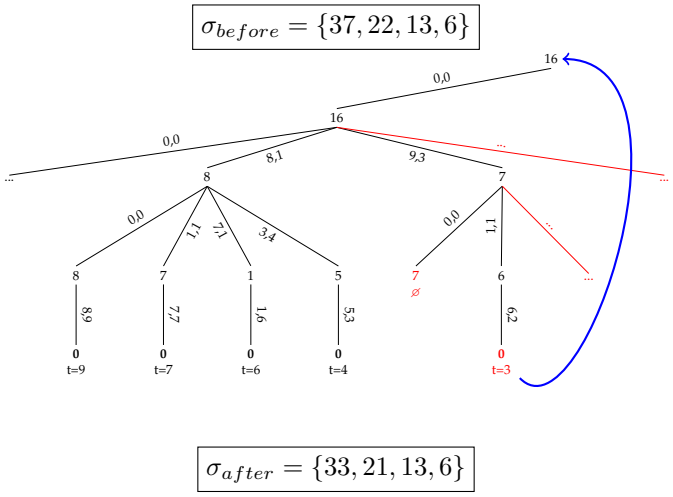

Fig. 11. Example: Keeping on applying *HPOPTA* on the search space.

this key operation, *Save*.

Having said that, backtracking to an ancestor node is one of the operations performed after each generated solution. Now, we describe how the ancestor node is chosen for the backtracking. From the leaf pertaining to the current solution, we traverse up the tree to the node with the maximum execution time. The parent of this node will be the backtracking target. For example, again consider the solution with distribution $\{(0,0),(8,1),(3,4),(5,3)\}$ and execution time 4 in Figure 9. For this solution, the node with the maximum execution time will be at level L_2 (the node labelled by 8). Therefore, the algorithm will backtrack to its parent, node 16 at level L_1, as indicated by a blue arc in Figure 9. Performing this backtracking effectively means that the algorithm will not generate and process the remaining solution leaves descending from the node 8 at level L_2 which are highlighted in red in Figure 9.

The reason for this is that the children of node 8 are examined in a non-decreasing order of times taken by processor P_2 to execute its workload in the corresponding solutions. Therefore, no edge coming out of node 8 after the edge (3,4) can have a label with the execution time less than 4. This makes further expansion of node 8 meaningless as no solution resulting from this expansion will have execution time less than 4, which is necessary to improve the currently best solution. Therefore, we backtrack to its ancestor, node 16 at level L_1. We will call this key operation *Backtrack*. After backtracking to node 16, the solution saved for workload size 8 at level L_2 becomes *final* because the corresponding distribution of workload of size 8 between processors P_2 and P_3 is the optimal one.

After backtracking to node 16 at level L_1, next node to examine will be node 7 at level L_2. The expansion of this node results in two children as shown in Figure 11. Giving zero workload to P_2 results in the workload of size 7 at level L_3, which exceeds the size threshold $\sigma_3 = 6$ and therefore results in no-solution. The second child yields a solution, which has the parallel execution time of 3. The algorithm updates the time threshold, τ, making it 3. As the time threshold decreased, the vector of size thresholds is updated to $\{33, 21, 13, 6\}$. The solution then is saved. For L_1, the memorized information includes the problem size assigned to P_1, which is 9, the last examined index which is equal to 1, and the parallel execution time of the solution for processors $\{P_1, P_2, P_3\}$, which is 3. For L_2, the saved information includes the problem size assigned to P_2, which is 1, the index of current element in the corresponding time function, which is equal to 0, and the parallel execution time of the solution for processors $\{P_2, P_3\}$, which is equal to 2. After this, *HPOPTA* backtracks to the root.

HPOPTA proceeds in this manner from the root until it comes to node 8 at level L_1 as illustrated in Figure 12. Here, as the optimal distribution of the workload of size 8 between processors P_2 and P_3 has been already found and saved, the best solution coming out of node 8 at level L_2 will be just retrieved from the memory. We call this key operation, *ReadMemory*. Since the parallel execution time of the retrieved solution is equal to 4, which is greater than the current time threshold $\tau = 2$, this solution is ignored. The algorithm then moves to the next child, which results in the solution $\{(8,1),(8,1),(0,0),(0,0)\}$ with the parallel

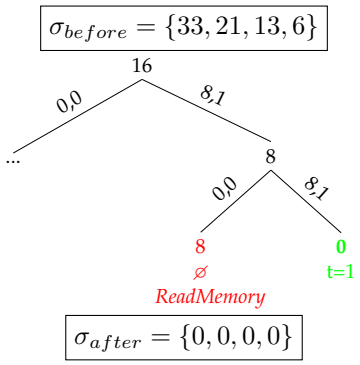

Fig. 12. Example: Finding the optimal solution and using Mem to find solutions.

execution time of 1. For this solution, the time threshold, τ, is updated to 1. The corresponding reduction of the search space results in the situation when no more data points in the time functions are left for further examination. Therefore, the algorithm terminates.

The optimal execution time is given by the last value of the time threshold. The optimal workload distribution is given by the workload distribution associated with this time threshold. So, there are four key operations in the algorithm, which are a) *Cut*, b) *Save*, c) *Backtrack*, and d) *ReadMemory*.

In the next section, we give a pseudocode of our algorithm, which uses these key operations as the fundamental building blocks.

5 Formal Description of *HPOPTA*

In this section, we describe the pseudocode of *HPOPTA*, which is shown in Algorithm 1. The inputs to *HPOPTA* are: the workload size, n, the number of heterogeneous processors, p, and a array of p time functions, $T = \{T_0, T_1, \cdots, T_{p-1}\}$. T_i represents the time function of processor P_i and consists of m pairs (x_{ij}, t_{ij}), $j \in [0, m)$, where x_{ij} is the j-th workload size in the time function and t_{ij} is its execution time by processor P_i. The outputs are the optimal workload distribution, D_{opt}, and the optimal parallel execution time, t_{opt}. It should be noted that the number of processors selected by the algorithm in the optimal workload distribution may be less than p.

The algorithm first sorts each time function in non-decreasing order of time (Line 2). It then determines the load-equal distribution. The array, D_{opt}, and the time threshold, τ, are initialized to the load-equal distribution and its corresponding execution time respectively (Lines 3-5). Then the vector of size thresholds, σ, is determined using the function *SizeThresholdCalc* (Line 6).

In line 7, the memorization data structure, matrix Mem, consisting of $(p-2) \times (n+1)$ elements, is initialized. It will save the found solutions for processors $P_1, \cdots, P_{p-2}$. Then, *HPOPTA* invokes the recursive routine, *HPOPTA_Kernel*, to find the optimal workload distribution.

Function GetTime(T, x) (called in Line 5) returns the execution time of workload size x in time function T. It return 0 when x equals to 0. It should be mentioned that pseudocodes of all functions used in Algorithms 1 and 2 and the structure of Mem are explained in the supplemental which is available online.

Algorithm 1 Algorithm Finding Optimal Workload Distribution of Size n for Maximizing Performance

1: **function** HPOPTA$(n, p, T, D_{opt}, t_{opt})$
INPUT:
Workload size, $n \in \mathbb{Z}_{>0}$
Number of processors, $p \in \mathbb{Z}_{>0}$
Time functions, $T = \{T_0, ..., T_{p-1}\}$,
$T_i = \{(x_{ij}, t_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, t_{ij} \in \mathbb{R}_{>0}\}$.
OUTPUT:
Optimal workload distribution, $D_{opt} = \{d_{opt}[0], ..., d_{opt}[p-1]\}$,
$d_{opt}[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\}, i \in [0, p)$.
Parallel execution time, $t_{opt} \in \mathbb{R}_{>0}$

2: $T \leftarrow Sort_{\uparrow}(T)$
3: $d_{opt}[i] \leftarrow \frac{n}{p}, \forall i \in [0, p-1]$
4: $d_{opt}[i] \leftarrow d_{opt}[i] + 1, \forall i \in [0, n\%p)$
5: $\tau \leftarrow \max_{i=0}^{p-1}$ GetTime$(T_i, d_{opt}[i])$
6: $\sigma \leftarrow$ SizeThresholdCalc(p, T, τ)
7: $Mem[i][j] \leftarrow \langle 0, 0, 0 \rangle, \forall i \in [1, \cdots, p-2], j \in [0, \cdots, n]$
8: HPOPTA_Kernel$(n, p, 0, T, \tau, \sigma, NULL, D_{cur}, Mem, D_{opt})$
9: $t_{opt} \leftarrow \tau$
10: **return** (D_{opt}, t_{opt})
11: **end function**

5.1 Recursive Algorithm $HPOPTA\_Kernel$

The recursive function, *HPOPTA_Kernel* (Algorithm 2), invokes the core operations, *Cut*, *Save*, *ReadMemory* and *Backtrack*. The level of the tree that is processed in this function is given by c. So, the first invocation of *HPOPTA_Kernel* deals with L_0, the next recursive invocation deals with L_1 and so on. It is important to note that D_{opt} holds the best distribution found so far. The array D_{cur} is used to store workloads currently assigned to processors $P_i (i \in [0, p-1])$.

Function *Cut* (given in the supplemental) compares workload size (n) with the corresponding size threshold (σ_c) to decide whether to expand the node or cut the subtree at level c (Lines 2-4).

Lines 5-11 process the solutions found in the last level L_{p-1}. When a solution, D_{cur}, is found, the routine *Process-Solution()* is invoked to perform the following operations :

- If D_{cur} is faster than the current best solution, D_{opt}, the time threshold τ will be reduced to the time of D_{cur} and D_{opt} will be updated by D_{cur}.
- When τ decreases, the vector of size thresholds, σ, is correspondingly updated.
- D_{cur} is memorized by invoking the operation *Save*.
- Using D_{cur}, the index of the level, bk, with the maximal execution time is found. If there are more than one level with this time, the level, which is closer to the root, is chosen.

Line 12 sets idx to -1. Variable idx, ranging from -1 to $m-1$, is used to store indexes of data points in the sorted time functions. If idx equals to -1, the workload size $x_{i\ idx}$ is set to the zero workload size (Lines 28-30), else $x_{i\ idx}$ is the idx-th workload size in the time function T_i.

Before expanding a node at a given level c to generate distributions of the workload of size n associated with this node, the function *ReadMemory* is called to check if any solution distributing workload n between processors $\{P_c, \cdots, P_{p-1}\}$, is currently saved in Mem and retrieve it if this is the case (Lines 13-27). The function also updates idx (Line 14) to determine the point from where the examination of data points should be resumed.

A memory cell in Mem saves either optimal or intermediate solution. The memory cell containing the optimal

distribution is labelled *Finalized*. The intermediate solution is a solution which may not be optimal. The variable *status* determines the type of the retrieved solution. If no solution has been saved for the node or the parallel execution time of the retrieved solution is equal to or greater than τ (given by the status, *NOT_SOLUTION*), we return from *HPOPTA_Kernel*. If the saved solution in the Mem is the optimal one (given by the status, *SOLUTION*), the retrieved solution is used and we return from *HPOPTA_Kernel*. However, if the retrieved solution is not *Finalized* (given by the status, *SOLUTION_RESUME*), the function *ProcessSolution* is invoked to process this solution (Line 21). Then the function *Backtrack* is invoked (Line 23) to determine whether the routine backtracks or resumes the process from the data point ($x_{c\ idx}$, GETTIME($T_i, x_{c\ idx}$)) where idx has been set by the function *ReadMemory*. If none of the above cases takes place, the routine resumes from data point idx (Line 31).

The $while$ loop (Lines 31-47) scans the time function T_c from left to right examining the data points with execution times less than the time threshold, τ. In each iteration, the data point idx is extracted from the time function T_c. Its workload size $x_{c\ idx}$ is stored in array D_{cur} (Line 32). If this workload size ($x_{c\ idx}$) is equal to n, we found a solution. In this case, the solution is processed using *ProcessSolution()* (Lines 33-35). Otherwise, if $x_{c\ idx}$ is less than n, *HPOPTA_Kernel* is re-invoked to solve *HPOPT* for the remaining workload $n - x_{c\ idx}$ at the next level L_{c+1}. If $x_{c\ idx}$ greater than n, *HetW_Kernel* drops this data point and moves to the next one.

After data point $x_{c\ idx}$ is examined, the function *Backtrack* is called to decide whether the algorithm backtracks or continues the examination at level L_c (Line 40).

Lines 43-46 check if the algorithm reaches the end of the time function T_c. If this is the case, the *while* loop (Line 31-47) terminates and the corresponding memory cell is finalized (Line 48). Otherwise, idx is incremented moving to the next data point in the time function T_c.

Algorithm 2 Algorithm of Recursive Kernel Invoked by Algorithm 1

```
1: function HPOPTA_KERNEL(n, p, c, T, τ, σ, bk, D_cur, Mem, D_opt)
2:     if CUT(n, σ_c) then
3:         return
4:     end if
5:     if c = p − 1 then
6:         if GETTIME(T_c, n) < τ then
7:             d_cur[c] ← n
8:             PROCESSSOLUTION(p, T, τ, σ, bk, D_cur, Mem, −1, D_opt)
9:         end if
10:        return
11:    end if
12:    idx ← −1
13:    if c > 0 ∧ c ≤ p − 2 then
14:        status ← READMEMORY(n, p, c, τ, T, D_cur, Mem, idx)
15:        if status = NOT_SOLUTION then
16:            return
17:        else if status = SOLUTION then
18:            PROCESSSOLUTION(p, T, τ, σ, bk, D_cur, Mem, c, D_opt)
19:            return
20:        else if status = SOLUTION_RESUME then
21:            PROCESSSOLUTION(p, T, τ, σ, bk, D_cur, Mem, c, D_opt)
22:            tt ← GETTIME(T_c, x_{c idx})
23:            if BACKTRACK(n, c, bk, idx, tt, τ, Mem, TRUE) then
24:                return
25:            end if
26:        end if
27:    end if
28:    if idx = −1 then
29:        x_{c idx} ← 0
30:    end if
31:    while GETTIME(T_c, x_{c idx}) < τ do
32:        d_cur[c] ← x_{c idx}
33:        if x_{c idx} = n then
34:            d_cur[i] ← 0, ∀i ∈ [c + 1, ··· , p − 1]
35:            PROCESSSOLUTION(p, T, τ, σ, bk, D_cur, Mem, −1, D_opt)
36:        else if n > x_{c idx} then
37:            HPOPTA_KERNEL(n − x_{c idx}, p, c + 1, T, τ, σ, bk, D_cur)
38:        end if
39:        tt ← GETTIME(T_c, x_{c idx})
40:        if BACKTRACK(n, c, bk, idx, tt, τ, Mem, FALSE) then
41:            return
42:        end if
43:        if idx + 1 = m then
44:            break
45:        end if
46:        idx ← idx + 1
47:    end while
48:    MAKEFINAL(Mem[c][n])
49: end function
```

5.2 Theoretical Analysis of *HPOPTA*

Proposition 5.1. *The algorithm* HPOPTA *always returns a distribution of the workload of size n between p heterogeneous processors that minimizes its parallel execution time.*

Proposition 5.2. *The time complexity of* HPOPTA *is $O(m^3 \times p^3)$. The total memory used by the algorithm is $O(p \times (m + n))$.*

The proofs of the propositions can be found in the supplemental file which is available online.

6 Experimental Analysis of *HPOPTA*

In this section, we experimentally examine our proposed algorithm, *HPOPTA*. We also present speedup compared to solutions returned by load-balancing algorithms based on functional performance model and constant performance model. Two sets of experiments are conducted. The first set is carried out on a real heterogeneous server, while the second is performed on simulated clusters of heterogeneous nodes. Finally, we analyse a hierarchical two-level workload distribution algorithm that uses *HPOPTA* and *POPTA* [9].

6.1 Experimental Platform and Applications

We perform our experiments on *HCLServer* containing an Intel Haswell multicore CPU, Nvidia K40c GPU, and Intel Xeon Phi 3120P, whose specifications are given in Tables 1, 2 and 3 respectively.

We experiment with two widely known scientific data-parallel applications, Matrix Multiplication and 2D discrete Fourier Transform, configured for execution on *HCLServer* as explained in Section 1. Each application consists of three computational kernels running in parallel on three abstract processors of the *HCLServer*.

We would like to mention that the incorporation of the cost of communications is out of the scope of this paper.

6.2 Data Partitioning on Hybrid Server

In this section, we examine our proposed algorithm on *HCLServer*. For each application, the input to *HPOPTA* are three time functions representing the performance profiles of the CPU, GPU, and Xeon Phi abstract processors respectively.

As explained in Section 1, the time functions of an application are built simultaneously on all abstract processors to take into account resource contention. It should be mentioned that there is no specific reason for choosing particular problem sizes in our time functions. *HPOPTA* can deal with any time function represented by a discrete set of data points. However, if the consecutive problem sizes are separated by a large step size, the shape of the speed functions becomes smoother thereby disallowing any opportunity for optimization.

We compare the speedup of *HPOPTA* over load-balancing algorithms based on functional performance model (*FPM*) [45] [5] [6] and constant performance model. Just for comparison purposes, we will call the load-balancing algorithms based on *FPM*, *smooth-FPM* algorithms. The percentage speedup of *HPOPTA* against *smooth-FPM* algorithm is calculated as follows: $Speedup_{FPM}(\%) = \frac{t_{smooth-FPM} - t_{HPOPTA}}{t_{HPOPTA}} \times 100$, where $t_{smooth-FPM}$ and t_{HPOPTA} respectively are the execution times of solutions found by executing *HPOPTA* using smoothed and actual time functions. $t_{smooth-FPM}$ is estimated as follows. First, the workload distribution for a given workload size is found by executing *HPOPTA* using smoothed time functions as input. Then, the execution time for this distribution is calculated using the original, not smoothed, time functions. Thus, the smoothed time functions are used for finding the FPM workload distribution, and its execution time is then found using the real time functions.

The percentage speedup of *HPOPTA* against load-balancing algorithm based on constant performance model is calculated as follows: $Speedup_{cpm}(\%) = \frac{t_{CPM} - t_{HPOPTA}}{t_{HPOPTA}} \times 100$, where t_{CPM} and t_{HPOPTA} respectively are the execution times of solutions found by executing the CPM-based load-balancing algorithm and *HPOPTA* using actual time functions. The constant performance model (CPM) uses relative speeds of processors, which are constant floating-point numbers. We use three CPMs for comparison and these are determined from three different data points in speed functions of the processors.

We now summarize the experimental results on *HCLServer* using two data parallel applications Matrix Multiplication and FFT.

6.2.1 Matrix Multiplication

Heterogeneous matrix multiplication application uses three kernels to perform computation on CPUs, GPUs and Xeon Phis which has been explained in detail in Section 1. It is a data parallel application enabling in-card and out-of-card matrix multiplication on CPU, GPU and Xeon Phi. In our experiments, the Intel MKL and CUDA versions used are 2017.0.2 and 7.5 respectively.

For a problem size n^2 in the speed function, the speed is calculated as $\frac{2 \times n^3}{t}$ where t is execution time taken to multiply two $n \times n$ square matrices. For GPU and Intel Xeon Phi, the execution time includes the transfer of matrices from the host to the device and the results from the device to the host. Figure 13 shows the speed functions for the three processors. Each speed function of DGEMM (and its equivalent time function) is represented by a discrete set of cardinality (m) equal to 700 data points with problem sizes $x = \{64^2, 128^2, \cdots, 44800^2\}$. Out-of-card DGEMM

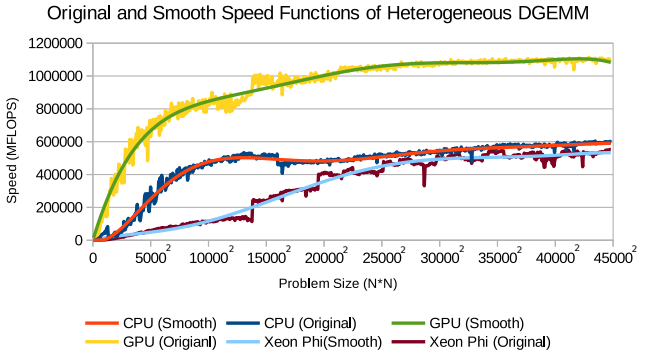

Fig. 13. Original and smoothed speed functions of Heterogeneous Matrix Multiplication on our server. MKL DGEMM is invoked for CPU and Xeon Phi. For GPU, CUBLAS is used. The original functions are smoothed using polynomial trend line in LibreOffice Calc.

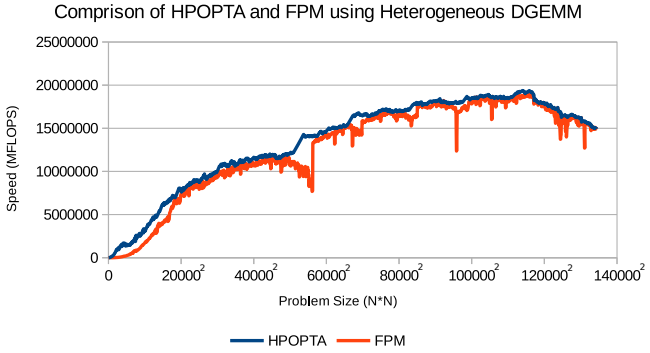

Fig. 14. Speed functions of heterogeneous matrix multiplication for whole *HCLServer*. The application is executed for each workload size n using two different workload distributions *HPOPTA* and *FPM*.

invocations are performed on GPU and Xeon Phi when workload size exceeds the size of main memory on the accelerators.

To obtain smooth speed function from the actual speed function, we smooth the actual speed function using polynomial trend line in LibreOffice Calc and construct its equivalent time function. Figures 13 shows the original and smoothed speed functions of DGEMM.

To determine the percentage improvements given by *HPOPTA*, we create an experimental data set for DGEMM whose data points ranges from $(\frac{p}{3} \times 64 \times 100)^2$ to $(p \times 64 \times 700)^2$ with step size of 64^2. Since there are three abstract processors in the *HCLServer*, p is equal to 3 in this experiment. Figure 14 shows the speed of heterogeneous DGEMM on *HCLServer* when executed using *HPOPTA* in comparison with *FPM* workload distribution. *HPOPTA* gives the minimum, average, and maximum percentage of improvement of 0, 14, 261 percent respectively in comparison with *FPM*. Since $t_{smooth-FPM}$ equals t_{HPOPTA} for some workloads, we have observed zero percentage of improvement for them. The maximum improvement belongs to the workload size 6784^2 where the problem sizes for CPU, GPU and Xeon Phi found using original functions (*HPOPTA*) are $\{1856, 4928, 0\}$ and using the smooth functions are $\{576, 4736, 1472\}$.

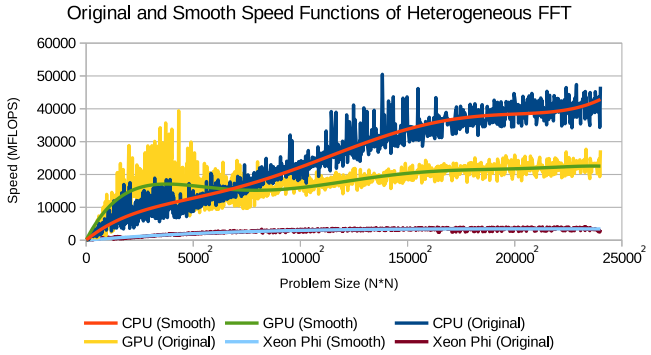

Fig. 15. Original and smoothed speed functions of Heterogeneous FFT on our server. MKL FFT is invoked for CPU and Xeon Phi. For GPU, CUFFT is used. The original functions are smoothed using polynomial trend line in LibreOffice Calc.

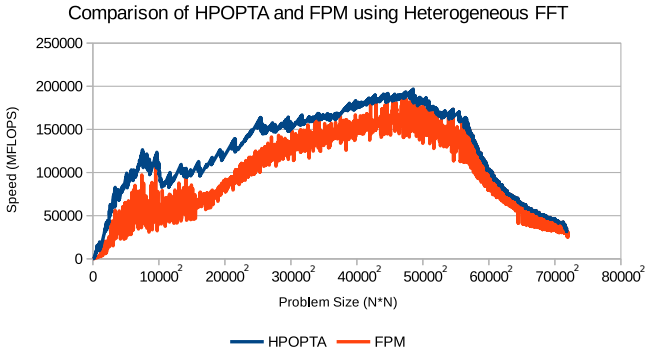

Fig. 16. Speed functions of heterogeneous FFT for whole *HCLServer*. The application is executed for each workload size n using two different workload distributions *HPOPTA* and *FPM*.

For *CPM*, we use the relative speeds of the processors based on execution of one problem size. We select three different problem sizes from the speed functions for this purpose. One at the beginning, one in the middle, and one in the end. These are 4736, 28672 and 44800 and therefore there are three constant relative performance models, $\{0.34, 0.59, 0.07\}$, $\{0.27, 0.55, 0.18\}$, $\{0.27, 0.49, 0.24\}$ where the first element in each set is relative speed of CPU, the second is relative speed of GPU, and the last one represents the relative speed of Xeon Phi. The average percentage improvements are respectively 122, 106, and 82 percent.

Since the number of data points in speed functions are limited, there are workload sizes whose CPM workload distributions contain problem sizes, which exceed the largest problem size in the speed functions. That is, for these workload sizes, CPM-based load balancing algorithm does not find any solution. Therefore, we ignored these workload sizes to calculate maximum and average of $Speedup_{cpm}$.

6.2.2 FFT

Heterogeneous FFT application uses three kernels to perform computation on CPUs, GPUs, and Xeon Phis which has been explained in detail in Section 1. It is a data parallel application enabling in-card fast Fourier transform on the three abstract processors in *HCLServer*. In our experiments, the Intel MKL and CUDA versions used are 2017.0.2 and 7.5 respectively.

For a problem size n^2 in the speed function, the speed is calculated as $\frac{n^2 \times \log_2 n^2}{t}$ where t is execution time taken to compute 2D DFT of size n^2. The FFT speed functions are shown in the Figure 15. The discrete set for the FFT speed functions has the cardinality 1090 and contains problem sizes, $\{16^2, 32^2, \cdots, 24000^2\}$. It does not include problem sizes, which cannot be factored into primes less than or equal to 127. For these problem sizes, CUFFT for GPU gives failures. Unlike DGEMM, all the FFT invocations are performed in-card.

Figures 15 shows the original and smoothed speed functions of FFT. We again apply polynomial trend line in LibreOffice Calc on actual speed function of FFT to obtain its smooth function.

To analyse FFT, the experimental data set includes data points ranging from $(\frac{p}{3} \times 16 \times 100)^2$ to $(p \times 16 \times 1500)^2$ with step size of 16^2 ($p = 3$ for *HCLServer*). Figure 16 compares the speed of heterogeneous FFT when executed using *HPOPTA* with the speed when the workload is distributed using *FPM*. *HPOPTA* gives the minimum, average, and maximum percentage of improvements of 0, 40, and 502 percent respectively in comparison with *FPM*. The maximum improvement happens for the workload size, 1920^2. The problem sizes given to CPU, GPU and Xeon Phi using original functions (*HPOPTA*) are $\{464, 1456, 0\}$ and using the smooth functions are $\{656, 1168, 96\}$.

Like DGEMM, to compare *CPM*-based load balancing algorithm with *HPOPTA*, we use the relative speeds based on three different problem sizes, 4320, 13824, and 24000, from the speed functions. These points result in three CPMs, $\{0.18, 0.78, 0.04\}$, $\{0.69, 0.26, 0.05\}$, $\{0.60, 0.35, 0.05\}$. The average percentage improvements are 301, 164, and 129 percent respectively.

Since the number of data points in speed functions are limited, there are workload sizes whose CPM workload distributions contain problem sizes, which exceed the the largest problem size in the speed functions. In addition to out-of-range problem sizes, there is no speed for problem sizes, which cannot be factored into primes less than or equal to 127. This is due to failure of CUFFT calls for these problem sizes. That is, for these workload sizes, CPM-based load balancing algorithm does not find any solution. Therefore, we ignored these workload sizes to calculate the maximum and average of $Speedup_{cpm}$.

6.2.3 Discussion

We observed a tight correlation between the average variations in speed functions and the average performance improvements. To study this correlation further, we create speed bands for DGEMM and FFT speed functions as mentioned in [47]. By looking at DGEMM speed functions in Figures 13 and 15, it can be observed that there are maximum differences of 29% and 150% approximately between lower and upper bands in DGEMM and FFT speed functions. These differences confirm the achieved improvements where the average $Speedup_{FPM}$ of FFT is about four times greater than that of DGEMM.

We also observed that sometimes the number of processors in the optimal solutions determined by *HPOPTA* is less than p. For instance, the optimal solution for FFT for matrix size 1200×1200 uses just one abstract processor, GPU, meanwhile for matrix size 19632×19632 the optimal distribution only uses CPU and GPU.

6.3 Using *HPOPTA* for Data partitioning on Clusters of Heterogeneous Nodes

In the supplemental (Section 5), we describe how *HPOPTA* can be used to optimally distribute workload between processors in a cluster of heterogeneous nodes. We present a hierarchical two-level workload distribution approach based on *HPOPTA* and *POPTA* [9], which not only reduces the computational complexity but also allows parallel computation for finding optimal workload distribution.

7 Conclusion

Modern high performance computing platforms have become highly heterogeneous due to tight integration of multicore CPU processors and accelerators. This tight integration causes contention on shared resources such as Last Level Cache (LLC), DRAM, PCI-E links, etc. Due to this serious contention and NUMA, the performance profiles of data-parallel applications executing on these heterogeneous platforms are not smooth and deviate greatly from the shapes that supposed by state-of-the-art load-balancing algorithms to find optimal solutions.

In this paper, we formulate the problem of finding optimal distribution on heterogeneous clusters of hybrid nodes and propose a novel model-based data partitioning algorithm to minimize the execution time for general performance profiles of data-parallel applications executing on clusters of heterogeneous nodes. The inputs to the algorithm are p discrete time functions, which represent the performance profiles of p processors existing in the heterogeneous cluster. The time complexity of the proposed algorithm is $O(m^3 \times p^3)$ where m and p respectively represent the maximum cardinality of input time function and the number of heterogeneous processors. We study the optimality of solutions found by the proposed algorithm using two well-known data-parallel applications, matrix multiplication and two-dimensional discrete fast Fourier transform. According to the experimental results, the proposed algorithm demonstrates considerable improvements in average and maximum performance for the two applications in comparison with state-of-art load-balancing algorithms.

The software implementation for *HPOPTA* is available at [48].

In our future work, we aim to design and implement parallel versions of the algorithms to reduce the theoretical complexity. We would also consider cost of communications.

Acknowledgements

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

References

[1] Top500, "Top500," 2017. [Online]. Available: https://www.top500.org/lists/2017/11/

[2] M. Cierniak, M. J. Zaki, and W. Li, "Compile-time scheduling algorithms for a heterogeneous network of workstations," *The Computer Journal*, vol. 40, no. 6, pp. 356–372, 1997.

[3] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1033–1051, 2001.

[4] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *J. Parallel Distrib. Comput.*, vol. 61, no. 4, Apr. 2001.

[5] A. L. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 104.

[6] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007.

[7] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2017.

[8] P. K. Smolarkiewicz and W. W. Grabowski, "The multidimensional positive definite advection transport algorithm: Nonoscillatory option," *J. Comput. Phys.*, vol. 86, no. 2, Feb. 1990.

[9] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017.

[10] OpenBLAS, "OpenBLAS: An optimized BLAS library," 2016. [Online]. Available: http://www.openblas.net/

[11] FFTW, "FFTW: A fast, free c FFT library," 2016. [Online]. Available: http://www.fftw.org/

[12] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Computing*, vol. 33, no. 12, Dec. 2007.

[13] A. Ilić, F. Pratas, P. Trancoso, and L. Sousa, "High-performance computing on heterogeneous systems: Database queries on CPU and GPU," *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, pp. 202–222, 2010.

[14] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, no. 02, pp. 195–217, 2011.

[15] D. Clarke, A. L. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 7155. Springer-Verlag, 2012.

[16] X. Liu, Z. Zhong, and K. Xu, "A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms," *Future Generation Computer Systems*, vol. 56, pp. 759–765, 2016.

[17] M. Radmanović, D. Gajić, and R. Stanković, "Efficient computation of galois field expressions on hybrid CPU-GPU platforms." *Journal of Multiple-Valued Logic & Soft Computing*, vol. 26, 2016.

[18] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 2012, pp. 683–690.

[19] J. Colaço, A. Matoga, A. Ilic, N. Roma, P. Tomás, and R. Chaves, "Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems," in *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 693–703.

[20] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms." in *PARCO*, 2013, pp. 203–212.

[21] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *Computers, IEEE Transactions on*, vol. 64, no. 9, pp. 2506–2518, 2015.

[22] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "Out-of-core implementation for accelerator kernels on heterogeneous clouds," *The Journal of Supercomputing*, vol. 74, no. 2, pp. 551–568, 2018.
[23] H. Khaleghzadeh, , R. Reddy, Z. Zhong, and A. Lastovetsky, "ZZGEMMOOC: Out-of-core package for out-of-core dgemm on GPU," 2017. [Online]. Available: https://git.ucd.ie/hcl/zzgemmooc.git
[24] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "XeonPhiOOC: Out-of-core package for out-of-core dgemm on Xeon Phi," 2017. [Online]. Available: https://git.ucd.ie/manumachu/xeonphiooc.git
[25] A. T. Chronopoulos, D. Grosu, A. M. Wissink, M. Benche, and J. Liu, "An efficient 3d grid based scheduling for heterogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 827 – 837, 2003, special Section on the Best Papers from the 2002 International Parallel and Distributed Processing Symposium.
[26] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–10.
[27] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 19–28.
[28] H. Khaleghzadeh, H. Deldari, R. Reddy, and A. Lastovetsky, "Hierarchical multicore thread mapping via estimation of remote communication," *The Journal of Supercomputing*, pp. 1–20, 2017.
[29] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, no. 2. ACM, 2008, pp. 287–296.
[30] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, no. 4, pp. 121–130, Feb. 2009.
[31] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Aug. 2009.
[32] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 365–376.
[33] K. Kyriakopoulos, A. T. Chronopoulos, and L. Ni, "An optimal scheduling scheme for tiling in distributed systems," in *Cluster Computing, 2007 IEEE International Conference on*. IEEE, 2007, pp. 267–274.
[34] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 59–59.
[35] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–11.
[36] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.
[37] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 4, pp. 289–299, 2005.
[38] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, Jun. 2004.
[39] R. L. Cariño and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *The Journal of Supercomputing*, vol. 44, no. 1, pp. 41–63, 2008.
[40] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE," *J. Supercomput.*, vol. 58, no. 2, Nov. 2011.
[41] A. Lastovetsky and R. Reddy, "A novel algorithm of optimal matrix partitioning for parallel dense factorization on heterogeneous processors," in *International Conference on Parallel Computing Technologies*. Springer, 2007, pp. 261–275.
[42] 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*, vol. 50, no. 10, pp. 1052–1070, 2001.
[43] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 46–51.
[44] R. Wyrzykowski, L. Szustak, K. Rojek, and A. Tomas, "Towards efficient decomposition and parallelization of mpdata on hybrid cpu-gpu cluster," in *International Conference on Large-Scale Scientific Computing*. Springer, 2013, pp. 457–464.
[45] A. Lastovetsky and R. Reddy, "Data partitioning for multiprocessors with memory heterogeneity and memory constraints," *Scientific Programming*, vol. 13, no. 2, pp. 93–112, 2005.
[46] W. Zhang, X. Ji, B. Song, S. Yu, H. Chen, T. Li, P. C. Yew, and W. Zhao, "Varcatcher: A framework for tackling performance variability of parallel workloads on multi-core," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1215–1228, April 2017.
[47] 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*. ACM, 2006, pp. 746–753.
[48] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "HPOPTA: Heterogeneous model-based data partitioning algorithm for optimization of data-parallel applications for performance," 2017. [Online]. Available: https://git.ucd.ie/hkhaleghzadeh/hpopt.git

Hamidreza Khaleghzadeh is a PhD researcher at Heterogeneous Computing Laboratory at the School of Computer Science, University College Dublin. He got his BSc and MSc degrees in Computer Engineering (software) in 2007 and 2011, respectively. He ranked as a 2nd position holder in his MS program. His main research interests include performance and energy consumption optimization in massively heterogeneous systems, high performance heterogeneous systems, energy efficiency, and parallel/distributed computing.

Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science, University College Dublin in 2005. His main research interests include high performance heterogeneous computing, high performance linear algebra, parallel computational fluid dynamics and finite element analysis.

Alexey Lastovetsky received a PhD degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models, and programming tools for high performance heterogeneous computing. He has published over a hundred technical papers in refereed journals, edited books, and international conferences. He authored the monographs Parallel computing on heterogeneous networks (Wiley, 2003) and High performance heterogeneous computing (Wiley, 2009).