Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing

Alexey Lastovetsky, *Member, IEEE*, Lukasz Szustak, *Member, IEEE* and Roman Wyrzykowski, *Member, IEEE*

Abstract—Load balancing is a widely accepted technique for performance optimization of scientific applications on parallel architectures. Indeed, balanced applications do not waste processor cycles on waiting at points of synchronization and data exchange, maximizing this way the utilization of processors. In this paper, we challenge the universality of the load-balancing approach to optimization of the performance of parallel applications. First, we formulate conditions that should be satisfied by the performance profile of an application in order for the application to achieve its best performance via load balancing. Then we use a real-life scientific application, EULAG MPDATA kernel, to demonstrate that its performance profile on a modern parallel architecture, Intel Xeon Phi, significantly deviates from these conditions. Based on this observation, we propose a method of performance optimization of scientific applications through load imbalancing. In the case of data parallel application, the method uses functional performance models of the application to find partitioning that minimizes its computation time but not necessarily balances the load of processors. We apply this method to optimization of MPDATA on Intel Xeon Phi. Experimental results demonstrate that the performance of this carefully optimized load-balanced application can be further improved by 15% using the proposed load-imbalancing technique.

Index Terms—functional performance model, data partitioning, Intel Xeon Phi, EULAG, MPDATA, load imbalancing.

1 Introduction

Load balancing is a widely accepted technique for optimization of the computational performance of scientific applications on parallel architectures. Indeed, the intuition suggests that unlike unbalanced applications, the balanced ones do not waste processor cycles on waiting at points of synchronization and data exchange, maximizing this way the utilization of the processors.

In this paper, we challenge the universality of the load-balancing approach to optimization of the computational performance of parallel applications. First, we try to understand the limitations of the load-balancing approach. We formulate conditions that should be satisfied by the performance profile of an application in order for the application to achieve its best performance via load balancing.

Then we use a real-life scientific application to demonstrate that its performance profile on a modern parallel architecture does not satisfy these conditions. The application we use implements the Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), which is one of the major parts of the dynamic core of the EULAG geophysical model [1]. EULAG (Eulerian/semi-Lagrangian fluid solver) is an established numerical model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios [2], [3]. In particular, it can be used in numerical weather prediction (NWP), simulation of urban flows, areas of turbulence, ocean currents, etc. This solver, originally developed for conventional HPC systems, is currently being re-written for modern HPC platforms. In particular, MPDATA has been recently re-written and optimized for execution on an Intel Xeon Phi coprocessor [4], [5], [6].

In our experiments, we observe significant deviations of the MPDATA performance profile from the conditions required for applicability of the load-balancing techniques. Based on this observation, we propose a general method of performance optimization of scientific applications through load imbalancing as well as an algorithm that tries to find the optimal, possibly imbalanced, configuration of a data parallel application on a set of homogeneous processors. This algorithm uses functional performance models of the application [7], [8] to find the partitioning that minimizes its computation time but not necessarily balances the load of the processors. Finally, we apply this algorithm to optimization of MPDATA on Intel Xeon Phi. Experimental results demonstrate that the performance of this carefully optimized load-balanced application can be further improved by 15% using the proposed load-imbalancing method.

The contributions of the work presented in the paper are as follows:

- Formulation of the conditions that should be satisfied to guarantee that load balancing will minimize the computation time of parallel application.
- Building the performance profile of a real-life scientific application on a modern HPC platform and demonstration of its significant deviation from the conditions that guarantee that load balancing be a safe technique for performance optimization.
- A new optimization method that uses the performance profile for optimization of the application

- *A. Lastovetsky is with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland.*
E-mail: alexey.lastovetsky@ucd.ie,
- *L. Szustak and R. Wyrzykowski are with the Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland. E-mail: lszustak@icis.pcz.pl, roman@icis.pcz.pl*

through its imbalancing.
- Application of the proposed method to optimization of MPDATA on Intel Xeon Phi, resulting in further acceleration of this carefully optimized load-balanced application by up to 15%.

The rest of the paper is structured as follows. Section 2 overviews load-balancing techniques and formulates the conditions when these techniques would minimize the computation time of parallel applications. Section 3 analyzes the performance profile of MPDATA on Intel Xeon Phi and introduces the new approach to minimization of the computation time through load imbalancing. Section 4 introduces a partitioning algorithm for (uneven) distribution of computations between homogeneous processors, minimizing the computation time of the application. Section 5 applies this algorithm to optimization of MPDATA on Xeon Phi, and Section 6 concludes the paper.

2 Load balancing and performance

In this section, we review load-balancing techniques used for optimization of the performance of parallel scientific applications on both homogeneous and heterogeneous platforms. We also formulate the conditions when application of these techniques will optimize the computational performance.

Load balancing is a widely accepted and commonly used approach to performance optimization of scientific applications on parallel architectures. Load balancing algorithms can be classified as static or dynamic. Static algorithms (for example, those based on data partitioning) [8], [9], [10], [11], [12], [13] require a priori information about the parallel application and platform. This information can be gathered either at compile-time or runtime. Static 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. Static algorithms are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms are unable to balance on non-dedicated platforms, where load changes with time. Dynamic algorithms (such as task scheduling and work stealing) [14], [15], [16] balance the load by moving fine-grained tasks between processors during the calculation. Dynamic algorithms do not require a priori information about execution but may incur significant communication overhead due to data migration. Dynamic algorithms often use static partitioning for their initial step due to its provably near-optimal communication cost, bound tiny load imbalance, and lesser scheduling overhead [17].

Whatever load balancing algorithm is used, the goal is always to minimize the computation time of the application. The intuition behind the assumption that balancing the application improves its performance is the following: a balanced application does not waste processor cycles on waiting at points of synchronization and data exchange, maximizing this way the utilization of the processors and minimizing the computation time. Is this assumption always true? To answer this question, let us formulate the assumption in a mathematical form. Consider an application, the computational performance of which can be modeled by speed functions. Namely, let p parallel processors be used to execute the application, and let $s_i(x)$ be the speed of execution of the workload of size x by processor i. Here the speed can be measured in floating point operations per second or any other fix-sized computation units per unit time. The size of workload can be characterized by the problem size (for example, the number of cells in the computational domain or the matrix size) or just by the number of equal-sized computational units. The speed $s_i(x)$ is calculated as $\frac{x}{t_i(x)}$, where $t_i(x)$ is the execution time of the workload of size x on processor i. Using these definitions, we can formulate the following theorem.

Theorem 1: Let $s_i(x) > 0$ $(x > 0)$ be the speed of processor $i \in \{1, \dots, p\}$, and $\forall \Delta x > 0\colon\ \frac{s_i(x)}{x} \geq \frac{s_i(x+\Delta x)}{x+\Delta x}$. Let $x_1 + \dots + x_p = n > 0$ and $\frac{s_1(x_1)}{x_1} = \dots = \frac{s_p(x_p)}{x_p}$. Then, $\forall y_1, \dots, y_p > 0$ such that $(y_1, \dots, y_p) \neq (x_1, \dots, x_p)$ and $y_1 + \dots + y_p = n$: $\max_i \frac{y_i}{s_i(y_i)} \geq \frac{x_1}{s_1(x_1)}$.

Proof: As $(y_1, \dots, y_p) \neq (x_1, \dots, x_p)$ and $y_1 + \dots + y_p = x_1 + \dots + x_p$, then there exists $k \in \{1, \dots, p\}$ such that $y_k > x_k$. Therefore, $\max_i \frac{y_i}{s_i(y_i)} \geq \frac{y_k}{s_k(y_k)} = \frac{x_k + (y_k - x_k)}{s_k(x_k + (y_k - x_k))} \geq \frac{x_k}{s_k(x_k)} = \frac{x_1}{s_1(x_1)}$.

Theorem 1 states that in order to guarantee that the balanced configuration of the application will execute the workload of size n faster than any unbalanced configuration, the speed functions $s_i(x)$, characterizing the performance profiles of the processors, should satisfy the condition:

$$\forall \Delta x > 0\colon\ \frac{s_i(x)}{x} \geq \frac{s_i(x+\Delta x)}{x+\Delta x} \tag{1}$$

Geometrically, it can be illustrated as follows. If we plot a speed function as shown in Figure 1, then the angle $\alpha(x)$ between the straight line, connecting the point $(0,0)$ and the point $(x, s(x))$ on the speed curve, and the x-axis will be inversely proportional to the execution time of the workload of size x by the processor. Indeed, the cotangent of this angle is directly proportional to the ratio $\frac{x}{s(x)}$ representing the execution time of the workload x. Therefore, larger angles correspond to shorter execution times. The condition 1 means that the increase of the workload, x, will never result in the decrease of the execution time, or equivalently in the increase of the angle $\alpha(x)$.

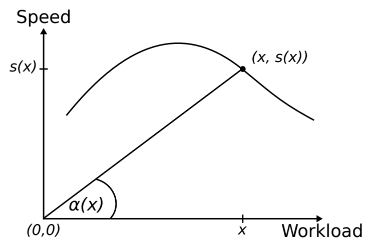

Fig. 1. Example of speed function suitable for minimization of computation time through load balancing. Angle $\alpha(x)$ represents the computation time: the greater the angle, the shorter the computation time.

The main body of the load balancing algorithms designed for performance optimization explicitly or implicitly assume that the speed of processor does not depend on the size of workload [9], [10], [18], [19], [20], [21]. In other words, the speed functions $s_i(x)$ are assumed to be positive constants, in which case the condition 1 is trivially satisfied. More advanced algorithms are based on functional performance models (FPMs), which represent the speed of processor by a continuous function of the problem size [7], [22]. However, the shape of the function is not arbitrary but has to satisfy the following assumption [8]: Along each of the problem size variables, either the function is monotonically decreasing, or there exists point x such that

- On the interval $[0, x]$, the function is
  - monotonically increasing,
  - concave, and
  - any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.
- On the interval $[x, \infty)$, the function is monotonically decreasing.

These restrictions on the shape of speed functions guarantee that the efficient load balancing algorithms, proposed in [23], [24], [25], [26], [27], [28], will always return a unique solution, minimizing the computation time. At the same time, it is easy to show that the restrictions imposed on FPMs will make them comfortably satisfy the condition 1.

Thus, the state-of-the-art load balancing algorithms designed for optimization of the computational performance of parallel applications assume that their performance profiles satisfy the condition 1. Therefore, correct application of such algorithms requires that the experimental speed points be approximated by a function satisfying this condition. This approximation step may significantly distort the actual performance profile and lead to a substantially non-optimal solution.

3 Optimization of Parallel Applications through Load Imbalancing

In this section, we demonstrate that the performance profile of real-life scientific applications on modern parallel platforms may significantly deviate from the conditions, which guarantee that load balancing will always optimize their computational performance. Based on this observation, we propose an optimization method that uses the performance profile for optimization of the application through its imbalancing.

In this work, we build the performance profile of MPDATA on Intel Xeon Phi. MPDATA is a core component of EULAG (Eulerian/semi-Lagrangian fluid solver), which is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. Its carefully optimized data-parallel implementation on a 61-core Intel Xeon Phi [5] partitions the 3D rectilinear $n \times n \times l$ domain into four equal $\frac{n}{2} \times \frac{n}{2} \times l$ sub-domains, each allocated to a team of 15 cores. This configuration of the application is the best load-balanced configuration identified in [5].

The experimentally constructed speed functions of these four teams, each processing (in parallel) a $120 \times m \times 128$ sub-domain, are shown in Figure 2. In these experiments, the application runs on 60 cores while one core is always reserved for OS. Each of the 60 cores runs four threads of the application making the total number of threads 240. The experimental points for a given m are obtained by execution of the application for a $120 \times 2m \times 128$ domain and measuring the execution time of each team of cores (each team will process a sub-domain of size $120 \times m \times 128$ in this case).

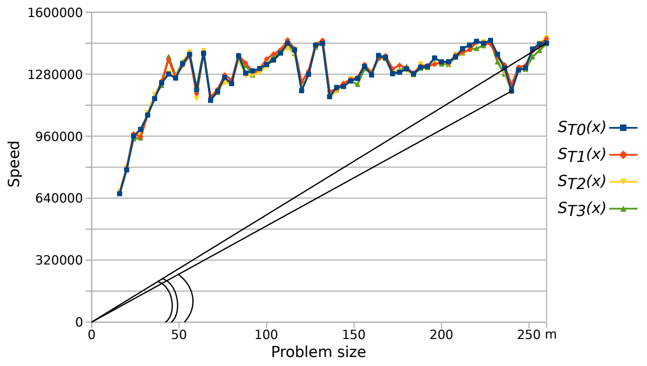

Fig. 2. Speed functions of Intel MIC built for four 15-core teams, each processing in parallel a $120 \times m \times 128$ sub-domain. The speed is measured in cells per second, while the problem size is represented by m.

The graph in Figure 2 clearly shows that for many m and Δm the speed of processing of the $120 \times m \times 128$ sub-domain will be significantly lower than the speed of processing of the $120 \times (m + \Delta m) \times 128$ sub-domain. Moreover, we can also see that $\alpha(m+\Delta m) > \alpha(m)$ for some such m and Δm, which means that the time of processing of the $120 \times m \times 128$ sub-domain will be longer than the time of processing of the $120 \times (m + \Delta m) \times 128$ sub-domain. This is illustrated in Figure 2 by two straight lines that geometrically represent the execution time of the problem for two different sizes. In general, the lower the line, the longer is the execution time. One can see that in our case the lower line corresponds to a smaller problem size and the upper line to a larger one.

This observation can be used to speed up the execution of the application as follows. First, the four speed functions in Figure 2 are practically identical, differing by less than 3%. Therefore, these four functions can be accurately approximated by a single function (for example, by their average). We can use this single function to compare the execution time of the application for various partitions of a $120 \times 2m \times 128$ domain along the second dimension. Namely, the execution time for the equally partitioned domain, when each team is given a sub-domain of size $120 \times m \times 128$, will be same for all teams and characterized by a line with angle $\alpha(m)$, also characterizing the execution time of the whole application. If we re-partition the domain so that two teams get $120 \times (m+\Delta m) \times 128$ sub-domains and two other teams get $120 \times (m - \Delta m) \times 128$ sub-domains, then the execution time of the first two teams will be characterized by lines with angle $\alpha(m - \Delta m)$ while the execution time of the second two teams will be characterized by lines with angle $\alpha(m + \Delta m)$. The time of parallel execution for this configuration will be given by the lowest of these lines, that is, by the line with angle $\min\{\alpha(m + \Delta m), \alpha(m + \Delta m), \alpha(m - \Delta m), \alpha(m - \Delta m)\} = \min\{\alpha(m+\Delta m), \alpha(m-\Delta m)\}$. Now if $\min\{\alpha(m+\Delta m), \alpha(m-\Delta m)\} > \alpha(m)$, then the unequal partitioning will result in faster execution than the equal one.

In general, if the performance profile of an application violates the condition 1, that is,

$$\exists i \in \{1, \ldots, p\}, \exists x > 0, \exists \Delta x > 0\colon \frac{s_i(x)}{x} < \frac{s_i(x + \Delta x)}{x + \Delta x} \tag{2}$$

and the balanced configuration of the application allocates the workload of size x to processor i, then the application can be accelerated if we reduce the accumulated workload of all processors but processor i by Δx so that none of these processors would increase its execution time, and allocate this additional workload to processor i. This method can be applied to optimization of parallel applications on both heterogeneous and homogeneous platforms.

4 Model-based partitioning algorithm for optimal load imbalancing

In this section, we develop the proposed approach for a relatively simple case and introduce a straightforward partitioning algorithm that aims to find the optimal distribution of computations of an application between homogeneous processors using the functional performance model of the application. While simple and not general, this algorithm is sufficiently efficient to be used for optimization of our target application, MPDATA.

Consider the following problem. Let p identical parallel processors be used to execute the workload of size n, and let $s(x)$ be the speed of execution of the workload of size x by a processor. Let Δx be the minimal granularity of workload so that each processor can be only allocated a multiple of Δx. The problem is to find the distribution of the workload of size n between the p processors, which minimizes the computation time of its parallel execution.

To further simplify the problem, we assume that $\frac{n}{p}$ be a multiple of Δx and p be an even number. Then the proposed partitioning algorithm, shown below as Algorithm 1, proceeds as follows. It starts from the balanced distribution of the workload when each processor is assigned the same workload, $\frac{n}{p}$, and calculates the time of parallel execution of the workload for this configuration using the speed function $s(x)$. Then, all processors are divided into two equal teams, called the left team and the right team, and the algorithm goes through a number of iterations. At each iteration, the workload of each processor from the right team is increased by Δx, while the workload of each processor from the left team is decreased by the same amount, Δx. This will not change the total amount of the workload but further imbalance its distribution between the processors. For this redistributed workload, the time of its parallel execution is calculated (again using the speed function $s(x)$). The iterations are repeated until the workload cannot be further redistributed. The algorithm returns the distribution with the minimal calculated execution time.

We do not claim neither optimality nor generality of the presented algorithm. This is an *ad hoc* approximate algorithm designed for a very specific and restricted case, motivated by the application we aim to optimize. A generic optimization algorithm is still required to solve the formulated problem.

Algorithm 1 Distribution of workload n between p homogeneous processors of speed $s(x)$

```
$x_{ropt} = x_{lopt} = x_r = x_l = \frac{n}{p}$
$t_{min} = \frac{\frac{n}{p}}{s(\frac{n}{p})}$
**repeat**
  $x_r = x_r + \Delta x$
  $x_l = x_l - \Delta x$
  $t = \max(\frac{x_r}{s(x_r)}, \frac{x_l}{s(x_l)})$
  **if** $t < t_{min}$ **then**
    $t_{min} = t$
    $x_{ropt} = x_r$
    $x_{lopt} = x_l$
  **end if**
**until** $x_r < n$ & $x_l > 0$
```

It is obvious that if we replace the speed function $s(x)$ by any function $a \times s(x)$, where $a = const$, then this algorithm will return the same solution. We will use this property when applying Algorithm 1 in Section 5.4.

5 Application: Optimization of MPDATA on Intel Xeon Phi

In this section, we apply the partitioning algorithm proposed in Section 4 to optimization of MPDATA on Intel Xeon Phi.

5.1 Intel MIC overview

The Intel MIC architecture is a relatively new system for high performance computing [29]. Intel MIC combines many integrated Intel CPU cores into a single chip. This architecture is built to provide a general-purpose programming environment similar to that provided for Intel CPUs. It is capable of running applications written in industry-standard programming languages such as Fortran, C, and C++. The Intel Xeon Phi (codenamed Knights Corner) is the first product based on Intel MIC architecture. This coprocessor is delivered in form factor of a PCI express device, and can not be used as a stand-alone processor. However, it allows users to directly run individual applications in the native mode without the support of CPU.

In this study, we use the top-of-the-line Intel Xeon Phi 7120P coprocessor. It contains 61 cores clocked at 1.238 GHz, and 16 GB of on-board memory. As the Intel MIC architecture supports four-way hyper-threading, it totally gives 244 logical cores (threads) for a single chip. This coprocessor provides 352 GB/s of memory bandwidth. An important component of each Intel Xeon Phi processing core is its vector processing unit (VPU) [5], that significantly increases the computing power. Each VPU supports a new 512-bit SIMD instruction set called Intel Initial ManyCore Instructions. The theoretical peak performance of Intel Xeon Phi 7120P is 1208 GFlop/s for double precision numbers.

5.2 Introduction to MPDATA

The MPDATA algorithm is a general approach for integrating the conservation laws of geophysical fluids on micro-to-planetary scales [30]. It belongs to the class of methods for the numerical simulation of fluid flows which are based on the sign-preserving properties of upstream differencing. The MPDATA scheme allows for solving advection problems, and offers several options to model a wide range of complex geophysical flows.

MPDATA is a collection of stencils kernels, which are commonly known as memory bound [31], [32], [33]. The stencils computations have been investigated by many authors over the years [34], [35], [36], [37], [38], [39], [40], [41], [42]. The one of the main direction of improving the efficiency of stencil computations is focused around different strategies of domain decomposition, like space and temporal blocking techniques [34]. These strategies provide as nearly as possible balanced workload of computing resources, and have been adapted to a wide range of multi-/manycore systems (see, e.g., [36], [41], [42]). The main assumption for these techniques is to attempt to better exploit data locality by performing operations on data blocks of a suitable size before moving on to the next block. This assumption has been aggressively used by us in [42] to improve the efficiency of implementing 2D stencil codes on hybrid CPU-GPU platforms.

The MPDATA algorithm corresponds to the group of nonoscillatory forward-in-time algorithms. The number of required time steps depends on a type of simulated physical phenomenon, and can exceed few millions especially when considering the MPDATA algorithm as a part of the EULAG model. For detailed description of the MPDATA mathematical scheme, the reader is referred to [1], [3], [30].

Each MPDATA time step is determined by a set of 17 computational stages, where each stage is responsible for calculating elements of a certain matrix. These stages represent stencil codes which update grid elements according to different patterns. Listing 1 shows a part of the 3D MPDATA stencil-based implementation for the 8-th stage.

Listing 1. Part of 3D MPDATA stencil-based implementation

```
/* ... */
//stage 8
for( ... ) // i - dimension
  for( ... ) // j - dimension
    for( ... ) // k - dimension
      mx[i,j,k]=max(x[i][j][k],
                    x[i-1][j][k], x[i+1][j][k],
                    x[i][j-1][k], x[i][j+1][k],
                    x[i][j][k-1], x[i][j][k+1]);
/* ... */
```

The stages are dependent on each other: outcomes of prior stages are usually input data for the subsequent computations. Every stage reads a required set of matrices from the main memory, and writes results to the main memory after computation. In consequence, a significant traffic to the main memory is generated, which mostly limits the performance of novel architectures. A single MPDATA time step requires 5 input matrices, and returns one output matrix that is necessary for the next step.

5.3 Adaptation of 3D MPDATA to Intel Xeon Phi coprocessor

In our previous work [4], [5], we proposed the adaptation of 3D MPDATA to Intel Xeon Phi coprocessors. The proposed decomposition contributes to ease the memory and communication bounds, and to better exploit computation resources of Intel Xeon Phi. The resulting adaptation is based on the following methodology:

- (3+1)D decomposition of MPDATA heterogeneous stencil computations;
- partitioning of threads into independent work teams;
- parallelization of MPDATA computations;
- scheduling for multicore and manycore systems.

To alleviate the memory-bound nature of MPDATA, we proposed the (3+1)D decomposition of MPDATA stencil computation [5]. The main aim of this decomposition is to take advantage of cache memory reuse by transferring the data traffic associated with all intermediate computation from the main memory to the cache hierarchy. This aim is achieved by using a combination of two well-known loop optimization techniques: loop tiling and loop fusion. Such an approach allows us to reduce the main memory traffic at the cost of additional computations associated with extra areas (*halo*) of all intermediate matrices. Another advantage of this approach is the possibility of reducing the main memory consumption because all intermediate results are stored only in the cache memory.

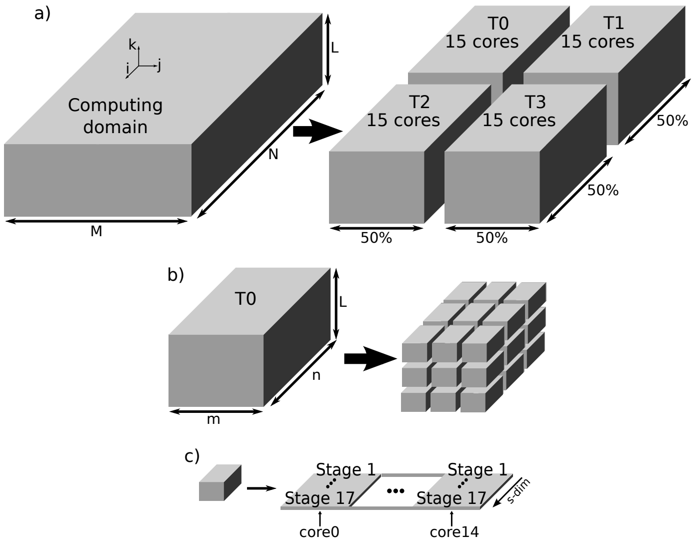

Fig. 3. Hierarchical domain decomposition of the MPDATA computation: a) domain partitioning into sub-domains according to work teams, b) sub-domain decomposition into blocks of size adapted to the cache capacity, and c) parallel execution of MPDATA stages within a single block by work team

The proposed decomposition moves the bulk of data traffic from the main memory to the cache hierarchy. In consequence, a lot of inter- and intra-cache communications are generated between more than 200 Intel MIC's processing cores. To improve the efficiency of the (3+1)D decomposition on Intel Xeon Phi, we provided [5] the partitioning of available cores (threads) into independent work teams. The best numbers of work teams and threads per each team are determined empirically. The usage of work teams allows us to reduce inter- and intra-cache communication overheads due to data transfers between neighbour threads, as well as their synchronization. These advantages are achieved at the cost of some extra computation performed by teams.

Figure 3 illustrates the hierarchical domain decomposition of the MPDATA computation. At the start point, the MPDATA computing domain is partitioned into sub-domains (Figure 3a) according to a given number of work teams. Every sub-domain is processed by a single work team of threads. Within every time step, the work teams execute computations in parallel and independently of each other. After each time step, the work teams are synchronized. Each sub-domain is further partitioned into a number of blocks (Figure 3b), where the size of block is adapted to the cache capacity. The subsequent blocks are processed one by one, and each block is processed in parallel by the corresponding work team. A sequence of all the MPDATA stages is executed in parallel within every block (Figure 3c), taking into account the data dependencies.

The best performance results on a single Intel Xeon Phi obtained in [4], [5] were achieved by partitioning the 3D MPDATA domain in two dimensions into four equal sub-domains, so that there was one-to-one mapping between these sub-domains and the teams of cores arranged in a 2×2 grid as illustrated in Figure 3. This homogeneous partitioning balances the load of the core teams and minimizes the execution time of the application in comparison with all other homogeneous partitioning shapes.

5.4 Applying model-based partitioning algorithm to MPDATA decomposition

Thus, it was *previously* and *experimentally* determined that the best *load-balanced* configuration of the MPDATA application on a Intel Xeon Phi arranges its cores in four 15-core teams as shown in Figure 3 and evenly partitions the $N \times M \times L$ computation domain between these teams, allocating a $\frac{N}{2} \times \frac{M}{2} \times L$ sub-domain to each of the teams. It is important to note that only *balanced* homogeneous distributions were considered in this previous work. In this paper, we start with that *balanced* distribution and use the performance model to *imbalance* the distribution in order to achieve a better performance. To find a better partitioning of the computation domain between these teams of cores, we employ the data-partitioning algorithm, presented in Section 4.

As a first step, we build speed functions of the teams so that the speed of each team be represented by a function of problem size. In the case of MPDATA, the problem size is characterized by the size of the domain processed by the team and therefore represented by three parameters n, m and l. In real-life NWP simulations l is fixed [5]. Therefore, we build speeds of teams as functions of two parameters n and m, setting l to 128, the value typically used in NWP simulations.

In general, the speed should be measured in equal-sized computation units performed per one time unit [8], for example, in flops. In the case of MPDATA, it is difficult to estimate the amount of arithmetic operations that will be executed during the processing of a $n \times m \times l$ computation domain. We know however that with a very high level of accuracy this amount is directly proportional to the number of cells in this domain. Therefore, we measure the speed in cells per second.

The speed functions are built empirically by benchmarking the work teams for a range of problem sizes. For each problem size (n, m), the speed is calculated as $\frac{n \times m \times 128}{t}$, where t is the measured execution time.

It has been shown [13], [43] that in modern multicore, manycore and hybrid platforms, where processing elements are coupled and share resources, the speed of one group of elements may depend on the load of others due to resource contention. Therefore, the groups cannot be considered as independent processing units and their speed cannot be measured separately and independently. In this work, we use the performance measurement method proposed in [13]. According to this method, the speed of each homogeneous processing unit will depend not only on the executed workload but also on the number of other processing units executing the same workload in parallel. Therefore, the performance of the four teams of cores is measured simultaneously rather than separately, thereby taking into account resource contention. To ensure the reliability of measurements, we repeat measurements multiple times. We only measure the computation time of every team without the overheads of inter-team synchronization required after each time step. If the measurements were conducted separately, the measured performance of these teams would not reflect their actual performance during the execution of the application, and therefore performance optimization decisions based on the corresponding performance models would be inaccurate. Figure 4 demonstrates the difference between the speed of team T_0 measured separately and simultaneously with other teams.

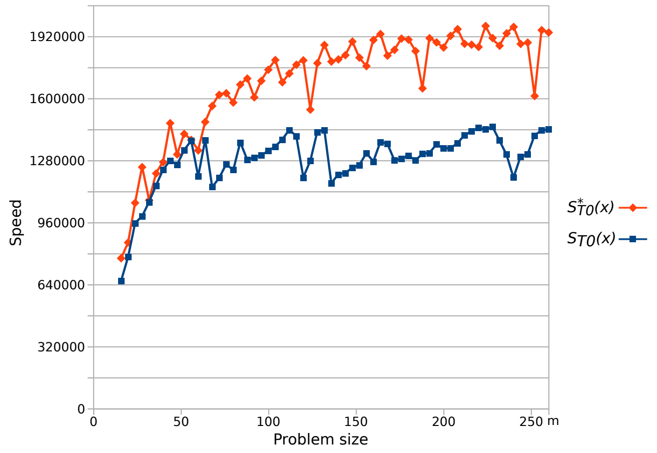

Fig. 4. Comparison of speed functions of team T_0, measured separately($S^*_{T_0}(x)$) and simultaneously with other three teams ($S_{T_0}(x)$) executing the same workload

If we aim to partition a $N \times M \times 128$ domain between the four teams of cores, then for different values of variables n and m ($0 < n < N$, $0 < m < M$) we find the speed of processing of a subdomain of size $n \times m \times 128$ by each of the four teams, making sure that all they start their computations simultaneously. Figure 5 illustrates how the obtained experimental points are used to approximate the speed of team T_0 as a function of n and m. The experimental points for the speed function were obtained with steps $\Delta n = \Delta m = 4$ for both n and m.

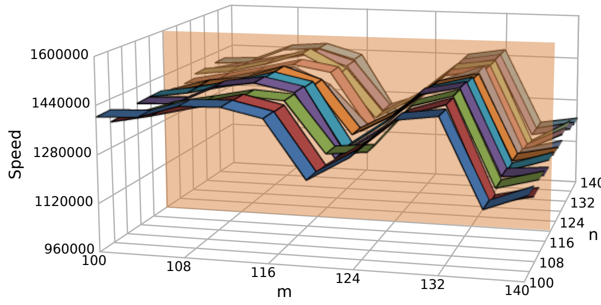

Fig. 5. Experimentally built speed of execution of the MPDATA workload by team T_0 as function of two parameters n and m ($l = 128$)

We can see that for a fixed value of m the speed varies very slowly and very little with variation of n, staying nearly constant. More detailed analysis of the speed functions confirms that the speed of team strongly depends on m and very little depends on n. This observation allows us to assume that with a high level of accuracy the optimal (or at least a near optimal) partitioning of the $N \times M \times 128$ domain between the four teams can be obtained from the optimal even load-balanced partitioning, which allocates a sub-domain of size $\frac{N}{2} \times \frac{M}{2} \times 128$ to each team, by fixing n to $\frac{N}{2}$ and varying m.

Mathematically, it means that we only have to deal with speed functions of just one parameter, m. These functions are obtained from the previously built speed functions of two parameters, n and m, by fixing the parameter n. Geometrically, this can be illustrated as follows. The two-parameter speed functions are represented by surfaces. By fixing parameter n to $\frac{N}{2}$, we cut the surfaces by a vertical plane $n = \frac{N}{2}$ as shown in Figure 5. Curves obtained on this plane will represent the one-parameter speed functions as shown in Figure 6 for $n = 120$.

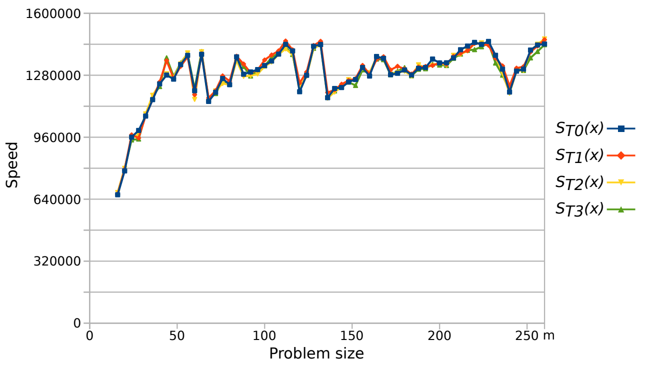

Fig. 6. Speeds of four teams built simultaneously as functions of parameter m ($n = 120$ and $l = 128$)

Finally, as all four teams have very close speed functions (as can be seen in Figure 6), we calculate their average (shown in Figure 7) and use it as input to Algorithm 1 to find the optimal value of m for each team.

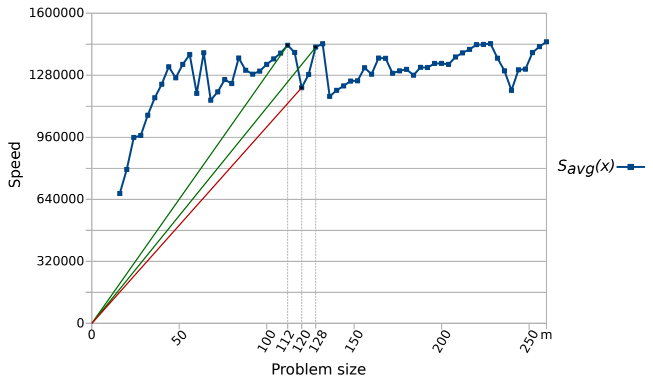

Fig. 7. Averaged speed of teams built as a function of parameter m ($n = 120$ and $l = 128$). The statistical properties of averaging are given by the relative standard deviation (RSD) less than 3.0%, and the average value of RDS equal to 0.74%.

More specifically, let the MPDATA domain be of size $240 \times 240 \times 128$. Then, we consider our four teams as four identical abstract processors, $p = 4$, the speed of each of which is given by the speed function shown in Figure 7. Note that in this function, the amount of workload is given in frames of cells of size 120×128, while the speed is given in cells per second. As pointed in Section 4, despite the unit of workload used to measure the speed (axis y) is 120×128 times greater than the unit of workload used to measure the size of workload (axis x), we can safely use this function as input to Algorthm 1.

The solution returned by Algorthm 1 allocates $m = 112$ frames to even abstract processors and $m = 128$ frames to odd processors. This corresponds to partitioning of the $240 \times 240 \times 128$ domain into two sub-domains of size $120 \times 112 \times 128$, allocated to teams T_0 and T_2, and two sub-domains of size $120 \times 128 \times 128$, allocated to teams T_1 and T_3. The traditional load-balanced approach partitions the domain in four equal sub-domains of size $120 \times 120 \times 128$. This is illustrated in Figure 8.

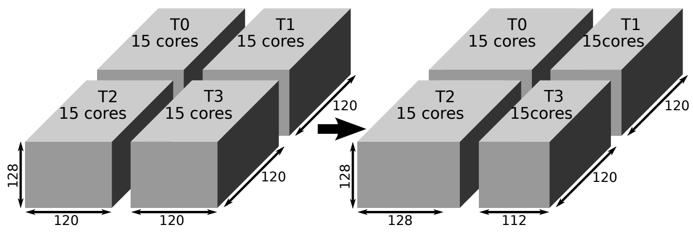

Fig. 8. Optimal partitioning of MPDATA of size $240 \times 240 \times 128$ between 4 teams

In general, the theoretical execution time is calculated using the formula:

$$t = \max(\frac{x_r}{s(x_r)}, \frac{x_l}{s(x_l)}),$$

where x_r is the number of cells processed by each of the right teams, x_l is the number of cells processed by each of the left teams, and $s(x_r)$ and $s(x_l)$ are the speeds of processing of the cells. For the even partitioning, it gives us:

$$t_e = \max(\frac{120 \cdot 120 \cdot 128}{1240376}, \frac{120 \cdot 120 \cdot 128}{1240376}) = 1.486[s],$$

while for the uneven partitioning returned by Algorithm 1, we have:

$$t_u = \max(\frac{120 \cdot 128 \cdot 128}{1418579}, \frac{120 \cdot 112 \cdot 128}{1436742}) = \\ = \max(1.386, 1.197) = 1.386[s].$$

Figure 7 illustrates these two solutions. The red line represents the execution time of the evenly partitioned configuration when all teams take the same time to complete their computations. The two green lines represent the execution time of the left and right teams for the uneven partition. The upper line corresponds to the execution time of each of the right teams processing smaller sub-domains, while the lower green line – each of the left teams. The overall time of parallel execution will be given by the lower green line. We can see that this green line is positioned above the red one, which indicates that the execution time of the unevenly partitioned configuration is less than that of the evenly partitioned. We can also clearly see that any attempt to reduce the imbalance by brining the two green lines closer to each other will result in the lower line moving further down, which means the increase of the overall execution time

5.5 Experimental results

In this subsection, we experimentally evaluate the optimization technique presented in Section 5.4.

The performance results presented in this subsection are obtained for double precision MPDATA computations corresponding to 40 time steps. All the benchmarks are compiled as native executables using the Intel compiler (v.15.0.2), and run on the Intel Xeon Phi 7120P coprocessor. To ensure the reliability of the results, measurements are repeated several times, and average execution times are used. We find the confidence interval and stop the measurements if the sample mean lies in the interval with the confidence level 95%. We use Student's t-test, assuming that the individual observations are independent and their population follows the normal distribution.

Table 1 includes both theoretical and experimental execution times of MPDATA for the domain of size $240 \times 240 \times 128$. These results are obtained for different configurations of partitioning, including the traditional "load-balanced" partitioning ($\Delta m = 0$) and a range of "unbalanced" partitioning for different $\Delta m > 0$. The theoretically optimal Δm returned by Algorithm 1 is equal to 8, which corresponds to the configuration where each odd or even team processes the sub-domain of size $120 \times 128 \times 128$ or $120 \times 112 \times 128$ respectively. In this case, the estimated execution time of 1.386 seconds is very close to the real computation time which is 1.364 seconds. According to experiments, the shortest execution time is achieved for $\Delta m = 9$, when computations take 1.348 seconds.

Comparing the experimental and theoretical times, we can see that the accuracy of theoretical prediction is very good, with prediction errors being as small as $2 - 4\%$. In general, we can identify two main factors contributing into the prediction error:

- While the experimentally built speed functions of teams T_0, T_1, T_2 and T_3 are not identical, suggesting some degree of their heterogeneity in execution of the MPDATA workload, our theoretical model considers them homogeneous and represents their speed by the average of the real speed functions, which is then used as input to Algorithm 1.
- During the construction of the speed functions, the speed of a team for problem size $n \times m \times l$ is measured when other teams process in parallel sub-domains of the same size, $n \times m \times l$. However, during the execution of the application in our experiments different teams process sub-domains of slightly different sizes when $\Delta m \neq 0$.

TABLE 1
Theoretical and experimental execution times for MPDATA domain of size $240 \times 240 \times 128$ with different configurations of partitioning. The odd work teams process the sub-domain of size $120 \times (120 + \Delta m) \times 128$, while the even teams –$120 \times (120 - \Delta m) \times 128$.

Offset Δm	Theoretical time [s]	Experimental time [s]	Speedup
0	1.486	1.548	1.000
4	1.470	1.470	1.053
6	1.401	1.374	1.127
7	1.422	1.361	1.137
8	1.386	1.364	1.135
9	1.398	1.348	1.148
10	1.397	1.352	1.145
11	1.429	1.372	1.129
12	1.402	1.368	1.131

Table 1 also demonstrates the performance gain from applying the proposed load-imbalancing optimization. For the imbalanced configurations presented in this table, we notice a better performance than for the load-balanced configuration of the MPDATA decomposition. The largest performance gain is achieved for $\Delta m = 9$, giving the speedup of 1.148x.

TABLE 2
Experimental time for all work teams with different partitionings: the odd work teams process the sub-domain of size $120 \times (120 + \Delta m) \times 128$, while the even teams –$120 \times (120 - \Delta m) \times 128$.

Offset	Experimental time [s]				
Δm	Team 0	Team 1	Team 2	Team 3	Total
0	1.515	1.498	1.518	1.503	1.548
4	1.456	1.247	1.455	1.249	1.470
6	1.364	1.161	1.359	1.162	1.374
7	1.355	1.161	1.341	1.168	1.361
8	1.355	1.166	1.349	1.172	1.364
9	1.340	1.155	1.335	1.161	1.348
10	1.345	1.141	1.337	1.152	1.352
11	1.363	1.156	1.357	1.154	1.372
12	1.360	1.163	1.353	1.165	1.368

Table 2 complements the results in Table 1 giving experimental execution times of the individual teams. We can clearly see a significant difference between the execution times measured for the odd and even teams when $\Delta m \neq 0$. Obviously, this difference is caused by the unbalanced workloads for the odd and even teams. However, the total execution time is shorter than in the case of balanced workloads ($\Delta m = 0$).

Table 2 also shows that the total execution time is always slightly longer than the maximum time among all teams. It is mainly due to the fact that the computation time of every team is measured without the overheads of inter-team synchronization required after each time step. In addition, the results in Table 2 are presented in Figure 9 in a graphical form.

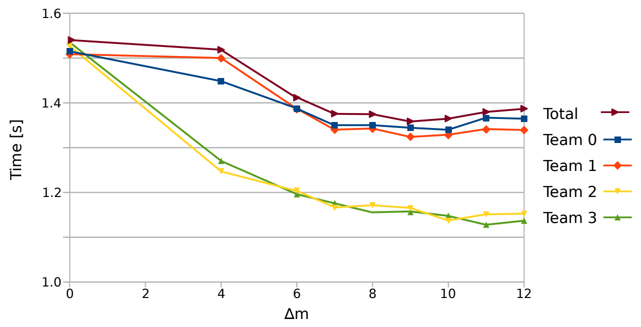

Fig. 9. Experimental execution times measured for individual work teams and the total execution time measured for the whole MPDATA workload

Finally, we evaluate the proposed model-based partitioning algorithm for the MPDATA domain of size $480 \times 480 \times 128$. As in the previous case, the application is executed for different configurations of partitioning, for a range of Δm. In this case, however, the theoretically optimal configuration returned by Algorithm 1 is exactly the same as the experimentally optimal one, both achieved when $\Delta m = 20$. The prediction errors are also smaller in this case, not exceeding 3%. The experimental execution time for $\Delta m = 20$ is 5.338 seconds, in comparison with 6.140 seconds for the even partitioning.This allows us to accelerate the MPDATA computations by 1.15 times. Moreover, the performance gain is also observed for other unbalanced configurations, but it is smaller than 1.15x. The results of these experiments are included in Table 3.

TABLE 3
Theoretical and experimental execution times for MPDATA domain of size $480 \times 480 \times 128$ with different configurations of partitioning. The odd work teams process the sub-domain of size $240 \times (240 + \Delta m) \times 128$, while the even teams – $240 \times (240 - \Delta m) \times 128$.

Offset Δm	Theoretical time [s]	Experimental time [s]	Speedup
0	6.136	6.140	1.000
4	5.731	5.681	1.081
8	5.809	5.806	1.058
12	5.543	5.453	1.126
16	5.509	5.418	1.133
20	5.499	5.338	1.150
24	5.624	5.477	1.121

5.6 Self-adaptable implementation of MPDATA

MPDATA is typically used in long running simulations, such as numerical weather prediction, that require several thousand time steps. Therefore, in the context of real-life simulations, MPDATA is executed many thousand times for the same given size of domain. What is important is that for a given domain decomposition the speed of execution of one time-step will be the same for any time step, whether it is first, second or 1234th. We can exploit these facts and use a small number of the initial time steps to build the speed function, find the optimal division of the workload and then use this optimal division for the rest of the execution. This way we can make MPDATA self-adaptable.

In our design, we do not build at runtime the full speed function but only its small part, sufficient for finding the optimal division for the given size of domain. More specifically, for a given problem size we fix n to the value of the balanced solution and only vary m, building the speed as a function of m for a limited range of values around the balanced solution.

Figure 10 illustrates how the necessary part of the speed function is built for a given problem size. In this example, all work teams process together the MPDATA domain of size $240 \times 240 \times 128$. In the initial MPDATA time steps, the computation are executed with the load-balanced configuration where the domain is partitioned in four equal sub-domains of size $120 \times 120 \times 128$ (Figure 10a). In the subsequent MPDATA time step, the total execution time (Figure 10b) and the speed (Figure 10c) are obtained for $\Delta m = 0$ and $m = 120$. Then, the first redistribution takes place with $\Delta m = 4$, where the odd work teams process the sub-domains of size $120 \times (120 + \Delta m) \times 128$, while the even teams – the sub-domains of size $120 \times (120 - \Delta m) \times 128$ (see Figure 10d). In the next time step, this new configuration of the application is executed and the total execution time (Figure 10e) and speed (Figure 10f) for both odd ($m = (120 + \Delta m)$) and even ($m = (120 - \Delta m)$) teams are determined. This gives us two additional points, for $m = 116$ and $m = 124$, in the approximation of the speed function (Figure 10f). This procedure is repeated for other redistributions with incremented Δm (Figures 10g–10o) until a significant drop in performance is registered for $\Delta m = 16$ (Figures 10m–10o). The constructed part of the speed function (Figure 10o) is then used as input to Algorithm 1, returning the optimal configuration for the given problem size. As a result, the indicated division of the workload is used for the rest of the execution.

The proposed method has been successfully applied as a run-time mechanism for self-adaptive implementation of MPDATA. During the initial time steps, this mechanism builds the required part of the speed function, and then uses Algorithm 1 to find the optimal partition. After the optimal partition is found, MPDATA continues its execution with this fixed partition as normal.

The potential overhead of this mechanism can come from three additional operations: (i) evaluation of the speed for a given distribution, (ii) execution of the data partitioning algorithm, and (iii) redistribution.

In general, the proper evaluation of the speed for any given distribution would require repetition of the execution of the same time step several times to ensure the acceptable accuracy of measurements. However, in the case of MPDATA the sufficient number of repetitions has proved to be just one because of the stable computational intensity

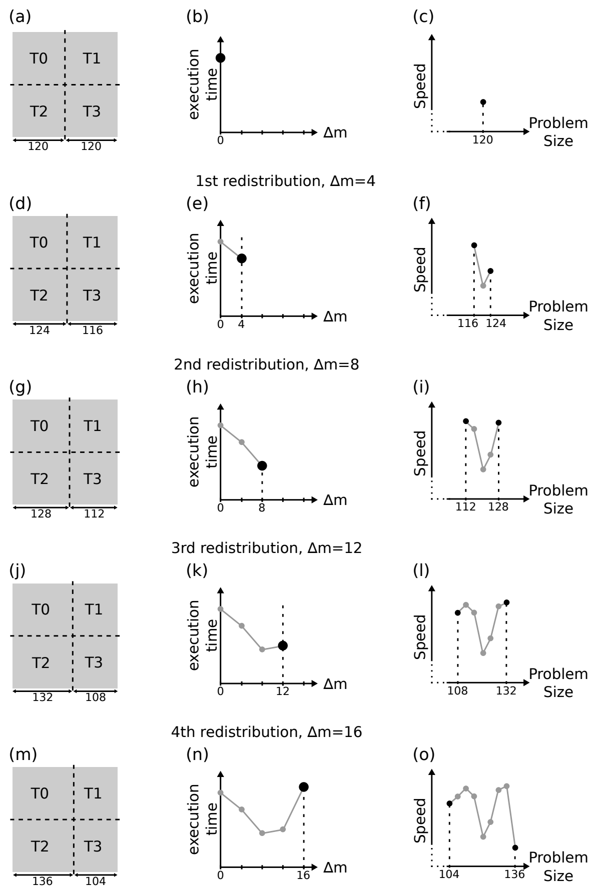

Fig. 10. Workflow for building a part of desired speed function

of each MPDTA time step. The second run never improved the estimation of the speed on the experimental platform to the extent that would change the solution returned by Algorithm 1. This property of MPDATA allowed us to significantly reduce the related overhead.

Finally, the redistribution does not cause an additional overhead associated with data locality because the repartitioning takes place between subsequent time steps when all data has to be reloaded within the main memory and caches, regardless of whether the redistribution happens or not.

In all our experiments, the self-adaptable MPDATA managed to find the optimal division in less than 20 time steps. Furthermore, the total overhead associated with the proposed run-time mechanism was consistently less than 0.05 second. Given that the number of initial times steps involved in the model construction has never exceeded twenty, this means that the overhead has never exceeded 0.0025 second per time step involved in finding the optimal partition. For the domain size of $480 \times 480 \times 128$, this accounts for less than 2% of the execution time of one time step, and this percentage will further decrease with the increase of the domain size. In numerical weather prediction, where MPDATA is used, the simulation runs for several thousand time steps (over 16000 for a 2-day prediction). Thus, the potential cost of this optimization mechanism is less than 0.005% of the total execution time of the application.

To conclude, the cost of this overhead is negligibly small. Despite the distribution of computations during these initial time steps is not optimal, the overall performance gains reach up to 15%, which is particularly noticeable for long time simulations. Generally, assuming that the optimal decomposition can be found during the first 20 time steps, the cost of the proposed self-adapting mechanism is stable for any problem size and for any number of time steps. The cost of this mechanism has a negligible impact on the performance in the case of a very small number of time steps and/or a very small domain size, while in all other cases it has practically no negative effect at all.

6 Conclusion

Modern compute nodes are characterized by both the increasing number of (possibly, heterogeneous) processing elements and a high level of complexity of their integration. Various resources such as caches and data links are shared in an hierarchical and non-uniform way. This makes the development of efficient applications for such platforms a very difficult and challenging task. It would be naive to expect that the performance profile of real-life scientific applications on these platforms will always be comfortably nice and smooth to suit traditional load-balancing techniques used for minimization of their computation time. Therefore, new optimization approaches that do not rely on such increasingly unrealistic assumptions are needed. This work has presented one such approach and demonstrated its applicability to optimization of a real-life application on a modern HPC platform.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474. This research was conducted with the financial support of NCN under grants no. UMO-2011/03/B/ST6/03500, and no. UMO-2015/17/D/ST6/04059. This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

The authors are grateful to the Czestochowa University of Technology for granting access to Intel Xeon Phi coprocessors provided by the MICLAB project no. POIG.02.03.00.24-093/13.

References

[1] P. Smolarkiewicz, "Multidimensional Positive Definite Advection Transport Algorithm: An Overview," *Int. J. Numer. Meth. Fluids*, vol. 50, pp. 1123–1144, 2006.

[2] Z. Piotrowski, A. Wyszogrodzki, and P. Smolarkiewicz, "Towards Petascale Simulation of Atmospheric Circulations with Soundproof Equations," *Acta Geophysica*, vol. 59, pp. 1294–1311, 2011.

[3] P. Smolarkiewicz and W. Grabowski, "The Multidimensional Positive Definite Advection Transport Algorithm: Nonoscillatory Option," *J. Comput. Phys.*, vol. 86, pp. 355–375, 1990.

[4] L. Szustak, K. Rojek, and P. Gepner, "Using Intel Xeon Phi coprocessor to accelerate computations in MPDATA algorithm," *Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2013, Part I. LNCS*, vol. 8384, pp. 582–592, 2014.

[5] L. Szustak, K. Rojek, T. Olas, L. Kuczynski, K. Halbiniak, and P. Gepner, "Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor," *Scientific Programming*, http://dx.doi.org/10.1155/2015/642705, 2015.

[6] L. Szustak, K. Rojek, R. Wyrzykowski, and P. Gepner, "Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture," *In: HiStencils'14 Proc. 1st International Workshop on High-Performance Stencil Computations*, pp. 51–56, 2014.

[7] A. Lastovetsky and J. Twamley, "Towards a realistic performance model for networks of heterogeneous computers," in *High Performance Computational Science and Engineering*. Springer, 2005, pp. 39–57.

[8] 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.

[9] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *GPGPU-2*. ACM, 2009, pp. 46–51.

[10] C. Yang, F. Wang, Y. Du *et al.*, "Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing," in *Cluster'10*, 2010, pp. 19–28.

[11] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *IPDPS 2008*, 2008, pp. 1–10.

[12] K. Rojek and R. Wyrzykowski, "Parallelization of 3D MPDATA Algorithm using Many Graphics Processors," *Lecture Notes in Computer Science*, vol. 9251, pp. 445–457, 2015.

[13] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2506–2518, 2015.

[14] M. D. Linderman, J. D. Collins, H. Wang *et al.*, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, 2008.

[15] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *EuroPar'09*, 2009, pp. 56–65.

[16] 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, pp. 121–130, 2009.

[17] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *ICS '12*. ACM, 2012, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304625

[18] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO-42*, 2009, pp. 45–55.

[19] M. Cierniak, M. Zaki, and W. Li, "Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations," *Computer J.*, vol. 40, pp. 356–372, 1997.

[20] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520–535, 2001.

[21] J. Martínez, E. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE," *J. Supercomput.*, vol. 58, no. 2, pp. 151–159, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11227-009-0350-1

[22] A. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. Santa Fe, New Mexico, USA: IEEE Computer Society, 26-30 April 2004.

[23] A. Ilic, F. Pratas, P. Trancoso, and L. Sousa, "High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU," in *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*. IOS Press, 2011.

[24] J. Colaco, A. Matoga *et al.*, "Transparent Application Acceleration by Intelligent Scheduling of Shared Library Calls on Heterogeneous Systems," in *PPAM 2013, Part I*, 2014, pp. 693–703.

[25] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Computing*, vol. 33, no. 12, pp. 757–779, 2007.

[26] 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.

[27] A. AlOnazi, D. Keyes, A. Lastovetsky, and V. Rychkov, "Design and Optimization of OpenFOAM-based CFD Applications for Hybrid and Heterogeneous HPC Platforms," *arXiv preprint arXiv:1505.07630*, 2015.

[28] D. Clarke, A. 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*. Springer, 2012, pp. 450–459.

[29] *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors*. Colfax International, 2013.

[30] P. Smolarkiewicz and L. Margolin, "MPDATA-A Multipass Donor Cell Solver for Geophysical Flows," in *Godunov Methods: Theory and Applications*, E. Toro, Ed. Springer, 2001, pp. 833–839.

[31] K. Rojek, M. Ciznicki, B. Rosa, P. Kopta, M. Kulczewski, K. Kurowski, Z. Piotrowski, L. Szustak, D. Wojcik, and R. Wyrzykowski, "Adaptation of fluid model EULAG to graphics processing unit architecture," *Concurrency and Computations: Practice and Experience*, vol. 27 (4), pp. 937–957, 2015.

[32] M. Wittmann, G. Hager, J. Treibig, and G. Wellein, "Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters," *Parallel Process. Lett.*, vol. 20 (4), pp. 359–376, 2010.

[33] W. Xue, C. Yang, H. Fu, X. Wang, Y. Xu, J. Liao, L. Gan, Y. Lu, R. Ranjan, and L. Wang, "Ultra-Scalable CPU-MIC Acceleration of Mesoscale Atmospheric Modeling on Tianhe-2," *Computers, IEEE Transactions on*, vol. 64, no. 8, pp. 2382–2393, 2015.

[34] R. Cruz, M. Araya-Polo, and J. Cela, "Introducing the Semi-stencil Algorithm," *Lect. Notes in Comp. Sci.*, vol. 6067, pp. 496–506, 2010.

[35] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *SIAM Rev.*, vol. 51(1), pp. 129–159, 2009.

[36] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," *In: SC 08 Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2008.

[37] G. Hager and M. Wittmann, "Introduction to High Performance Computing for Science and Engineers," *CRC Press,*, vol. 20 (4), pp. 359–376, 2011.

[38] T. Malas, G. Hager, H. Ltaief, and D. Keyes, "Towards energy efficiency and maximum computational intensity for stencil algorithms using wavefront diamond temporal blocking," *arXiv preprint arXiv:1410.5561*, 2014.

[39] G. Rivera and C.-W. Tseng, "Tiling Optimizations for 3D Scientific Computations," *In: SC00 Proc. 2000 ACM/IEEE Conf. on Supercomputing*, 2000.

[40] J. Treibig, G. Wellein, and G. Hager, "Efficient multicore-aware parallelization strategies for iterative stencil computations," *Journal of Computational Science*, vol. 2, pp. 130–137, 2011.

[41] D. Unat, X. Cai, and S. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," *In: ICS 11 Proc. Int. Conf. on Supercomputing*, pp. 214–224, 2011.

[42] R. Wyrzykowski, L. Szustak, and K. Rojek, "Parallelization of 2D MPDATA EULAG Algorithm on Hybrid Architectures with GPU accelerators," *Parallel Computing*, vol. 40, pp. 425–447, 2014.

[43] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore platforms," in *2011 IEEE International Conference on Cluster Computing (Cluster 2011)*. Austin, Texas, USA: IEEE Computer Society, Sept 26-30 2011, pp. 580–584.

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).

Lukasz Szustak received his M.Sc. in Computer Science from the Czestochowa University of Technology in 2008 and his PhD in 2012. During this period, his doctoral research focused on adaptation of high performance computing to modern parallel architectures including hybrid platforms. Since 2012, Dr. Szustak is employed at Czestochowa University of Technology. His current work is associated with the development of efficient methods of scheduling, load balancing, and adaptations of stencil based computations to Intel MIC and CPUs architectures.

Roman Wyrzykowski received M.Sc. and Ph.D degrees from the Kiev Polytechnic Institute in Computer Science in 1982 and 1986, respectively. Since 1982, he is employed at the Czestochowa University of Technology, Poland, where currently he is the head of Department of Computer and Information Science. His fields of expertise are: parallel and distributed computing, mapping algorithms onto parallel architectures, cluster and cloud technologies with applications. Since 1994, he has chaired the program committee of the PPAM series of international conferences on parallel processing and applied mathematics.