Check for updates

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2019, Vol. 33(3) 534–553
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342019828153
journals.sagepub.com/home/hpc
SAGE

Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern Intel processors

Lukasz Szustak and Pawel Bratek

Abstract
In this work, we take up the challenge of performance portable programming of heterogeneous stencil computations across a wide range of modern shared-memory systems. An important example of such computations is the Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), the second major part of the dynamic core of the EULAG geophysical model. For this aim, we develop a set of parametric optimization techniques and four-step procedure for customization of the MPDATA code. Among these techniques are: islands-of-cores strategy, (3+1)D decomposition, exploiting data parallelism and simultaneous multithreading, data flow synchronization, and vectorization. The proposed adaptation methodology helps us to develop the automatic transformation of the MPDATA code to achieve high sustained scalable performance for all tested ccNUMA platforms with Intel processors of last generations. This means that for a given platform, the sustained performance of the new code is kept at a similar level, independently of the problem size. The highest performance utilization rate of about 41–46% of the theoretical peak, measured for all benchmarks, is provided for any of the two-socket servers based on Skylake-SP (SKL-SP), Broadwell, and Haswell CPU architectures. At the same time, the four-socket server with SKL-SP processors achieves the highest sustained performance of around 1.0–1.1 Tflop/s that corresponds to about 33% of the peak.

Keywords
Parallel programming, performance portability, shared-memory systems, heterogeneous stencils, EULAG model, MPDATA, code parameterization, Skylake, Knights Landing

1. Introduction

Multi- and manycore architectures of emerging microprocessor designs (Ang et al., 2014; Rico-Gallego et al., 2017) have become increasingly complex, hierarchical, and heterogeneous (Malik et al., 2016; Rojek and Szustak, 2012; Szustak et al., 2017). With the quick development of computing platforms and software environments, application developers are forced to contend with a variety of parallel architectures. Since applications far outlive any computer system, porting application codes has become a difficult, time-consuming, and significant challenge for scientific and commercial environments (Szustak et al., 2016; Unat et al., 2014).

In this work, we take up the challenge of providing the performance portability for a rather complex scientific application. The application we study implements the Multidimensional Positive Definite Advection Transport Algorithm (MPDATA; Smolarkiewicz, 2006) that contains a set of stencil-based kernels with heterogeneous patterns. Besides the GCR solver, MPDATA is the second major part of the dynamic core of the EULAG (Eulerian/semi-Lagrangian) geophysical model (Smolarkiewicz and Charbonneau, 2013; Wyrzykowski et al., 2012b). EULAG is an established numerical model developed for simulating thermo-fluid flows across a wide range of scales and physical scenario, including numerical weather prediction, simulation of urban flows, turbulences, and ocean currents (Kumar et al., 2016; Smolarkiewicz et al., 2016; Strugarek et al., 2016).

Institute of Computer and Information Science, Faculty of Mechanical Engineering and Computer Science, Czestochowa University of Technology, Czestochowa, Poland

Corresponding author:
Lukasz Szustak, Institute of Computer and Information Science, Faculty of Mechanical Engineering and Computer Science, Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland.
Email: lszustak@icis.pcz.pl

The goal of our research is to investigate and improve the process of porting the MPDATA applications for a wide range of modern computing systems with the shared-memory architecture. We target that the same piece of code—from the user perspective—can be executed on different architectures with the high sustained performance.

In particular, the Intel processors are becoming the fastest growing choice for high performance computing in the last years (see, e.g. `https://top500.org`). That is why methods proposed in this work are evaluated on a variety of Intel microarchitectures released in the last 5 years. We choose the five Intel platforms, including four- and two-socket ccNUMA servers based on Skylake (SKL), Broadwell (BDW), Haswell (HSW), as well as Knights Landing (KNL) chips.

In this article, we propose a parameterized adaptation (or transformation) of the MPDATA application to shared-memory systems. We expect to exploit the parallel processing benefits of high performance systems in conjunction with the adaptive application based on a new version of reconfigurable code. In general, we propose to provide the configurability of the MPDATA parallel code to customize it to a given platform. The proposed idea involves parameterizing characteristics of different computing platforms. As a result, the adaptive MPDATA has to follow along with a variety of hardware architectural issues such as memory hierarchy, threading, vectorization, and their interaction.

In order to ensure performance portability among a variety of Intel processors, we deliver a set of parametric optimization techniques for the MPDATA application. The first technique—(3+1)D decomposition—overcomes the memory and communication constraints by improving both cache reusing and the data locality. The second one—islands-of-cores strategy—allows us to take advantage of multi-socket ccNUMA platforms by searching trade-off and synergy between computations and communications. The third one is responsible for the workload distribution across available cores and threads, by providing efficient and flexible usage of data parallelism and simultaneous multithreading (SMT). In the fourth technique, a nonconventional process of synchronization is implemented, which—in contrast to the barrier approach—makes synchronization only between interdependent threads. Finally, the main goal of the last technique is to ensure the performance portability while vectorizing MPDATA computations.

Furthermore, to make the high performance portable, we provide a flexible tuning of the parameterized transformation of the MPDATA code. The main assumption is to calibrate the proposed parametric optimization techniques for a given computing platform, before the execution of a specific numerical simulation. To meet this aim, we propose to develop an external mechanism—not included in the MPDATA code—that produces the customized configuration of the MPDATA parallel code.

The proposed combination of the parametric optimization techniques allows us to improve radically the efficiency of MPDATA and perform computation with the high sustained performance over all the tested platforms. The proposed adaptation accelerates the MPDATA computation about 10×, 8×, 7×, 6×, and 9× for respectively four-socket server with SKL CPUs, two-socket SKL, BDW, and HSW-based servers, as well as the KNL processor. Furthermore, high utilization rates of around 41–46% of the theoretical peak performance are achieved for the two-socket servers. At the same time, the four-socket platform reaches about 33% of the peak performance (1.0–1.1 Tflop/s), while a single KNL processor gets 20% of the peak.

This article is organized as follows. Section 2 outlines shared-memory systems based on Intel processors, while Section 3 introduces the MPDATA application. Section 4 presents the idea of the parameterized adaptation of MPDATA to shared-memory systems, while Section 5 reveals the parametric optimization techniques used in this adaptation. Section 6 presents a four-step procedure for the adaptive customization of the new MPDATA code. Section 7 describes the evaluation of the proposed approach using five servers with various Intel processors. Section 8 discusses related works, while Section 9 concludes the article.

2. Introduction to shared-memory systems based on Intel processors

In this work, we use five computing platforms based on various Intel microarchitectures: one four-socket server with SKL-SP architecture, three two-socket servers with SKL-SP, BDW, and HSW architectures, and one platform with the KNL processor. Table 1 summarizes parameters of these platforms.

Among parameters that characterize every platform are: frequency, numbers of sockets and cores/threads, SIMD type, number of memory controllers, as well as details of the memory hierarchy. All the platforms feature the out-of-order execution model, and excluding the HSW-based server, the rest of them support the SMT using either two threads per core for Intel Xeon CPUs or four threads per core for KNL.

For all platforms, each core is equipped with 64 KB of L1 cache. The SKL-SP processors have a larger L2 cache, and smaller non-inclusive L3 cache in comparison with BDW or HSW CPUs (Intel, 2018). The key difference is that instead of copying data both to L2 and L3 caches, data are loaded directly into L2 cache in the novel SKL-SP CPUs (Intel, 2018). At the same time, the KNL processor includes 32 tiles (Jeffers et al., 2016), each of which contains two cores that share 1 MB of L2 cache. All L2 caches of KNL are fully coherent. In addition, the KNL processor is equipped with 16 GB of high bandwidth, on-package MCDRAM memory that offers a much higher bandwidth than the traditional DDR memory.

Whereas BDW and HSW cores are connected by a single ring bus, cores of SKL-SP and tiles of KNL communicate through the 2D mesh interconnection (Intel, 2018; Jeffers et al., 2016). Furthermore, each platform

Table 1. Specification of computing platforms.

Platform		A	B	C	D	E
Computing resources		4 × Intel Xeon Platinum 8180 (4 × SKL-SP)	2 × Intel Xeon Gold 6148 (2 × SKL-SP)	2 × Intel Xeon E5-2697v4 (2 × BDW)	2 × Intel Xeon E5-2697v3 (2 × HSW)	1 × Intel Xeon Phi 7250F (1 × KNL)
Base freq. [GHz] (SIMD frequency)		2.5 (1.7)	2.4 (1.6)	2.3 (2.0)	2.6 (2.2)	1.4 (1.2)
Sockets		4	2	2	2	1
Threads		4 × 56	2 × 40	2 × 36	2 × 14	1 × 272
Cores		4 × 28	2 × 20	2 × 18	2 × 14	1 × 68
SMT threads		2	2	2	1*	4
SIMD		AVX-512	AVX-512	AVX2	AVX2	AVX-512
L2 [MB]		4 × 28 × 1	2 × 20 × 1	2 × 18 × 0.25	2 × 14 × 0.25	68 × 0.5
L3 [MB]		4 × 38.5	2 × 27.5	2 × 45	2 × 35	—
Memory		512 GB DDR4-2666	192 GB DDR4-2666	128 GB DDR4-2400	128 GB DDR4-2133	16 GB MCDRAM 96 GB DDR4-2400
Memory control.		4 × 2	2 × 2	2 × 2	2 × 2	1 × 4
Peak perf. [Gflop/s]	Scalar	560	192	165	145	190
	SIMD	3046	1024	576	492	1305
Max. SIMD gain		5.4×	5.3×	3.5×	3.4×	6.8×

Source: https://ark.intel.com.
SKL-SP: Skylake-SP; BDW: Broadwell; HSW: Haswell; KNL: Knights Landing; SMT: simultaneous multithreading.
*Intel HT is disabled.

Table 2. Percentage of extra elements for the MPDATA domain of size 1024 × 512 × 64.

# of subdomains	1	2	4	8
% of extra elements	0.00	0.25	0.74	1.73

MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

consists of at least two memory controllers per processor (four for KNL, two for other processors). These controllers are either placed on opposite ends of the ring (HSW and BDW) or evenly placed on the mesh (SKL-SP and KNL) (Intel, 2018; Jeffers et al., 2016). The expected consequence is that some cores may feature a higher latency while trying to access data via the memory controller placed on the other end.

The studied Intel processors support legacy and modern vector instruction sets, from 64-bit Multimedia Extensions (MMX) to the new 512-bit AVX-512 set (Eltablawy and Vladimirov, 2015). SKL and KNL cores use 512-bit registers, while BDW and HSW cores support 256-bit vectors. The most notable new feature of AVX-512 compared to AVX/AVX2 is the 512-bit vector register width, which is twice the size of AVX/AVX2 registers. Depending on the architecture, all the tested platforms are able to operate on vectors of up to eight double precision floating point numbers (Eltablawy and Vladimirov, 2015; Intel, 2018).

Each core of the considered platforms is equipped with different kind and number of floating point units (Intel, 2018) that typically are able to start up to two scalar (or vector) floating point instructions at every next cycle. For example, a SKL-SP core delivers a throughput of two scalar (or two vector) add instruction per every cycle because it has two FP Add Execution Units connected to ports 0 and 5 (Intel, 2018).

The processors are clocked by the base frequency listed in Table 2. This frequency is used only for non-SIMD workloads, while for workloads heavy in AVX-512 or AVX2, the CPU reduces the clock frequency (Eltablawy and Vladimirov, 2015) (see Table 1). At the same time, all the tested processors are capable of frequency scaling thanks to the Intel Turbo Boost technology, where the maximum turbo frequency depends on the type and intensity of workload, as well as the number of active cores (Intel Xeon Processor, 2017, 2018).

Summarizing, the studied platforms offer the theoretical peak performance (Eltablawy and Vladimirov, 2015) from 492 to 3046 Gflops/s for double precision floating point operations. The presented values of the peak performance take into account the usage of SIMD vectorization, base SIMD frequency, and throughput for the non-fused multiply-add type of instructions. For the fused multiply-add instruction, all platforms offer twice more of the peak performance.

3. Overview of MPDATA

The MPDATA application implements a general approach to integrating the conservation laws of geophysical fluids on micro-to-planetary scales (Smolarkiewicz and Margolin, 1998). The MPDATA algorithm enables solving the continuity equation describing the advection of a nondiffusive quantity Ψ in a flow field (Rosa et al., 2015), namely

$$\frac{\partial \Psi}{\partial t} + div(V\Psi) = 0$$

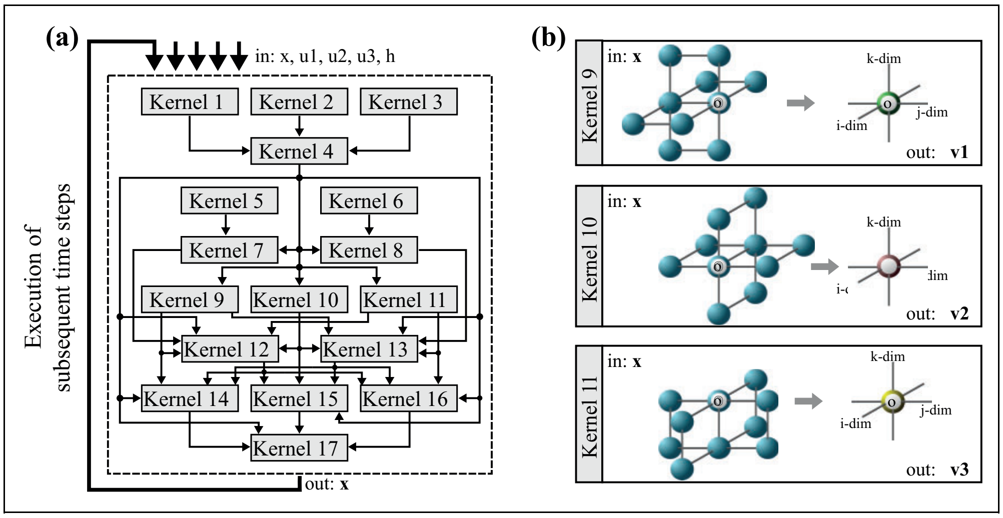

Figure 1. Computational structure for MPDATA application: (a) graph of data dependencies between MPDATA kernels and (b) examples of heterogeneous stencil patterns for MPDATA kernels. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

where V is the velocity vector. The algorithm is positive defined, and by appropriate flux correction (Smolarkiewicz and Margolin, 1998) can be also monotonic. It is a very important feature for the advection of positive definite variables such as specific humidity, cloud water, rain, snow, aerosol particles, and gaseous substances. For a detailed description of the MPDATA mathematical scheme, the reader is referred to the work of Smolarkiewicz and Charbonneau (2013) and Smolarkiewicz and Margolin (1998).

MPDATA corresponds to the group of nonoscilatory forward-in-time algorithms (Szustak et al., 2015), and is typically used for long-running simulations executing several thousand time steps. In this article, we consider solving three-dimensional (3D) problems, when MPDATA is defined in a 3D domain of size $n \times m \times l$ according to i-, j-, and k-dimensions. Moreover, since the accuracy of computation plays a key role for MPDATA, these simulations usually are performed using the double precision floating point format.

The general execution scheme for the MPDATA application is presented in Figure 1. Every MPDATA time step operates on five input 3D matrices (arrays), and returns one output 3D array that is used in the next step. Calculating outcomes of a single time step requires intermediate computations that include 17 stencil kernels. In general, these kernels depend on each others, as the outcomes of prior kernels are usually inputs for the subsequent ones. Figure 1(a) illustrates data dependencies between MPDATA kernels. Each kernel is a stencil code that updates elements of its 3D output array, according to a specific pattern. Although every kernel features a specific stencil pattern, the shape and size of these patterns are quite similar. Typically, such a similarity is observed for a group of three neighbor kernels, where the same shape is slightly transformed in three dimensions. Figure 1(b) shows an example of three stencil patterns with the same shape; they correspond to kernels 9, 10, and 11.

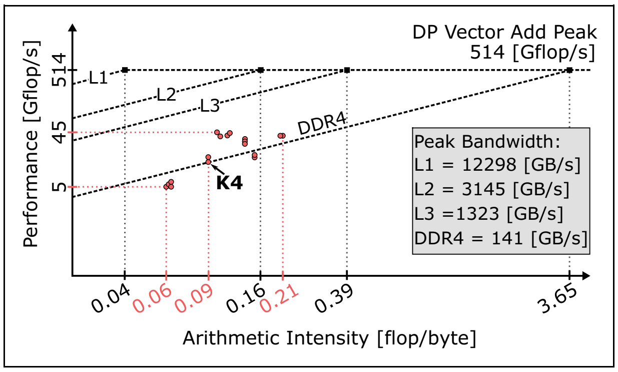

Figure 2. Intel Advisor Roofline plots derived for platform with two Intel Xeon E5-2697v4 CPUs when performing parallel computations for MPDATA domain of size 256 × 256 × 64. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

In a naive, non-optimized implementation of the MPDATA application, the consecutive MPDATA kernels are executed sequentially, one by one, where each kernel is processed in a parallel way using OpenMP. Every MPDATA kernel reads a required set of matrices from the main memory and writes results to the main memory after computation. The consequence is the significant traffic to the main memory. For example, a single iteration of the innermost loop for the fourth kernel (K4) reads eight double precision elements from five input arrays, then performs seven floating point operations, and finally writes one value to xOut-array. Thus, the operational (arithmetic) intensity for K4 is $(7flops)/(9 \times 8bytes) = 0.097\frac{flop}{byte}$.

We use the Intel Advisor Roofline Analysis tool (Jeffers et al., 2016; Vladimirov et al., 2015) to generate the computational profile of the MPDATA application. This cache-aware analysis addresses all levels of the memory/cache hierarchy, and enables us to identify high-impact, under-optimized critical regions of a given code. Figure 2 illustrates a roofline chart determined for the platform with two BDW CPUs when performing parallel computations for the MPDATA domain of size $256 \times 256 \times 64$. The arithmetic intensity for the MPDATA kernels is in the range from 0.06 to 0.21. The Intel Advisor tool places a dot for every MPDATA kernel (loop) in the roofline plot (Figure 2). Most of the loops are around the main memory roof although they are vectorized. Thus, this roof represents a performance bottleneck that has to be addressed by better utilizing the cache memory.

4. Parameterized adaptation of MPDATA to shared-memory systems: General concept

In this work, to achieve the portable high performance, we propose a flexible tuning based on a parameterized transformation of the MPDATA code. We deliver a new parameterized adaptation of the MPDATA application to shared-memory systems, which enables us to provide the configurability of the MPDATA parallel code in order to customize it to a given platform. As a result, the adaptive MPDATA has to follow along with a variety of the hardware architectural issues such as memory hierarchy, multi-/manycore, threading, vectorization, and their interaction.

The key point is providing a suitable parameterization of the parallel MPDATA code that takes into account the interaction of application with hardware for a variety of shared-memory systems. In order to ensure the performance portability across various computing platforms, we develop the following parametric optimization techniques for the MPDATA application:

- (3+1)D decomposition of MPDATA—the main goal of this approach is to overcome the memory and communication constraints for a variety of shared-memory systems by improving both cache reusing and the data locality.
- Partitioning cores into independent work teams—in particular, this technique allows adapting the MPDATA application to take advantages of multi-socket platforms based on the ccNUMA architecture by searching trade-off and synergy between computations and communications.
- Data parallelism and grouping SMT threads into work groups—the purpose is to distribute the MPDATA workload across both physical and logical cores, in a flexible and efficient way.
- New strategy for synchronization—the main idea is to synchronize only interdependent threads instead of using the barrier approach that, in contrast to the developed approach, synchronize all threads.
- Vectorization—the last technique is responsible for ensuring the performance portability for the vectorization of MPDATA computations.

5. Parameterized adaptation of MPDATA: Parametric optimization techniques

5.1. (3+1)D decomposition of MPDATA

Our previous works (Szustak et al., 2014a, 2015) outlined the adaptation of MPDATA to KNC coprocessors, the first generation of the Intel MIC architecture. The aim was to better exploit the cache hierarchy by reducing the data traffic between the main memory and cache hierarchy. To reach this goal, we proposed to reorganize MPDATA computations using a mixture of the loop fusion and loop tiling optimization techniques. As a result, we developed a new strategy of workload distribution for the MPDATA application, which is called *(3+1)D decomposition* of MPDATA.

The main idea of the proposed decomposition is to eliminate accesses to the main memory related to MPDATA intermediate computations (for all kernels). Our intention is to keep intermediate outcomes of MPDATA computations in the cache hierarchy within every time step—without transferring them to the main memory. As a result, the main memory traffic corresponds precisely to transfers related to input and output data (six arrays for every MPDATA time step). Since intermediate results are stored in the cache only, the proposed approach significantly reduces the main memory consumption, by storing 6 arrays of size $n \times m \times l$, instead of 21 such arrays.

The idea of the (3+1)D decomposition is depicted in Figure 3, which for simplicity shows solving an 1D problem with two stencil kernels. In this example, `kernel_1 (K1)` is executed before `kernel_2 (K2)` (Figure 3(a)). In the new version, the original execution scheme is reorganized using the combination of the loop fusion and loop tiling techniques (Figure 3(b)). As a result, the computational domain is decomposed into blocks of computations—the consecutive blocks are processed sequentially, one by one, but each block executes all kernels. In this way, we are able to keep outcomes of kernel K1 in the cache memory only. To achieve this, every block has to be extended by extra computations because of data dependencies between kernels (Figure 3(b)).

This overhead can be reduced or even avoided (Szustak et al., 2015) by leaving partial results in cache when executing successive blocks. Since the extra computations, which correspond to ghost regions of kernels and data arrays, are repeated within adjacent blocks, we propose to keep in cache some partial results of a given block to be used for the next block (Figure 3(c)). As a result, every block is extended by ghost regions, but only some of them have to be calculated within this block, while the others come from the previous block. This strategy splits execution of a block into two phases: (i) computation phase and (ii) phase of leaving/keeping partial results in cache. Its implementation requires maintaining the right order of processing blocks along one of the three dimensions, and mapping partial results onto the cache.

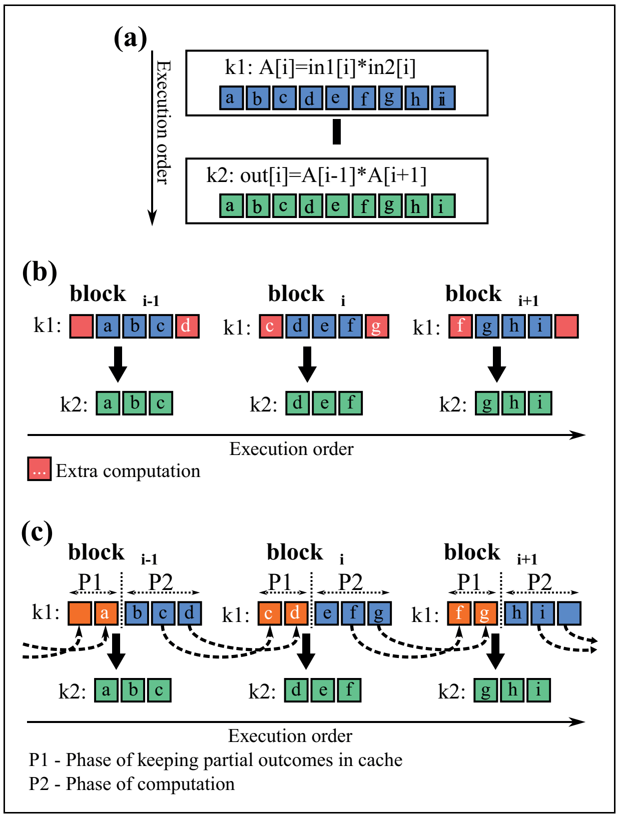

Figure 3. Idea of (3+1)D decomposition of stencil computations: (a) an example of 1D problem with two stencil kernels; (b) partitioning the computational domain into set of blocks using a mixture of loop fusion and loop tiling; and (c) strategy of leaving partial outcomes in cache memory.

In order to implement the (3+1)D decomposition of MPDATA, its domain is partitioned into a set of blocks each of size $nB \times mB \times lB$ that allows us to keep all intermediate results in the cache memory (Figure 4(a)). The consecutive blocks are processed sequentially, one by one, where every block includes all MPDATA kernels (Figure 4(b)). For each block, computations are performed on corresponding chunks of intermediate MPDATA arrays, returning an adequate part of the output array. In consequence, each chunk of a given MPDATA kernel has to be expanded by ghost regions along *i*-, *j*-, and *k*-dimensions. The final size of a given chunk is determined individually for each kernel taking into account data dependencies between kernels (Szustak et al., 2014b, 2015). As a result, a given *r* th MPDATA kernel executes computations on expanded chunks of size $(a_r^i + nB + b_r^i) \times (a_r^j + mB + b_r^j)\times\ (a_r^k + lB + b_r^k)$.

As noticed earlier, since the extra computations are repeated for two adjacent blocks, we can reduce or avoid this overhead by leaving/keeping in the cache partial outcomes of a given block to be used in the next one (see Figure 3(c)). This strategy imposes the execution order for MPDATA blocks along a certain dimension. This choice allows us to avoid extra computations related to one of the three ghost regions of a given block. Since the stencil patterns of MPDATA kernels are identical along all three

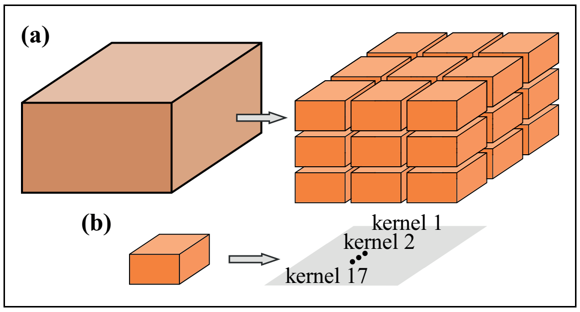

Figure 4. (3+1)D decomposition of MPDATA: (a) partitioning MPDATA domain into set of blocks and (b) execution of all 17 kernels within every block. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

dimensions, this method can be applied successfully for any of the three ghost regions. At the same time, the key point is development of an efficient mechanism for mapping partial results onto the cache space. To reach this aim, the mapping has to be implemented for contiguous memory regions. As a result, the extra computations along *i*-dimension are fully avoided.

This choice also creates opportunities for reducing the extra computations along the other two dimensions. Here the key point is selecting an appropriate shape of MPDATA blocks. To minimize the total amount of extra computations, we propose to define the size nB as small as possible (minimum $nB = 1$), and set the other two sizes mB and lB as large as possible to keep all necessary data in cache. It is worth noticing that extra computations can be avoided at all if $mB = m$, $lB = l$, and the cache capacity of a given device is large enough to keep all required data for a given size nB. Otherwise, the MPDATA block is split into $q = 2, 3, 4, \dots$ smaller blocks of size $nB \times \frac{mB}{q} \times lB$ until each block is small enough to keep all data in cache. Naturally, in this case it is not possible to avoid extra computations, and their amount grows up with increasing q.

5.2. Islands-of-cores strategy: Partitioning cores into independent work teams

In our previous article (Szustak et al., 2017), we have taken up the challenge of harnessing the heterogeneous nature of SMP/NUMA communications for the MPDATA application. We proposed the *islands-of-cores strategy* that exposes the correlation between computation and communication for stencils, and enables the efficient management of trade-off between computation and communication costs, in accordance with features of SMP/NUMA systems. In this article, we propose to adopt this strategy for shared-memory platforms that reveal physical groups of cores as it is in ccNUMA and KNL architectures.

The idea of the islands-of-cores strategy is illustrated in Figure 5. An example of 1D stencil computations with two kernels K1 and K2 is shown in Figure 5(a), while Figure 5(b) and (c) presents two general scenarios for the

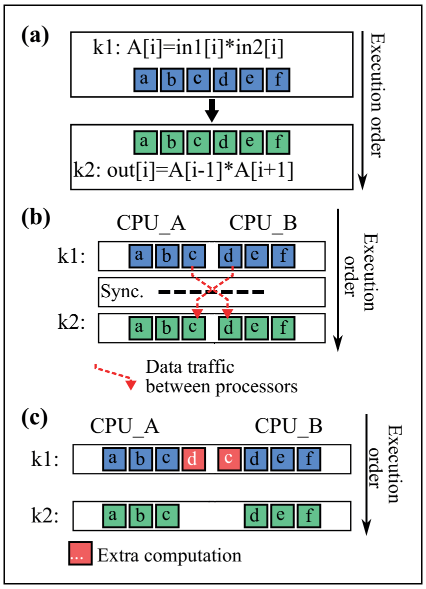

Figure 5. Idea of islands-of-cores strategy: (a) example of stencil computations with two kernels; (b) parallelization with implicit data transfers between processors; and (c) avoiding data transfers and synchronization, at the cost of extra computations. *Source*: Szustak et al. (2017). MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

parallelization of computations using two interconnected processors.

The first scenario (Figure 5(b)) illustrates the interprocessor communication due to data dependencies. These implicit data transfers are implemented through the cache hierarchy; they take place on borders of computation subdomains distributed across processors. In particular, the output element *c* computed in CPU<u>A</u> by the kernel K1 is transferred to CPU<u>B</u> in order to compute element *d* in the kernel K2. At the same time, the element *d* obtained in CPU<u>B</u> by the first kernel is copied to CPU<u>A</u> to compute element *c* by the second kernel. Furthermore, the synchronization point is required to ensure the correctness of parallel computations. When using a single processor, this traffic is restricted to the cache hierarchy of this CPU. However, in the case of the multiprocessor server, the required data are implicitly transferred between caches of neighbor processors through QPI or UPI links (Intel, 2018).

The second scenario (Figure 5(c)) permits us to avoid exchanging data between processors at the cost of extra computations. Instead of transferring the elements *c* and *d* computed by the first kernel, let both processors compute them once more. In consequence, CPU<u>A</u> computes one extra element *d*, and CPU<u>B</u> delivers the additional element *c* within the kernel K1. As a result, like independent islands, both processors are able to perform computations independently of each other within every time step, at the cost of computing some amount of extra elements.

To sum up, the first scenario performs less computations but requires more data traffic, while the second one allows us to replace the implicit data traffic between processors by replicating some computations. Thus, the second scenario seems to fit perfectly to reduce inter-processor communications between caches in a ccNUMA shared-memory system. At the same time, the first scenario is well suited to be implemented inside each processor, where a more efficient, local memory hierarchy is used to implement the data traffic between cores.

Therefore, the key point is to customize the islands-of-cores strategy to the MPDATA application, which features a definitely more complex computational structure than the example shown in Figure 5. To implement this strategy for ccNUMA architectures, the abstraction of the MPDATA islands of cores is applied across *P* interconnected processors of a given platform. As a result, the MPDATA domain is partitioned into *P* subdomains that have to be mapped onto *P* processors. Each processor corresponds now to a work team of cores (MPDATA island), and these work teams perform the following activities in each time step:

1. All work teams share input data, utilizing the first-touch policy with parallel initialization.
2. At the cost of extra computations, each team independently executes the set of blocks within its subdomain, following the (3+1)D decomposition.
3. After completing the whole time step, each team returns outcomes to the main memory. Additionally, all teams synchronize their operations, in order to ensure the correctness of input data for the next time step.

Assuming that every team consists of the same number of cores, the MPDATA domain is decomposed into uniform subdomains of size $nP \times mP \times lP$, where the total number of subdomains is equal to the number of physical teams of cores. Only 1D variants of partitioning the MPDATA domain are considered. The reason for avoiding 2D and 3D partitionings is that data layouts of all the MPDATA arrays allow performing transfers of contiguous areas of memory only along the first dimension. In consequence, we expect too high communication overheads when the MPDATA domain is partitioned across two or three dimensions.

Each work team is able to perform computations independently of other teams at the cost of extra computation. In consequence, as shown in Figure 5(c), every MPDATA subdomain has to be extended by ghost regions. More specifically, each kernel is extended by ghost regions of which the sizes are determined individually based on data dependencies of kernels. Since the MPDATA kernels feature heterogeneous stencil patterns, the final sizes of ghost regions of various kernels differ from each other. Our previous articles (Szustak et al., 2014b; Wyrzykowski et al., 2014) discussed how dependencies between MPDATA kernels affect both the sizes of ghost regions and final amount of extra elements. As an example, Table 2 presents how the total number of extra elements increases with the

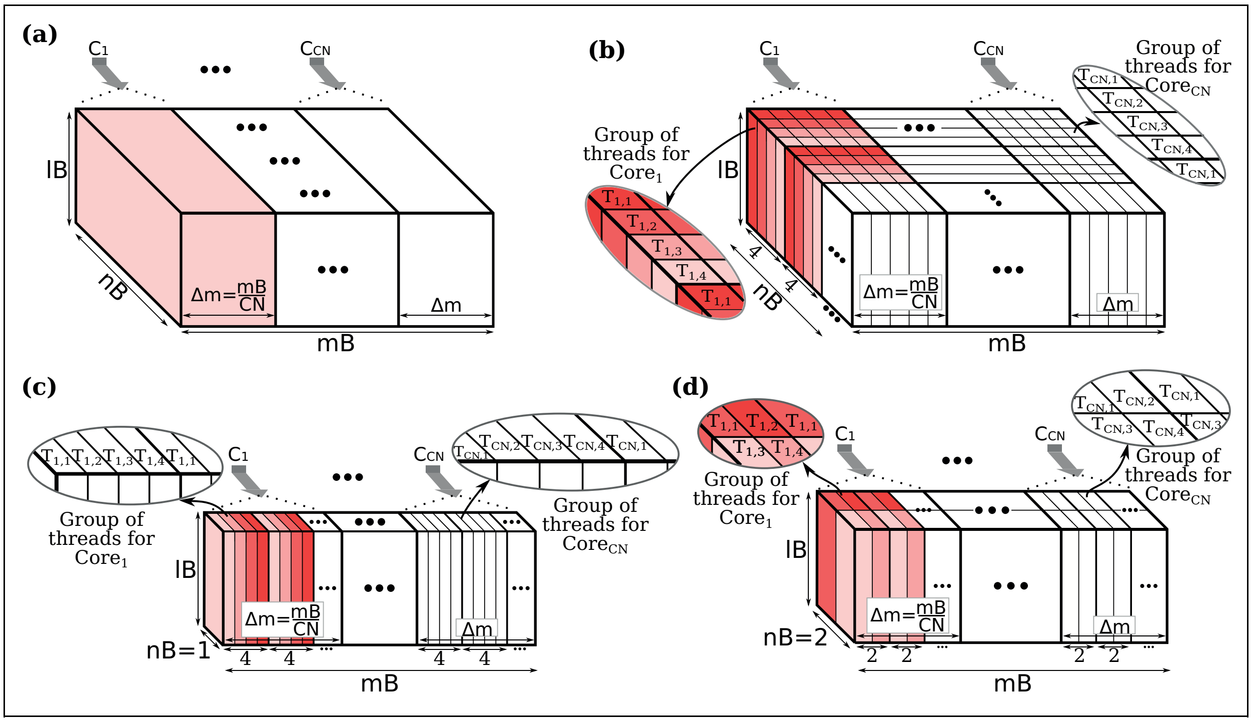
(a)
C1
CCN
lB
nB
Δm=mB/CN
Δm
mB
(b)
Group of threads for Core1
Group of threads for CoreCN
T1,1 T1,2 T1,3 T1,4 T1,1
TCN,1 TCN,2 TCN,3 TCN,4 TCN,1
4
(c)
nB=1
(d)
nB=2
2

Figure 6. Data parallelism for MPDATA: (a) distributing MPDATA block of size $nB \times mB \times lB$ across CN cores; strategy for grouping threads into work groups for $S = 4$ with $nB \geq 4$ (b), $nB = 1$ (c), and $nB = 2$ (d). MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

number of subdomains for the 1D partitioning of domain of size $1024 \times 512 \times 64$.

5.3. Data parallelism and grouping threads into work groups

Assuming the combination of the (3+1)D decomposition and islands-of-cores strategy, the MPDATA domain is decomposed into uniform subdomains (islands), then each subdomain is partitioned into a set of blocks following the (3+1)D decomposition, and finally this set is executed by all the physical cores assigned to the corresponding MPDATA island. As a result, every MPDATA block of size $nB \times mB \times lB$ is implemented in parallel.

The important point here is to distribute computations of each MPDATA block across both cores/threads and vector units, in portable and efficient way. To reach this aim, we propose to: (i) exploit data parallelism across *i*- and *j*-dimensions, based on distributing data across available cores/threads; (ii) incorporating the vectorization (see the next subsection) along *k*-dimension.

Since the typical shape of blocks assumes that $nB << mB$ with a relatively small value of nB (even $nB = 1$), the MPDATA block is partitioned evenly into a set of subblocks of size $nB \times \frac{mB}{CN} \times lB$, where CN is the number of physical cores assigned to a given work team (MPDATA island). Each core of a given team executes computations related to all MPDATA kernels performed on chunks of corresponding data arrays. The data parallelism corresponds to distributing data across cores of a team along *j*-dimension. Since the data traffic takes place only between adjacent MPDATA subblocks, the adjacent subblocks are mapped onto cores that are physically closely connected to each other, in order to reduce the communication paths. The scheme of the proposed distribution is presented in Figure 6(a).

The goal of this work is to minimize data transfers not only between cores, but also inside a single core, by exploring the possibilities of running multiple threads within a single core using SMT. Therefore, we propose to group multiple threads into work groups in a way that permits executing a single MPDATA subblock in a compact manner. The main rationale is to increase reusing of data processed by a group of $S > 1$ threads.

We distinguish three scenarios for mapping SMT threads onto MPDATA subblocks; they are illustrated in Figure 6(b) to (d). In the first scenario, each MPDATA subblock is partitioned into a set of parts of size $S \times 1 \times l$, and then S threads are parallelized inside every part. This scenario is directly applied in the case of $nB \geq S$, as it is shown in Figure 6(b), where $S = 4$. However, this scenario is not applicable if $nB = 1$. In this case, a single MPDATA subblock is decomposed into parts of size $1 \times S \times l$ that are parallelized using a work group of S threads. An example of this scenario is presented in Figure 6(c). The last scenario is designed for $nB > 1$, when $nB < S$. Figure 6(d) shows this scenario for $nB = 2$, and $S = 4$. In this example, every subblock is split into parts of size $2 \times 2 \times l$, which are executed in parallel by four threads of a given work group.

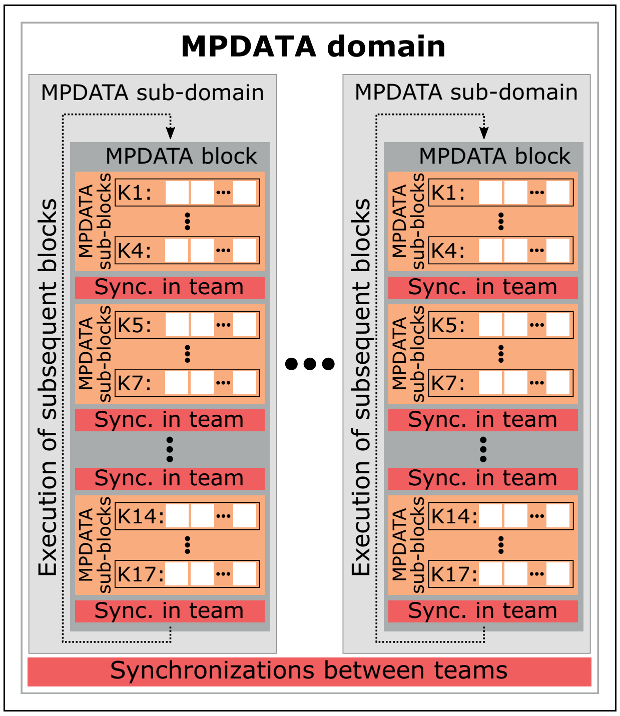

Figure 7. Synchronization scheme for a single time step of MPDATA. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

In general, the last two scenarios can be used instead of the first one. However, the stencil nature of MPDATA kernels, as well as dependencies between them, make the first scenario the best choice because of the highest rate of data reusing. The proposed approach allows us to define any number of threads per a single group, but this number is constrained by the number of logical threads available for a given platform.

Using the OpenMP standard (OpenMP, 2015) to implement MPDATA gives programmers a simple and rather flexible interface for developing portable parallel applications. In particular, OpenMP offers a useful interface to control the thread affinity (OpenMP, 2015). However, the complexity of the proposed adaptation makes it impossible to implement efficiently the multithreading parallelization of MPDATA using the OpenMP constructs such as #pragma omp for. So instead, we develop a proprietary scheduler responsible for the workload distribution and data parallelism. Following the proposed adaptation, our scheduler explicitly define the scope of work for each thread.

5.4. New strategy for synchronization

Because of data dependencies between the MPDATA kernels, the parallelization of an MPDATA block requires providing the synchronization of threads. The general scheme of synchronizations for MPDATA computations is presented in Figure 7. In particular, four synchronization points are necessary within each MPDATA block, to ensure the correctness of executing subsequent kernels. Moreover, following the three optimization techniques already presented in this section, each work team executes independent computations within every time step, but all teams have to synchronize their works after each time step.

Therefore, two synchronization levels have to be considered: inside every team (first level) and between all teams (second level). Within a certain time step, the amount of synchronization points depends on the number of MPDATA blocks. In consequence, a huge number of synchronization points is expected for the first level. Its efficiency becomes critical for the overall performance. For this reason, in our previous work (Szustak, 2018) we proposed a novel strategy for the data flow synchronization in shared-memory systems. The main idea of this strategy is to synchronize only interdependent threads instead of using the barrier approach that—in contrast to the proposed strategy—synchronize all threads (see also Laccetti et al., 2016). An inherent part of this strategy is revealing the scheme of thread interrelationships for a given application.

Implementing this strategy for the MPDATA application needs to determine the inter-thread traffic, by considering the dependencies between parallel computations inside every MPDATA block. It can be concluded that the inter-thread data traffic occurs only between threads pinned to adjacent physical cores. In fact, threads assigned to a given physical core C_i depend only on outcomes computed by threads pinned to the core C_{i-1} on the left side, as well as to threads assigned to the core C_{i+1} on the right side (Figure 6). At the same time, threads assigned to the cores C_{i-1} and C_{i+1} do not depend on each other.

Unlike the barrier approach, we propose to synchronize only interdependent cores/threads. To reach this aim, we define $(CN - 1)$ synchronization groups of threads that correspond to the subsequent pairs of adjacent cores, where a certain i th group includes all threads assigned to the cores C_{i-1} and C_i, while the next $(i + 1)$ th group encompasses threads pinned to the cores C_i and C_{i+1}. As a result, all threads assigned to a core are now affiliated to two synchronization groups on its left and right sides. The exceptions to this rule are the first and last cores that are affiliated only to a single group. The total number of threads per synchronization group is equal to $2 \times S$. This strategy allows us to reduce the cost of synchronization since each group includes radically less cores/threads to synchronize than the barrier approach. Figure 8 illustrates interrelationships between cores/threads in the data flow synchronization strategy used for MPDATA. Based on this strategy, a data flow synchronization algorithm was also developed (Szustak, 2018), and successfully adopted to the application.

5.5. Vectorization

The next goal of this work is to ensure the performance portability for vectorization of MPDATA computations. In general, there exists a wide range of alternative methods and tools for implementing vectorization (Eltablawy and

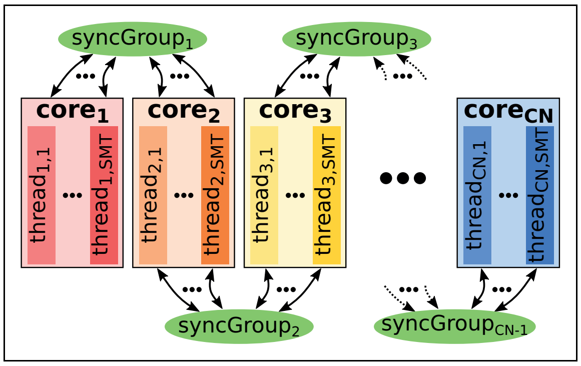

Figure 8. Data flow synchronization strategy for MPDATA stencil computations. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

Vladimirov, 2015; Jeffers et al., 2016), that differ in terms of complexity, flexibility, and portability. Two main approaches to take advantages of vector registers and vectorization benefits can be distinguished (Eltablawy and Vladimirov, 2015): (i) manual vectorization using assembly and/or intrinsic functions and (ii) automatic vectorization by the compiler.

Although the first method is more controllable, it lacks the portability, which in fact is the main goal of this work. In contrast, the automatic vectorization provides a strong basis to ensure compatibility of the code with various architectures (Jeffers et al., 2016). The compiler detects operations in the program that can be executed in parallel, and then converts sequences of operations into parallel vector operations. Furthermore, if the compiler is able to vectorize a code for an older vector instruction set, only a recompilation is necessary to generate an executable program for a new processor architecture (Eltablawy and Vladimirov, 2015). As a result, the compiler-based automatic vectorization seems to be the most convenient method for this work.

The challenge is to specify the process of vectorization entirely and correctly, so that a compiler will make vectorization automatically. To reach this aim, we use OpenMP SIMD directives. The main advantages of using these directives is the ability for code portability between not only different microarchitectures but also compilers (OpenMP, 2015). OpenMP SIMD directives can force the compiler to generate the vectorized code, but it is the programmer's responsibility to ensure that there are no limitations which might impact the correctness of computations.

Furthermore, the compilation commands have to be properly adjusted for a given microarchitecture to support the full use of vector hardware. For example, for Intel compilers 18.x the automatic vectorization is enabled at the default optimization level -O2, but two more compiler options `-xCore-AVX512` and `-qopt-zmm-usage=high` are necessary for generating codes for the newest Intel Xeon Scalable Performance (SKL-SP) CPUs (Eltablawy and Vladimirov, 2015).

To make the automatic vectorization more efficient, the Intel Advisor tool (Jeffers et al., 2016) is used to help MPDATA reach the full performance potential of modern processors. This tool allows us to find what is blocking vectorization for various computing platforms. Additionally, as the compiler does not make all the work for the automatic vectorization (Jeffers et al., 2016), in this article we propose how to overcome all issues that affect the effective vectorization of MPDATA. In practice, the innermost loop is commonly recommended as a target for vectorization (Jeffers et al., 2016; Vladimirov et al., 2015). The efficiency of vectorization relies on the ability to load multiple array elements into wide vector register, but these elements have to reside in memory contiguously to optimize the performance of computations.

For MPDATA, the data layout used for storing arrays compels us to perform vectorization along *k*-dimension of MPDATA blocks, to provide the contiguity of memory accesses. As a result, vectorization corresponds indeed to the innermost loops of MPDATA kernels. In addition, these loops are manually marked up with `#pragma omp simd`, in order to force the compiler to vectorize the correct loop of kernels. Furthermore, if necessary, we align the range of loop index (value of parameter *l*) for each innermost loop to be divisible by the vector length, in order to avoid the not vectorized, remainder part of loops. For a given execution, this range is defined as a constant variable to inform the compiler when a loop will exit to safely push multiple iterations for executing by the vector processing unit.

One of the key methods to aid the efficient vectorization is to perform data alignment and assist the compiler to recognize aligned data (Eltablawy and Vladimirov, 2015; Jeffers et al., 2016). We ensure alignment using the dedicated memory allocation functions (see Jeffers et al., 2016), and telling the compiler that data are aligned through `#pragma omp simd aligned(list[: linear-step])`. As a result, the innermost loops of MPDATA kernels will be able to start operating on data addresses aligned to specific byte boundaries. However, codes of some kernels will still include accesses to both aligned and unaligned data that are produced as an effect of irregularities in the stencil patterns.

In general, there are a lot of loop features that potentially can reduce the efficiency of vectorized code, or even prevent vectorization (Jeffers et al., 2016). This leads to some restrictions on the types of loops that can be vectorized efficiently. An important example are loops with the read-after-write dependency (see Listing 1) that appears if a variable is written in one iteration and read in a subsequent one. In order to solve this issue, which occurs for two MPDATA kernels, we propose to slightly modify the code by using the loop splitting optimization technique. It allows us to eliminate the read-after-write loop dependencies by breaking the original loop into two loops which iterate over an additional temporal data object. Listing 2 shows the result of using this technique for the code shown in Listing 1.

Another example of such restrictions are loops containing conditional statements with if construction. Few MPDATA kernels include loops of this type as a result of MPDATA boundary conditions. These loops include one or two conditional statements of the following type:

- `int km1 = ((k==first)? k: k-1);`
- `int kp1 = ((k==last)? k: k+1);`

To overcome this limitation, we add properly selected paddings for the corresponding data arrays, and insert suitable data to them, just before executing a given kernel, as illustrated in Listing 3.

The compiler cannot determine, without help, whether there may be dependencies between loop iterations when using pointers in C/C++ (Eltablawy and Vladimirov, 2015; Jeffers et al., 2016). As pointers are used in the MPDATA code when accessing data arrays, the special hints (e.g. *#pragma ivdep*) and keywords (e.g. *restrict*) are used to avoid pointers aliasing issues. As a result, we help the compiler assist successfully in vectorization of MPDATA loops considering all the tested platforms.

Listing 1. Loop with read-after-write

```
A[0]=0.0;
#pragma omp simd
for(unsigned int k=1;k<l;++k) {
  A[k] = A[k-1] - (B[k] + C[k]);
}
```

Listitng 2. Result of applying the loop splitting optimization technique to eliminate the read-after-write dependency

```
#pragma omp simd aligned(tmp,B,C)
for(unsigned int k=0;k<l;++k) {
  tmp[k] = (B[k] + C[k]);
}
#pragma omp simd aligned(A,tmp)
for(unsigned int k=0;k<l;++k) {
   const int km1 = ((k==0)?k:k-1);
  A[k] =  tmp[km1] -  tmp[k];
}
```

Listing 3. Idea of avoiding the `if` statement in the boundary conditions inside the vectorized loop,

```
B[-1] = B[0];
C[l] = C[l-1];
#pragma omp simd aligned(A,B,C)
for(unsigned int k=0;k<l;++k) {
   const int km1 = k-1;
   const int kp1 = k+1;
  A[k]=B[k-1] + C[k+1];
}
```

6. Parametric customization of MPDATA code

In order to ensure the performance portability for the new code of MPDATA, we propose a flexible tuning of the parameterized transformation of the code. The main assumption is to calibrate the proposed parametric optimization techniques for a given computing platform, before the execution of a specific numerical simulation. For this aim, a four-step customization procedure is developed, and then implemented as an external mechanism—not included in the MPDATA code—that produces the customized configuration of the MPDATA parallel code.

In this way, input parameters that characterize a given computing platform are transformed to generate an optimal configuration for each parametric optimization technique. The resulting configuration returns compile-time constants for the MPDATA code, and thus not cause a run-time degradation of performance. Furthermore, once generated the configuration can be successfully used repeatedly.

6.1. Parameterization of computing platforms

The proposed tuning requires parameterization of characteristics of different computing platforms. In consequence, for a given platform, the following set of parameters is determined:

1. total number TCN^P of physical cores per platform;
2. number S^P of SMT threads per each core;
3. SIMD width VS^P (in bits);
4. number TN^P of physical teams of cores;
5. number CN^P of cores per every physical team;
6. total effective cache size TCS^P; and
7. effective cache size CS^P for each physical team.

Values of all these parameters have to be carefully selected. The information about the first three parameters is commonly available (see https://ark.intel.com), but determining adequate values of others requires a deep knowledge about a given architecture.

For ccNUMA-based platforms, the number TN^P of physical teams of cores is determined first of all by the number of available processors. At the same time, CPUs in the considered platforms contain two memory controllers per processor, which may affect the main memory latency differently for different cores. For this reason, we define other two physical groups of cores inside every processor. As a result, the final value of TN^P is twice the number of available processors. Additionally, since a single KNL processor includes four physical groups of cores evenly placed on a mesh inside a chip, and connected to four memory controllers of MCDRAM (Jeffers et al., 2016), the value of TN^P should be defined as 4. The considered platforms have an equal number of cores per every physical team (Intel, 2018; Jeffers et al., 2016). As a result, the number CN^P of cores per every physical team is defined as $CN^P = TCN^P/TM^P$.

For BDW and HSW CPUs, the total effective cache size TCS^P can be successfully fixed as the size of the last level cache (LLC) (Intel, 2018), which in fact corresponds to the size of L3 cache. For KNL processors, the size of LLC is calculated as the aggregate size of all L2 caches located in all KNL tiles (Jeffers et al., 2016). On the contrary, since

Table 3. Parameterization of computing platforms.

	Platforms:	A	B	C	D	E
Parameters:	TCN^P	112	40	36	24	68
	S^P	2	2	2	1	4
	VS^P [bit]	512	512	256	256	512
	TN^P	8	4	4	4	4
	CN^P	14	10	9	7	17
	TCS^P [MB]	112	40	90	70	34
	CS^P [MB]	14	10	22.5	17.5	8.5

SKL-SP processors feature the non-inclusive L3 cache (Intel, 2018), the value of TCS^P in this case corresponds rather to the aggregate size of L2 caches of all cores than to the total size of L3 cache. The reason is that instead of copying data both to the L2 and L3 caches as in the case of previous generations of CPUs, now data are loaded directly into the L2 cache of a given core. Based on the size TCS^P, we fix the effective size CS^P of caches utilized jointly by all cores of a given physical team. Since all physical teams in the considered platforms include equal numbers of cores, the last parameter is determined as $CS^P = TCS^P/TN^P$. The results of parameterization of the tested platforms are shown in Table 3.

6.2. Four-step procedure for MPDATA code customization

Figure 9 presents the general scheme of the four-step procedure developed for the parametric customization of the MPDATA code. Its first step is responsible for calibrating the islands-of-cores strategy. Initially, two parameters are determined: (i) the total number NT of MPDATA work teams (islands) of cores and (ii) numbers CN_i of cores per every i th work team, $i = 0, 1, \dots \dots NT - 1$. For a given platform, TN is equal to the number TN^P of physical teams of cores. Since all physical teams of any considered platform include the same number of cores, we are entitled to fix CN_i as equal to the parameter CN^P already determined for the computing platform. Also, we have to indicate the physical cores that are associated with the physical teams, in order to correctly assigned them to the MPDATA work teams.

Afterward, the number and size of MPDATA subdomains are determined following the islands-of-cores strategy. For a given MPDATA simulation, the size of MPDATA domain is specified as $n \times m \times l$. This domain is decomposed into equal subdomains of size $nP \times mP \times lP$ using the 1D scheme of partitioning with the total number of subdomains equal to the number NT of MPDATA work teams. Therefore, each subdomain will be of size $nP = \frac{n}{nT}$, $mP = m$, $lP = l$. Finally, we assign the work teams of cores to the subsequent MPDATA subdomains to enable parallel execution of subdomains by different work teams.

Every MPDATA subdomain has to be further partitioned according to the (3+1)D decomposition. The main constraint is to keep all necessary data in the cache associated with a given work team. This is the goal of step 2, which carefully selects the number, size, and optimal shape of MPDATA blocks for each MPDATA subdomain, considering the restrictions of the (3+1)D decomposition. Also, this step customizes the size and shape of the ghost regions for all MPDATA kernels that effectively differ from each other.

The three parameters nB, mB, and lB define the size of MPDATA blocks and further impose the optimal shape of ghost regions. Following the previous section, the parameter nB is set as small as possible (minimum $nB = 1$), while the other two parameters mB and lB are fixed as large as possible to keep all necessary data in the cache available for a given work team, where the cache size is limited by the parameter CS^P already determined for the platform. Considering constraints of the (3+1)D decomposition, the optimal configuration that allows avoiding extra computations is fixed as $mB = mP$, $lB = lP$ if the cache capacity for a given work team is large enough to keep all necessary data for a given size nB. Otherwise, we iterate over $mB = \frac{mP}{q}$, with $q = 2, 3, 4, \dots$, until the size of block is small enough to keep all data in cache. The procedure for determining the optimal size of block is summarized in Algorithm 1, which is dedicated for the numerical simulation of weather prediction, where typically $lP = l \in [64, 128]$.

Algorithm 1. Determining the optimal size nB × mB × lB of MPDATA block

```
$nB = 1$
$mB = mP$
$lB = lP$
$q = 1$
while $getCacheUsage(nB, mB, lB) < CS^P$ do
    $q = q + 1$
    $mB = \frac{mP}{q}$
end while
$mB = \frac{mP}{q-1}$
while $getCacheUsage(nB, mB, lB) < CS^P$ do
    $nB = nB + 1$
end while
$nB = nB - 1$
```

The combination of the first two steps specifies the scheme of MPDATA decomposition (see also Figure 7). While MPDATA subdomains are executed in parallel by various work teams, every subdomain is decomposed into blocks of size $nB \times mB \times lB$ that are further executed one by one. Finally, each block is processed by all physical cores assigned to a corresponding work team. The main goal of step 3 is to customize the data parallelism in each MPDATA block, as well as grouping threads into work groups.

Firstly, we determine the partitioning of every MPDATA block into a set of CN subblocks (see Figure 6(a)), each of size $nB \times \frac{mB}{CN} \times lB$. In consequence, each subblock will be processed by one of CN work groups of

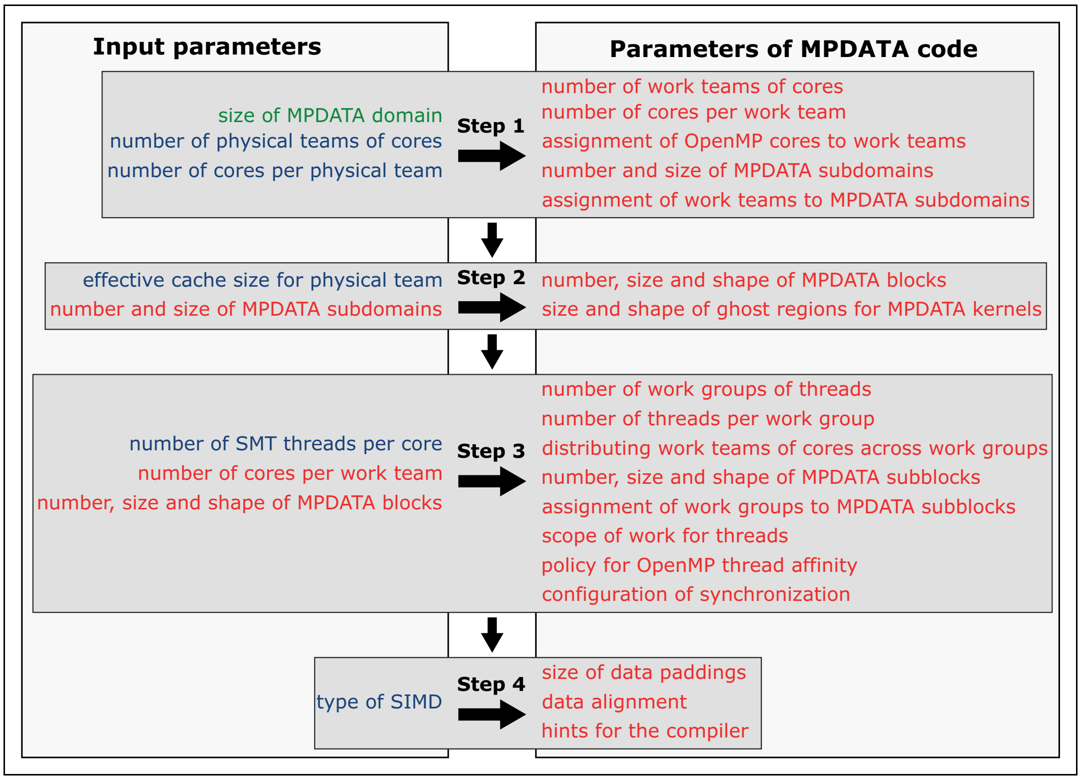
Input parameters
Parameters of MPDATA code
size of MPDATA domain
number of physical teams of cores
number of cores per physical team
Step 1
number of work teams of cores
number of cores per work team
assignment of OpenMP cores to work teams
number and size of MPDATA subdomains
assignment of work teams to MPDATA subdomains
effective cache size for physical team
number and size of MPDATA subdomains
Step 2
number, size and shape of MPDATA blocks
size and shape of ghost regions for MPDATA kernels
number of SMT threads per core
number of cores per work team
number, size and shape of MPDATA blocks
Step 3
number of work groups of threads
number of threads per work group
distributing work teams of cores across work groups
number, size and shape of MPDATA subblocks
assignment of work groups to MPDATA subblocks
scope of work for threads
policy for OpenMP thread affinity
configuration of synchronization
type of SIMD
Step 4
size of data paddings
data alignment
hints for the compiler

Figure 9. Four-step procedure for MPDATA code customization. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

threads, each with S threads, where S is equal to the number S^P of SMT threads per core. Additionally, the cores of a given work team are distributed across work groups, and successive work groups with S threads are assigned to subsequent MPDATA subblocks.

Secondly, if $S > 1$, the optimal way for parallelization of threads inside every subblock is selected. In particular, we select one of the following scenarios for mapping threads of a work group onto a given subblock:

1. In the first scenario, each MPDATA subblock is partitioned into a set of parts of size $S \times 1 \times lB$, and then each part is processed in parallel using a work group with S threads. This scenario is used in the case of $nB \geq S$ (see Figure 6(b)).
2. The second scenario is applicable if $nB = 1$. In this case, a single MPDATA subblock is decomposed into parts of size $1 \times S \times lB$ (Figure 6(c)).
3. The last scenario is based on a combination of the previous scenarios. It is used for $nB > 1$ with $nB < S$. Now every subblock is split into parts of size $S_a \times S_b \times lB$, where $S_a \times S_b = S$ (Figure 6(d)).

Afterward, we explicitly define the scope of work for all threads of every work group, and select the correct policy for OpenMP thread affinity. Also, step 3 allows customizing the way how the proposed data flow synchronization is implemented.

The last step is responsible for ensuring the performance portability for the vectorization of MPDATA computations. In this step, based on the knowledge about SIMD hardware, including the SIMD width VS^P, the size of data padding is selected, as well as details of data alignment for MPDATA arrays and hints for the compiler.

7. Performance evaluation

The proposed parameterized adaptation of the MPDATA application to shared-memory systems is evaluated for five servers with HSW, BDW, SKL-SP, and KNL processors (see Table 1). In these benchmarks, the Intel icpc compiler (v.18.0.1) is used with the optimization flag -O3 and properly chosen compiler arguments that support the full use of SIMD hardware. The Intel Turbo Boost technology is enabled in all servers. The KNL platform is configured in the quadrant clustering mode with the MCDRAM memory used in the flat mode (Jeffers et al., 2016).

In this work, we focus on using MPDATA for the numerical weather prediction, where the double precision floating point format is essential, and the size of grids is typically limited (Lastovetsky et al., 2017; Szustak et al., 2015) by $n \leq 2048$, $m \leq 1024$, $l = 64$. The performance results are obtained for 5000 time steps. In order to guarantee the reliability of benchmark results, the measurements of execution time are repeated several times, and the median value of measurements is used finally.

Table 4. Performance results of MPDATA obtained for various computing platforms, with the domain of size 2048 × 1024 × 64.

Platform	Execution time [s] Basic code	Execution time [s] Optimized code	Speedup	Perfor. [Glops/s]	% of peak	Team config.	Size of block	#SMT groups
A 4 × SKL-SP	1472.5	151.3	9.73	1048.1	34.4	8 × 1	2 × 341 × 64	2 × 1
B 2 × SKL-SP	2948.9	352.6	8.36	447.9	43.7	4 × 1	1 × 341 × 64	1 × 2
C 2 × BDW	4382.9	598.6	7.32	263.3	45.7	4 × 1	1 × 512 × 64	1 × 2
D 2 × HSW	4714.1	777.7	6.06	203.1	41.3	4 × 1	2 × 341 × 64	1 × 1
E 1 × KNL	5357.3	609.5	8.78	261.9	20.1	4 × 1	4 × 128 × 64	4 × 1

SKL-SP: Skylake-SP; BDW: Broadwell; HSW: Haswell; KNL: Knights Landing; SMT: simultaneous multithreading; MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

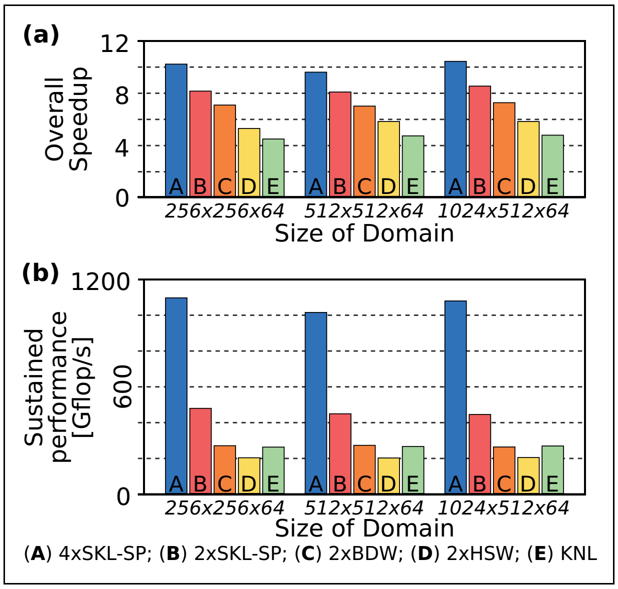

Figure 10. Performance results of MPDATA achieved for different problem sizes and various platforms: (a) overall speedup of the optimized version of MPDATA against the basic version and (b) sustained performance. SKL-SP: Skylake-SP; BDW: Broadwell; HSW: Haswell; KNL: Knights Landing; MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

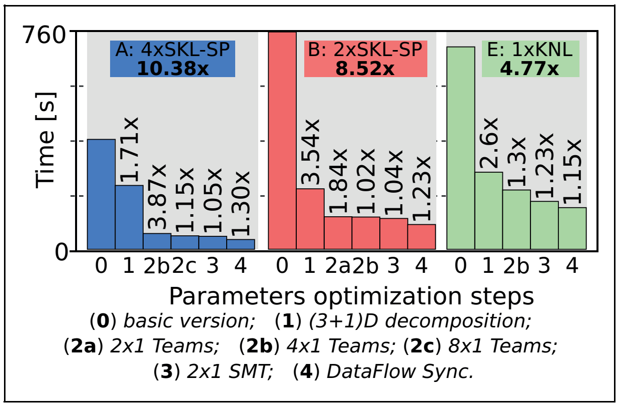

Figure 11. Comparison of the impact of optimization techniques on the performance gain achieved for various platforms, with MPDATA domain of size 1024 × 512 × 64. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm; SMT: simultaneous multithreading.

Table 4 and Figure 10 show the performance results of MPDATA achieved for different problem sizes and various computing platforms. Besides the execution time, the overall speedup of the optimized code against the original version of MPDATA is shown. Also, the sustained performance (in Gflop/s) is included for the new code, as well as the utilization rate in comparison with the theoretical peak performance of the servers. Finally, we present details of MPDATA configurations returned by the proposed four-step customization procedure.

The main conclusion is that the proposed methodology allows us to improve radically the efficiency of MPDATA computations in comparison with the basic version of code. As shown in Figure 10(a), the proposed parameterized adaptation accelerates the MPDATA application about 10×, 8×, 7×, 6×, and 9× for 4 × SKL-SP, 2 × SKL-SP, 2 × BDW, 2 × HSW, and 1 × KNL platforms, respectively. What is important, for all platforms and different problem sizes, we try to find empirically better configurations of the MPDATA code than returned by the four-step procedure, but without success.

For a given platform, the sustained performance of the new MPDATA code is kept at a similar level, independently of the problem size (see Figure 10(b)). As expected, the most powerful computing platform is the four-socket server with SKL-SP CPUs. This platform allows us to execute computations with the sustained performance of around 1.0–1.1 Tflop/s (33% of peak) for all tests. On the other hand, the two-socket servers with SKL-SP, BDW, and HSW processors feature the highest utilization rate of the theoretical peak performance (about 41–46%).

Remark: For the KNL platform, there is a seeming discrepancy between Table 4 where the new code achieves the speedup of about nine times against the basic code, and Figure 10(a) where we have speedups of around five times. The reason for this effect is a sharp fall in the performance of the basic code for large domains. At the same time, the new version allows keeping the sustained performance at a stable level of around 260 Gflop/s, that gives about 20% of the peak.

Figure 11 presents the comparison of impact of various optimization techniques on the achieved performance gain for different platforms. It can be concluded that in all cases

Table 5. Execution times for scalar and vectorized versions of new MPDATA code, as well as speedup of vectorization achieved both with and without the use of Turbo Boost (domain of size 1024 × 512 × 64).

Platform	A: 4 × SKL-SP		B: 2 × SKL-SP		C: 2 × BDW		D: 2 × HSW		E: 1 × KNL	
Intel Turbo mode	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF
Time [s]										
Scalar	102.0	116.0	244.5	285.0	315.2	372.5	437.9	—	586.9	634.6
SIMD	37.1	38.4	89.0	90.0	149.9	153.1	192.9	—	147.9	157.4
Speedup	2.75	3.02	2.75	3.20	2.10	2.43	2.27	—	3.97	4.03

SKL-SP: Skylake-SP; BDW: Broadwell; HSW: Haswell; KNL: Knights Landing; MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

the MPDATA performance is significantly increased by using the (3+1)D decomposition. It allows us to alleviate the memory and communication constraints by increasing both cache reusing and the data locality. As a result, the speedup in the range from about 1.7× to about 3.5× is achieved against the basic code.

The next technique, islands-of-core strategy, fits perfectly into ccNUMA architectures. As shown in Figure 11, a significant performance gain is noticed for the four-socket server, that using the configurations with 4 × 1 teams performs computation 3.87× faster against the pure (3+1)D decomposition. At the same time, the two-socket server allows accelerating MPDATA 1.84× (configuration with 2 × 1 teams). Utilizing the islands-of-core strategy inside each processor gives also some performance gain, however, much smaller. In this case, the acceleration reaches 1.15× for the four-socket server (configurations with 8 × 1 teams), and only 1.02× for the two-socket platform (configuration with 4 × 1 teams). On the contrary, the platform with a single KNL processor achieves the performance gain of about 1.3× using four teams per processor.

Then we evaluate grouping threads for executing on a single core. As expected, the highest performance gain is obtained for KNL that supports four threads per core. This processor gives a constant performance gain of about 20% for all tested MPDATA sizes. For the rest of platforms that support two SMT threads per core, the performance profit is modest (up to 5%). Finally, we investigate the advantages of using the data flow synchronization that—in contrast to the barrier approach—synchronize only interdependent threads. As shown in Figure 11, the performance improvements of up to 30% is achieved against the MPDATA code that involves the three techniques already evaluated.

Table 5 presents the execution times for the scalar and vectorized versions of the new MPDATA code achieved with and without the use of the Intel Turbo Boost technology. The speedup of vectorization is included as well, to show the advantages of using the automatic vectorization for various platforms.

The automatic vectorization allows achieving about two to four times better performance against the scalar version for all benchmarks. At the same time, the new code with the disabled automatic vectorization achieves quite a good performance. The key to success in this case is quite large clock frequency of processors (especially with Turbo Boost), which is significantly reduced for workloads heavy in AVX-512 or AVX2 instructions (see Table 1). For example, the base clock frequency of the Intel Xeon Platinum 8180 CPU is 2.5 GHz, but when all active cores are heavily utilized with AVX-512, the processor clocks down to 1.7 GHz (Intel Xeon Processor, 2018). The Intel Turbo Boost technology plays a key role here since it enables increasing the CPU frequency definitely, for non-AVX workloads. This affects the execution time for both the scalar and vectorized version of new MPDATA code.

Nevertheless, the new code still does not utilize the maximum of the theoretical capacity of vectorization. For example, using two SKL-SP CPUs with the disabled Turbo Boost, we achieve the speedup of vectorization of about 3.2×, while theoretically the speedup of vectorization is limited to around 5.3× (see Table 1). The Intel Advisor tool shows us that the first four MPDATA kernels are not able to fully utilize SIMD units, in contrast to other kernels. The tool reveals that kernels 1–4 are able to utilize vector units at 60–70% only, while the other kernels reach the maximum (100%). As expected, the main reason is that cores remain idle as they wait for input data to arrive from the main memory. This effect increases the overall execution time and reduces the SIMD efficiency. Another factor inhibiting the MPDATA performance is the synchronization overhead that can even refer up to 15% of the total execution time (see Szustak, 2018).

The memory-intensive parallel codes can suffer from the memory bandwidth saturation as more threads are used. As an increasing number of threads or processes share the limited resources of bandwidth, the scalability of the new MPDATA code typically decreases for all the tested platforms. Figure 12 shows the parallel efficiency achieved for different number of cores on a single Intel Xeon Platinum CPU. In this case, the highest performance corresponding to about 80% of linear scaling is achieved when utilizing all 28 cores. At the same time, other platforms that use CPUs with less cores provide better parallel efficiency, which is typically greater than 90% of linear scaling when running MPDATA on all available cores of a given processor.

Finally, Table 6 shows the parallel efficiency expressed as percentage of linear scaling achieved on the four-socket

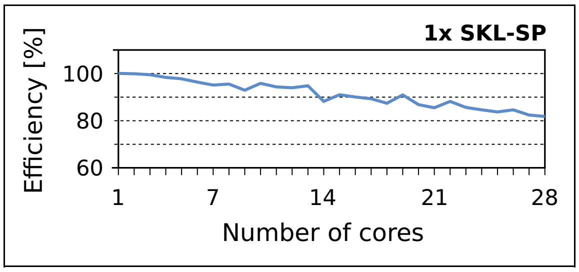

Figure 12. Parallel efficiency for different number of cores (% of linear scaling) achieved with a single Intel Xeon Platinum 8180 CPU and domain size of 1024 × 512 × 64.

Table 6. Parallel efficiency for different number of processors expressed as percentage of linear scaling, achieved with four-socket platform and domain of size 1024 × 512 × 64.

Number of CPUs	1	2	3	4
% of linear scaling	100	97.91	94.06	91.84

server for different number of processors. In this benchmark, the maximum sustained performance of about 1.07 Tflop/s is obtained with four processors, which corresponds to about 92% of linear scaling. Also, a point worth noting is 97.5–99% of linear scaling achieved on the two-socket platforms.

8. Related work

The code portability is an important issue for scientific and commercial environments (Hager and Wellein, 2011; Unat et al., 2014). This challenge was imperceptible for application developers in the prior decades, especially when development of microprocessors was focused on increasing the clock rate (Bobulski, 2016; Czarnul, 2018a). However, over the last decade we have observed major changes in computing systems which are becoming increasingly complex, massively parallel, hierarchical, and also heterogeneous (Czarnul, 2017; Szustak et al., 2016a).

The cost of data movement has turned into the dominant factor in HPC systems (Malik et al., 2016; Rico-Gallego et al., 2017). To reduce this cost, applications have to be redesigned and tuned for data movements both in the memory hierarchy and between processing units (Hager and Wellein, 2011; Wyrzykowski et al., 2012a).

Emerging computing architectures are characterized by a large number of parameters whose diversity makes it difficult to ensure performance portability. The scientific problems aimed to be solved in this work are strongly related with the quick development of current and future computer architectures. While many details of the future architectures are still undefined, an abstract machine model (AMM) outlined in the works of Ang et al. (2014) and Kogge and Shalf (2013) enables researchers to focus on the aspects of a machine that are important or relevant to performance and portable code structure.

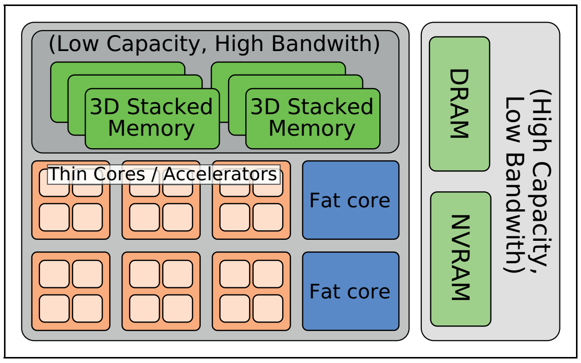
(Low Capacity, High Bandwith)
3D Stacked Memory
3D Stacked Memory
Thin Cores / Accelerators
Fat core
Fat core
DRAM
NVRAM
(High Capacity, Low Bandwith)

Figure 13. AMM for emerging hardware. *Source*: Ang et al. (2014). AMM: abstract machine model.

Figure 13 depicts an AMM that captures the anticipated evolution of existing node architectures based on industry road maps (Ang et al., 2015).

All computing platforms considered in this work fit perfectly into respective features of this AMM architecture. It can be concluded also that the proposed combination of parametric optimization techniques in conjunction with the adaptive customization can be successfully accommodated for shared-memory designs based on AMM.

The efficient porting of applications to novel HPC systems has become a significant challenge (Czarnul, 2018b; Szustak et al., 2016b; Unat et al., 2014). Originally, application codes are often developed without considering advanced architectures and related tool chains. The portable parallel programming is a broad field that comprises a number of different approaches, encompassing data partitioning and distribution, management of data locality and optimization of data movement, workload scheduling, performance/energy modeling, autotuning, and so on.

The stencil-based algorithms have traditionally been optimized by many authors over the years (Henretty et al., 2011; Szustak et al., 2015). One of the main directions of improving the efficiency of stencil computations is focused around different strategies of domain decomposition (Lastovetsky et al., 2017; Zhou et al., 2012), like space and temporal blocking (Bandishti et al., 2013; Datta et al., 2009), diamond and multi-dimensional tiling (Bertolacci et al., 2015; Malas et al., 2016; Strzodka et al., 2011). Similarly to our study, these strategies provide as nearly as possible balanced workload of computing resources, where the main aim of these works is to attempt to better exploit the data locality. However, in contrast to this work, they investigate the homogeneous stencils computations with a single pattern, and typically the code transformations take place between successive time steps. In our work, we develop a combination of parametric optimization techniques within every MPDATA time step that are dedicated for a heterogeneous set of stencils with different patterns.

Furthermore, considering the previously mentioned works, a rather complex interaction between the proposed code transformations and the underlying architecture makes it difficult to find an optimal set of transformations (and variant of code) for any given stencil computations (Bandishti et al., 2013; Bertolacci et al., 2015; Strzodka et al., 2011). Given the complexity of emerging computing systems where small parameter changes may lead to large variations in performance, hand-tuning algorithms have become impractical (Davidson and Owens, 2012). An efficient way to solve this problem and develop self-adaptable applications is the software automatic tuning (autotuning in short), which is a paradigm enabling the software adaptation to a variety of computational conditions (Naono et al., 2011). Autotuning has been successfully applied in a number of widely used HPC algorithms and applications (Falch and Elster, 2015). However, the autotuning method is just a general concept of optimization of computations. Each algorithm requires often a specific analysis and deployment of this concept for a particular class of computer architectures.

The stencil computations often rely on autotuning techniques, based on the iterative compilation or machine learning, to achieve portable high performance (Cosenza et al., 2017). Iterative compilation autotuning is a challenging and time-consuming problem that may be unaffordable in many scenarios. In this case, the autotuning problem is modeled as a classification problem, where each class corresponds to a code variant. These variants expose a large and complex space of equivalent implementations. There are many algorithms such as artificial neural networks or support vector machines (Leather et al., 2009) that are used to select the optimal variant. However, the large number of code variants to execute for a stencil may be too large to be consistently covered in a training phase.

Alternatively, the work of Cosenza et al. (2017) presents an interesting solution to overcome the aforementioned problems with the iterative search and machine learning approaches. The authors present an approach for modeling the performance of stencil computations using the structural learning. They look at the inner structure of the training data to reorganize the stencil code executions by type, input size, and tuning parameters. In this work, an ordinal regression formulation is used in order to compare and rank different stencil variants without executing them.

In contrast to the previously mentioned works, in our study the adaptive code of MPDATA follows along with a variety of the hardware architectural issues such as memory hierarchy, threading, vectorization, and their interaction. In general, the hardware features of a given platform implies the optimal configuration for the MPDATA code transformation. Such a formulation of the MPDATA code is enabled by the parametric customization proposed in this work.

9. Conclusions and future works

Achieving performance portability for real-life scientific applications requires deep knowledge of their interaction with various computing systems. The sustained performance of a given application is affected by such features of computing architectures as number and characteristics of processors, cores, and threads; type of SIMD hardware; parameters of memory hierarchy; relationship between scalar and SIMD frequencies; and many others.

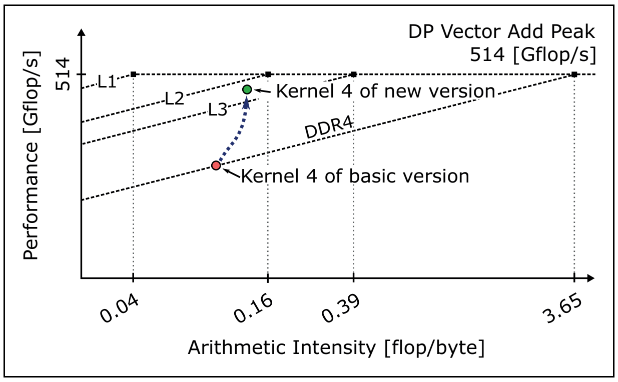

Figure 14. Comparison of Intel Advisor Roofline plots for the fourth kernel of basic and new versions of MPDATA application executed on platform with two Intel Xeon E5-2697v4 CPUs. MPDATA: Multidimensional Positive Definite Advection Transport Algorithm.

Understanding and utilizing full capabilities of modern architectures is quite impossible for programmers without a background in computer architecture. Threading and vectorization are essential functionality of powerful processors. In general, both threading and vectorization are necessary to effectively use all computing resources in applications. Together, they offer definitely more performance than each of techniques alone. Nevertheless, the memory and communication constraints can strongly limit the attainable performance on modern HPC systems.

Apart from using such common optimization techniques as loop tiling/blocking, loop fusion, and others, the application developers need a set of programming abstractions to describe the data locality in new computing ecosystems. Accelerating memory access by arranging data in an appropriate way is vital for achieving the high application performance. Applications with a poor data locality reduce the efficiency of memory hierarchy, causing long waiting times for access to data. A purposeful utilization of various optimization techniques and increasing the data locality play a key role in enabling applications to run on different architectures, while ensuring the performance portability.

In this work, we focus on developing the set of parametric optimization techniques and four-step procedure for customization of the MPDATA code that ensure the performance portability for the MPDATA stencil-based application across a variety of Intel architectures released in the last few years. Among these techniques are islands-of-cores strategy, (3+1)D decomposition, exploiting data parallelism and SMT, data flow synchronization, and vectorization. The proposed adaptation methodology helps us to develop the automatic transformation of the MPDATA code to achieve high sustained scalable performance for all tested servers. This means that for a given platform, the sustained performance of the new code is kept at a similar level, independently of the problem size. The highest performance utilization rate of about 41–46% of the theoretical peak, measured for all benchmarks, is provided for any of the two-socket servers that are based on SKL-SP, BDW, and HSW CPU architectures. At the same time, the four-socket server with SKL-SP processors achieves the highest sustained performance of around 1.0–1.1 Tflop/s that corresponds to about 33% of the peak.

Although the proposed methodology definitely improve the efficiency of MPDATA accelerating its execution up to 10 times, we are still not able to get a full potential of computing systems used in benchmarks. Figure 14 shows a comparison of Intel Advisor Roofline plots for one of the kernels of the basic and new versions of MPDATA. These plots tell us that cores remain idle as they wait for data to arrive in the case of both the basic and new codes. However, the basic version is mainly limited by the DRAM peak bandwidth, while the new MPDATA code allows us to overcome this limitation, but is still finally constrained by the peak bandwidth of L2 cache.

Summarizing, the developed combination of optimization techniques can also help in understanding the evolution of computing architectures, as well as in generalizing our approach for resolving performance portability issues. Our future work will also be focused on developing the MPDATA Benchmark Framework that enables comparing the performance metrics of MPDATA on a wide range of current and future architectures delivered from chip-making vendors such as Intel, IBM, AMD, and ARM.

Authors' Note

This work was partially performed during a STSM scientific visit of Lukasz Szustak at IT4Innovations, Technical University of Ostrava, Czech Republic, supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Acknowledgements

The authors are grateful to Intel Technology Poland, Gdansk, Poland, as well as Poznan Supercomputing and Networking Center, Poland for granting access to HPC platforms. The authors would also like to thank Georg Zitzlsberger from IT4Innovations, Technical University of Ostrava, Czech Republic for his helpful advice on various technical issues.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was conducted with the financial support of the National Science Centre (Poland) under grant no. UMO-2017/26/D/ST6/00687.

References

Ang JA, Barrett RF, Benner RE, et al. (2014) Abstract machine models and proxy architectures for exascale computing. In: *Hardware-software co-design for high performance computing (Co-HPC)*, New Orleans, Louisiana, USA, 17 November 2014, pp. 25–32. IEEE.

Bandishti V, Pananilath I and Bondhugula U (2013) Tiling stencil computations to maximize parallelism. In: *SC'12 proceedings of the international conference on high performance computing, networking, storage and analysis*, Salt Lake City, USA, 10–16 November 2012, pp. 1–11. IEEE.

Bertolacci I, Olschanowsky CRM, Harshbarger B, et al. (2015) Parameterized diamond tiling for stencil computations with chapel parallel iterators. In: *Proceedings of the 29th ACM international conference on supercomputing* (eds KShin-ya P Andrzej, Jerzy P, Imed E and Janusz K), California, USA, 8–11 June 2015, pp. 197–206. NY, USA: ACM.

Bobulski J (2016) Parallel facial recognition system based on 2DHMM. In: *International multi-conference on advanced computer systems (ACS 2016)*, Miedzyzdroje, Poland, 19–21 October 2016, pp. 258–265. Cham: Springer.

Cosenza B, Durillo J, Ermon S, et al. (2017) Autotuning stencil computations with structural ordinal regression learning. In: *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, Sankt Goar, Germany, 12–13 June 2017, pp. 72–75. NY, USA: ACM.

Czarnul P (2017) Benchmarking performance of a hybrid Xeon/Xeon Phi system for parallel computation of similarity measures between large vectors. *International Journal of Parallel Programming* 45(5): 1091–1107.

Czarnul P (2018a) *Parallel Programming for Modern High Performance Computing Systems*. Boca Raton: Chapman and Hall/CRC.

Czarnul P (2018b) Parallelization of large vector similarity computations in a hybrid CPU+GPU environment. *The Journal of Supercomputing* 74(2): 768–786.

Datta K, Kamil S, Williams S, et al. (2009) Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* 51(1): 129–159.

Davidson A and Owens J (2012) Toward techniques for auto-tuning GPU algorithms. In: *Proceedings of the 10th international conference on applied parallel and scientific computing, PARA 2010* (ed Jónasson K), Reykjavík, Iceland, 6–9 June 2010, pp. 110–119. Berlin Heidelberg: Springer.

Eltablawy A and Vladimirov A (2015) *Capabilities of Intel AVX-512 in Intel Xeon Scalable Processors (Skylake)*, Sunnyvale, USA. Colfax International.

Falch T and Elster AC (2015) Machine learning based auto-tuning for enhanced opencl performance portability. In: *International parallel and distributed processing symposium workshop (IPDPSW)*, Hyderabad, India, 25–29 May 2015, pp. 1231–1240. IEEE.

Hager G and Wellein G (2011) *Introduction to High Performance Computing for Science and Engineers*. Boca Raton: CRC Press.

Henretty T, Stock K, Pouchet LN, et al. (2011) Data layout transformation for stencil computations on short-vector SIMD architectures. In: *International conference on compiler construction (CC'11/ETAPS'11)*, Saarbrücken, Germany, 26 March–3 April 2011, pp. 225–245. Berlin Heidelberg: Springer-Verlag.

Intel (2018) Intel 64 and IA-32 architectures optimization reference manual. Available at: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf (accessed 1 July 2018).

Intel Xeon Processor (2017) Intel Xeon Processor E7-8800/4800 v4 Product Family Specification. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e7-v4-spec-update.pdf (accessed 1 July 2018).

Intel Xeon Processor (2018) Intel Xeon Processor Scalable Family Specification. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf (accessed 1 July 2018).

Jeffers J, Reinders J and Sodani A (2016) *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Burlington: Morgan Kaufmann.

Kogge P and Shalf J (2013) Exascale computing trends: adjusting to the "new normal" for computer architecture. *Computing in Science & Engineering* 15(6): 16–26.

Kumar S, Bhattacharyya R, Joshi B, et al. (2016) On the role of repetitive magnetic reconnections in evolution of magnetic flux ropes in solar corona. *The Astrophysical Journal* 830(2): 1–12.

Laccetti G, Lapegna M and Mele V (2016) Loosely coordinated model for heap-based priority queues in multicore environments. *International Journal of Parallel Programming* 44(4): 901–921.

Lastovetsky A, Szustak L and Wyrzykowski R (2017) Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing. *IEEE Transactions on Parallel and Distributed Systems* 28(3): 787–797.

Leather H, Bonilla E and O'Boyle M (2009) Automatic feature generation for machine learning based optimizing compilation. In: *CGO'09 Proceedings of the 7th annual IEEE/ACM international symposium on code generation and optimization*, Seattle, USA, 22–25 March 2009, pp. 81–91. IEEE.

Malas T, Hornich J, Hager G, et al. (2016) Optimization of an electromagnetics code with multicore wavefront diamond blocking and multi-dimensional intra-tile parallelization. In: *2016 IEEE international parallel and distributed processing symposium*, Chicago, USA, 23–27 May 2016, pp. 142–151. IEEE.

Malik T, Rychkov V and Lastovetsky A (2016) Network-aware optimization of communications for parallel matrix multiplication on hierarchical HPC platforms. *Concurrency and Computation: Practice and Experience* 28(3): 802–821.

Malik T, Szustak L, Wyrzykowski R, et al. (2016) Network-aware optimization of MPDATA on homogeneous multi-core clusters with heterogeneous network. In: *ICA3PP 2016 collocated workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS*, pp. 30–42. Springer.

Naono K, Teranishi K, Cavazos J, et al. (eds) (2011) *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, Granada, Spain, 14–16 December 2016, pp. 30–42. Berlin: Springer.

OpenMP (2015) OpenMP application programming interface version 4.5. Available at: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (accessed 1 July 2018).

Rico-Gallego JA, Lastovetsky A and Diaz-Martin JC (2017) Model-based estimation of the communication cost of hybrid data-parallel applications on heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems* 28(11): 3215–3228.

Rojek K and Szustak L (2012) Parallelization of EULAG model on multicore architectures with GPU accelerators. In: *Proceedings of 9th international conference on parallel processing and applied mathematics, PPAM 2011*, Torun, Poland, 11–14 September 2011, pp. 391–400. Springer.

Rosa B, Szustak L, Wyszogrodzki AA, et al. (2015) Adaptation of multidimensional positive definite advection transport algorithm to modern high-performance computing platforms. *International Journal of Modeling and Optimization* 5(3): 171–176.

Smolarkiewicz PK (2006) Multidimensional positive definite advection transport algorithm: an overview. *International Journal for Numerical Methods in Fluids* 50(10): 1123–1144.

Smolarkiewicz PK and Charbonneau P (2013) EULAG, a computational model for multiscale flows: an MHD extension. *Journal of Computational Physics* 236: 608–623.

Smolarkiewicz PK and Margolin L (1998) MPDATA: a finite-difference solver for geophysical flows. *Journal of Computational Physics* 140(2): 459–480.

Smolarkiewicz PK, Szmelter J and Xiao F (2016) Simulation of all-scale atmospheric dynamics on unstructured meshes. *Journal of Computational Physics* 322(C): 267–287.

Strugarek A, Beaudoin P, Brun AS, et al. (2016) Modeling turbulent stellar convection zones: sub-grid scales effects. *Advances in Space Research* 58(8): 1538–1553.

Strzodka R, Shaheen M, Pajak D, et al. (2011) Cache accurate time skewing in iterative stencil computations. In: *2011 international conference on parallel processing (ICPP)*, Taipei City, Taiwan, 13–16 September 2011, pp. 571–581. IEEE.

Szustak L (2018) Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems. *The Journal of Supercomputing* 74(4): 1534–1546.

Szustak L, Halbiniak K, Kuczynski L, et al. (2016) Porting and optimization of solidification application for CPU–MIC hybrid platforms. *International Journal of High Performance Computing Applications* 32(4): 523–539. DOI: 10.1177/1094342016677740.

Szustak L, Halbiniak K, Kulawik A, et al. (2016a) Using hStreams Programming Library for accelerating a real-life application on Intel MIC. In: *ICA3PP 2016 collocated workshops: SCDT, TAPEMS, BigTrust, UCER, DLMCS*, Granada, Spain, 14–16 December 2016, pp. 373–382. Cham: Springer.

Szustak L, Halbiniak K, Kulawik A, et al. (2016b) Toward parallel modeling of solidification based on the generalized finite difference method using Intel Xeon Phi. In: *Proceedings of 11th international conference on parallel processing and applied mathematics, PPAM 2015*, Krakow, Poland, 6–9 September 2011, pp. 411–412. Cham: Springer.

Szustak L, Rojek K and Gepner P (2014b) Using Intel Xeon Phi coprocessor to accelerate computations in MPDATA algorithm. In: *Proceedings of 10th international conference on parallel processing and applied mathematics, PPAM 2013* (eds Roman W, Jack D, Konrad K and Jerzy W), Torun, Poland, 8–11 September 2013, pp. 582–592. Berlin: Springer.

Szustak L, Rojek K, Olas T, et al. (2015) Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. *Scientific Programming* 2015: 1–14.

Szustak L, Rojek K, Wyrzykowski R, et al. (2014a) Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture. In: *Proceedings of 1st international workshop on high-performance stencil computations, HiStencils 2014, in conjunction with HiPEAC 20–22 January 2014*, Vienna, Austria, 21 January 2014, pp. 51–56.

Szustak L, Wyrzykowski R and Jakl O (2017) Islands-of-cores approach for harnessing SMP/NUMA architectures in heterogeneous stencil computations. In: *Proceedings of 14th international conference on parallel computing technology, PaCT 2017* (ed Victor M), Nizhni Novgorod, Russia, 4–8 September 2017, pp. 351–364. Cham: Springer.

Unat D and others (eds) (2014) Programming abstractions for data locality, Report no. 01083080, v1, November 2014. Available at: https://hal.inria.fr/hal-01083080/file/PADAL-report.pdf (accessed 1 June 2018).

Vladimirov A, Asai R and Karpusenko V (2015) *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*, 2nd ed. Sunnyvale, USA: Colfax International.

Wyrzykowski R, Rojek K and Szustak L (2012a) Model-driven adaptation of double-precision matrix multiplication to the cell processor architecture. *Parallel Computing* 38: 260–276.

Wyrzykowski R, Rojek K and Szustak L (2012b) Using blue Gene/P and GPUs to accelerate computations in the EULAG model. In: *Proceedings of 8th international conference on large-scale scientific computations, LSSC 2011* (eds Ivan L, Svetozar M and Jerzy W), Sozopol, Bulgaria, 6–10 June 2011, pp. 670–677. Berlin: Springer.

Wyrzykowski R, Szustak L and Rojek K (2014) Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators. *Parallel Computing* 40(8): 425–447.

Zhou X, Giacalone JP, Garzarán MJ, et al. (2012) Hierarchical overlapped tiling. In: *CGO'12 Proceedings of 10th international symposium on code generation and optimization*, San Jose, California, 31 March – 04 April 2012, pp. 207–218. NY, USA: ACM.

Author biographies

Lukasz Szustak received his MSc 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 modern HPC architectures.

Pawel Bratek is a student of both computer and mathematical science at the Faculty of Mechanical Engineering and Computer Science, Czestochowa University of Technology, Poland. His scientific activity focus on code optimization of numerical application for multi- and manycore parallel architectures.