J Supercomput (2018) 74:1534–1546
https://doi.org/10.1007/s11227-018-2239-3

Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems

Lukasz Szustak[1]

Published online: 6 January 2018

Abstract In this paper, an innovative strategy for the data-flow synchronization in shared-memory systems is proposed. This strategy assumes to synchronize only inter-dependent threads instead of using the barrier approach that—in contrast to our approach—synchronize all threads. We demonstrate the adaptation of the data-flow synchronization strategy to two complex scientific applications based on stencil codes. An algorithm for the data-flow synchronization is developed and successfully used for both applications. The proposed approach is evaluated for various Intel microarchitectures released in the last 5 years, including the newest processors: Skylake and Knights Landing. The important part of this assessment is the performance comparison of the proposed data-flow synchronization with the OpenMP barrier. The experimental results show that the performance of the studied applications can be accelerated up to 1.3 times using the proposed data-flow synchronizations strategy.

Keywords Shared-memory platforms · Data-flow synchronization · Barrier · Stencil codes

1 Introduction

The huge capacity of modern HPC platforms allows complex problems, previously thought impossible, to be solved [12]. Reaching this goal requires to develop more effi-

This work was supported by the National Science Centre (Poland) under Grant no. UMO-2017/26/D/ST6/00687. The author is grateful for granting access to computing infrastructure provided by the MICLAB project no. POIG.02.03.00.24-093/13.

✉ Lukasz Szustak
lszustak@icis.pcz.pl

[1] Czestochowa University of Technology, Czestochowa, Poland

cient programming abstractions enabling better utilization of computing resources [7]. In particular, the still growing parallelism of emerging computing systems makes the synchronization issues of critical importance for scaling application performance on these systems [4]. The present paper meets this challenge for stencils algorithm running on advanced multi- and manycore computing platforms such as Intel architectures released in the last 5 years, including Ivy Bridge, Haswell, Broadwell, Skylake, Knights Corner and Knights Landing.

In this paper, we propose an innovative 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 our approach—synchronize all threads. An inseparable part of this strategy is the scheme of thread interrelationships for a given application. In fact, the data dependencies, workload distribution, way of parallelization and inter-thread data traffic play a key role in the effective adaptation of this strategy to a given application.

This paper demonstrates the adaptation of the proposed strategy to two complex scientific applications. The first one is the Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), the main module of the multiscale fluid model EULAG [10,15]. The EULAG model is an innovative solver in the field of numerical modeling of multiscale geophysical flows. Another application area tackled in the work refers to the phase-field method, which is a powerful tool for solving interfacial problems in materials science [11]. In this paper, we focus on the parallel implementation of a numerical model of the dendritic solidification process in the isothermal conditions [12]. These two applications belong to the class of the forward-in-time algorithms, which assume the iterative execution of multiple time steps. Each of the considered applications consists of a set of stencil-based computing kernels. Applications of this kind are used typically for long running simulations such as the numerical weather prediction that needs to execute several thousand time steps for a given size of domain.

Based on the proposed strategy, a data-flow synchronization algorithm is developed and successfully used for both applications. It is then evaluated experimentally to compare the efficiency of the data-flow synchronization with the OpenMP barrier. The results of tests show that the performance of these applications can be accelerated up to 1.3 times using the proposed approach.

The State-of-the-Art Synchronization algorithms differ in trade-offs between communication complexity, length of the critical path and memory footprint [9]. The barriers are an essential synchronization approach for parallel models of many shared-memory programming languages such as OpenMP, OpenCL or Cilk. They can be grouped into three categories: centralized, tree and butterfly. In addition, work [9] presents a hybrid barrier implementation dedicated to the first generation of Intel Xeon Phi accelerators.

Each synchronization algorithm features its own set of trade-offs, where the real profit is largely dependent on the structure of a computing systems. The optimization of barrier synchronization has been widely studied [2,3,6,8]. Typically, the barriers algorithms [4] are usually split into three phases: arrival, waiting and departure. The arrival phase shifts all arriving threads at the barrier into a waiting state. Once the execution of all threads has been locked by the barrier, the departure (or release) phase

is entered, which releases all threads from their busy state at once. As a result, all threads pass the barrier.

The main aim of this work is to avoid global barriers: the synchronization process should proceed only between carefully selected threads that depend on each others. An excellent justification and study of related work meeting this challenge can be found in the work of Bhatti et al. [1]. The authors presented an approach to implement stencil computations as dynamic task graphs, aiming at minimizing the synchronization overheads. In contrast to our approach, their method requires to redesign the structure of a parallel code using Intel Threading Building Blocks. At the same time, the synchronization strategies that base on data-flow communication layers are very popular in distributed-memory programming standards, including MPI or hStreams programming library [5]. In both cases, the synchronization between the interdependent processing elements is explicitly defined according to communication flows of data, using the specific commands such as MPI_Send and MPI_Recv in the case of MPI. However, this is achieved on a totally different level of programming abstraction than in the proposed approach.

2 Strategy for data-flow synchronization in stencils

The idea of strategy for the data-flow synchronization is to synchronize only interdependent threads/cores instead of using the barrier approach. This strategy needs to define the synchronization groups of threads/cores execution of which depend on each others. For this aim, the inter-thread traffic has to be determined for a given application, by considering the data dependencies of parallel computation.

An example of parallel computations including three stencil kernels is demonstrated in Fig. 1a. This example illustrates not only the general scheme of parallelizing computations across three cores, but also outlines the interrelationship between them. Particularly, in order to compute the third kernel, $core\_B$ and $core\_C$ are responsible for passing the outcomes computed, respectively, by $core\_A$ and $core\_B$ into the second kernel. At the same time, to compute the second kernel, $core\_A$ and $core\_B$ need to transfer the results calculated, respectively, by $core\_B$ and $core\_C$ by the first kernel. The available cores have to be synchronized because of the inter-core traffic. The classical strategy for the synchronization is based on the barrier approach that synchronizes all cores (Fig. 1b). However, not all cores have to be synchronized in this example, since $core\_A$ and $core\_C$ do not depend on each other.

In contrast to the barrier approach, we propose to synchronize only interdependent cores. To reach this aim, two synchronization groups of cores are defined in this example: the first group includes $core\_A$ and $core\_B$, while the second one embraces $core\_B$ and $core\_C$. The key assumption for our strategy is to perform synchronization inside every group of cores. The general idea is shown in Fig. 1c. It is expected to reduce the synchronization costs since each group should include definitely less cores/threads to synchronize than the barrier approach. Additionally, the proposed approach can alleviate the negative effect of load imbalance between cores that arises when the workload cannot be partitioned evenly. This advantage is also illustrated in

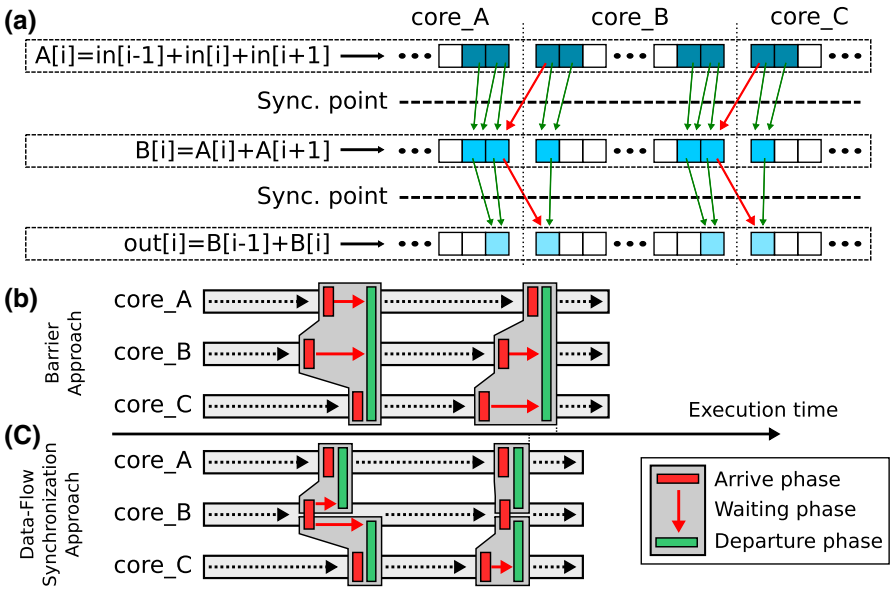

Fig. 1 Synchronization strategies for stencil parallel computation: **a** example of three stencil kernels and their parallelization; **b** barrier synchronization; **c** data-flow synchronization

Fig. 1c, where *core_A* can execute the second kernel earlier because it does not wait for *core_C*.

3 Adaptation of data-flow synchronization strategy to MPDATA

The MPDATA application implements a general approach for integrating the conservation laws of geophysical fluids on micro-to-planetary scales [10]. It allows solving advection problems and offers various options to model a wide range of complex geophysical flows. MPDATA corresponds to the group of iterative algorithms, where each time step [15] operates on five input arrays and returns one output array that is necessary for the next step. Every time step encompasses a set of 17 stencil kernels. Each of them represents a stencil code which updates elements of 3D grid according to a specific pattern. All these kernels are dependent on each others, where the outcomes of prior kernels are usually input data for the subsequent ones. The four synchronizations points are placed after the 4th, 7th, 13th, and 17th MPDATA kernels.

In our previous works [7,14,15], we proposed methods for the adaptation of MPDATA to multi-/manycore systems. These methods contribute to ease memory and communication bounds and to better exploit computation resources of shared-memory systems, including CPUs and Intel Xeon Phi. The main challenge of these works was to minimize data transfers between the main memory and cache hierarchy and better exploit the computing resources. To reach this goal, we proposed the (3+1)D decomposition of MPDATA computation [15] based on a combination of the loop fusion and loop tiling optimization techniques. The proposed approach yields a

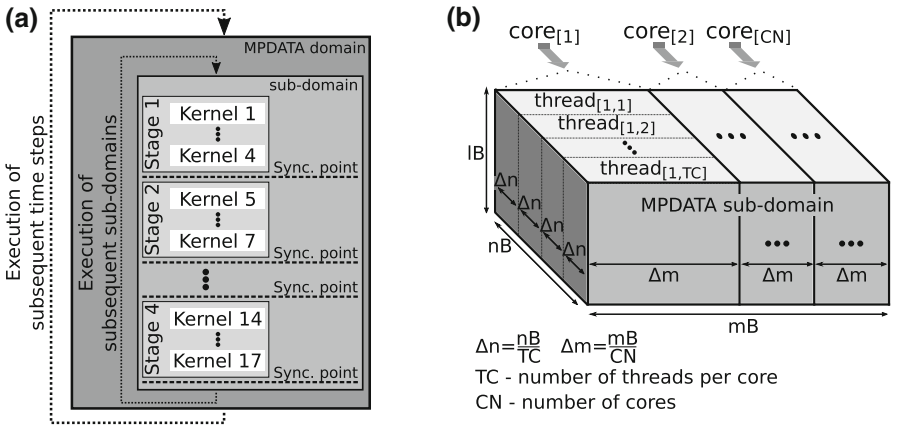

Fig. 2 Parallelization of MPDATA: **a** (3+1)D decomposition of MPDATA; **b** workload distribution across threads and cores for every MPDATA sub-domain of size $nB \times mB \times lB$

significant performance gain for Intel Xeon CPUs and Intel MICs (up to about four times).

The proposed decomposition requires partitioning the MPDATA grid into a set of sub-domains of size that enables to keep all the intermediate data in the cache memory. The consecutive sub-domains are processed sequentially, one by one, where each sub-domain is responsible for computing all the MPDATA kernels that perform computations on blocks of the corresponding arrays, and returns an adequate part of the output array. Since kernels depend on each others, the parallelization process for each MPDATA sub-domain requires implementing four synchronizations points. As a result, processing every sub-domain includes four stages, where the first stage contains four MPDATA kernels, the second one—three kernels, the third one—six kernels, while the last stage includes four kernels. The synchronization of threads is performed after every stage. In consequence, the total amount of synchronization points depends on the numbers of sub-domains and time steps.

The general scheme of the proposed decomposition is presented in Fig. 2a. The parallelization process is performed within every sub-domain using all the available threads. For this aim, each sub-domain of size $nB \times mB \times lB$ is partitioned evenly into parts of size $\frac{nB}{TC} \times \frac{mB}{CN} \times lB$, where CN denotes the number of physical cores, while TC is the number of threads per each core. All these parts are assigned to threads in compact fashion, in order to enable their efficient cooperation (Fig. 2b). As a result, the neighbor parts of each sub-domain are processed by the adjacent physical cores, including threads pinned to them.

The execution of an MPDATA stage requires to load outcomes from the kernels assigned to the previous stages. Furthermore, the stencil pattern for a given stage corresponds to the aggregate patterns of MPDATA kernels belonging to this stage. Based on data dependencies between MPDATA kernels, we are able to define the general 27-point stencil pattern (Fig. 3a), that in fact is suitable for every stage. According to this pattern, a given grid element processed at any stage requires delivering of all

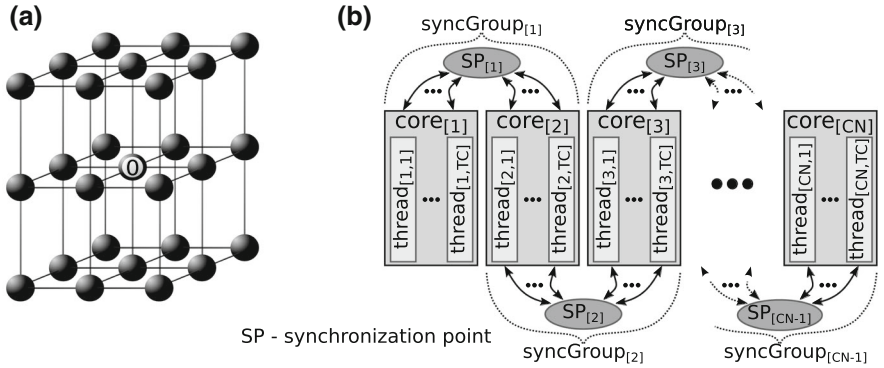

Fig. 3 Interrelationships between cores, threads and synchronizations for MPDATA: **a** aggregate stencil pattern of MPDATA stages; **b** structure of synchronization groups of threads

neighbor elements returned by previous stages. These elements are located in a 3D subspace corresponding to the nearest neighborhood of the processed element. By considering the elements computed on the border between sub-domains, it can be concluded that the inter-thread data traffic takes place only between all the threads pinned to adjacent physical cores. In fact, the threads assigned to a given physical core C_i depend only on the outcomes computed by the threads pinned to the core C_{i-1} on its left side, as well as the threads pinned to the core C_{i+1} on its right side. At the same time, the threads associated with the cores C_{i-1} and C_{i+1} do not depend on each others.

Therefore, to improve the efficiency of synchronization in MPDATA computation, we propose to synchronize only the threads pinned to every pair of adjacent cores. To reach this aim, we define $(CN - 1)$ synchronization groups of threads (see Fig. 3b) that include the subsequent pairs of adjacent physical cores, where a given ith pair includes all threads assigned to the cores C_{i-1} and C_i, while the next $(i + 1)$ group encompasses threads pinned to the cores C_i and C_{i+1}. As a result, the threads assigned to a core are now associated with two synchronization groups on their left and right sides. An exception to this rule is the first and last cores that are assigned only to a single group. If TC is the number of threads per core for a given computing platform, then the total number of threads per every synchronization group is equal to $2 \times TC$.

To implement the data-flow synchronization approach, we develop a new synchronization algorithm based on the centralized counter-based algorithm [4]. In the proposed Algorithm 1, every synchronization group of threads (shortly syncGroup) is equipped with a single synchronization point (syncPoint) that contains two shared synchronization flags: (i) a counter initially containing the number of threads per a given syncGroup and (ii) a global release flag (global sense). Basically, once a given thread arrives to the syncPoint (arrival phase) the counter is atomically decremented, and the thread waits (waiting phase) for a global release flag to change its state. If the value of the counter reaches zero, the flag is updated to release all waiting threads assigned to a given syncGroup (departure phase). In the algorithm, a separate sense-reversal flag [4]

Algorithm 1 Data-Flow Synchronization algorithm for MPDATA stencils

Input: $leftLocalSens$; $rightLocalSens$; $leftPoint$; $rightPoint$

```
L := fetch_and_add(leftPoint.counter, -1)                          ▷ Arriving phase
R := fetch_and_add(rightPoint.counter, -1)
if L = 0 and R = 0 then                                            ▷ Scenario 1
    leftPoint.counter := leftPoint.threadsNum
    leftPoint.globalSense := leftLocalSens
    rigthPoint.counter := rightPoint.threadsNum
    rightPoint.globalSense := rightLocalSens
else if L = 0 then                                                 ▷ Scenario 2
    leftPoint.counter := leftPoint.threadsNum
    leftPoint.globalSense := leftLocalSens
    while rightPoint.globalSense ≠ rightLocalSens do               ▷ Waiting phase
    end while
else if R = 0 then                                                 ▷ Scenario 3
    rightPoint.counter := rightPoint.threadsNum
    rightPoint.globalSense := rightLocalSens
    while leftPoint.globalSense ≠ leftLocalSens do                 ▷ Waiting phase
    end while
else                                                               ▷ Scenario 4
    while leftPoint.globalSense ≠ leftLocalSens or                 ▷ Waiting phase
          rightPoint.globalSense ≠ rightLocalSens do
    end while
end if
leftLocalSens := !leftLocalSens                                    ▷ Departure phase
rightLocalSens := !rightLocalSens
```

is used to free all the threads once the last thread reaches a given syncPoint. Therefore, each thread includes also one local release flag of its private sense (localSense) per every synchronization point. Finally, since every thread is associated with two synchronization groups, the synchronization phases for these two groups are merged to avoid a synchronization deadlock.

In Algorithm 1, a given thread atomically fetches and decrements two counters assigned to its left and right synchronization points (arrival phase), while the further execution of this thread is dependent on states of these counters and corresponds to one of the four possible scenarios. In the first scenario, a given thread reaches both the left and right counters as the last thread, changing values of these counters to zero. Then, this thread changes the state of the global sense flags assigned to the left and right synchronization points, in order to releases all threads from both synchronization groups (departure phase). The second scenario corresponds to the case when a given thread reaches only the left counters as the last one. In this case, the thread updates only its left global sense, in order to release all waiting threads assigned to the left synchronization group, and then will wait for the right global release flag to change its state (waiting phase). In the next scenario, the right counter is decremented to zero that enables releasing the right group of threads. This allows passing a given thread to the waiting phase for the left syncGroup. In the last scenario, the values of both counters are greater than zero, so a given thread has to wait for all threads associated with the left and right synchronization groups.

4 Adaptation of data-flow synchronization strategy to solidification modeling application

Another subject of our research is adaptation of the data-flow synchronization strategy to the numerical modeling of 2D solidification problems [12]. The resulting computing scheme belongs to the group of forward-in-time, iterative algorithms. The application code includes five computing kernels that are processed sequentially one by one, in every time step. The computing kernels represent stencil codes which updates elements of the 2D grid according to 9-point stencil patterns, which involve grid points located in the nearest neighborhood. All these kernels depend on each others: outcomes returned by a given kernel are the input data for the next one. In consequence, the five synchronization points are required per every time step.

In our previous works [5,12,13], we developed a methodology for porting the parallel code of this application to hybrid platforms with CPUs and Intel Xeon Phi coprocessors, assuming no significant modifications of the code. In particular, we evaluated different setups for the OpenMP parallelization. Independently of selecting `static` (for coprocessors) or `dynamic` (for CPUs) scheduling option, the chosen scheme of scheduling of loop iterations tries to assign evenly consecutive chunks of iterations among subsequent threads. In consequence, the 2D domain is partitioned along one dimension into a set of sub-domains, where a given sub-domain is processed by a single thread t_i. For the 9-point stencil pattern, the inter-thread data traffic takes place only between subsequent couples of neighbor threads, where a given ith couple includes threads t_{i-1} and t_i, while the next couple embraces threads t_i and t_{i+1}. The total number of these couples corresponds to the total number of available threads minus one, where every thread is associated with two couples. In general, the thread t_i depends only on the outcomes of the thread t_{i-1} on its left side, and the thread t_{i+1} on its right side, but these two threads (t_{i-1} and t_{i+1}) do not depend on each other. Since only the adjacent threads depend on each other, we propose to perform the synchronization inside every couple of threads instead of using the barrier approach. So each of these couples becomes now a synchronization group of two threads.

Such a synchronization scheme is similar to the MPDATA case, where only threads assigned to each couple of adjacent cores are synchronized. The main difference is the number of synchronization groups and the number of threads per each group. In fact, if a given computing platform supports only a single thread per each physical core, the synchronization scheme is exactly the same for both applications. As a result, the proposed Algorithm 1 can be successfully reused again in the solidification modeling application.

5 Experimental results

The proposed approach is evaluated experimentally for various Intel microarchitectures released in the last 5 years (https://ark.intel.com). We use six computing platforms based on Ivy Bridge, Haswell, Broadwell, Skylake, Knights Corner and Knights Landing. The first four platforms represent 2-socket servers with Intel Xeon processors, while the last two ones are based, respectively, on the first and second generations of

Intel Xeon Phi chips. In all tests, the Intel icpc compiler (v.17.0.1) with the optimization flag -O3 is used. To ensure reliability of results, all performance measurements are repeated several times, and averaged execution times are used. Moreover, since the accuracy of computation plays a key role for the studied applications, the double-precision floating-point format is selected.

The OpenMP standard is used to parallelize both applications. While testing the proposed synchronization approach, calls to the OpenMP barrier are replaced with invocations of a function implementing the proposed Algorithm 1. Since every synchronization point shares the synchronization flags within its synchronization group of threads, the compiler intrinsic of `fetch-and-add` instruction is used for updating the counter flags atomically. Additionally, the procedure for pinning threads to selected synchronization groups is invoked before executing the parallel region of the application code.

Table 1 presents the performance results of the MPDATA application obtained for the domain of size $256 \times 256 \times 64$, and 500 time steps. This table shows the overall execution time for MPDATA simulation, as well as the minimum (min) and maximum (max) times for the synchronization process. The presented values of the min and max parameters are selected among measurements taken for all threads. Additionally, the percentage of the synchronization cost in relation to the overall execution time is determined. Finally, this table demonstrates the overall performance gain from the usage of the proposed data-flow synchronization approach for MPDATA. In all performed tests, the proposed approach allows reducing the execution time of MPDATA simulations for all computing platforms. In particular, the best acceleration of about 1.3 times against the OpenMP barrier is achieved for the two newest platforms based on Knights Landing and Skylake. At the same time, the lowest speedup of 1.14x is obtained for Intel Ivy Bridge (platform *A*).

The performed tests reveal that the cost of synchronization refers to a significant part of the MPDATA execution time, which reaches even 30% for the OpenMP barrier, while it is reduced to 17% for the proposed approach. The main reason of this overhead is a huge number of synchronization points. Their amount depends on the number of sub-domains. In these tests, the best performance results are achieved for sub-domains of size $4 \times 256 \times 64$ that corresponds to partitioning the MPDATA domain into 64 parts. In consequence, since each sub-domain requires four synchronization points, 256 synchronization points are needed for a single time step of MPDATA. Furthermore, there are variations in costs of synchronization processes across threads. Such differences are results of an uneven workload partitioning between available threads. This generates performance losses, and using the data-flow synchronization allows reducing them about twice.

Figure 4 summarizes the experimental evaluation of the proposed synchronization strategy for the MPDATA application. It shows the performance gain in relation to the OpenMP barrier, for different domain sizes, and a variety of computing platforms. In these tests, the acceleration of the MPDATA execution is in the range from $1.14\times$ to $1.33\times$ (Fig. 4a). It could be also observed that the cost of synchronization is reduced for all experiments by the factor in the range from 1.63 to 2.89 (Fig. 4b).

The summary of tests performed for the solidification modeling application is presented in Fig. 5. Similarly to MPDATA, the proposed approach improves the overall

Table 1 Performance results of MPDATA for domain of size 256 × 256 × 64 and 500 time steps, achieved for different Intel microarchitectures, including Ivy Bridge (A), Haswell (B), Broadwell (C), Skylake (D), Knights Corner (E), and Knights Landing (F)

Computing platforms	Type of Sync.	Execution time (s)	Sync. time		Cost of sync. (%)	Overall speedup
			max (s)	min (s)		
2×Intel CPUs E5-2695v2 (**A**)	OMP barrier	5.14	0.90	0.58	17.51	–
	Data-flow	4.50	0.55	0.15	12.22	1.14
2×Intel CPUs E5-2697v3 (**B**)	OMP barrier	4.21	0.80	0.70	19.00	–
	Data-flow	3.54	0.37	0.16	10.45	1.19
2×Intel CPUs E5-2682v4 (**C**)	OMP barrier	3.95	0.80	0.60	20.25	–
	Data-flow	3.40	0.28	0.20	8.24	1.16
2×Intel CPUs gold 6148 (**D**)	OMP barrier	2.87	0.88	0.70	30.66	–
	Data-flow	2.23	0.38	0.14	17.04	1.29
Intel xeon phi 7120P (**E**)	OMP barrier	7.06	1.80	1.10	25.50	–
	Data-flow	6.00	1.00	0.50	16.67	1.18
Intel xeon phi 7210 (**F**)	OMP barrier	3.58	0.92	0.48	25.70	–
	Data-flow	2.70	0.46	0.17	17.04	1.33

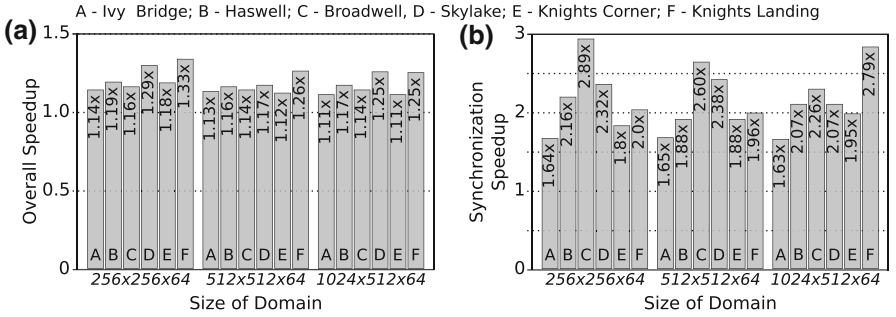

Fig. 4 Performance results for MPDATA achieved for different problem sizes, 500 time steps, and various Intel microarchitectures: **a** overall speedup of MPDATA using the data-flow synchronization; **b** speedup of data-flow synchronization against OpenMP barrier

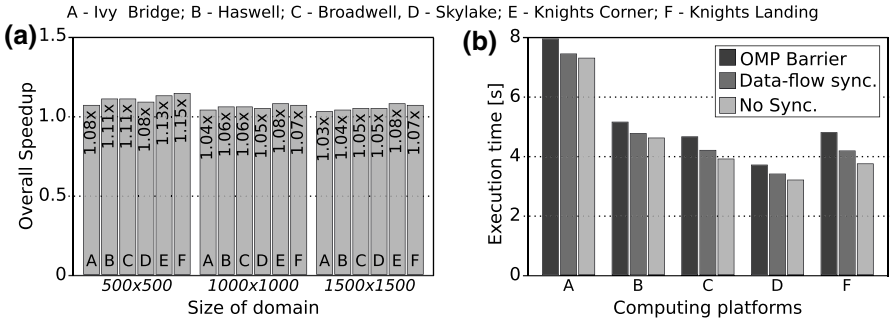

Fig. 5 Performance results of the solidification modeling application achieved for 2000 time steps, and various Intel microarchitectures: **a** overall speedup of the application using the data-flow synchronization obtained for different problem sizes, **b** execution time for the problem of size 500 × 500 and different synchronization scenarios

performance for all tests (Fig. 5a). However, we observe a lower performance gain than in the MPDATA case, with the speedups in the range from 1.03x to 1.15x, where the highest gain of 1.15x is achieved for the platform based on the newest Skylake processors. In fact, the solidification modeling application requires significantly less synchronizations points than MPDATA. As a result, the synchronization cost constitutes a smaller part of the total execution time than in the case of MPDATA (Fig. 5b). At the same time, the acceleration of the synchronization process is at a similar level, with the speedups in the range from 1.81 to 4.22 times.

6 Conclusions and future work

Modern multi-/manycore architectures feature both the increasing number of processing elements, and a high level of complexity of their integration. In consequence, accelerating the synchronization process becomes vital for attaining high performance on emerging computing platforms. In this work, we present the new approach for reducing the synchronization overhead for stencils by avoiding the global barriers. The

innovative strategy for the data-flow synchronization in shared-memory platforms is proposed. The main idea of this strategy is to synchronize only interdependent threads instead of using the barrier approach that—in contrast to our approach—synchronize all threads. We also develop the algorithm for the data-flow synchronization that is successfully used for two stencil-based scientific applications.

The proposed approach is evaluated experimentally for various Intel architectures, including Ivy Bridge, Haswell, Broadwell, Skylake, Knights Corner and Knights Landing chips. An important part of this evaluation is the performance comparison between the proposed data-flow synchronization and the Intel OpenMP barrier. All tests prove that the data-flow synchronizations yields better performance results then the standard approach. The overall performance of both the MPDATA and solidification modeling applications is improved for all computing platforms. In particular, the best acceleration of about 1.3 times is obtained for two platforms based on the newest processors with Knights Landing and Skylake microarchitectures.

The proposed data-flow synchronization strategy allows reducing considerably the synchronization costs. For a given application, the real profit depends on the problem size and characteristics of computing platforms. The reduction of synchronization costs in the range from about 1.6 to 4.2 times is achieved in comparison with the OpenMP barrier. The proposed approach enables also alleviating the effect of load imbalance between cores that arises when the computing workload cannot be partitioned evenly across available threads.

References

1. Bhatti ZW, Wuyts R, Costanza P, Preuveneers D, Berbers J (2013) Efficient synchronization for stencil computations using dynamic task graphs. Procedia Comput Sci 18:2428–2431
2. Brooks E (1986) The butterfly barrier. Int J Parallel Program 15(4):295–307
3. Caballero D, Duran A, Martorell X (2013) An OpenMP* barrier using SIMD instructions for Intel Xeon Phi coprocessor. In: International Workshop on OpenMP, pp 99–113
4. Dolbeau R (2014) Address selection for efficient barriers on the Intel Xeon PHI. http://www.dolbeau.name/dolbeau/publications/barrierphi.pdf. Accessed 12 Dec 2017
5. Halbiniak K, et al (2016) Exploring OpenMP Accelerator Model in a real-life scientific application using hybrid CPU-MIC platforms. In: Proceedings 3rd International Workshop on Sustainable Ultrascale Computing Systems, Sofia, Bulgaria, pp 11–14
6. Hensgen D, Finkel R, Manber U (1988) Two algorithms for barrier synchronization. Int J Parallel Program 17(1):1–17
7. Lastovetsky A et al (2017) Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing. IEEE Trans Parallel Distrib Syst 28(3):787–797
8. Mellor-Crummey J, Scott M (1991) Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans Comput Syst 9(1):21–65
9. Rodchenko A et al (2015) Effective barrier synchronization on intel xeon phi coprocessor. LNCS 9233:588–600

10. Smolarkiewicz P (2006) Multidimensional positive definite advection transport algorithm: an overview. Int J Numer Meth Fluids 50(10):1123–1144
11. Steinbach I (2009) Phase-field models in materials science. Model Simul Mater Sci Eng 17(7):073001
12. Szustak L, Halbiniak K, Kuczynski L, Wrobel J, Kulawik A (2016) Porting and optimization of solidification application for CPU -MIC hybrid platforms. Int J High Perform Comput Appl. https://doi.org/10.1177/1094342016677740
13. Szustak L, Halbiniak L, Kulawik A, Wyrzykowski R, Uminski P, Sasinowski M (2016) Using hstreams programming library for accelerating a real-life application on intel MIC. LNCS 10049:373–382
14. Szustak L, Jakl O, Wyrzykowski R (2017) Islands-of-cores approach for harnessing SMP/NUMA architectures in heterogeneous stencil computations. LNCS 10421:351–364
15. Szustak L, Rojek K, Olas T, Kuczynski L, Halbiniak K, Gepner P (2015) Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. Sci Program 2015:1–14