Toward Heterogeneous MPI+MPI Programming: Comparison of OpenMP and MPI Shared Memory Models

Lukasz Szustak(✉), Roman Wyrzykowski, Kamil Halbiniak, and Pawel Bratek

Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland
{lszustak,roman,khalbiniak,pbratek}@icis.pcz.pl

Abstract. This paper introduces our research on investigating the possibility of using heterogeneous all-MPI programming for the efficient parallelization of real-world scientific applications on clusters of multicore SMP/ccNUMA nodes. The investigation is based on verifying the efficiency of parallelizing a CFD application known as MPDATA, which contains a set of stencil kernels with heterogeneous patterns. As the first step of the research, we consider the level of SMP nodes, and compare the performance achieved by the MPI Shared Memory model of MPI-3 versus the OpenMP approach. In contrast to other works, this paper aims to evaluate these two programming models in conjunction with the parallelization methodology recently proposed [1] for performance portable programming across multicore SMP/ccNUMA platforms. We show that the shared memory extension of MPI delivers portable means for implementing all steps of this methodology efficiently, to take advantages of emerging multicore ccNUMA architectures.

Keywords: MPI shared memory · Multicore SMP/ccNUMA · MPDATA

1 Introduction

The Message Passing Interface (MPI) [2] is a dominant parallel programming model for distributed memory systems, including large clusters with tightly coupled SMP nodes. In the recent past, applications written with nothing except MPI were able to deliver an acceptable and portable performance, as well as scalability. However, as the number of cores per node has increased, programmers have increasingly took advantage of the hybrid (heterogeneous) parallel programming with MPI for internode communications in conjunction with shared memory programming systems, such as OpenMP, to manage intranode parallelism [3]. While this hybrid model, known as MPI+X [4], provides a lot of flexibility and performance potential, it burdens programmers with the complexity of using two parallel programming systems in the same application [5]. Apart from problems with a proper work of interface between two systems, there are other open issues, e.g., who manages the cores and how is that negotiated?

U. Schwardmann et al. (Eds.): Euro-Par 2019 Workshops, LNCS 11997, pp. 270–281, 2020.
https://doi.org/10.1007/978-3-030-48340-1_21

Version 3.0 of the MPI standard introduces another option for hybrid programming that uses the new MPI Shared Memory (SHM) model [6] to build MPI-everywhere codes for clusters with SMP nodes. In this MPI+MPI model, the MPI SHM extension enables programmers to create regions of shared memory that are directly accessible by MPI processes within the same shared memory domain. Also, several functions were added that enable MPI ranks within a shared memory region to allocate shared memory for direct load/store access. The ability to directly access a region of memory shared between ranks can improve performance in comparison with the pure MPI option, by reducing memory motion and footprint [3,5].

This paper introduces our research on investigating the possibility of using heterogeneous all-MPI programming for the efficient parallelization of real-world scientific applications on clusters of multicore SMP/ccNUMA nodes. The investigation is based on verifying the efficiency of parallelizing a CFD application known as MPDATA (Multidimensional Positive Definite Advection Transport Algorithm) [7]. It contains a set of stencil kernels with heterogeneous patterns.

As the first step of our research, we consider the level of SMP nodes, and compare the performance achieved by the MPI SHM model versus the OpenMP approach. The latter has become [4] a dominant choice for parallel programming of modern shared memory systems used as cluster nodes. The capabilities of such systems are constantly growing as a result of quick progress in multicore technology. It is quite easy to build SMP nodes with 112 or even 224 cores (2×56 cores with Intel Xeon Platinum 9282 or 8×28 with Intel Xeon Platinum 8280, see https://ark.intel.com). Thus, efficient harnessing of multicore SMP nodes with high degree of parallelism becomes of vital importance for the total performance of applications.

This paper is organized as follows. Section 2 discusses related works, while Sect. 3 presents a brief comparison of MPI SHM and OpenMP models. The MPDATA application is introduced in Sect. 4, which presents also the parallelization methodology for shared memory multi- and manycore architectures. Mapping MPDATA decomposition onto OpenMP and MPI Shared Memory is revealed in Sect. 5, while results of experimental evaluation of these two options are presented and discussed in Sect. 6. The paper is concluded in Sect. 7.

2 Related Work

The MPDATA code has been recently re-written and optimized for execution on HPC platforms with multicore CPUs and Intel MIC accelerators. The new C++ implementation proposed in [8] allows a more efficient distribution of computations across the available resources. It makes use of the (3+1)D decomposition strategy for heterogeneous stencils, that transfers the data traffic from the main memory to the cache hierarchy by reusing caches properly. Also, to improve the efficiency of computations, the algorithm groups the cores/threads into independent work teams in order to reduce inter-cache communication overheads due to transfers between neighbor cores.

Next, to harness the heterogeneous nature of communications in shared memory systems with ccNUMA architecture, the islands-of-cores approach was proposed in [9]. It allows a flexible management of the trade-off between computation and communication costs in accordance with features of multicore ccNUMA architectures. Finally, to reduce the synchronization overheads, an innovative strategy for the data-flow synchronization in shared memory systems was developed in [10]. As all designed codes were implemented with OpenMP, their direct extension on clusters with multicore SMP nodes requires utilizing the hybrid MPI+OpenMP approach.

This approach has already been applied quite successfully to real scientific applications [11,12]. For the CFD simulation considered in [11], the hybrid code can outperform a pure MPI version by up to 20%, while pure MPI still outperforms hybrid MPI+OpenMP in modeling of granular materials [12]. Recent scientific works enlighten the complexity of many aspects of the hybrid model that affect the overall performance and development costs of hybrid programs [2,3,5]. Thus, choosing a right option for parallel programming of real-world applications on clusters requires further research. In particular, there are surprisingly few works on performance comparison between MPI+MPI and MPI+OpenMP approaches. An example is work [13] on a performance evaluation of the MPI SHM model compared to OpenMP, using two relatively simple case studies: the matrix-matrix multiplication and Conway's game of life. The latter is an example of an 8-point stencil application. In contrast to this work, our research aims at evaluating these two programming models in conjunction with the parallelization methodology proposed recently [1] for performance portable programming across multicore SMP/ccNUMA systems. What is important, this methodology is not tailored to a particular programming approach.

3 MPI Shared Memory Model Compared to OpenMP

By default MPI codes are executed under the distributed memory model that assumes the private data allocation for each MPI process. In consequence, all processes have to communicate with each other using calls to MPI functions that typically moves data explicitly or perform some collective operations [14]. In this memory model, data are not shared automatically across MPI processes.

The MPI-3 RMA (Remote Memory Access) interface extends the default memory model with a new unified model [3] that is exposed through the MPI window. An MPI window object can be used to allocate shared memory regions [14] using the collective MPI routine `MPI_Win_allocate_shared`. It enables also the non-contiguous shared memory allocation by specifying the key info parameter `alloc_shared_noncontig` in order to fully utilize ccNUMA architectures. In addition, the function `MPI_Win_shared_query` is provided to query pointers to the memory allocated on MPI processes, that enable them immediate load/store operations with automatically propagated updates of data. As a result, data are automatically shared between MPI processes in a similar fashion as for OpenMP codes, where all OpenMP threads access data in parallel and coherent way [15].

MPI requires the explicit control of data parallelism. It is responsibility of programmers to formulate explicitly the workload distribution strategy. Nevertheless, the richness of the MPI library makes this shortage relatively easy to overcome. In contrast, OpenMP offers a straightforward mechanism for data parallelism that can automatically split the workload across available threads. However, as shown in [1], the parallel efficiency of an application can be significantly improved by replacing the standard solution for data parallelism, such as `#pragma omp for`, by a custom strategy for workload distribution adapted to the application, as well as to a target architecture. As a result, the data or loop parallelism with threads often requires a specific parallelisation strategy which in fact is similar to that of MPI, especially for shared memory programming.

4 Overview of MPDATA Parallelization

4.1 Introduction to MPDATA Application

The MPDATA application implements a general approach to modeling a wide range of complex geophysical flows on micro-to-planetary scales [7]. MPDATA belongs to the class of methods for the numerical simulation of fluid flows that are based on the sign-preserving properties of upstream differencing. It is mainly used to solve the advection problems on moving grids for a sequence of time steps, that classifies MPDATA into the group of forward-in-time algorithms. In this paper, we consider solving 3D problems. The MPDATA numerical scheme is described in detail in [7].

MPDATA is typically used for long simulations that run thousands of time steps. A single step operates on five input matrices (arrays), and returns a single output array that is used in the next step. Each MPDATA step performs a collection of 17 kernels that depend on each other (the outcomes of prior kernels typically are inputs for the subsequent ones). Each kernel is a stencil code that updates elements of its 3D output array, according to a specific pattern.

4.2 Parallelization Methodology for Shared Memory Systems

In the basic version of MPDATA (Listing 4.1) all kernels are executed sequentially, one by one, with each kernel processed in parallel using OpenMP. This version exploits data parallelism across i-dimension, based on distributing data across available resources by `#pragma omp for` directive, and then incorporates vectorization along k-dimension using `#pragma vector` directive [16].

The operational intensity of each MPDATA kernel is not high enough [1,17] to utilize computing resources of modern processors efficiently. Since the code is not optimized for cache reusing, the performance of this MPDATA version is limited by the main memory bandwidth. To alleviate these constraints, we developed [1,8–10] a parallelization methodology for MPDATA heterogeneous stencil computations. It contributes to ease the memory and communication bounds, and to better exploit resources of multicore ccNUMA/SMP systems.

This methodology consists of the following parametric optimization steps:

Listing 4.1. Part of 3D MPDATA basic version, corresponding to the 4-th kernel

```
/*...*/
//Kernel 4
#pragma omp for
for( ... ) // i - dimension
 for( ... ) // j - dimension
  #pragma vector
  for( ... ) // k - dimension
   x[i,j,k]=XIn[i,j,k]-(((F1[i+1,j,k]-F1[i,j,k])+(F2[i,j+1,k]
            -F2[i,j,k])+(F3[i,j,k+1]-F3[i,j,k]))/H[i,j,k]);
/*...*/
```

- *(3+1)D decomposition of MPDATA* [8] – the prime goal of is to take advantage of cache reusing by transferring the data traffic between kernels from the main memory to the cache hierarchy. For this aim, a combination of loop tiling and loop fusion optimization techniques is used, that allows reducing the main memory traffic at the cost of additional computations.
- *Data-flow strategy of synchronization* [10] – the main purpose is to synchronize only interdependent threads instead of using the barrier approach that typically synchronize all threads. This strategy reduces the cost of synchronization. Implementing this strategy for MPDATA needs to reveal the scheme of inter-thread data traffic during execution of MPDATA kernels.
- *Partitioning cores into independent work teams* [9] – this strategy delivers two scenarios for executing MPDATA kernels: the first one performs less computations but requires more data traffic, while the second scenario allows us to replace the implicit data traffic by replicating some of computations. As a result, the second scenario is successfully used to reduce inter-processor communications between caches in ccNUMA systems, while the first scenario is applied inside each processor.
- *Vectorization* – the last step is responsible for ensuring the performance portability of vectorizing MPDATA computations. In paper [1], we proposed the 7-step procedure for the MPDATA code transformation to allow the compiler to perform the vectorization automatically.

Figure 1 illustrates the hierarchical decomposition of MPDATA according to the proposed methodology. In general, the MPDATA domain is partitioned into p sub-domains that are processed by p processors of a given ccNUMA platform (Fig. 1a). Now each processor embraces a work team of cores, where each work team processes a sub-domain following the (3+1)D decomposition (Fig. 1b). Furthermore, each sub-domain is decomposed into blocks of size that enables keeping all the required data in the cache memory. The successive blocks are processed sequentially, one by one, where a given block exploits data parallelism across i- and j-dimensions (Fig. 1c) to distribute workload across available cores/threads. Each core of a given work team executes computations corresponding to all MPDATA kernels, that are performed on appropriate chunks of data arrays.

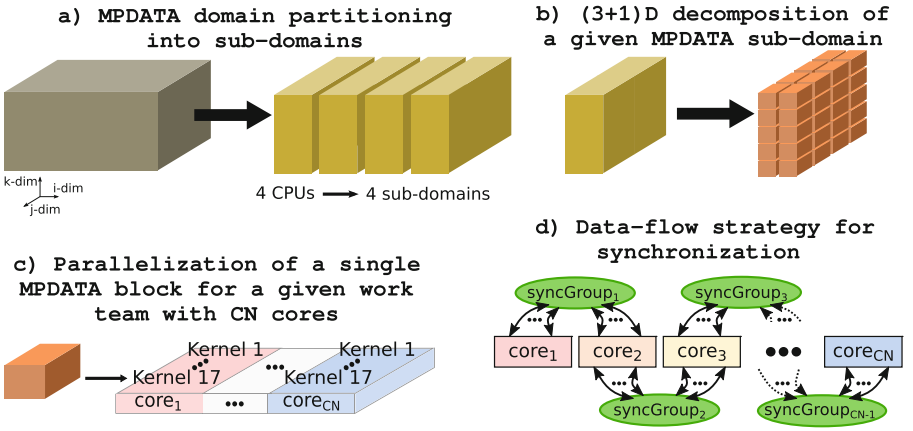

Fig. 1. Decomposition of MPDATA: a) domain partitioning into sub-domains, b) sub-domain decomposition into blocks of size adjusted to cache capacity, c) parallel execution of kernels within a single block by a given work team, and d) synchronization

Finally, the data layout used for storing arrays enforces performing the vectorization along k-dimension.

Because of data dependencies between the kernels, two synchronization levels have to be considered: inside every work team (first level), and between all work teams (second one). The parallelization of every block requires providing five synchronization points inside every work team. To improve the efficiency for the first level, only interdependent threads are synchronized according to data dependencies of kernels (Fig. 1d). Additionally, all work teams have to be synchronized after each time steps to ensure the correctness of simulation.

In order to implement the parallelization methodology automatically, we proposed [1] the parameterized transformation of the MPDATA code to achieve the high sustained and scalable performance for ccNUMA shared memory systems. As a result, the adaptive MPDATA code follows along with parameters of hardware components such as memory hierarchy, multi-/manycore, threading, vectorization, and their interaction with MPDATA computations.

5 Mapping MPDATA Decomposition onto Shared Memory Programming

5.1 Data Parallelism

The complexity of the proposed hierarchical decomposition (see Fig. 1) makes it impossible to efficiently implement parallelization across available cores using general approaches for data parallelism, such as `#pragma omp for` construct of OpenMP. Instead, based on the four-step procedure for MPDATA code customization [1], we developed a dedicated scheduler that is responsible for the

management of workload distribution and data parallelism. The main assumption is to calibrate the developed hierarchical domain decomposition for a given computing platform, before the execution of a specific numerical simulation.

Following the proposed customization, our scheduler explicitly define the scope of work for all available computing resources of a given ccNUMA system. As a result, each physical core is assigned to a given work team that process all MPDATA blocks from its sub-domain, and then is linked to appropriately selected pieces of distributed data for all MPDATA kernels within every block. This is achieved by providing a suitable loop-level management of loop iterations distributed across computing resources. A simplified structure of loop-level management for the proposed hierarchical decomposition is shown in Listing 5.1.

Since the scope of work for each core is individually determined, the proposed methodology can be successfully implemented for any shared memory model that supports data parallelism. To map efficiently the proposed decomposition onto shared memory programming systems, such as OpenMP and MPI SHM, each OpenMP thread or MPI process has to be associated with the workload defined for a given physical core, using its ID (OpenMP thread ID or MPI rank). A part of this issue is selecting a correct policy for binding OpenMP threads or MPI processes to physical cores that can guarantee optimality of both data parallelism and inter-core communication paths. Both Intel MPI and Intel OpenMP offer a flexible interface to control thread/process affinity [14,16].

5.2 Memory Allocation and Data Sharing

The MPDATA code distinguishes two groups of data: (i) a large set of 3D arrays (matrices) of floating-point type processed during MPDATA computation, and relatively small packages of data of various types required for the loop-level management with the proposed scheduler. For performance reasons, it is of vital importance to allocate the first group of data *closest* to a physical core on which a given OpenMP thread or MPI process is executed. For the OpenMP version, achieving this goal is based on utilizing the first-touch policy with parallel initialization. For the MPI version, specifying the `alloc_shared_noncontig` info key enables to allocate the first group of data in noncontiguous memory regions, and as a result allow eliminating negative ccNUMA effects.

The noncontiguous memory allocation strategy also permits us to avoid replications of data of the first group between MPI processes. In contrast, we propose replicating the read-only data of the second group to expose their copies individually to each MPI process. Because of heterogeneity and fine-grain nature of these data, this replication data strategy definitely simplifies the structure of code at the negligible cost of extra memory consumption.

5.3 Synchronization

Besides solving issues of data parallelism, memory allocation and data sharing, the new version of MPDATA requires also providing an efficient synchronization mechanism. We solved [10] this issue for the OpenMP code by developing the

Listing 5.1. Structure of loop-level management of new 3D MPDATA implementation

```
for(...) // i-dim for sub-domains
  for(...) // j-dim for sub-domains
    for(...) // i-dim for MPDATA blocks
      for(...) // j-dim for MPDATA blocks
        for(...) // k-dim for MPDATA blocks
          {
          //Parallelization across cores
          for(...) // i-dim for sub-blocks of 1st kernel
            for(...) // j-dim for sub-blocks of 1st kernel
              //Vectorization
              for( ..) // k-dim for sub-blocks of 1st kernel
                /*.... Kernel 1 ...*/
          /*...  Synchronization Points  ...*/
          /*... and other MPDATA kernels ...*/
          }
```

custom mechanism for our data-flow strategy. This mechanism uses low-level compiler intrinsincs such as `fetch-and-add` instruction. This solution negatively affects the code portability across emerging compilers and CPU architectures, due to the need for validation of the correctness of code before its real use.

In contrast, the MPI-3 version delivers programming solutions that allow the portable implementation of the proposed synchronization strategy. This implementation is based on the non-blocking barrier `MPI_ibarrier` and corresponding `MPI_Waitall` routine used for the subsequent completion. Following the scheme of inter-core data traffic in the MPDATA application outlined in [10], the execution of computations by a given core depends on outcomes generated by two neighbor cores placed on its right and left sides (see also Fig. 1d). As a result, the data-flow strategy can be successfully implemented by starting the non-blocking synchronization for the left neighbor of every core, next for its right neighbor, and afterward waiting until all of the cooperated cores complete the synchronization operations identified by MPI requests (Listing 5.2).

The MPI SHM interface assumes also an explicit use of synchronization to ensure memory consistency, as well as the visible of changes in memory to the other processes [14]. In consequence, we select the passive target synchronization model, defined by the pair of `MPI_Win_lock_all` and `MPI_Win_unlock_all` functions. These functions specify the time interval, called an RMA access epoch,

Listing 5.2. A code snippet for MPI version of data flow strategy

```
MPI_Win_sync(MPDATA_Win_to_Sync);
MPI_Ibarrier(MPDATA_LEFT_MEMBERS_COMMUNICATOR, MPIreq+0);
MPI_Ibarrier(MPDATA_RIGHT_MEMBERS_COMMUNICATOR, MPIreq+1);
MPI_Waitall(2, MPIreq);
```

when memory operations are allowed to occur. Afterward, the `MPI_Win_sync` function has to be used to ensure completion of memory updates before using the `MPI_ibarrier` that synchronize all processes in time [14].

6 Benchmarking MPDATA Codes

We benchmark four versions of MPDATA: **(A)** basic, non-optimized implementation; **(B)** code with (3+1)D decomposition of MPDATA domain; **(C)** version B with data-flow synchronization; **(D)** version C with partitioning cores into independent work teams. All versions are implemented using both MPI and OpenMP shared memory programming. A series of experiments is performed on three shared memory ccNUMA platforms (Table 1). Among them are 2-socket servers with either Cascade Lake-SP (CLX-SP) or Skylake-SP (SKL-SP) Intel Xeon CPUs, and 4-socket server with Broadwell (BDW-EX) Intel Xeon CPUs. The MPDATA codes provide vector-friendly data structures that enable us to easy switch between AVX 2.0 and AVX-512, by setting a properly chosen compiler arguments [1]. All experiments are compiled using Intel compiler version 18.0.5 with the optimization flag -O3 and properly chosen compiler arguments for enabling auto-vectorization. The MPI codes are developed with Intel MPI Library 2018 Update 4. All tests are repeated 10 times, and average execution times are used to obtain statistically sound results, with the relative standard deviation (RSD) less than 1%.

Figure 2 depicts comparison of execution times (in seconds) for OpenMP and MPI codes of all MPDATA versions, achieved on three computing platforms outlined in Table 1 for the domain of size $2048 \times 1024 \times 64$. In addition, both OpenMP and MPI implementations of all MPDATA versions are compared for different sizes of domain. An example of such comparison is illustrated in Fig. 3.

Table 1. Specification of computing platforms (https://ark.intel.com)

Computing resources		2× Intel Xeon Platinum 8280L (CLX-SP)	2× Intel Xeon Platinum 8168 (SKL-SP)	4× Intel Xeon E7-8890v4 (BDW-EX)
Scalar/SIMD Turbo freq. [GHz]		3.3/2.4	3.4/2.5	2.6
Sockets		2	2	4
Cores/Threads		56/112	48/96	96/192
SIMD		AVX-512	AVX-512	AVX2 (256 bits)
Main memory		$2 \times 6 \times 16$ GB DDR4-2933	$2 \times 6 \times 16$ GB DDR4-2666	$4 \times 4 \times 16$ GB DDR4-2400
Memory bandwidth [MB/s]		281.5	255.9	204.8
Peak performance* [Gflop/s]	Scalar	369.6	326.4	499.2
	SIMD	2150.4	1920.0	1996.8

*Refers to multiplication instructions performed with Turbo frequency

The presented performance results correspond to the double precision floating point format, and 5000 time steps.

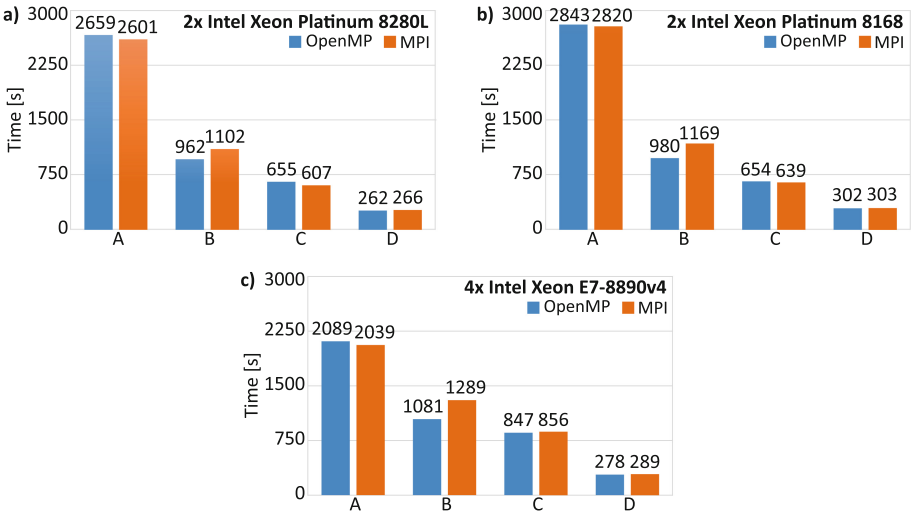

Fig. 2. Comparison of execution times of different MPDATA versions (A, B, C and D) achieved for both OpenMP and MPI, assuming the domain of size $2048 \times 1024 \times 64$, while using various computing platforms: a) 2× CLX-SP, b) 2× SKL-SP, and c) 4× BDW-EX

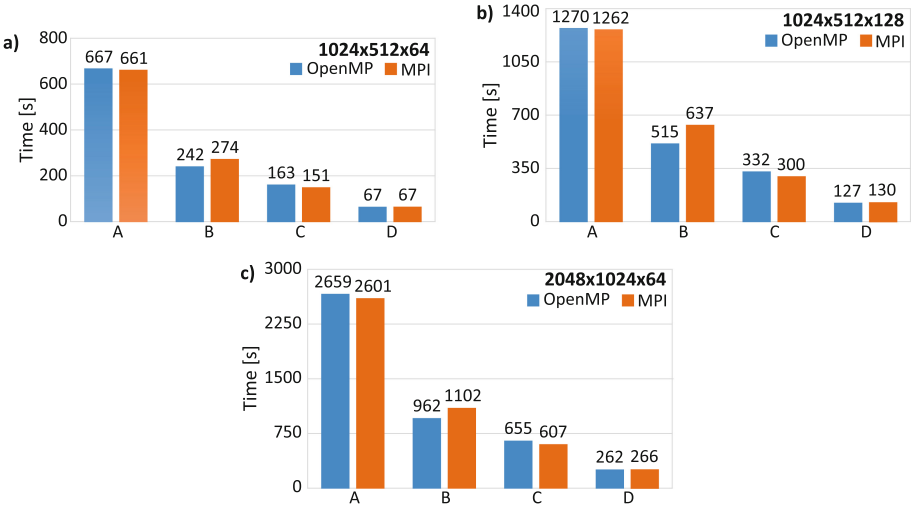

Fig. 3. Comparison of execution times of different MPDATA versions obtained for both OpenMP and MPI with various problem sizes on the platform equipped with two Intel Xeon Cascade Lake-SP CPUs (2× CLX-SP)

The benchmark results achieved for the first version (**(A)**) confirm a slightly high performance of the MPI code against the OpenMP implementation. This is an effect of overheads introduced by OpenMP runtime scheduling, while the MPI implementation from the beginning uses our scheduler that performs the loop distribution before computations.

In contrast, the OpenMP implementation of the version **B** returns better performance results for all performed tests. In fact, this benchmark reveals a negative impact of large number of synchronization points required by the (3+1)D decomposition of MPDATA [9] on the overall performance, with the MPI barrier resulting in greater performance losses than the OpenMP barrier.

The version **C** allows us to solve the synchronization issue for both MPI and OpenMP. As a result, the achieved performance is kept on a similar level for both programming models, with some advantage of MPI on the platforms with two CPUs. Finally, MPI and OpenMP implementations of the resulting version **D** feature practically the same performance, since the differences in the execution time between OpenMP and MPI models do not exceed 4% in favour of OpenMP.

7 Conclusions and Future Works

This paper demonstrates that the shared memory extension added in MPI-3 is efficient enough to take advantages of emerging multicore ccNUMA architectures. An example of such architectures is the newest Cascade Lake Intel Xeon Platinum 9282 processor, which packs two whole processors in a single socket offering 56 cores totally. Another remarkable example is the second generation of AMD EPYC processors, known as Rome. Using the multi-chip design with 4 modules interconnected via AMD Infinity Fabric, these emerging architecture is expected to deliver up to 64 cores per CPU.

The presented benchmarks show very similar performance results for both OpenMP and MPI shared memory implementations of the MPDATA CFD application on ccNUMA platforms with 2 and 4 CPUs. What is important is that MPI SHM delivers portable means to implement efficiently all steps of the parallelization methodology recently proposed for performance portable programming across multicore SMP/ccNUMA platforms. As a result, the resulting MPI code allows us to accelerate the MPDATA application more than 9 times as compared to the original version, achieving the sustained performance of 583 Glop/s for the server with two Cascade Lake Intel Xeon processors (each with 28 cores).

The aim of our future paper is to extend these results on the cluster level, in order to verify if heterogeneous MPI+MPI programming is able to successfully replace the common MPI+OpenMP hybrid programming model, providing portable application programming across forthcoming HPC platforms.

Acknowledgments. This research was supported by the National Science Centre (Poland) under grant no.UMO-2017/26/D/ST6/00687 and by the project financed within the program of the Polish Minister of Science and Higher Education under the name “Regional Initiative of Excellence” in the years 2019–2022 (project no.

020/RID/2018/19, the amount of financing 12 000 000 PLN). The authors are grateful to Intel Technology Poland for granting access to HPC platforms.

References

1. Szustak, L., Bratek, P.: Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern Intel processors. Int. J. High Perform. Comput. Appl. **33**(3), 507–526 (2019)
2. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, pp. 427–436 (2009)
3. Hoefler, T., et al.: MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory. Computing **95**(12), 1121–1136 (2013). https://doi.org/10.1007/s00607-013-0324-2
4. Rabenseifner, R., Hager, G., Blaas-Schenner, C., Reichl, R.: MPI+X—Introduction to Hybrid Programming in HPC (2019). https://moodle.rrze.uni-erlangen.de/course/view.php?id=388
5. Gropp, W.: MPI+MPI: Using MPI-3 Shared Memory As a Multicore Programming System. https://www.caam.rice.edu/mk51/presentations/SIAMPP2016_4.pdf
6. Brinskiy, M., Lubin, M., Dinan, J.: MPI-3 shared memory programming introduction. In: High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches, vol. 2, pp. 305–318. Morgan Kaufmann (2015)
7. Smolarkiewicz, P.K.: Multidimensional positive definite advection transport algorithm: an overview. Int. J. Numer. Methods Fluids **50**(10), 1123–1144 (2006)
8. Szustak, L., Rojek, K., Olas, T., Kuczynski, L., Halbiniak, K., Gepner, P.: Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. Sci. Program. **2015**, Article ID 642705, 14 p. (2015). https://doi.org/10.1155/2015/642705
9. Szustak, L., Halbiniak, K., Wyrzykowski, R., Jakl, O.: Unleashing the performance of ccNUMA multiprocessor architectures in heterogeneous stencil computations. J. Supercomput. **75**(12), 7765–7777 (2018). https://doi.org/10.1007/s11227-018-2460-0
10. Szustak, L.: Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems. J. Supercomput. **74**(4), 1534–1546 (2018)
11. Ouro, P., Fraga, B., Lopez-Novoa, U., Stoesser, T.: Scalability of an Eulerian-Lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelism. Comput. Fluids **179**(3), 123–136 (2019)
12. Yan, B., Regueiro, R.A.: Comparison between pure MPI and hybrid MPI-OpenMP parallelism for Discrete Element Method (DEM) of ellipsoidal and poly-ellipsoidal particles. Computat. Particle Mech. **6**(2), 271–295 (2018). https://doi.org/10.1007/s40571-018-0213-8
13. Karlbom, D.: A performance evaluation of MPI shared memory programming. Master's thesis, KTH, Sweden (2016)
14. Intel MPI Library Developer Guide for Linux OS, March 2019
15. OpenMP Application Programming Interface Version 5.0, November 2018
16. Intel C++ Compiler 19.0 Developer Guide and Reference, March 2019
17. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM International Conference on Supercomputing, pp. 207–216 (2015)