INSTITUTE OF MATHEMATICS

THE CZECH ACADEMY OF SCIENCES

Acceleration of a parallel BDDC solver by using GPUs on subdomains

Jakub Šístek
Tomáš Oberhuber

Preprint No. 54-2022

PRAHA 2022

Acceleration of a parallel BDDC solver by using GPUs on subdomains

Jakub Šístek[a,b], Tomáš Oberhuber[b]

[a] *Institute of Mathematics of the Czech Academy of Sciences, Czech Republic*
[b] *Czech Technical University in Prague, Czech Republic*

Abstract

An approach to accelerating a parallel domain decomposition (DD) solver by GPUs is investigated. The solver is based on the Balancing Domain Decomposition Method by Constraints (BDDC), which is a nonoverlapping DD technique. Two kinds of local matrices are required by BDDC. First, dense matrices corresponding to local Schur complements of interior unknowns are constructed by the sparse direct solver. These are further used as part of the local saddle-point problems within BDDC. In the next step, the local matrices are copied to GPUs. Repeated multiplications of local vectors with the dense matrix of the Schur complement are performed for each subdomain. In addition, factorizations and backsubstitutions with the dense saddle-point subdomain matrices are also performed on GPUs. Detailed times of main components of the algorithm are measured on a benchmark Poisson problem. The method is also applied to an unsteady problem of incompressible flow, where the Krylov subspace iterations are performed repeatedly in each time step. The results demonstrate the potential of the approach to speed up realistic simulations up to 5 times with a preference towards large subdomains.

Keywords: domain decomposition, BDDC, GPUs, MAGMA, TNL

1. Introduction

A large number of scientific and engineering simulations are based on solving of partial differential equations (PDEs). Consequently, many numerical methods suitable for their discretization have been developed, such as finite difference, finite volume, or finite element methods. Their efficient implementations suitable for high-performance computing and their updating hand-in-hand with changes in computer architectures enable mod-

Email addresses: sistek@math.cas.cz (Jakub Šístek), tomas.oberhuber@fjfi.cvut.cz (Tomáš Oberhuber)

els with unprecedented resolution to be solved through efficient utilization of the most advanced parallel supercomputers.

The Balancing Domain Decomposition by Constraints (BDDC) was introduced by Dohrmann (2003). Together with the Finite Element Tearing and Interconnecting - Dual/Primal (FETI-DP) method by Farhat et al. (2000), these methods can be considered as the most advanced nonoverlapping domain decomposition techniques. These methods aim at solving discretized PDEs by dividing the computational domain into smaller parts (subdomains) and combining local and global corrections in an iterative manner. Due to the large amount of local work, they are very suitable for parallel processing.

The underlying theory of BDDC was presented by Mandel and Dohrmann (2003), while Mandel et al. (2005) showed that the BDDC method is spectrally equivalent to the FETI-DP method. While BDDC can be used as a standalone solver, it is more common to use one step of the method as a preconditioner within a Krylov subspace method, e.g., the preconditioned conjugate gradient method (PCG) for symmetric positive definite (SPD) problems. The monograph by Toselli and Widlund (2005) provides details and analysis of many domain decomposition methods and iterative substructuring.

If the number of subdomains reaches thousands, the coarse problem of BDDC becomes a bottleneck of scalability, and it is beneficial to apply another step of BDDC on the coarse problem, giving rise to the three-level BDDC first introduced by Tu (2007). If the idea is repeated, we arrive at the multilevel BDDC by Mandel et al. (2008).

The multilevel BDDC method was combined with the adaptive selection of coarse unknowns and implemented in an open source parallel solver *BDDCML* by Sousedík et al. (2013). Pechstein and Dohrmann (2017) provide a recent overview of adaptive BDDC. Multilevel BDDC is seen as a method with the potential to scale to the largest supercomputers, as was demonstrated by Badia et al. (2016) for up to 0.5 million cores.

For more than a decade, the use of graphics processing units (GPUs) for scientific and engineering computations has become the main way to reach high performance on modern parallel computers. In particular, a majority of supercomputers, including those in the TOP500 list by Strohmaier et al. (2022), are accelerated by GPUs nowadays. For many of these supercomputers, a large part of the installed performance is due to these accelerators. Consequently, it is crucial for applications to utilize these accelerators for achieving high performance on these machines.

The field of numerical linear algebra has pioneered the creation of libraries that can be called from various applications through a well-defined interface. *LINPACK* by Dongarra et al. (1979) is a great example of an early library promoting modularity and standardized interface, later followed by *BLAS* by Dongarra et al. (1988b,a) and *LAPACK* by Anderson et al. (1999).

An early adopter of GPUs has been the *MAGMA* numerical library by Agullo et al. (2009) which implements a lot of the functionality of *BLAS* and *LAPACK* targeting GPUs. A modern numerical library targeting GPUs is the Templated Numerical Library (*TNL*) by Oberhuber et al. (2020). While offering a number of features for scientific computing, such as handling of structured and unstructured computational meshes, it also implements a subset of *BLAS* subroutines.

Despite the importance of employing GPUs for domain decomposition computations, the topic is not well covered by the existing literature. An exception aiming at the utilization of GPUs for a large portion of the FETI domain decomposition method was presented by Papadrakakis et al. (2011).

In this work, a different approach is followed. Namely, we investigate the potential of using GPUs via dense matrices of local Schur complements. Two local subdomain problems of BDDC are considered. The first is related to multiplication with local Schur complements, for which only a multiplication of a vector with a dense matrix is needed in each iteration of PCG. The second one requires a repeated solution with a saddle-point-type matrix in which the local Schur complement is wrapped by the matrix of constraints. In this operation, a backsubstitution is performed in each PCG iteration. Since these problems present a dominant cost of the BDDC method, they are good candidates for offloading their solution to GPUs in order to accelerate the overall computation.

2. The algorithm of BDDC

2.1. Iterative substructuring

In this paper, we focus on the standard (two-level) BDDC method for problems with a symmetric positive definite matrix. We stress the components of the BDDC algorithm relevant for subsequent processing by GPUs. In this scenario, a step of BDDC is used as a preconditioner for solving the problem reduced to the interface between subdomains.

Let $\Omega \subset \mathbb{R}^2$ or $\mathbb{R}^3$ be a bounded domain where we want to solve a linear partial differential equation, and let $\partial\Omega$ denote its boundary. In this paper, we will use the example of a Poisson problem,

$$-\Delta u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \partial\Omega, \tag{2}$$

where u is an unknown function and f is a prescribed right-hand side.

We aim at solving problem (1) by means of the finite element method (FEM), see, e.g., the monograph by Elman et al. (2005).

Domain Ω is divided into N nonoverlapping subdomains Ω_i for $i = 1, \ldots, N$. In finite element computations, the division into subdomains is

typically aligned with the elements of a computational mesh. Hence, subdomains can be seen as parts of the computational mesh.

In the i-th subdomain, a finite element function is determined by a sum of basis functions multiplied with unknown coefficients. Let us define the *interface* Γ as the set of unknown coefficients shared by more than one subdomain. This allows us to define a local interface of the i-th subdomain $\Gamma_i = \overline{\Omega_i} \cap \Gamma$. The remaining unknowns of the subdomain are called *interior* unknowns.

Let A_i be the local subdomain matrix obtained by the assembly of the local matrices of finite elements in Ω_i. Similarly, the local vector on the right-hand side f_i is obtained for problem (1) by integrating and assembling the contributions of finite elements in Ω_i.

The unknown coefficients of the local solution vector u_i can be divided into those belonging to Γ_i, denoted as u_i^Γ, and those in the interior of the subdomain u_i^I. Assuming that the interior unknowns are ordered first, the subdomain solution can be divided into blocks as $u_i = \begin{bmatrix} u_i^I u_i^\Gamma \end{bmatrix}^T$. Accordingly, the local matrix is blocked as

$$A_i = \begin{bmatrix} A_i^{II} & A_i^{I\Gamma} \\ A_i^{\Gamma I} & A_i^{\Gamma\Gamma} \end{bmatrix}. \tag{3}$$

Finally, the local right-hand side vector reads $f_i = \begin{bmatrix} f_i^I f_i^\Gamma \end{bmatrix}^T$. In the finite element context, matrix A_i is sparse.

The first step in *iterative substructuring* methods is a reduction of the problem to the interface Γ_i. Namely, we can construct the Schur complement of the local interior unknowns as

$$S_i = A_i^{\Gamma\Gamma} - A_i^{\Gamma I} \left(A_i^{II}\right)^{-1} A_i^{I\Gamma}. \tag{4}$$

The global Schur complement can be formally assembled as

$$S = \sum_{i=1}^{N} R_i^T S_i R_i, \tag{5}$$

where R_i is the 0–1 matrix that selects the unknowns of the local interface Γ_i from the global interface vector on Γ. However, the global Schur complement matrix is not assembled in implementations.

Let us denote the solution at the global interface u^Γ. In iterative substructuring, we first solve problem

$$S u^\Gamma = g, \tag{6}$$

where

$$g = \sum_{i=1}^{N} R_i^T \left(f_i^\Gamma - A_i^{\Gamma I} \left(A_i^{II}\right)^{-1} f_i^I \right) \tag{7}$$

is the reduced vector on the right-hand side.

Once this problem is solved, the interior unknowns are recovered in each subdomain by solving

$$A_i^{II} u_i^I = f_i^I - A_i^{I\Gamma} R_i u^\Gamma. \tag{8}$$

Problem (6) is solved by PCG. In this algorithm, only multiplications of the direction vectors by S are needed, which allows us to circumvent its construction by applying operators from the right-hand side of (5). In a standard implementation, even the construction of the local S_i is avoided and substituted by evaluating the right-hand side of (4). In this paper, we denote this approach as *implicit* multiplication with Schur complement, and compare it with an explicit construction of S_i, which is more amenable for acceleration by GPUs.

2.2. BDDC preconditioner

One step of the BDDC method is used as the preconditioner of problem (6) within PCG. The idea of the method is to define the *coarse* degrees of freedom at which we require the approximation of the solution to be continuous, whereas it is discontinuous at most of the interface. These coarse degrees of freedom can be solution values at selected nodes, called corners, and/or averages (arithmetic or weighted) at parts of the interface such as faces (interface unknowns shared by the same two subdomains) and edges (interface unknowns shared by the same more than two subdomains) of subdomains. In the presented computations, we use the values at corners and the arithmetic averages on faces and edges of subdomains as coarse degrees of freedom.

Let R_{Ci} be the 0–1 matrix that selects the coarse unknowns of subdomain Ω_i from the vector of global coarse unknowns. Locally, we define a matrix of constraints C_i, where the coarse degrees of freedom are defined in its rows. The number of columns corresponds to the number of unknowns on Γ_i.

The most demanding step in the BDDC setup is solving a saddle-point problem

$$\begin{bmatrix} S_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} \Psi_i \\ \Lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix}. \tag{9}$$

By its solution in each subdomain, we algebraically obtain the matrix of coarse basis functions Ψ_i, in which every column corresponds to a local coarse unknown.

As a side product, we also obtain a local subdomain matrix

$$S_{Ci} = \Psi_i^T S_i \Psi_i = -\Lambda_i, \tag{10}$$

from which the coarse problem matrix can be assembled as

$$S_C = \sum_{i=1}^{N} R_{Ci}^T S_{Ci} R_{Ci}. \tag{11}$$

Let us now describe the action of the BDDC preconditioner within the k-th PCG iteration, namely, producing a vector of the preconditioned residual u^k from the residual r^k. The first step is to obtain the local residual vector by

$$r_i = W_i R_i r^k, \tag{12}$$

where matrix W_i applies weights that satisfy the partition of unity, i.e., $\sum_{i=1}^{N} R_i^T W_i R_i = I$. The efficiency of the BDDC preconditioner depends largely on the choice of W_i, especially for problems with varying coefficients. However, in this paper, we use weights by cardinality, which is simply the inverse number of subdomains sharing the unknown.

The action of BDDC is generally split into a global coarse correction and local corrections, which are naturally parallel. Let us start with the coarse correction. The coarse residual is obtained as

$$r_C = \sum_{i=1}^{N} R_{Ci}^T \Psi_i^T r_i. \tag{13}$$

Then, we solve the coarse problem

$$S_C u_C = r_C, \tag{14}$$

we distribute the coarse solution to each subdomain, and we prolong it to the whole interface as

$$u_{Ci} = \Psi_i R_{Ci} u_C. \tag{15}$$

The subdomain problems are solved next. In particular, a saddle-point system

$$\begin{bmatrix} S_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} u_i \\ \mu \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix} \tag{16}$$

is solved in each subdomain. Here μ are Lagrange multipliers, and note that the matrix is the same as in (9). By solving this problem on each subdomain, we get the subdomain correction u_i.

At the end of the action of the BDDC preconditioner, we obtain the preconditioned residual u^k, $M_{BDDC} : r^k \to u^k$, by combining the subdomain corrections with the subdomain coarse solution as

$$u^k = \sum_{i=1}^{N} R_i^T W_i (u_i + u_{Ci}). \tag{17}$$

3. Implementation details

In this section, we describe the details of our implementation, which is based on the open source Adaptive-Multilevel BDDC solver *BDDCML* (version 2.6).

In the standard version of BDDC (here called *implicit*) implemented in the library, the local Schur complement matrix (4) is not constructed. Instead, the sparse direct solver *MUMPS* (version 5.3.3) is used to compute the Cholesky factorization of A_i^{II}, and a backsubstitution is performed in each S_i application following the right-hand side of (4).

In the approaches with explicit Schur complements, *MUMPS* is asked to construct the dense matrix of the Schur complement during this procedure, which makes the factorization more time and memory consuming but allows us to work with S_i further.

Hence, a dense matrix-vector product is performed in each iteration in each subdomain by multiplying a vector with S according to (5). This corresponds to the `dsymv` operation of *BLAS* Level 2, or the `dgemv` operation if we do not utilize the symmetry.

This operation can be accelerated by GPUs by transferring the matrix to the device memory. Then, only the vector to be multiplied and the resulting vector are transferred between the host memory and the device memory in each PCG iteration.

The explicit S_i is used further in (9). However, within the implicit version, a slightly modified matrix (with A_i in place of S_i and the matrices C_i extended by zeroes in the interiors) is factorized by the *MUMPS* solver using its LDL^T factorization. Nevertheless, with an explicit S_i, the whole matrix of (9) is considered and factorized as dense. In particular, the `dsytrf` function is used, and it is also compared with `dgetrf`, both from the LAPACK library. Problem (9) is then solved once as part of the preconditioner setup, while problem (16) is solved in each action of the preconditioner. These operations are performed using the `dsytrs` or the `dgetrs` LAPACK routines.

On CPUs, we perform the operations with the dense matrices using the Intel Math Kernel Library (*MKL*, version 2019, update 2).

On GPUs, we use the analogues of the *BLAS* and *LAPACK* functions in the *MAGMA* library (version 2.6.1). Namely, we use the `magma_dsymv` and `magma_dgemv` functions for multiplication with the Schur complement and the `magma_dgetrf_gpu` and `magma_dgetrs_gpu` functions for factorization and backsubstitution within the BDDC preconditioner.

Finally, we have also prepared an interface to the Templated Numerical Library (*TNL*), version from the `develop` branch from July 2021. It is used just for the matrix-vector multiplication, namely the `gemv` function.

4. Results

In this section, we evaluate the effect of the GPU acceleration on the overall performance of the BDDC solver. We aim at accelerating two operations, which present the main cost of a typical BDDC run, namely (i) the multiplication of a vector with the local Schur complement S_i, and (ii)

the solution of the subdomain problem (16). The considered versions of the solver are described in detail in the previous section and summarised in Table 1.

version	Solver setup		PCG iterations	
	Schur compls. (4)	Augmented probls. (9)	Schur compls. (5)	Augmented probls. (16)
Implicit	*MUMPS* LL^T	*MUMPS* LDL^T	*MUMPS* solve	*MUMPS* solve
MKL	*MUMPS* LL^T + dense Schur	`dsytrf`	`dsymv`	`dsytrs`
MKL getrf	*MUMPS* LL^T + dense Schur	`dgetrf`	`dsymv`	`dgetrs`
MAGMA	*MUMPS* LL^T + dense Schur	`magma_dgetrf_gpu`	`magma_dsymv`	`magma_dgetrs_gpu`
TNL	*MUMPS* LL^T + dense Schur	`magma_dgetrf_gpu`	*TNL* `dgemv`	`magma_dgetrs_gpu`

Table 1: Summary of solver configurations.

The multiplication of a vector by the local Schur complement $S_i p_i$. is a part of the application of the global Schur complement matrix in (5). In the version denoted as *implicit*, this is performed by evaluating the right-hand side of (4), that is, by evaluating three sparse matrix-vector products with matrices $A_i^{\Gamma\Gamma}$, $A_i^{\Gamma I}$, and $A_i^{I\Gamma}$. The application of $\left(A_i^{II}\right)^{-1}$ is performed by backsubstitution in the *MUMPS* direct solver.

If we let *MUMPS* construct the dense matrix S_i, we can use several numerical libraries to multiply a dense matrix with a vector. In particular, we consider the CPU implementation from *MKL* (`dsymv`) and two implementations for GPUs. Namely, we use *MAGMA* (`magma_dsymv`) and the implementation from *TNL*.

Solution of the subdomain problems (16).. First, the factorization of the (dense) matrix of problem (9) is created. It is natural to exploit the symmetry of the matrix by performing the LDL^T factorization, which corresponds to the `dsytrf` function of *MKL*. However, we also compare it with the *LU* factorization with partial pivoting in the `dgetrf` implementation. The backsubstitution is performed by the `dsytrs` and the `dgetrs` routines, respectively.

For GPUs, we use the *LU* version in the *MAGMA* library (`magma_dgetrf_gpu`), with the backsubstitutions performed by `magma_dgetrs_gpu`.

The aim of our numerical tests is two-fold. First, we compare the behaviour of the BDDC algorithm with the *implicit* and *explicit* local Schur complements on CPUs. In the second step, we can evaluate the benefits of employing GPUs for performing the operations with dense matrices in the *explicit* version.

The experiments have been performed on the computational cluster of the Research Center of Informatics in Prague. The cluster has 12 accelerated nodes, each equipped with 36 cores (2 x Intel Xeon Scalable Gold 6150) with the frequency 2.7GHz, 4 nVidia Tesla V100 GPUs, and 384 GB RAM.

4.1. Benchmark problem

We start with the widely used benchmark of the Poisson problem on a unit cube. More specifically, Ω is a unit cube and $f = 1$ in (1). The domain is divided into $N_E \times N_E \times N_E$ cubic subdomains, with a characteristic size of an edge denoted H. Each subdomain is composed of $n_E \times n_E \times n_E$ elements with cubic shape and tri-linear shape functions. The size of the edge of an element is $h = 1/(N_E n_E)$.

An important parameter of the problem for the performance of the BDDC method is $n_E = H/h$. It is known from the BDDC theory by Mandel and Dohrmann (2003) that the condition number of the operator preconditioned by BDDC is proportional to $(1 + \log^2(H/h))$. This means that convergence of BDDC is independent of the global size of the problem while it depends on the local subdomain size H/h.

We perform computations with a fixed number of subdomains, $4 \times 4 \times 4 = 64$, and for growing subdomain sizes, $H/h \in \{5, 10, 15, 20, 25, 30, 35, 40, 45\}$. This corresponds to global problem sizes from 9.3 thousand to 5.9 million degrees of freedom, with subdomain local sizes from 216 to 97 thousand degrees of freedom in volume and from 152 to 12 thousand unknowns on local subdomain interfaces. The latter corresponds to the size of the subdomain Schur complement. The problems are computed on 1 and 2 computational nodes, with 16 and 32 MPI processes, respectively. For one node, there are 4 processes and 16 subdomains per GPU, whereas in the second scenario, there are 4 processes with 8 subdomains per GPU.

In Fig. 1, we present the number of PCG iterations with respect to the H/h ratio. It can be seen that it behaves as expected from the BDDC literature.

The whole solution process is broken down into several important parts, for which the time is measured separately. A hierarchical scheme of the solver is presented in Fig. 2, while the detailed time measurements are presented in Figs. 3–10. Note that the higher-level components also include the times of the lower-level ones to evaluate the overall benefits. The plots are ordered according to the scheme in Fig. 2. We also include the relative amount of time spent in the block for the *implicit* version with $H/h = 45$ and one computational node.

In Fig. 3, we present the whole computational time required to solve one problem. We can see that for small subdomains (small H/h), the fastest solution is delivered by using the dense matrices processed on CPUs by *MKL*, while the overhead of the GPU processing penalizes these versions. Nevertheless, for large subdomain sizes, the *implicit* Schur complement handling starts to be the most efficient, followed by GPU-accelerated versions. We can also see that all versions scale well when moving from one to two computational nodes.

Having a look at Fig. 4, we can see that the setup of the solver is responsible for most of the overall time. With the only difference for small

subdomain sizes where the *implicit* version is equally fast as the CPU version with dense matrices, the *implicit* version is the fastest.

In Fig. 5, we plot the times for preparing the Schur complement matrix. From this plot, we can evaluate the cost of running the Cholesky factorization by the *MUMPS* solver on matrices A_i^{II} of problem (4). For the *implicit* variant, this is just the factorization of A_i^{II}. For the other versions, this includes the overhead of constructing the dense Schur complement S_i by *MUMPS*, which seems to be about 3–4 times more expensive. In addition, the GPU versions also include the transfer of the matrix S_i to the device memory of GPUs, which seems to be slightly more efficient in *TNL* than in *MAGMA*.

Let us now look at the times for the other significant part of the solver setup, namely, the BDDC preconditioner setup, which is dominated by the factorization of the matrix of the augmented problem (9) presented in Fig. 6. In the *implicit* version, this corresponds to an independent instance of the *MUMPS* solver performing a sparse LDL^T factorization, whereas it is an LDL^T or LU factorization of a dense matrix for the other versions. Without GPU acceleration, it is faster to do this by the *implicit* version. However, here we can see a benefit of using the GPU version from the *MAGMA* library, which provides factorization about 5 times faster than *MKL*. Note that also the *TNL* version relies on the *MAGMA* library for this operation, since matrix factorizations are not implemented in TNL.

The factorization is followed by backsubstitution with several right-hand sides to solve problem (9). Although this is a fast operation, we can see from Fig. 7 that using the LU factorization is faster than the LDL^T factorization of *MKL*, as it gets comparable to the *implicit* version, and the *MAGMA* version is the fastest for this operation.

Let us now turn to the second part of the solution process, that is, the time spent in the PCG iterations shown in Fig. 8. Note that this time also includes the effect of slightly increasing number of iterations, see Fig. 1. In these times, we can appreciate the effect of GPU acceleration, the trends of which are very different from the CPU versions. For small subdomains, the CPU versions are faster, whereas from H/h around 30, the GPU versions become beneficial, and for subdomain size $H/h = 45$, the *MAGMA* version becomes about 4 times faster than the CPU versions, whereas the *TNL* version is slightly slower.

These effects can be seen in more detail on time for one application of the Schur complement in Fig. 9, where we can see that the time for *MAGMA* is essentially independent of the subdomain size, with *MKL* being about 30% faster than the *implicit* version. The *TNL* version is slightly slower than *MAGMA* for large subdomains. Finally, we have a look at the time for backsubstitution in problem (16) in Fig. 10. Here, the *MAGMA* version is faster than the CPU versions for subdomains with H/h larger than 35 for one node and 30 for two nodes, while the CPU versions are comparable.

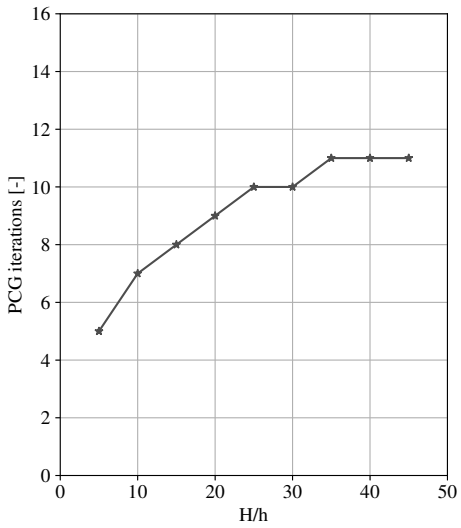

Figure 1: Dependence of the number of PCG iterations on H/h.

Due to the speed of this operation, we do not see scaling when going from 1 to 2 nodes, which is likely caused by the communication overhead.

To summarize this experiment, we can see that the *implicit* version is faster in the setup of the solver, whereas explicit Schur complements and especially their acceleration by GPUs become beneficial for the Krylov iterations with large subdomains.

In order to appreciate the domain decomposition strategy, we also include the time required for the solution by the distributed sparse direct solver *MUMPS*. In this case, this direct solver is run on the global distributed problem rather than on the subdomain blocks. It is compared with the *implicit* variant on two nodes and 64 processes in Fig. 11. In this case, we can clearly observe the benefits of the iterative strategy in the domain decomposition approach, which provides more than 10x speed-up compared to the direct approach for the largest problem with $H/h = 45$.

4.2. Application to computational fluid dynamics

The goal of this section is to explore whether the results of the benchmark problem can be used to accelerate the solution of real-life problems. We have chosen the problem of unsteady incompressible flow around rigid bodies, in which a Poisson problem for pressure is repeatedly solved in each time step. Here, we apply the solver to the benchmark problem of flow around a unit sphere with Reynolds number 300 inspired by Emblemsvåg et al. (2005). In this regime, the flow develops into a periodic vortex shedding, see Fig. 12. The problem has been solved by the pressure correction approach and parallel one-level domain decomposition solvers from the PETSc library by Šístek

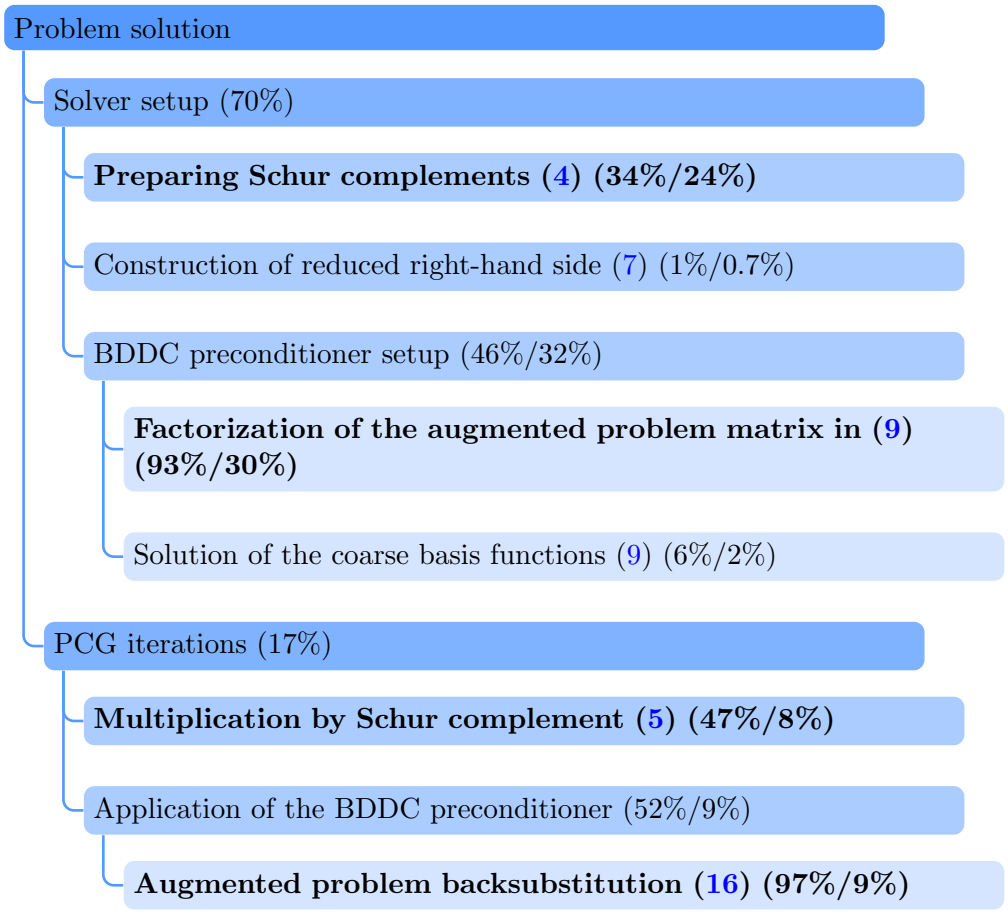

Figure 2: Scheme of the BDDC algorithm with a percentage of computational time of each block relative to the higher block and to the whole problem solution.

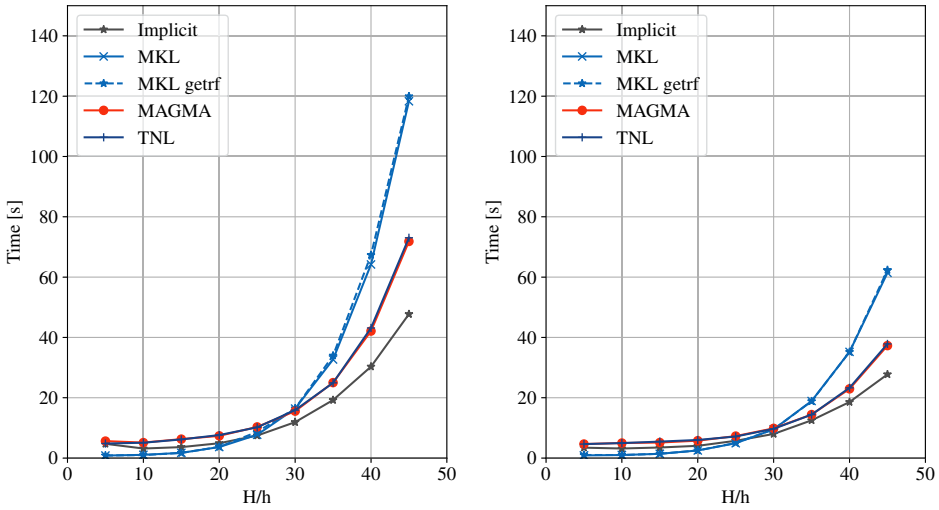

Figure 3: Problem solution time on 1 node (left) and 2 nodes (right).

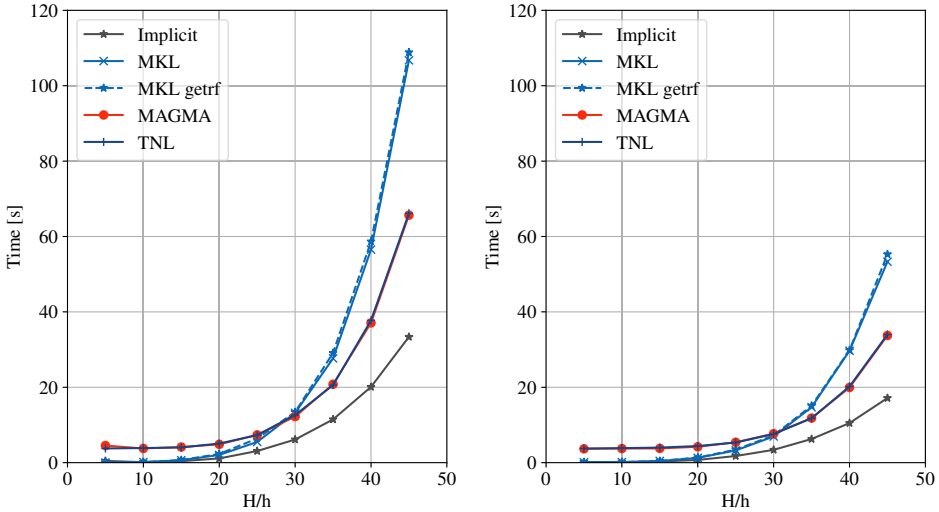

Figure 4: Solver setup on 1 node (left) and 2 nodes (right).

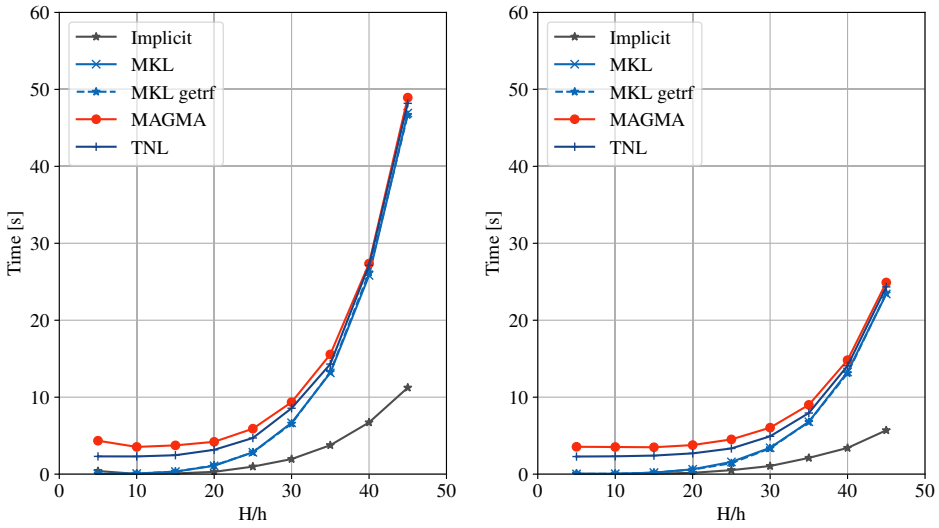

Figure 5: Preparing Schur complements on 1 node (left) and 2 nodes (right).

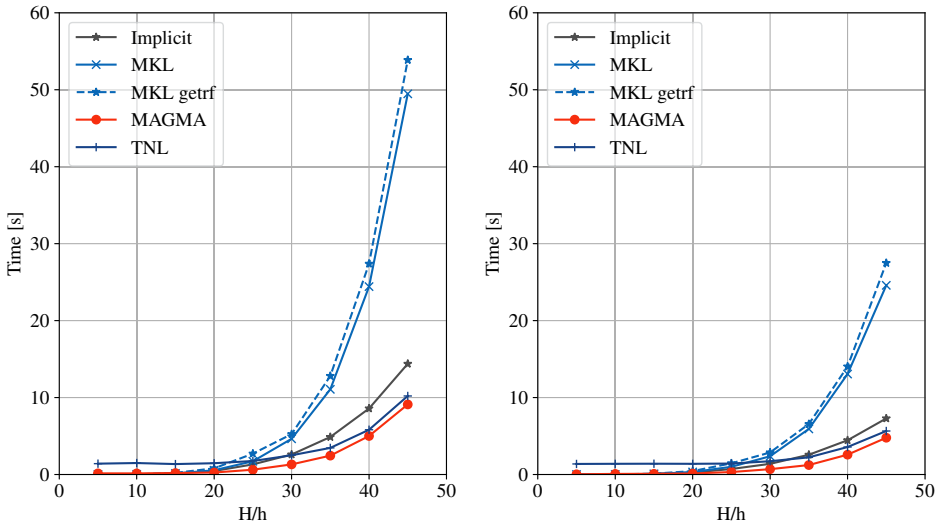

Figure 6: Factorization of the augmented problem on 1 node (left) and 2 nodes (right).

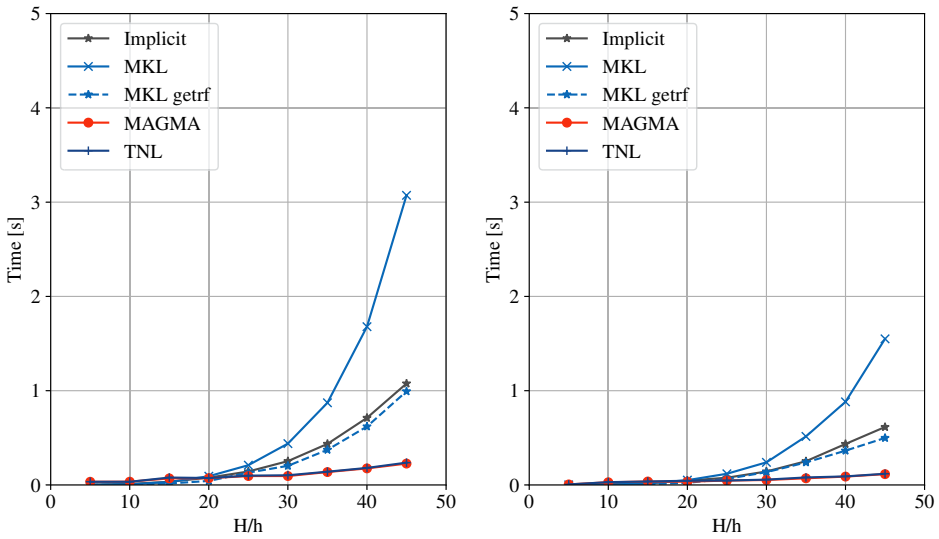

Figure 7: Solution of the coarse basis functions on 1 node (left) and 2 nodes (right).

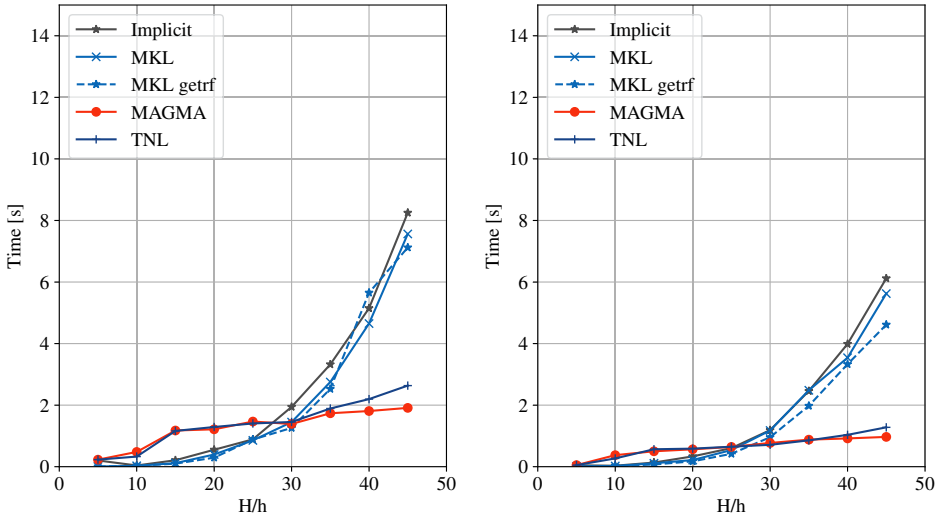

Figure 8: PCG iterations on 1 node (left) and 2 nodes (right).

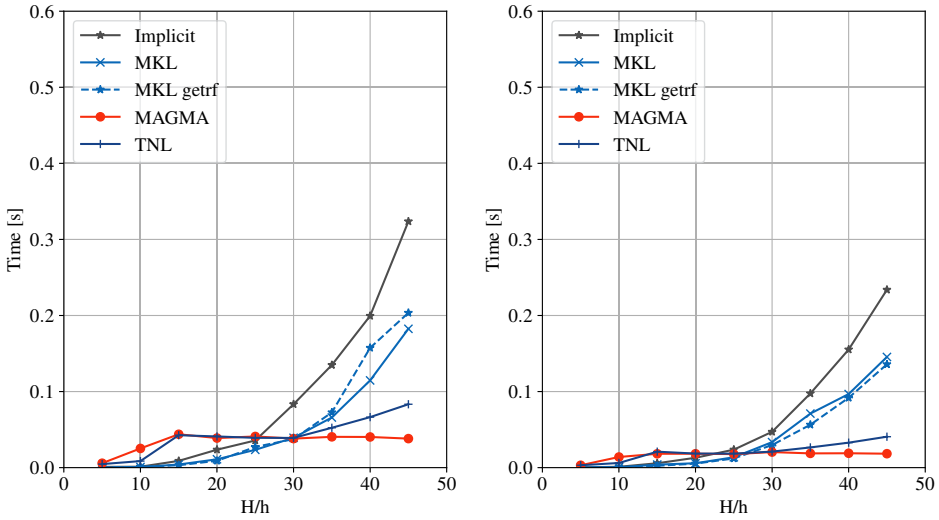

Figure 9: Schur complements multiplication with a vector on 1 node (left) and 2 nodes (right).

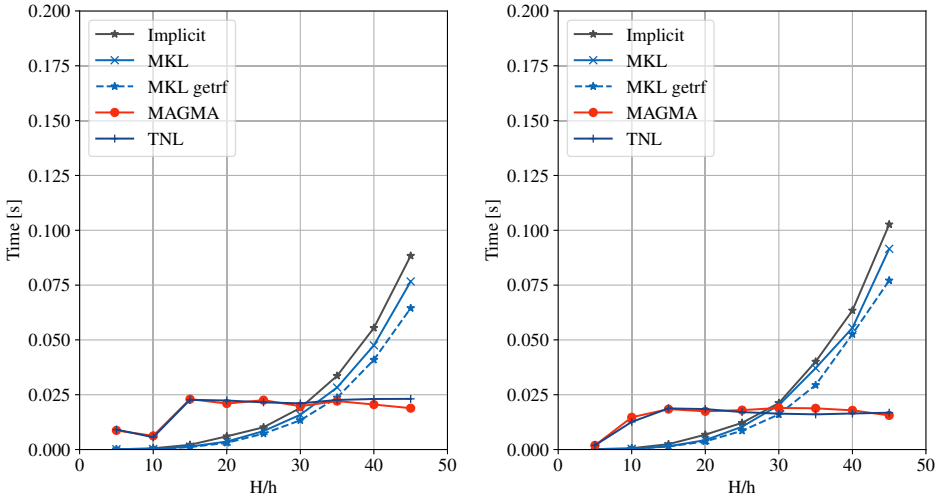

Figure 10: Augmented problem backsubstitution on 1 node (left) and 2 nodes (right).

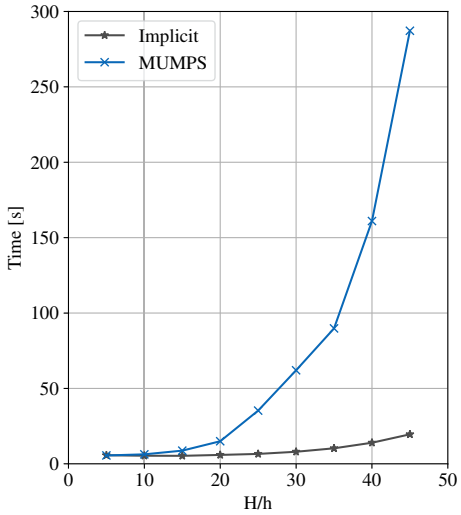

Figure 11: Comparison of the problem solution time of the distributed sparse direct solver (*MUMPS*) with the BDDC approach on two computational nodes and 64 MPI processes.

(2015). Recently, we have solved the Poisson problem of pressure by the multilevel BDDC method (Hanek and Šístek (2021)). In that reference, we have also studied the benefits of using various variants of BDDC for the arising sequence of problems. The goal of this paper is to evaluate the effect of GPU acceleration.

When the problem is solved by the finite element method, the matrix of the Poisson problem is fixed for all time steps, and only the right-hand side vector changes with time. Hence, most of the solver setup is performed just once for the whole sequence of time steps, and only the reduced right-hand side vector (7) needs to be constructed repeatedly. Then, of course, the PCG iterations are rerun in every time step.

This is a plausible use case for the GPU acceleration of the solver, since the greatest potential for acceleration has been observed for the Krylov iterations part in the previous section.

In this example, we no longer distinguish between the time of setup and the time of PCG solution, and we measure only the *problem solution time*. Nevertheless, we do differentiate between the time for the first time step, in which the complete solver setup is included, and the times of other time steps, which is an average time over 30 time steps. It has been shown by Hanek and Šístek (2021) that the number of iterations is almost constant throughout the time steps, which justifies the analysis of the initial 30 time steps rather than the 4000 time steps needed for the whole simulation.

The Poisson problem for pressure has 220 thousand finite elements with 225 thousand unknowns. We again perform the experiment for a variable subdomain size. This is in this case controlled by the number of subdomains,

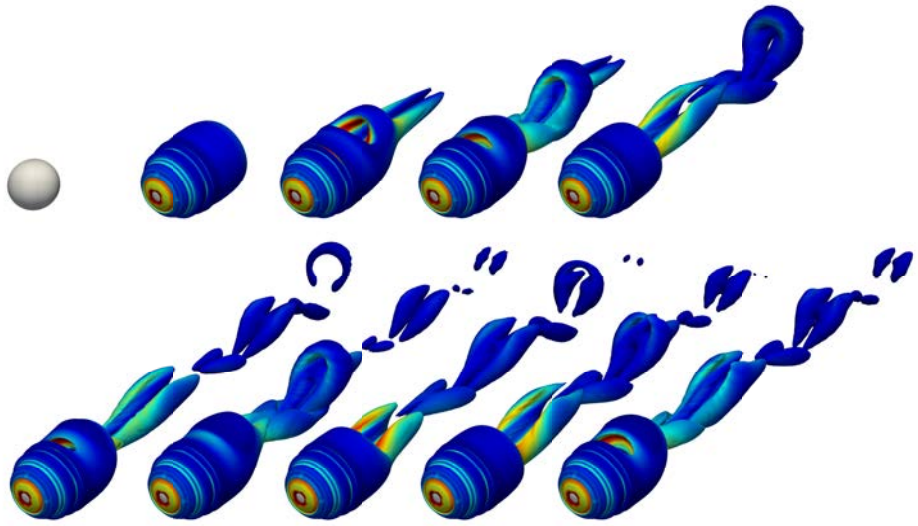

Figure 12: Incompressible flow around a unit sphere with Reynolds number 300. Vortices in the wake of the sphere are visualized by the magnitude of the vector of average corotation by Kolář et al. (2013) (isovalue 0.1) and coloured by the magnitude of vorticity. Flow after time in seconds: 0, 50, 70, 75, 80 (top line), 90, 100, 110, 150, and 200 (bottom line).

which is the same as the number of MPI processes in this case.

For this comparison, we select only the *implicit* version of the solver and the version accelerated by the *MAGMA* library. These versions are compared on one and two computational nodes, hence using 4 and 8 GPUs.

Figure 13 presents the resulting times. Let us first look at the left part of the figure with times for the first time step. We can clearly observe very different behaviour for both versions. While the *implicit* version enjoys a reasonable strong scalability for up to 16 subdomains, after which the subdomains become too small to scale further, the scalability of the setup of the *MAGMA* version worsens for more (and smaller) subdomains. This could be partly caused by the fact that the node has only 4 GPUs, and for smaller subdomains, more of them are assigned to each GPU. It is worth mentioning though that for four subdomains, which is the case of the largest subdomains, the *MAGMA* version gets even a bit faster than the *implicit* one. The scalability improves with employing two computational nodes with 8 GPUs. It can be seen that the cost of the first time step drops significantly, although it does not halve the time, as only a part of the solver setup is offloaded to GPUs. We can also observe that the accelerated version reduces time with the number of subdomains. However, its scalability stops much earlier than for the *implicit* version.

The right part of Fig. 13 presents the average times for the other time steps. Note that the time axis is 10 times finer than for the left plot. Also in

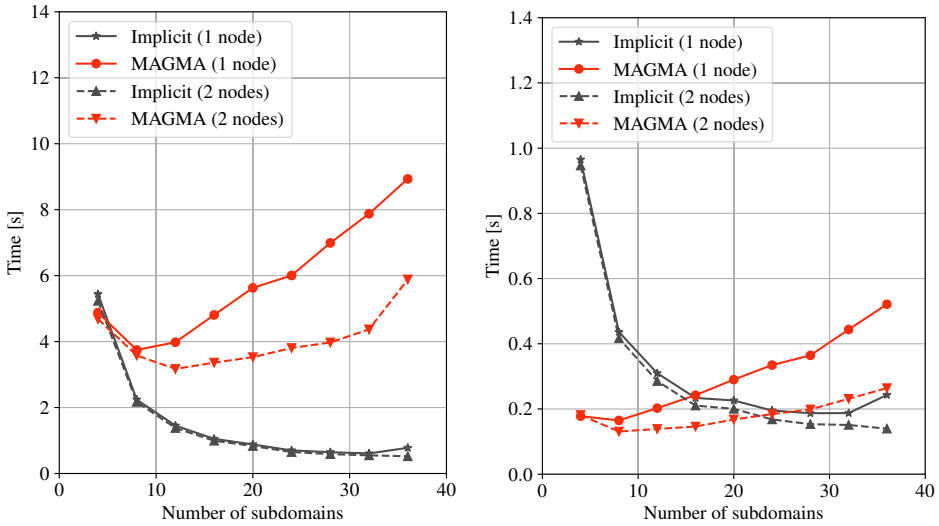

Figure 13: Solution times for the sequence of pressure Poisson problems. Time for the first time step (left) and the mean time for the next 30 time steps (right) on one and two computational nodes.

this figure, we can observe the largely different behaviour of the *implicit* and the *MAGMA* versions. While the *implicit* version provides a better strong scalability, the *MAGMA*-accelerated version is very attractive for larger subdomains (the case with a small number of subdomains). In particular, it is about 5 times faster than the implicit version for 4 subdomains and about 3 times faster on 8 subdomains. When moving from one to two computational nodes, we can also see a difference in time of the *implicit* (CPU) version. This is probably related to a faster access to memory when only half of the processes are placed at each node. For the accelerated version, we can see a significant drop of the time with increasing the number of subdomains, which indicates that the overhead of the GPU computation reduces with assigning less subdomain problems to each GPU. Naturally, this effect can be seen from eight subdomains, which corresponds to employing all GPUs available on the two nodes.

5. Conclusions

We have presented a minimalistic approach to accelerating a BDDC solver by GPUs. It is based on the explicit construction of the dense Schur complement matrix by the sparse direct solver, followed by its offloading to the GPU memory and using it for multiplying with a vector in the PCG method. The dense Schur complement is also used for constructing the dense saddle-point matrix of the subdomain problems, which is also offloaded to GPUs, where its factorizations and backsubstitutions are performed. Al-

though most of the BDDC algorithm still resides on the CPUs, the two offloaded operations are responsible for the majority of the solution time of a BDDC solver.

As a baseline for comparisons, we use the *implicit* version of the BDDC solver without forming an explicit Schur complement and relying on sparse direct solvers. For the version with explicit Schur complements, we first compare the *implicit* version with processing the dense matrices on CPUs by *MKL*. This includes multiplying a vector by the dense matrix on each subdomain and a factorization with repeated backsubstitution (either LU or LDL^T). The dense matrix operations were further assigned to the *MAGMA* library, whereas the *TNL* has also been evaluated for multiplying with the local Schur complements. Both libraries have used *CUDA* for nVidia GPUs.

The experiments have been performed on one and two cluster nodes with Intel CPUs and 4 nVidia Volta GPUs on each node.

The approaches have been first evaluated on a benchmark problem of a Poisson problem on a unit cube with varying subdomain size. A detailed time measurement has shown that while the setup phase of the preconditioner does not benefit from this GPU acceleration, the part with PCG iterations can benefit significantly from this approach, with benefits increasing with the subdomain size. We have also shown the benefits of using the DD strategy compared to a distributed sparse direct solver MUMPS used on the global problem.

Our observations were applied to a real-life Poisson problem of pressure within a numerical simulation of an unsteady incompressible flow around a sphere. In these simulations, the setup of the solver is performed just once, while the PCG iterations are repeatedly run in each time step. These results also confirm that this approach to GPU acceleration is beneficial for large subdomains, with up to 5-fold speedup for the largest subdomains. However, it should be emphasized that these are still realistic subdomain sizes that are often used in the applications of domain decomposition methods. We have also observed the benefit of employing more accelerated nodes, although the positive effect is stronger for smaller subdomains.

To summarize our findings, the potential of accelerating the subdomain operations through dense linear algebra and numerical libraries for GPUs has been evaluated. While it is not beneficial for small subdomains, where the overhead outweighs the larger performance of GPUs, it provides an interesting option for medium and larger subdomains, for which the computational times can be reduced up to 5 times. This applies to a repeated application of the Krylov subspace method with a fixed matrix and the preconditioner already residing on GPUs. Nevertheless, as all the approaches with an explicit construction of the dense Schur complement make the setup of the solver significantly more expensive, it does not seem beneficial to use this approach to GPU acceleration for problems in which the solution is performed just once.

Nevertheless, our study has provided an insight into what kind of problems may benefit from the presented approach. Performing a similar study with GPU accelerators from the latest generation by nVidia and AMD and studying the behaviour on a large cluster with GPU-accelerated nodes will be the subject of a future study.

Acknowledgements

This research was supported by the Ministry of Education, Youth and Sports of the Czech Republic (MEYS) under the OP RDE grant number CZ.02.1.01/0.0/0.0/16_019/0000765: Research Center for Informatics. It has also been supported by the Czech Science Foundation through grant 20-01074S and by the Czech Academy of Sciences through RVO:67985840.

References

Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P and Tomov S (2009) Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In: *Journal of Physics: Conference Series*, volume 180. IOP Publishing, p. 012037.

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide*. Third edition. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 0-89871-447-8 (paperback).

Badia S, Martín AF and Principe J (2016) Multilevel balancing domain decomposition at extreme scales. *SIAM J. Sci. Comput.* 38(1): C22–C52.

Dohrmann CR (2003) A preconditioner for substructuring based on constrained energy minimization. *SIAM J. Sci. Comput.* 25(1): 246–258.

Dongarra JJ, Croz JD, Hammarling SJ and Hanson R (1988a) Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 14: 18–32.

Dongarra JJ, Croz JD, Hammarling SJ and Hanson R (1988b) An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software* 14: 1–17.

Dongarra JJ, Moler CB, Bunch JR and Stewart GW (1979) *LINPACK Users' Guide*. SIAM.

Elman HC, Silvester DJ and Wathen AJ (2005) *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. New York: Oxford University Press.

Emblemsvåg JE, Suzuki R and Candler GV (2005) Cartesian grid method for moderate-Reynolds-number flows around complex moving objects. *AIAA Journal* 43(1): 76–86.

Farhat C, Lesoinne M and Pierson K (2000) A scalable dual-primal domain decomposition method. *Numer. Linear Algebra Appl.* 7: 687–714.

Hanek M and Šístek J (2021) Application of Multilevel BDDC to the problem of pressure in simulations of incompressible flow. In: *Domain Decomposition Methods in Science and Engineering XXVI*, Lecture Notes in Computational Science and Engineering. Springer. To appear.

Kolář V, Šístek J, Cirak F and Moses P (2013) Average corotation of line segments near a point and vortex identification. *AIAA J.* 51(11): 2678–2694.

Mandel J and Dohrmann CR (2003) Convergence of a balancing domain decomposition by constraints and energy minimization. *Numer. Linear Algebra Appl.* 10(7): 639–659.

Mandel J, Dohrmann CR and Tezaur R (2005) An algebraic theory for primal and dual substructuring methods by constraints. *Appl. Numer. Math.* 54(2): 167–193.

Mandel J, Sousedík B and Dohrmann CR (2008) Multispace and multilevel BDDC. *Computing* 83(2-3): 55–85.

Oberhuber T, Klinkovský J and Fučík R (2020) TNL: Numerical library for modern parallel architectures. *Acta Polytechnica* 61(SI): 122–134. DOI: 10.14311/AP.2021.61.0122.

Papadrakakis M, Stavroulakis G and Karatarakis A (2011) A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering* 200(13): 1490–1508. DOI:10.1016/j.cma.2011.01.013.

Pechstein C and Dohrmann CR (2017) A unified framework for adaptive BDDC. *Electron. Trans. Numer. Anal.* 46: 273–336.

Šístek J (2015) A parallel finite element solver for unsteady incompressible Navier-Stokes equations. In: Šimurda D and Bodnár T (eds.) *Proceedings of Topical Problems of Fluid Mechanics 2015*. Institute of Thermomechanics AS CR, pp. 193–198.

Sousedík B, Šístek J and Mandel J (2013) Adaptive-Multilevel BDDC and its parallel implementation. *Computing* 95(12): 1087–1119.

Strohmaier E, Dongarra J, Simon H and Meuer M (2022) Top500 list. URL https://www.top500.org.

Toselli A and Widlund OB (2005) *Domain Decomposition Methods—Algorithms and Theory*, *Springer Series in Computational Mathematics*, volume 34. Berlin: Springer-Verlag.

Tu X (2007) Three-level BDDC in three dimensions. *SIAM J. Sci. Comput.* 29(4): 1759–1780.