Project Title	SCAlable LAttice Boltzmann Leaps to Exascale
Project Acronym	SCALABLE
Grant Agreement No.	956000
Start Date of Project	01.01.2021
Duration of Project	36 Months
Project Website	www.scalable-hpc.eu

D4.1
Initial Report on Application Optimisation

Work Package	WP 4: Performance Engineering - Hardware/Middleware
Lead Author	Jayesh Badwaik(Forschungzentrum Jülich)
Contributing Authors	Jayesh Badwaik, Markus Holzer, Milena Veneva, Romain Cuidard, Kristian Kadlubiak
Reviewed By	Harald Köstler, Alois Sengissen
Due Date	01.01.2021
Date	14.06.2022
Version	1.0

Dissemination Level

☒ PU: Public

☐ PP: Restricted to other programme participants (including the Commission)

☐ RE: Restricted to a group specified by the consortium (including the Commission)

☐ CO: Confidential, only for members of the consortium (including the Commission)

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Deliverable Information

Deliverable	D4.1
Deliverable Type	Report
Deliverable Title	Initial Report on Application Optimisation
Keywords	
Dissemination Level	Public
Work Package	WP 4: Performance Engineering - Hardware/Middleware
Lead Partner	Forschungzentrum Jülich
Lead Author	Jayesh Badwaik
Contributing Authors	Jayesh Badwaik, Markus Holzer, Milena Veneva, Romain Cuidard, Kristian Kadlubiak
Reviewed By	Harald Köstler, Alois Sengissen
Due Date	01.01.2021
Planned Date	01.02.2021
Version	1.0
Final Version Date	14.06.2022

Disclaimer:

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of the SCALABLE partners nor of the European Commission.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Initial Report on Application Optimisation

Jayesh Badwaik, Markus Holzer, Milena Veneva,
Romain Cuidard, Kristian Kadlubiak

August 1, 2022

The report describes initial experiments with performance optimizations for LaBS and Walberla codes. In particular, we describe the optimizations in communication algorithms for LaBS and kernel launch optimizations for Walberla.

1 Introduction

One of the objectives of the SCALABLE project is to prepare LBM applications for exascale architectures. In particular, the objective is to prepare Widely Applicable Lattice Boltzmann from Erlangen (waLBerla) and LaBS for exascale architecture. For the purpose of initial steps in optimization, the strategy was to continue two different tasks in parallel, benchmarking and profiling the use cases of the project which is done in D2.2 and exploring different techniques that might be used for optimization the use cases later on, which is done in this document.

In particular, for LaBS, we explore the possibility of reducing time consumption in communication and synchronization for moving domains and domains with multiple resolution. In Walberla, we explore reducing the overhead of launching GPU kernels by exploring the use of CUDA Task Graph.

2 LaBS

2.1 Introduction

LaBS is divided on several step as described in Figure 1 a lot of those steps are used for generating a mesh and preparing the computation. Solver is the main step with all the physical computation.

For the industrial test cases, the solver step take more than 90 % of the simulation time. So in the first time our objective is to focus on the scalability (and performance in general) of that step. We will be working on other steps only if they are blocking the simulation.

2.2 Reduction of Wait Time for Moving Region

2.2.1 Introduction

In LaBS, usecases with moving region suffer of heavy time consumption in MPI communication and synchronisation in compare to standard cases.

In the LaBS, moving regions are limited to rotation invariant domains, so pre-computations are done on a complete rotation. During a rotation, a *border mobile* node will need information from many *fixed* nodes. Those *fixed* nodes can belong to several different processors, so each *border mobile* node can generate a lot of MPI exchanges. The wider the moving regions is and the more there is an MPI process, the more the number of MPI exchanges is important and thus the time passed in the MPI communication is important. This behaviour is not good for scalability.

2.2.2 Solution

For a given time, a *border mobile* node doesn't need to perform MPI exchanges with all nodes sweep during the complete rotation, and it's too expensive to determine for each time step which processor holds the information needed. So our solution is to slice the rotation. In the current implementation a rotation is sliced every 256 time intervals. The number 256 comes from empirical studies.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

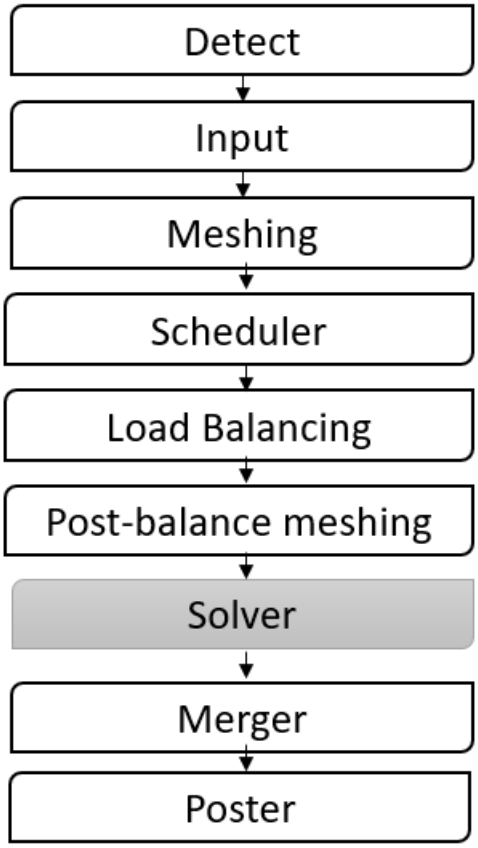

Figure 1: Description of Steps in LaBS Solver

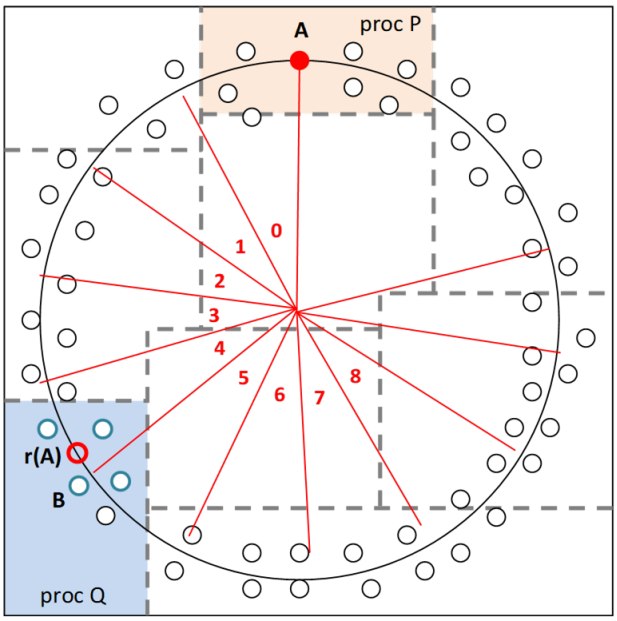

Figure 2: Time Step Based Slicing of a Rotating Domain

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

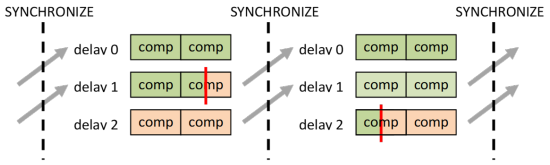

Figure 3: Delay Mechanism in LaBS

test case	nb cores	fuild nodes/cores	compute	send/recv	wait	total
S2A	64	0.7M	2 %	27 %	55 %	**21 %**
S2A long	64	0.7M	2 %	29 %	48 %	**19 %**
J84	160	0.9M	4 %	47 %	41 %	**20 %**
J84	288	0.5M	-4 %	19 %	71 %	**44 %**
J84 long	288	0.5M	2 %	19 %	40 %	**19 %**
J84	480	0.3M	-4 %	6 %	45 %	**19 %**
X62	96	0.7M	-4 %	46 %	58 %	**22 %**
X62	96	0.7M	-2 %	37 %	55 %	**24 %**

2.2.3 Results

With this optimisation, we have greatly reduced the data exchanged between processors and the time spend in communication. For a pulse on uniform mesh with a rotating domain we have obtained the following results:

number of processors		nanoseconds/(fine node*timestep)	exch. bytes/(fine node*timestep)
256	before	6693 ns	422 B
	after	876 ns	33 B
	gain	87 %	92 %
512	before	11753 ns	530 B
	after	1463 ns	46 B
	gain	88 %	91 %

2.3 Delay

2.3.1 Introduction

The LaBS suffer of heavy time consumption in communication. The main problem is that numerical computations are really fast (explicit scheme) and need a lot of synchronisation specially within multiple resolution domains. In that case load balancing must be really precise (around the nanosecond). Regarding the optimisation, our main work is to reduce time used in communication.

2.3.2 Solution

One of our answers to reduce the time passed in communication is a feature called "Delay". Between each synchronisation a *timeLimit* is set on all processors and during the solver step, when time elapsed in a computation on a processor exceeds the *timeLimit*, this computation is stopped (if it's possible) and the synchronisation is enforced. The computation will resume after the data sent to other processor.

The *timeLimit* is computed with forecast time of the different computation kernel specified in the tuning for the first cycle and then it's computed with B measured values of the previous cycle.

The user may specify a *maxDelay* limit, that prevent the computation to be delayed more than *maxDelay* times. If the user set the *maxDelay* to 1, all the delay mechanism is disable.

2.4 Vectorization

Mostly all of the HPC-oriented CPU architecture features vector units. These are pieces of hardware capable of applying a single operation on a vector of data in so called SIMD fashion. These SIMD units greatly increase the theoretical peak performance of the CPU. To access these resources, special instructions have to be generated. This process is called vectorization. Although, modern compilers are usually capable of automatic vectorization, in some situations such vectorization is inefficient or not possible at all. This is also the case with LaBS. To vectorize computation kernels in LaBS, a combination of techniques has been applied

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

	Baseline	Vectorisation		Cache Reuse	
cells	MLUPS	MLUPS	speedup	MLUPS	speedup
16384	3.17	4.89	1.54	5.27	1.66
65536	2.66	3.51	1.31	3.82	1.43
262144	2.66	3.28	1.23	3.84	1.44
1048576	2.02	2.94	1.45	3.20	1.58
4194304	2.27	3.51	1.54	3.93	1.72
16777216	2.23	3.04	1.36	3.27	1.46

Table 1: Performance Comparison in Millions Lattice Updates Per Second (MLUPS)

including loop permutation, code transformation, and compiler directives insertion. This way a more efficient vectorized code can be generated which improves single-core performance.

To evaluate the impact of manual vectorization, a fairly straightforward COVO test case has been chosen. This test case was chosen based on several features. Firstly, the test case requires almost no communication and mostly consists of computation. It allows the pinpointing impact of vectorization on the performance of the compute part in LaBS. Secondly, it requires evaluation of only several kernels with the top two consuming around 70% of the total computation time. Therefore we can demonstrate the benefit of manual vectorization by only modifying a couple of kernels. Finally, the size of the simulation domain can be easily specified, so vectorization can be tested on various problem sizes.

Table 1 contains the performance evaluation of vectorization in columns 2 and 3. It can be seen that the manual vectorization achieves 54% performance gain in the best case and around 40% gain on average.

2.5 Cache Reuse Optimization

The performance of many applications is limited by memory throughput. In such cases, optimizing cache subsystem utilization can provide a notable performance boost. One of the techniques used to improve data locality is loop fusion. This compiler optimization merges several loops together to increase data reuse. In LaBS, a similar effect is achieved by applying several kernels consecutively on a single block of data whose size can be optimized for concrete hardware. The aggregation of kernels has to respect data dependency between kernels. This is depicted in Figure 2.5 where function GradientStd requires information from adjacent cells thus creating a dependency. Functions that can be aggregated together are marked with the dashed line. The data reuse factor and effect of this optimization depend on the type of task and functions used.

For evaluation of cache reuse optimization the same COVO test cases used in vectorization performance analysis have been used. The amount of possible data reuse in these particular test cases is somewhat limited and cannot display the full potential of optimization. We chose these test cases for the ability to show additional benefits of cache reuse on top of vectorization.

Columns 3 and 4 of the Table 1 contains absolute performance in MLUPS and relative speedup respectively. This optimization provides 72% improvement in performance together with vectorization compared to 54% of vectorization alone. Additional performance ranges from 10% to 21% with an average of 14.3% compared to vectorization.

2.6 Hybridization

In general, the main limiting factor of the scalability of the application is communication. With an increasing number of processing elements involved in computation, the overhead of communication rises while computation time per individual processing element decreases. This leads to a situation where for a certain number of processing elements the run time of the application is entirely dominated by communication. There are two fundamentally different approaches to communication, one based on explicit messages passing and one using shared memory as means of communication. The explicit message passing advantage is the ability to facilitate communication between any interconnected processing elements and work on a variety of connections. This versatility comes at the expense of considerable overhead. The communication through shared memory, on the other hand, is very lightweight but is only applicable in an environment where memory is shared between processing elements which is usually a single computation node. The shared memory communication also introduces new challenges resulting from data sharing as correct placement and synchronization. The best performance is usually achieved using both methods resulting in a hybrid setup where shared memory communication is used within the compute node and message passing is used to communicate between compute nodes.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

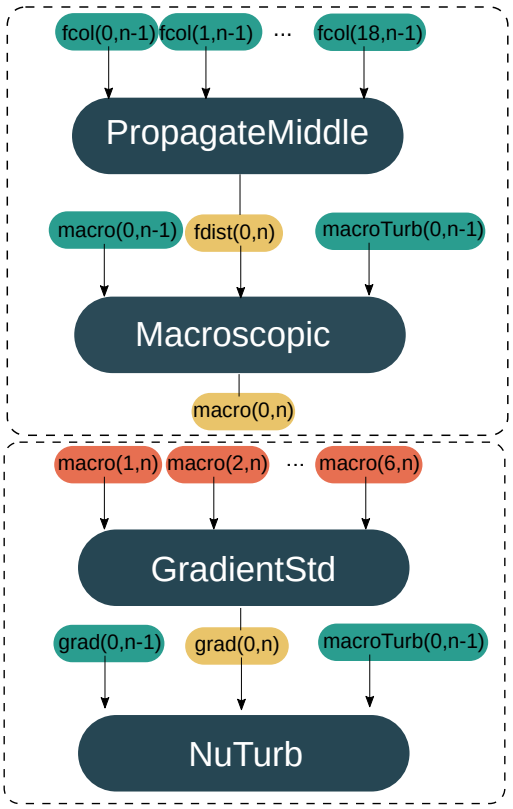

Figure 4: Dependencies between kernels and kernel aggregation

LaBS up to this point only used message-passing communication in form of an MPI library. Therefore to improve salability, a hybrid approach using the OpenMP standard for shared memory communication was implemented. Most of the work related to hybridization, revolved around adapting data structures, classes, and algorithms for sharing between several processing elements.

Due to technical difficulties and bugs in code, the effect of hybridization was evaluated only on a single node of Karolina supercomputer at IT4I Ostrava. Karolina node consists of 2 processors AMD Zen 2 EPYC 7H12 featuring 64 cores each.

Figure 2.6 shows the performance of various combinations of message passing processes and threads communicating through shared memory (x-axis) on various problem sizes. The right-most configuration represents a purely message passing run (e.g. LaBS implementation before this optimization) and the left-most represents a run using only communication through shared memory. In all cases, hybrid runs perform better than purely message passing runs even on a single node where message passing overhead is minimal as the network is not involved. The configuration consisting of 32 processes each with 4 threads achieves the best performance on almost all problem sizes, which is expected given the hardware organization of this processor. Performance of application on problem size 512^2 is unusually high. Currently, there is no explanation for this phenomenon and it will be investigated further.

In the future performance of hybridization will be tested on more communication-heavy problems using multiple nodes. In these scenarios, it is expected that hybridization will provide even more performance boost, and also the ratio of threads to processes of optimal configuration will be likely higher.

3 WaLBerla

The waLBerla is a modern open-source software framework that supports complex multiphysics simulations, and that is specifically designed to address the performance challenge in computeational science and engineering: exploiting the full power of the largest supercomputers. As mentioned, the waLBerla is a framework, and hence consists of a lot of components which can then be assembled together to create the different applications.

Figure 6 shows the core structure of the waLBerla framework. In that context, for the waLBerla, we explore the application of CUDA task graph to improve the performance of the LBM kernels.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

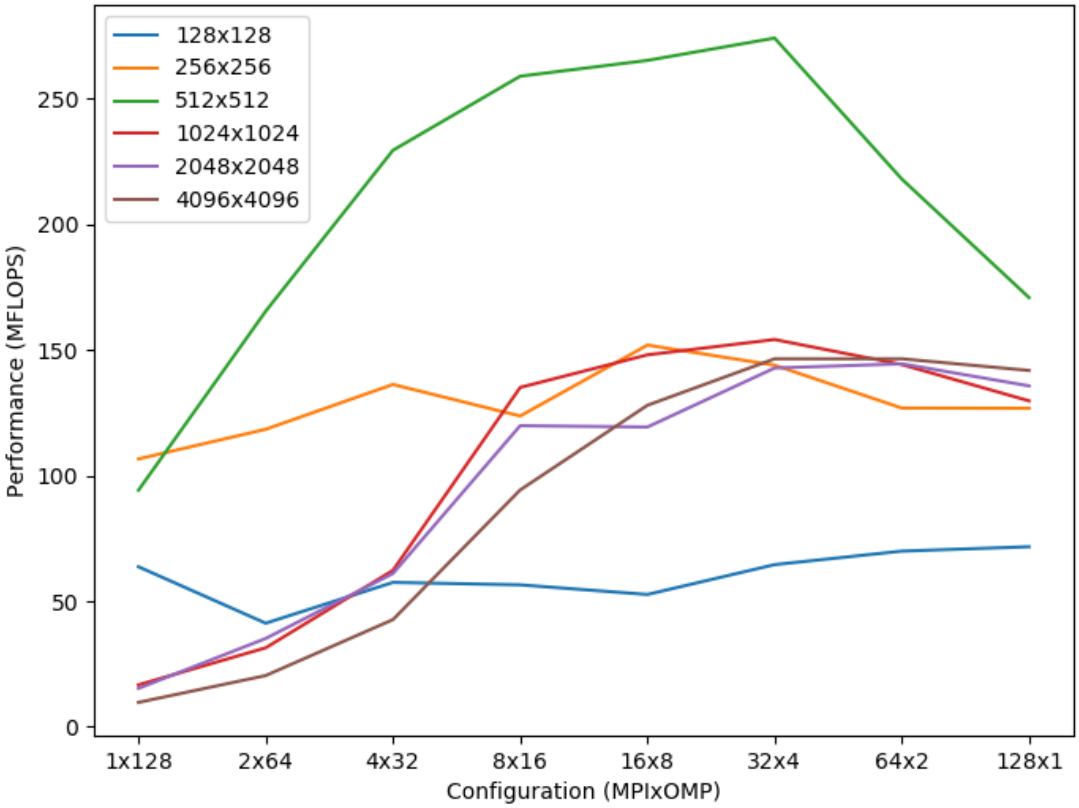

Figure 5: Performance of various configurations of hybrid run in LaBS Software

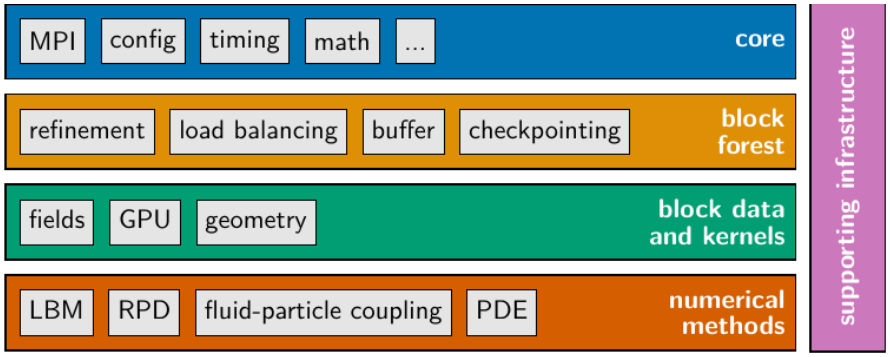

Figure 6: Overview of the waLBerla framework's software architecture and components.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

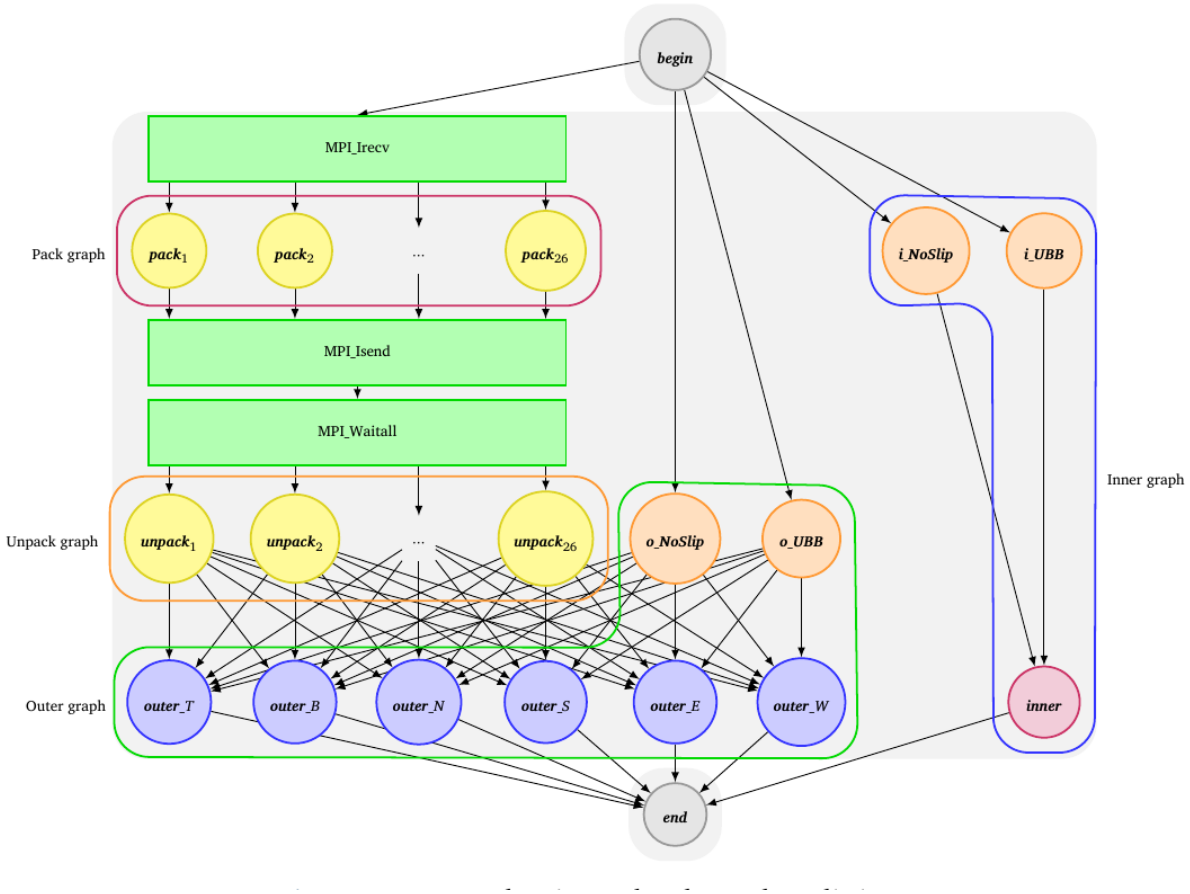

Figure 7: Kernel Dependency Graph

3.1 Optimization using CUDA task graphs

In modern Nvidia GPUs, as the processing of the kernels become faster, the kernel launch times can also be a bottleneck for execution of kernels. CUDA task graph allows the user to schedule kernels on the GPUs ahead of time, either directly or with a dependency to another kernel. And thereby, reduce the time spent in launching kernels. This can improve the performance of workloads where a lot of small kernels are launched in the application. If the application uses the same graph multiple times, then the cost of launching of graphs can be amortized.

For testing out this optimization, we started with an initial experiment of using the UniformGridGPU test case in the Walberla code. We chose the test case due to its underlying uniform grid, which allows the code to isolate the performance issues related to launching the kernel calls and ignore other issues like load balancing or kernel launch scheduling optimizations.

A typical dependency graph of kernels and other operations in the UniformGridGPU example looks like shown in Figure 7. The graph consists of kernels and MPI operations. The kernels can directly be added to the CUDA task graph, however, the MPI operations, even when using GPUDirect RDMA are scheduled from the CPU and therefore cannot be included as a part of the task graph. Therefore, we split the task graph into five parts, four of them them identified by an enclosing colored boxes and the fifth one consisting of the remaining functions. The four of them which are covered by the colored boxes are are CUDA task graphs and the remaining one represents the MPI calls.

The Table 3 shows performance improvement when using these 4 task graphs in various configurations of cells per block of the UniformGridGPU test case.

As we can see,the Table 3 shows that for small block sizes, the performance is almost double in compare to the performance without CUDA task graph, however, for the large block sizes, there is only minor performance improvement. This is in line with the expectations of the CUDA task graphs.

In future, Adaptive Mesh Refinement (AMR) on GPUs will be implemented in Walberla as a part of WP4. In AMR, scalability can be an issue for smaller blocks of the mesh. The expectation is that use of CUDA task graphs might reduce some of the overhead of launching of small blocks in that region and hence improve the

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.

Configuration	Cells Per Block	GPU Block Size
1	(64,64,64)	(32,1,1)
2	(128,128,128)	(32,1,1)
3	(256,256,256)	(32,1,1)
4	(320,320,320)	(32,1,1)

Table 2: Configuration

Configuration	No Graphs	4 Graphs
1	255.45	561.78
2	1314.27	561.78
3	2173.65	2287.76
4	2315.17	2442.24

Table 3: Performance Comparison in Millions Lattice Updates Per Second (MLUPS)

waLBerla scalability.

4 Conclusion

During the first phase of the SCALABLE project we have experimented with a couple of different optimizations. In case of the LaBS, the optimizations have already been implemented in the production code. Before the optimization, the performance in for simulations with moving meshes was much lower than the performance for static meshes. With the optimizations, the performance for moving mesh is now on parity with performance for static meshes. In case of the WaLBerla, we have achieved performance gains upto the order of 100% for kernels on small blocks. These gains should help us improve the performance of the applications for more complex use cases. The plan is now to mainstream the optimizations into the main code with the use cases under the tasks in WP4.

The Scalable project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 956000.