Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Scalable Delft3D Flexible Mesh for Efficient Modelling of Shallow Water and Transport Processes

M. Mogé[1a], M. J. Russcher[a,b], A. Emerson[c], M. Genseberger[b]

[a]*SURFsara, The Netherlands*
[b]*Deltares, The Netherlands*
[c]*CINECA, Italy*

Abstract

D-Flow Flexible Mesh ("D-Flow FM") [1] is the hydrodynamic module of the Delft3D Flexible Mesh Suite [2]. Since for typical, real-life applications there is a need to make D-Flow FM more efficient and scalable for high performance computing, we profiled and analysed D-Flow FM for representative test cases. In the current paper, we discuss the conclusions of our profiling and analysis. We observed that, for specific models, D-Flow FM can be used for parallel simulations using up to a few hundred cores with good efficiency. It was however observed that D-Flow FM is MPI bound when scaled up. Therefore, for further improvement, we investigated two optimisation strategies described below.
The parallelisation is based on mesh decomposition and the use of deep halo regions may lead to significant mesh imbalance. Therefore, we first investigated different partitioning and repartitioning strategies to improve the load balance and thus reduce the time spent waiting on MPI communications. We obtained small performance gains in some cases, but further investigations and broader changes in the numerical methods would be needed for this to be usable in a general case.
As a second option we tried to use a communication-hiding conjugate gradient method, PETSc's linear solver KSPPIPECG, to solve the linear system arising from the spatial discretisation, but we were not able to get any performance improvement or to reproduce the speedup published by the authors. The performance of this method turns out to be very architecture and compiler dependent, which prevents its use in a more general-purpose code like D-Flow FM.

Introduction

Delft3D [3] is used worldwide with a broad application range including the modelling of flooding, morphology and water quality, in coastal and estuarine areas, rivers and lakes, and from consultancy work to applied research. There are two different Delft3D versions: the Delft3D 4 Suite for structured computational meshes, and the newer Delft3D Flexible Mesh Suite [2] for unstructured computational meshes. D-Flow Flexible Mesh ("D-Flow FM") [1] is the hydrodynamic module of the Delft3D Flexible Mesh Suite. For typical real-life applications, for instance for highly detailed modelling and operational forecasting, there is a need to make D-Flow FM more efficient and scalable for high performance computing.[2]
This was the objective of a preparatory access type D project carried out by SURFsara, CINECA, and Deltares between 2017 and 2019. In particular, the goal was to bring the performances and scalabilities of the shallow water

[1] Corresponding author. E-mail address: maxime.moge@surfsara.nl
[2] The scalability of Delft3D-FLOW, the shallow water solver of the Delft3D 4 Suite, was studied before[4].

and transport solvers in the Delft3D Flexible Mesh Suite [2] closer to those required for Tier-0 systems, with a focus on D-Flow FM. This PRACE White Paper contains the results of this project.
In this paper, first the main computational methods of D-Flow FM are outlined. Then the selected test cases are described. The principal tasks of the work performed involved the scalability and performance analysis with these test cases. To further improve the observed scalability, based on the analysis, we identified the main bottlenecks and with this information several optimisation strategies were investigated.

Computational Methods of D-Flow FM

D-Flow FM solves the shallow-water equations [1] with the spatial discretisation being achieved by a staggered finite volume method on an unstructured mesh of cells of varying complexity (triangles to hexagons). After linearisation of the temporal discretisation, the resulting systems are solved with a semi-implicit method. This involves a linear system which is currently solved by a minimum degree algorithm to reduce system size and a preconditioned Krylov solver from PETSc [5]. Parallelisation is via domain decomposition with METIS [6] to distribute the computational work. At the interfaces between subdomains, halo regions are defined using degree 4 neighbours for a proper representation of discretised stencils at the interfaces and communication between subdomains via MPI.

Selection of Representative Test Cases

For benchmarking and testing possible improvements of D-Flow-FM, model applications were selected based on those currently under development at Deltares and which also impose a computational challenge. These are as follows:

- Schematic model of the Waal ("Waal_schematic") with 9000000 cells and 9015601 nodes. This depth-averaged model with groins and part of the floodplain of the Waal, one of the main rivers in the Netherlands, is used to estimate the effect of lowering the groins on the water level when the area is flooded [7]. The relatively large number of grid cells and the rectangular shape make it a good test case to start with for investigating scalability. For the depth-averaged shallow water solver WAQUA [8] a good scaling up to at least 80 processors was observed with this model [9]. The model has also been used in a previous PRACE project [4] to investigate and improve scalability of the shallow water solver Delft3D-FLOW[2].
- Global Tide and Surge Model with 9584149 cells and 8911362 nodes ("GTSM"). The main goal of the depth averaged Global Tide and Surge Model [10], [11] is to zoom in from global to regional scale and to study the impact of various assumptions in regional models. The unstructured grid is made in such a way that it represents coastal areas in more detail than the open oceans: this is of importance as much of the tidal energy is dissipated on the shelf, even on a global scale.
- North Sea models with 348842 cells and 353314 nodes for the depth averaged model ("North_Sea_2D") and with 8721050 cells and 9186164 nodes for both the three-dimensional model ("North_Sea_3D") and the three-dimensional model with salinity and temperature ("North_Sea_3D_ST"). The overall objective of these models is to have advanced modelling capabilities for assessing long-term ecosystem changes in the North Sea. The three-dimensional D-Flow FM models [12] have the same horizontal unstructured grid as the depth averaged model but use 25 so-called sigma layers in the vertical direction leading to a higher computational load per horizontal cell. Furthermore, for "North_Sea_3D_ST" additional salinity and temperature processes are switched on in D-Flow FM with proper forcing and boundary conditions. For this, next to the shallow water equations, D-Flow FM uses advection diffusion equations. This leads to a higher computational complexity which is representative for other water quality processes.
- Lake Marken model with 345184 cells and 175348 nodes ("Lake_Marken"). This model has been developed to enable an integrated approach in which the model can be used for different applications in the Lake Marken area [13], [14]. It contains a boundary fitted grid, with grid size depending on the location and smooth transitions in between. The key idea is to have enough resolution near the dikes (important for dike safety assessments and operational forecasting) and other important structures (for land reclamation for housing and natural islands) and a coarser resolution where possible in order to save computational time (important both for operational forecasting and water quality studies).
- Rhine branches models with 108143 cells and 109300 nodes ("Waal_40m") and with 1213410 cells and 1220927 nodes ("Rijntakken_20m"). The models have been developed for quantifying the cumulative effects of combined measures and to design optimal strategies [15], [16]. With these models measures are studied that counteract the effects of the bed level degradation that influence the morphology of the river bed, and therefore affect navigability.

Scalability and Performance Analysis

We used D-Flow FM Version 1.2.1.62244 with the configurations described in Table 1.

Table 1: Hardware and software used for the scalability and performance analysis

System	**Cartesius**	**MARCONI**
Partition	Thin nodes and Fat nodes	A2 (KNL)
Architecture	Intel Haswell and Sandy Bridge	Intel KNL
Interconnect	Infiniband	Intel Omni-Path
Compiler	Intel compiler 18.0.1	Intel compiler 18.0.5
Optimization flags	'-xAVX2 -O3'	'-xHost -axMIC-AVX512 -O3'
MPI implementation	Intel MPI 2018.1.163	Intel MPI 2018.4.274

On Cartesius we used the Sandy Bridge partition for Waal_40m test case since this D-Flow FM model has higher memory requirements. The nodes on the systems we used have significantly different architectures, with a lower core frequency on MARCONI (1.4 GHz for KNL cores) compared to that on Cartesius (2.6 GHz for Haswell cores, 2.7 GHz for Sandy Bridge cores).
For all test cases there is first a serial initialisation phase that we do not consider in the scalability analysis. We only consider the time spent in the time loop, which accounts for most of the running time in real life cases.
Test cases (detailed in previous section) are representative of D-Flow FM use cases from real life applications for which computational grids are tailor made with specific spatial scales. Therefore, we do not have different computational grids with different spatial scales for each test case to test weak scalability. However, test cases differ in the number of grid cells: from 108143 to 9584149.

Speedup and Efficiency of Test Cases

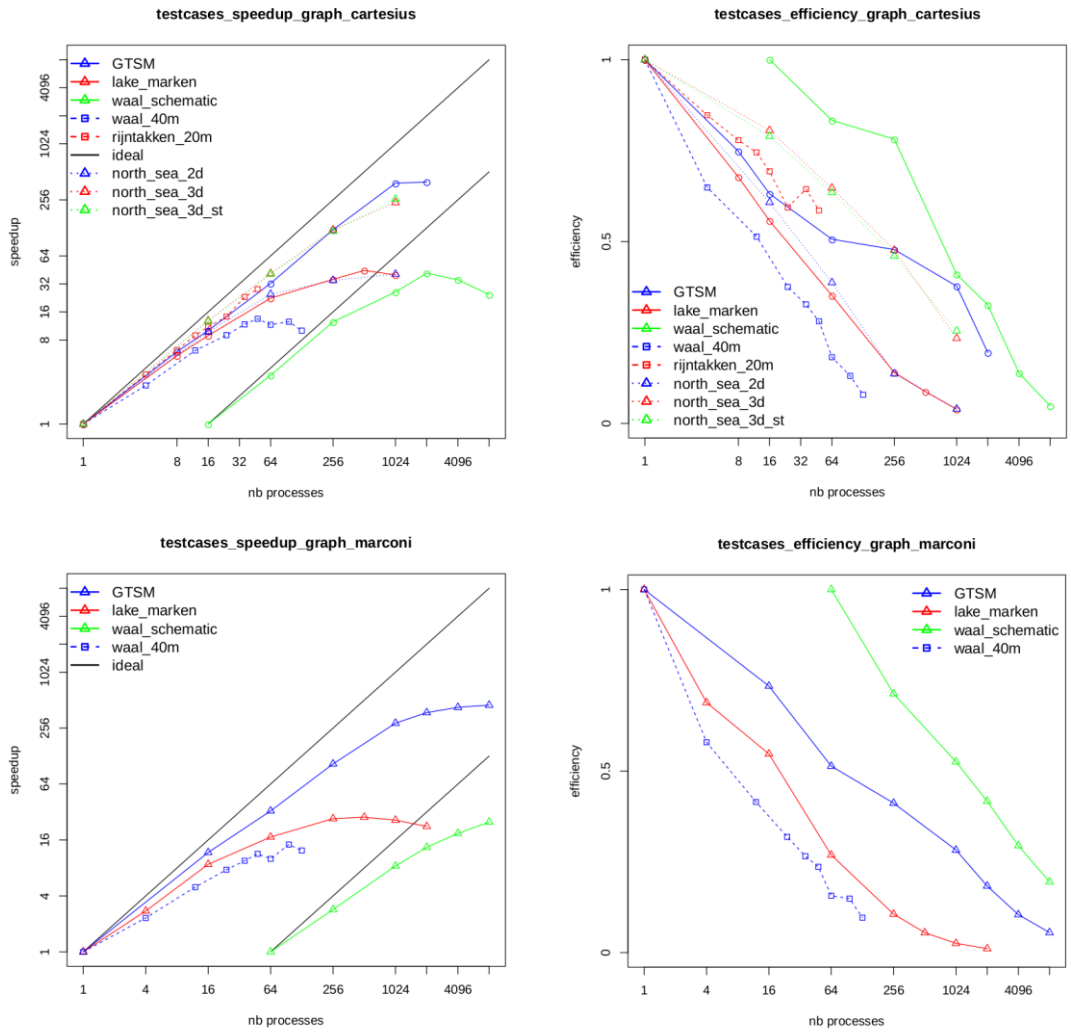

Figure 1: Speedup and efficiency of all test cases on Cartesius and MARCONI

From Figure 1 we observe the following:
1) Some test cases are too small to scale well for larger number of processes on Cartesius:
- For the Lake_Marken and North_Sea_2D test cases the parallel efficiency is good (> 0.50) for only up to 64 processes.
- For the Waal_40m test case the parallel efficiency is good (> 0.50) for only up to 48 processes.

2) The larger test cases (North_Sea_3D and North_Sea_3D_ST, Waal_schematic, and GTSM) show good scalability (efficiency > 0.50) up to 256 processes on Cartesius.
3) The scaling behaviour is similar on both machines, but the computation time on MARCONI is much higher than on Cartesius when the number of processes is the same (~4x higher for most test cases, up to ~10x higher for Waal_schematic, see Figure 5 for the Lake Marken and GTSM test cases).
Several previous scalability analyses showed similar results. However, these analyses were only for up to 100 cores [17], [19] and [20]. The observations can be largely explained as a result of the reduced size of the subdomains when scaling up and is expected. There is also a known scaling limit of PETSc (used for linear system solving in D-Flow FM) for local problem sizes smaller than ~20,000 unknowns and corresponds to the measured efficiency drop in our test cases.
Given the present parallelisation of D-Flow FM (see section Computational Methods of D-Flow FM), for an increasing number of partitions and relatively small test cases, the halo region of each partition (amount of ghost cells) becomes large compared to the amount of internal grid cells of the partition. This means that the maximum

attainable speedup will be sub-linear. Table 2 shows the increase in total size when increasing the number of partitions for the GTSM test case.

Table 2: total size of domain including ghost cells for GTSM test case

#partitions	Total #cells (including ghost cells)	Fraction of ghost cells in total # cells: #ghost cells / total #cells
1	9582942	0
16	9647149	0.01
64	9757414	0.02
256	10039319	0.05
1024	10719170	0.11
4096	12354089	0.22
8192	13838713	0.31

Efficiency of the PETSc solver compared to the rest of the time loop

D-Flow FM solves the discretised shallow water equations for the water levels implicitly in time, with momentum advection treated explicitly (for details, see [1]). The velocities and fluxes are then obtained by back substitution. The algorithm is a combined approach, where a configurable part of the system is solved by Gaussian elimination, and the remaining unknowns are solved iteratively with the conjugate gradient method available from PETSc.
We computed the efficiency of the PETSc solver and the rest of the time loop separately in Figure 2. The efficiency of the linear system solver with PETSc is significantly worse than the efficiency of the rest of the time loop, and eventually accounts for most of the runtime when we increase the number of MPI processes, as shown by the percentage of the runtime spent in KSPSolve, (PETSc's solver main routine) in Figure 3.
Figure 3 also confirms the scaling limit of PETSc for small local problems size (see section Speedup and Efficiency of Test Cases).

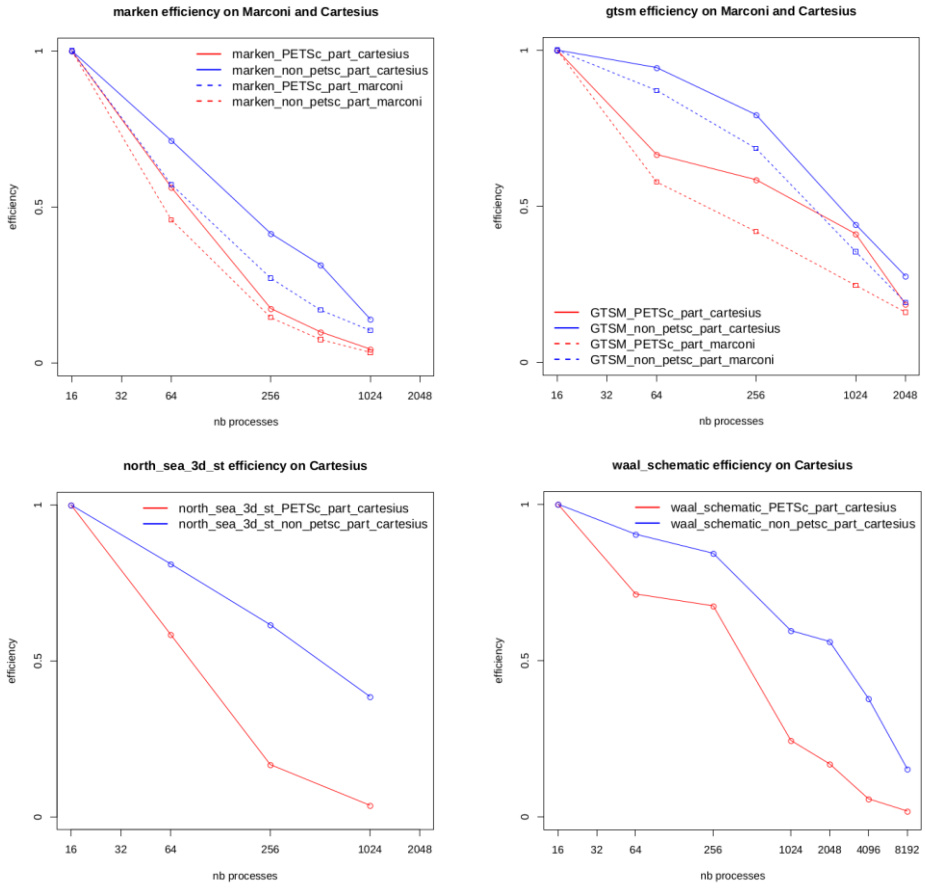

Figure 2: Efficiency of PETSc and non-PETSc part of the time loop for Lake_Marken, GTSM, North_Sea_3D_ST and Waal_schematic test cases on Cartesius and MARCONI

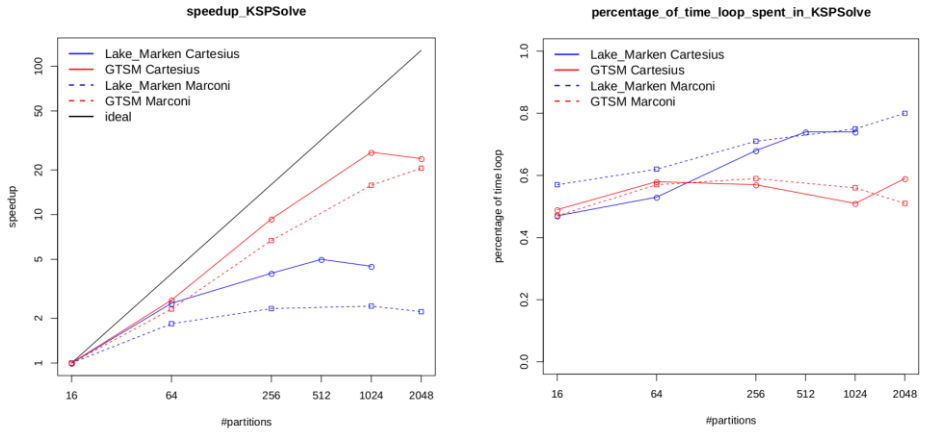

Figure 3: Portion of time loop spent in linear system solve with PETSc::KSPSolve (right) and speedup of KSPSolve (left) for Lake_Marken and GTSM test cases on Cartesius and MARCONI

MPI Profiling and Load Balance

We profiled the MPI communication using IPM, a portable profiling infrastructure that provides a low-overhead performance profile of the performance aspects and resource utilisation in a program [21], measuring the minimum and maximum time spent by a process in MPI operations (Figure 5). This shows that the time spent in MPI communication within D-Flow FM, including waiting time, does not scale well and prevents scaling for high numbers of processes, thereby accounting for most of the time spent in the time loop. Moreover, there is a significant imbalance between the minimum and the maximum MPI Time.
A Hotspot analysis of D-Flow FM with VTune (e.g. for GTSM test case in Figure 4) shows that almost all MPI communications are done in the linear solver (i.e. the conjugate gradient solver of PETSc), in MPI_Allreduce calls. Note also that the portion of elapsed time spent in MPI communications and its scaling behaviour are similar on Cartesius and MARCONI, even though the architectures are significantly different.

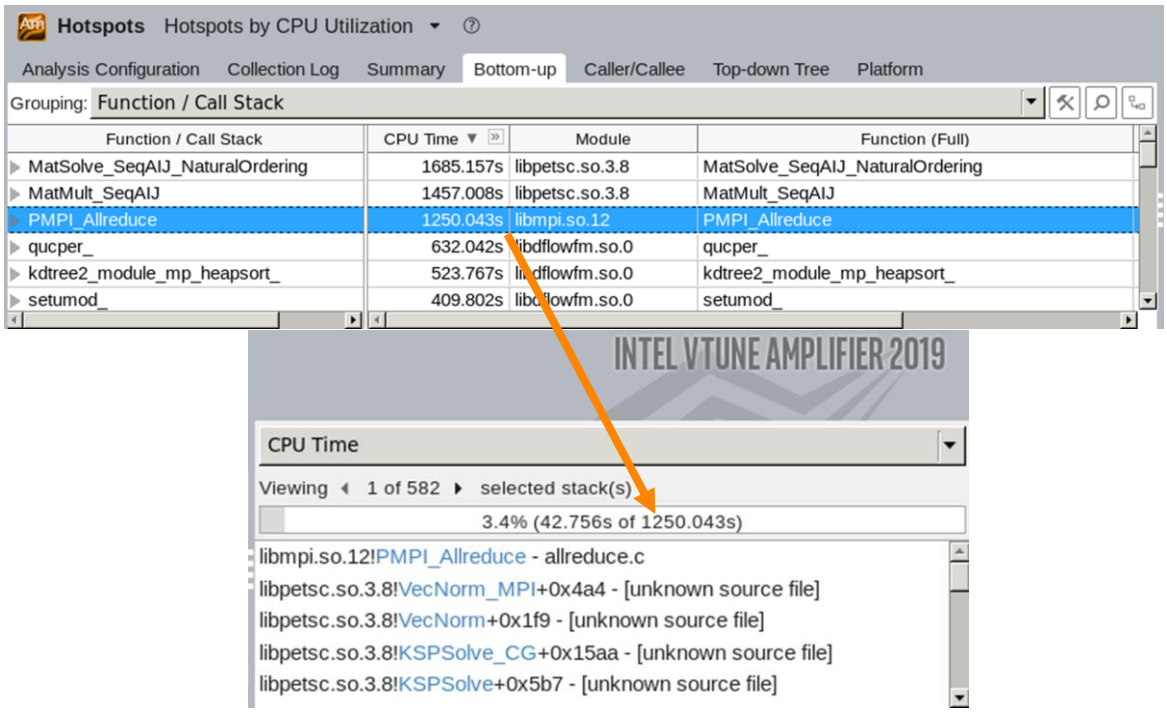

Figure 4: Hotspot analysis of GTSM test case with 64 MPI processes on Cartesius with VTune, with the call stack of PMPI_Allreduce

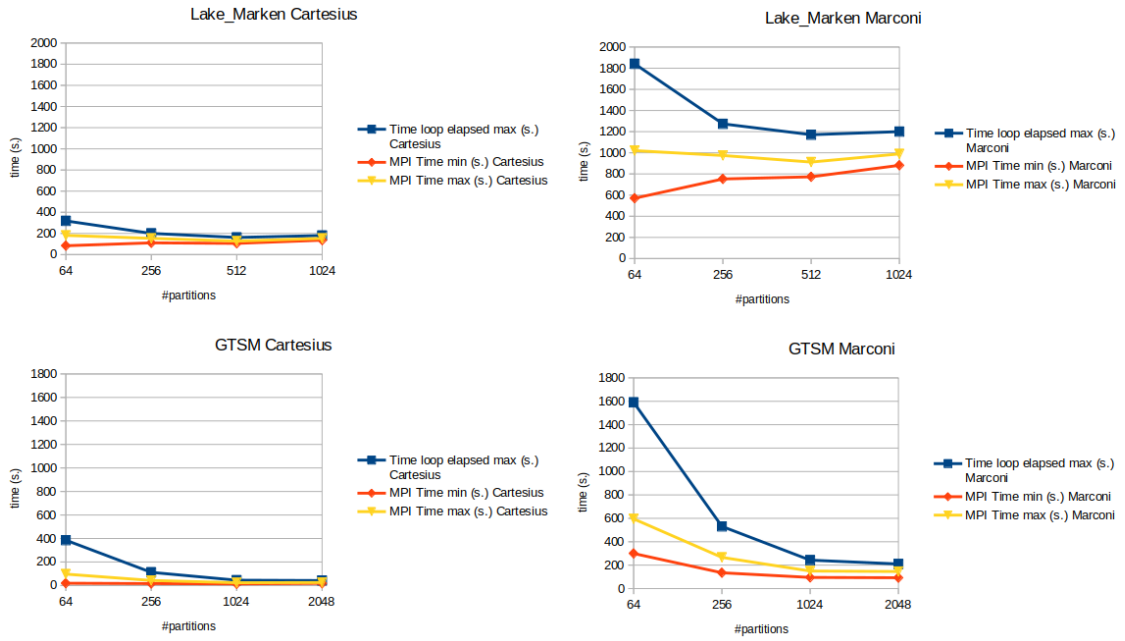

Figure 5: Elapsed and MPI Time (max and min) in Time Loop for Lake_Marken and GTSM test cases on Cartesius and MARCONI.

To better interpret our measurements, we used the metrics and methodology developed by POP COE for parallel performance analysis [22]. The communication efficiency (CommE) can be calculated by the maximum computation time divided by the runtime. A value below 0.80 is considered poor. In Table 3 and Table 4, we see that the communication efficiency is poor for high numbers of processes, which agrees with our observations from Figure 1. This implies that we need to investigate the MPI aspects of the code further.

Table 3: Communication Efficiency of time loop for different test cases on Cartesius

#partitions	**16**	**64**	**256**	**1024**	**2048**	**4096**	**8192**
Lake_Marken	0.91	0.72	0.43	0.23	-	-	-
GTSM	0.98	0.95	0.85	0.64	0.55	-	-
Waal_schematic	0.99	0.96	0.92	0.42	0.36	0.17	0.39

Table 4: Communication Efficiency of time loop for different test cases on MARCONI

#partitions	**16**	**64**	**256**	**1024**	**2048**
Lake_Marken	0.94	0.68	0.39	0.24	-
GTSM	0.95	0.81	0.72	0.59	0.54

The load balance (LB) metric is calculated by the average computation time divided by the maximum computation time. In Table 5 and Table 6, we see that for GTSM and Lake_Marken test cases the load balance decreases for high number of partitions. It is considered low when lower than 0.80, which is reached for 2048 processes for GTSM, 256 for Lake_Marken and 8192 for Waal_schematic. For Waal_schematic, the load balance stays quite good for larger number of partitions and then drops drastically with 8192 partitions. Therefore the parallel decomposition of the mesh also requires further investigation.

Table 5: Load balance of time loop for different test cases on Cartesius

#partitions	**16**	**64**	**256**	**1024**	**2048**	**4096**	**8192**
Lake_Marken	0.89	0.87	0.77	0.81	-	-	-
GTSM	0.97	0.92	0.89	0.84	0.78	-	-
Waal_schematic	0.99	0.97	0.95	0.92	0.90	0.86	0.14

Table 6: Load balance of time loop for different test cases on MARCONI.

#partitions	16	64	256	1024	2048
Lake_Marken	0.92	0.89	0.83	0.80	-
GTSM	0.95	0.88	0.83	0.78	0.68

The average time spent in an MPI_Allreduce call in D-Flow FM is quite high compared to the results measured using OSU benchmarks [18] with comparable message sizes. For example, for GTSM test case with 1024 partitions, and an average message size of 439 bytes, the average time per MPI_Allreduce call is 1174 µs. With osu_allreduce benchmark and 512 bytes messages and 1024 processes, the average time per MPI_Allreduce call is only 54 µs. This large difference is probably due to the computational imbalance which causes longer waiting times in MPI_Allreduce calls for the fastest subdomains. We are planning to further investigate this.

Main Identified Bottlenecks

- D-Flow FM is memory bound (already known from a previous study [19]).
- D-Flow FM is MPI bound on the selected test cases, and most of the MPI communications consist of MPI_Allreduce operations that are done in the linear solver (PETSc).

There is a significant difference between the lowest and the highest time spent in MPI communication by a process. The communication efficiency and the computational load balance are poor for large number of partitions. We think that the MPI imbalance is caused in large part by the computational imbalance, which is in turn largely caused by the mesh imbalance. Therefore, we focus on improving the computational load balance to improve the overall performances.

Partitioning: Lowering the Mesh Imbalance

Part of the observed imbalance in the computational load, appears to come from an uneven partitioning of the mesh. The partitioning algorithm succeeds very well at creating equally sized domains, but the halo regions are not accounted for. However, there is a non-negligible amount of computational work associated with the halo, and the halo regions sizes are imbalanced. Furthermore, its relative size increases when hard-scaling the application to a larger number of processes. Our strategy is therefore to reduce the imbalance in the partitioned mesh, with the contribution of the halo included.

Partitioning Strategy to Improve Mesh Imbalance

There are multiple ways to measure the imbalance of a mesh partitioning. We define the imbalance by the maximum number of mesh cells divided by the mean number of mesh cells in a partition:

```
imbalance = max_i(nCells_i)/ mean_nCells
```

In D-Flow FM, meshes are partitioned with METIS, using either recursive bisection of k-way partitioning. The imbalance is caused by the halo regions that are added after the mesh partitioning with METIS. The data in Table 7 and Table 8 show indeed that, although the inner subdomains are well-balanced, the size of their halo region varies substantially and even more so for a larger number of partitions.

Table 7: Mesh imbalance for GTSM test case – METIS k-way partitioning

#partitions	mean number of grid cells	minimum number of grid cells	maximum number of grid cells	standard deviation	imbalance
16	602947	600076	606626	1712	1.01
64	152460	150242	155655	1269	1.02
256	39216	37762	41576	773	1.06
1024	10468	9441	11590	418	1.11
2048	5551	4742	6545	310	1.18

Table 8: Mesh imbalance for Lake Marken test case – METIS k-way

#partitions	mean number of grid cells	minimum number of grid cells	maximum number of grid cells	standard deviation	imbalance
4	87197	86298	88090	749	1.01
16	22829	21978	23880	491	1.05
64	6385	5596	7117	311	1.11
256	2017	1400	2780	200	1.38

Halo-Aware Repartitioning

Starting from the original partitioning with METIS, we use ParMETIS to repartition taking into account the size of the ghost region of each subdomain (this is an adaptation of the algorithm proposed in [23]).

1. Compute the size of the halo region of each subdomain
2. For each subdomain, define weights on its internal cells derived from the total size of the subdomain (internal + ghost cells)
3. Perform a repartitioning using these weights, which should reduce the imbalance
4. If the imbalance is too large, use the new partitioning and go back to 1.

Using this algorithm, we were able to lower the imbalance (see Table 9 and Table 10). ParMETIS uses the k-way algorithm for repartitioning, so we used k-way partitioning with METIS as well for a more coherent performance comparison.

Table 9: Mesh imbalance for Lake Marken test case – METIS k-way partitioning vs. halo-aware repartitioning with ParMETIS

#partitions	partitioning algorithm	mean number of grid cells	minimum number of grid cells	maximum number of grid cells	standard deviation	imbalance
64	**METIS k-way**	6385	5596	7117	311	1.11
	METIS k-way + repartitioning	6438	6059	6776	136	1.05
256	**METIS k-way**	2017	1400	2780	200	1.38
	METIS k-way + repartitioning	1986	1758	2310	81	1.16

Table 10: Mesh imbalance for GTSM test case – METIS k-way vs. halo-aware repartitioning with ParMETIS

#partitions	partitioning algorithm	mean number of grid cells	minimum number of grid cells	maximum number of grid cells	standard deviation	imbalance
256	**METIS k-way**	39209	37655	41618	755	1.06
	METIS k-way + repartitioning	39576	37284	41231	677	1.04
1024	**METIS k-way**	10703	9296	12178	372	1.14
	METIS k-way + repartitioning	10662	9424	11843	306	1.11

The size of the biggest subdomain (which should correspond to the ‘slowest’ process) is reduced by

- 5% for Lake_Marken 64,
- 20% for Lake_Marken 256,

- 1% for GTSM 256,
- 3% for GTSM 1024.

Comparing the results before and after halo aware repartitioning (Table 11 to Table 14), we do not see a significant impact of the improved imbalance on the performances for Lake_Marken and GTSM test cases.

Table 11: Profiling of Lake Marken test case with METIS k-way partitioning and halo aware repartitioning using IPM and PETSc built-in profiler – on Cartesius

#partitions	type of partitioning	time loop (s)	min MPI time (s)	max MPI time (s)	KSPSolve Time	Ratio max/min Flops in KSPSolve
64	**METIS k-way**	311.2	70.2	171.6	172.2	1.2
	METIS k-way + repartitioning	308.6	81.7	166.6	175.5	1.2
256	**METIS k-way**	173.8	88.3	131.5	116.4	1.7
	METIS k-way + repartitioning	175.3	93.1	125.3	122.0	1.7

Table 12: Load Balance and Communication Efficiency of time loop for Lake_Marken test case with METIS k-way partitioning and halo aware repartitioning - on Cartesius

#partitions	type of partitioning	Load Balance	Communication Efficiency
64	**METIS k-way**	0.86	0.76
	METIS k-way + repartitioning	0.91	0.72
256	**METIS k-way**	0.80	0.48
	METIS k-way + repartitioning	0.86	0.46

Table 13: Profiling of GTSM test case with METIS k-way partitioning and halo aware repartitioning using IPM and PETSc built-in profiler – on Cartesius

#partitions	type of partitioning	time loop (s)	min MPI time (s)	max MPI time (s)	KSPSolve Time	Ratio max/min Flops in KSPSolve
256	**METIS k-way**	111.0	17.6	42.2	63.0	1.4
	METIS k-way + repartitioning	110.5	16.0	38.4	59.3	1.4
1024	**METIS k-way**	43.1	13.9	24.0	22.4	1.7
	METIS k-way + repartitioning	44.6	14.3	24.0	21.3	1.8

Table 14: Load Balance and Communication Efficiency of time loop for GTSM test case with METIS k-way partitioning and halo aware repartitioning - on Cartesius

#partitions	type of partitioning	Load Balance	Communication Efficiency
256	**METIS k-way**	0.92	0.83
	METIS k-way + repartitioning	0.90	0.85
1024	**METIS k-way**	0.89	0.66
	METIS k-way + repartitioning	0.86	0.66

Reducing the imbalance in the subdomain sizes including halo regions may reduce the load imbalance. However, the observed load imbalance may have a combination of causes. Possible causes, next to imbalance in subdomain sizes, are: less efficient PETSc part and/or non-PETSc part, the role of additional halo points due to stencil for

MPI communication, and the role of the "PETSc" limit. Further research is needed to determine the actual effects and hence importance of the various causes.

Hypergraph Partitioning

It is known that graphs do not model parallel communication volume well. Hypergraphs, on the other hand, are able to model the communication volume accurately (see [23]) and allow more complex objectives for partitioning, like minimizing the number of ghosts vertices or cells.
We used hMETIS, the hypergraph partitioner from the METIS suite. hMETIS uses k-way partitioning, so we used k-way partitioning with METIS as well for a more coherent performance comparison.

Table 15: Inner subdomain imbalance for Lake Marken test case– METIS k-way partitioning vs. hMETIS hypergraph k-way partitioning

#partitions	type of partitioning	mean number of grid cells in inner domain	minimum number of grid cells in inner domain	maximum number of grid cells in inner domain	inner domain imbalance
256	**METIS graph**	1343	1342	1344	1.00
	hMETIS hypergraph	1343	1262	1413	1.05
1024	**METIS graph**	336	335	336	1.00
	hMETIS hypergraph	336	250	363	1.08

Table 16: complete subdomain imbalance (including ghost cells) for Lake Marken test case– METIS k-way partitioning vs. hMETIS hypergraph k-way partitioning

#partitions	type of partitioning	mean number of grid cells in subdomain	minimum number of grid cells in subdomain	maximum number of grid cells in subdomain	subdomain imbalance
256	**METIS graph**	2017	1400	2780	1.38
	hMETIS hypergraph	1906	1415	2307	1.21
1024	**METIS graph**	717	445	1050	1.46
	hMETIS hypergraph	671	434	807	1.20

Table 17: Profiling of Lake Marken test case with METIS k-way partitioning and hMETIS hypergraph k-way partitioning using IPM and PETSc built-in profiler – on Cartesius

#partitions	type of partitioning	time loop (s)	min MPI time (s)	max MPI time (s)	KSPSolve Time	Ratio max/min Flops in KSPSolve
256	**METIS graph**	173.8	88.3	131.5	116.4	1.7
	hMETIS hypergraph	174.8	94.6	130.7	123.8	1.8
1024	**METIS graph**	228.7	181.0	200.0	184.6	2.1
	hMETIS hypergraph	223.8	177.2	197.0	189.6	2.8

Table 18: Load Balance and Communication Efficiency of time loop for Lake_Marken test case with METIS k-way partitioning vs. hMETIS hypergraph k-way partitioning - on Cartesius

#partitions	type of partitioning	Load Balance	Communication Efficiency
256	**METIS graph**	0.80	0.48
	hMETIS hypergraph	0.85	0.45
1024	**METIS graph**	0.77	0.19
	hMETIS hypergraph	0.77	0.19

The hypergraph partitioning significantly lowers the total number of ghost cells and the imbalance (see Table 16). However, the performances on this test case with the new partitioning obtained using a hypergraph are similar to the performances using the original METIS partitioning (Table 17 and Table 18). With a higher number of partitions (1024), we see a more significant benefit on the imbalance and a small speedup for the time loop.
Looking at the more detailed profiling in Table 17, we notice

- a small slowdown on the PETSc part,
- a small speedup on the MPI communication,
- an increase on the computational imbalance for the linear system solve (KSPSolve), as shown by the increased ratio max/min Flops in KSPSolve.

However, we were not able to use hypergraphs for other test cases with large meshes as hMETIS requires a lot of memory and in fact our attempts failed due to "Out of Memory" errors. Moreover, there are many parameters to tune for hMETIS to get the best possible output, so it is probably possible to get a better partitioning than the one used here.

Pushing the Scaling Limit of the Linear Solver with a Communication hiding Linear Solver

For linear system solving in D-Flow FM we use PETSc, which has a known scaling limit for local problem sizes smaller than ~20,000 unknowns. We reached this limit in our scalability analysis (Figure 1), using PETSc's conjugate gradient solver (KSPCG) with a block Jacobi preconditioner. To push that limit further, communication avoiding and communication hiding implementations of the conjugate gradient method have been developed. We tried to apply it to D-Flow FM.

Theoretical Advantage of Pipelined CG over classical CG

In PETSc, the pipelined CG method is implemented in KSPPIPECG. It uses only one non-blocking reduction per iteration instead of two blocking reductions for a standard CG. The non-blocking reduction is overlapped by the matrix-vector product and preconditioner application (see [24]).

Non-blocking MPI Collectives

Non-Blocking Collectives are part of the MPI-3 standard, and so are fairly recent features of MPI implementations. They are not well supported by all MPI implementations and need parameter and configuration tuning to be efficient. For most MPI implementations, this means using an asynchronous progress thread to manage communication in parallel with the application computation and, as a result, achieve better communication/computation overlap. With Intel MPI, this feature is supported for the multithreaded versions only ('release_mt' and 'debug_mt'). With MPICH, version 3.0 and later implement the MPI-3 standard and the default configuration supports use of threads.
Thread-based asynchronous progress is the approach adopted by most MPI communication models, but it suffers from the restriction that a background thread can make progress only for the process that spawned it. Thus, we need as many progress threads as MPI processes, and choose either to dedicate half of the cores to the progress threads, or to perform oversubscription of the cores (i.e. spawning more threads than cores) which can have a high impact on performances (see [25]). Furthermore, this model forces the MPI runtime to maintain multithreaded safety, which results in additional overhead because of lock contention and memory barriers.

Performance Testing of KSPPIPECG on a simple PETSc Test Case

To make sure that our MPI implementations and hardware allow good performances of KSPPIPECG, we did performance tests with KSPIPECG on a simple example from PETSc: SNES/ex48 with a 1024x1024 matrix. We compared the running time with asynchronous progress threads (1 progress thread per MPI process) and without asynchronous progress threads, using a block Jacobi preconditioner, for the KSPCG and KSPPIPECG solvers. We also used the running time with the single threaded version of the MPI library as a reference.

- with Intel MPI we set 'I_MPI_ASYNC_PROGRESS=1' and 'I_MPI_ASYNC_PROGRESS_THREADS=1' and we used the 'release_mt' version to enable asynchronous progress.
- with MPICH we set 'MPICH_ASYNC_PROGRESS=1' and 'MPICH_MAX_THREAD_SAFETY=multiple' to enable asynchronous progress.
- for the single threaded version, we used the Intel MPI 'release' version, and with MPICH we set 'MPICH_MAX_THREAD_SAFETY=single'.

In Table 19, we see that with asynchronous progress enabled, KSPPIPECG is faster than KSPCG on this example. However, the overhead caused by the use of the thread-safe MPI library is too large to get speedup compared to synchronous runs with KSPCG and KSPPIPECG. This holds for both MPICH and Intel.

Table 19: performances of a simple PETSc example using PIPECG compared to CG on GTSM test case on Cartesius and MARCONI

System and test case	MPI implem-entation	Solver	Time multithreaded MPI without async progress (s.)		Time multithreaded MPI with async progress enabled (s.)		Time single threaded MPI without async progress (s.)	
			Total time	KSPSolve	Total time	KSPSolve	Total time	KSPSolve
Cartesius 240P/10N	**Intel**	**CG**	1.13	1.06	2.61	2.26	1.14	1.07
		PIPECG	1.15	1.08	1.60	1.46	1.15	1.08
	MPICH	**CG**	1.55	1.47	2.39	2.18	1.47	1.39
		PIPECG	1.57	1.49	2.12	2.01	1.55	1.47
MARCONI 120P/2N	**Intel**	**CG**	60.90	55.65	108.4	97.61	60.01	54.36
		PIPECG	60.15	54.76	88.97	81.97	57.45	52.32
MARCONI 240P/4N	**Intel**	**CG**	32.86	29.94	72.29	64.65	33.34	29.96
		PIPECG	40.22	36.84	54.05	49.72	42.55	39.10

Performance Testing of KSPPIPECG in D-Flow FM

As expected from the results on a simple test case above, using KSPPIPECG instead of KSPCG in D-Flow FM compiled with Intel MPI does not give us any speedup on Cartesius (Table 20).

Table 20: performances of D-Flow FM using PIPECG compared to CG for GTSM test case on Cartesius

Test case	CG - Time single threaded MPI without async progress (s.)		CG - Time multithreaded MPI with async progress enabled (s.)		PIPECG - Time multithreaded MPI with async progress enabled (s.)	
	Total time	KSPSolve	Total time	KSPSolve	Total time	KSPSolve
GTSM 64P/3N	413	241	736	442	663	372
GTSM 256P/11N	116	68	309	206	253	150

The overhead of the thread-safe MPI library prevents speedup compared to synchronous runs. The performances of this solver are very hardware and MPI-implementation dependent, and we could not reproduce the performances published by the developers (in [24]).

Summary and conclusion

For typical real-life applications, for instance for highly detailed modelling and operational forecasting, there is a need to make D-Flow FM more efficient and scalable for high performance computing. The selected test cases from model applications Deltares are developing impose computational challenges for Deltares.

We successfully ported and tested D-Flow FM on Cartesius and MARCONI/KNL. Previous testing at Deltares did not go beyond a few hundred MPI processes, and we successfully ran representative simulations on up to 8192 MPI processes. The scalability depends on the architecture, and we measured some speedup up to 2048 on Cartesius, and up to 8192 on MARCONI, and a good efficiency (>0.5) up to 256 MPI processes on Cartesius and 1024 processes on MARCONI for large meshes. As a rule-of-thumb, partitions should have more than ~25000 cells to get good parallel efficiency. On MARCONI KNL, the performances are not as good as on Cartesius since the current version of D-Flow FM is not targeted for good performance with the specific capabilities of the KNL architecture. However, it was very useful to have access to two systems with different architectures to compare the behaviour of D-Flow FM, get a good overview of its performance and identify the bottlenecks on Tier-1 and Tier-0 systems. This yielded new insights and it also gave us the opportunity to test possible improvements in D-Flow FM for real-life applications. For the selected test cases from the current project, it has become clear which further steps have to be taken to be able to run the software efficiently on the Tier-0 systems.

Our optimisation efforts were aimed at two main bottlenecks: the linear solver scaling limit, and the mesh imbalance for strong scaling.

To improve the mesh imbalance arising from the mesh partitioning and large halo regions we investigated two strategies:

- Halo-aware repartitioning: the resulting partitionings including halo region are better balanced, but the imbalance on the internal subdomains (excluding the halo regions) increases.
- Using a hypergraph instead of the classic dual graph to represent and partition the mesh.

With both strategies the resulting partitionings including halo region are better balanced, but the imbalance on the internal subdomains (excluding the halo regions) increases. The performance results are not that significant yet and further research is needed to distinguish the different contributions and relations (PETSc part, non-PETSc part, role of additional halo points due to stencil for MPI communication, role of "PETSc limit").

We did experiments with the pipelined conjugate gradient (PIPECG) method implemented in PETSc. This is supposed to improve performances of the conjugate gradient (CG) method when communications are important. However, we were not able to get any performance improvement or to reproduce the speedup published by the authors. We tested CG and PIPECG on a simple example as well as on a real case with D-Flow FM, failing to get a speedup with PIPECG over CG with single threaded MPI. In addition, the results for the PIPECG method appear to be highly dependent on the MPI implementation and architecture.

References

[1] H. W. J. Kernkamp, A. Van Dam, G. S. Stelling, and E. D. De Goede, Efficient scheme for the shallow water equations on unstructured grids with application to the continental shelf, Ocean Dyn. 61 (8), 1175-1188, 2011.

[2] https://www.deltares.nl/en/software/delft3d-flexible-mesh-suite

[3] https://oss.deltares.nl/web/delft3d/home

[4] J. Donners, A. J. Mourits, M. Genseberger, and H. R. A. Jagers, Delft3D - performance benchmarking report, PRACE white paper, 2014. http://www.prace-project.eu/IMG/pdf/wp100.pdf

[5] https://www.mcs.anl.gov/petsc

[6] http://glaros.dtc.umn.edu/gkhome/views/metis

[7] M. F. M. Yossef and Zagonjolli, M., Modelling the hydraulic effect of lowering the groynes on design flood level, Deltares technical report 1002524-000, 2010.

[8] SIMONA WAQUA/TRIWAQ - two- and three-dimensional shallow-water flow model, 2016. http://simona.deltares.nl/release/doc/techdoc/waquapublic/sim1999-01.pdf
[9] Mart Borsboom, Menno Genseberger, Bas van 't Hof, and Edwin Spee, Domain decomposition in shallow-water modelling for practical flow applications, proceedings of 21st International Conference on Domain Decomposition Methods, 2012. http://www.ddm.org/DD21/homepage/dd21.inria.fr/pdf/borsboomgensebergerhofspee_mini_8.pdf
[10] S. Muis, M. Verlaan, H. C. Winsemius, J. C. Aerts, and P. J. Ward, A global reanalysis of storm surges and extreme sea levels. Nature communications, 7, 11969, 2016.
[11] https://www.deltares.nl/en/projects/global-storm-surge-information-system-glossis
[12] Firmijn Zijl, Jelmer Veenstra, and Julien Groenenboom, The 3D Dutch Continental Shelf Model - Flexible Mesh (3D DCSMFM) Setup and validation, Deltares report 1220339-000-ZKS-0042, 2018.
[13] Genseberger et al, A new D-Flow FM (Flexible Mesh) shallow water model for Lake Marken area, poster at SIAM Conference on Mathematical and Computational Issues in the Geosciences, 2017.
[14] Menno Genseberger, Carlijn Eijsberg - Bak, Asako Fujisaki en Christophe Thiange, Ontwikkeling Zesde generatie Markermeer en Veluwerandmeren model, Deltares rapport 11200569-009-ZWS-0013, 2018.
[15] Iris Niesten and Aukje Spruyt, Development of a new Rhine branches model with Delft3D-Flexible Mesh, proceedings NCR DAYS, 2019. https://ncr-web.org/wp-content/uploads/2019/01/ncr-43-ncrdays2019-bookofabstracts-web.pdf
[16] Mohamed Yossef, Jurjen de Jong, Aukje Spruyt, and Martin Scholten, Novel approaches for large-scale two-dimensional hydrodynamic modelling of rivers, proceedings River Flow, 2018. https://www.e3s-conferences.org/articles/e3sconf/abs/2018/15/e3sconf_riverflow2018_05040/e3sconf_riverflow2018_05040.html
[17] Menno Genseberger et al, Domain decomposition in shallow water modelling for practical applications, poster presented at 22nd International Conference on Domain Decomposition Methods, Switzerland, 2013.
[18] http://mvapich.cse.ohio-state.edu/benchmarks
[19] Bas van 't Hof, Profiling van D-Flow Flexible Mesh, VORtech memo BvtH/M14.030, 2014.
[20] Menno Genseberger, Andries Paarlberg, and John Donners, Delft3D as a Service (DaaS), FORTISSIMO Final Experiment Report, 2016.
[21] Integrated Performance Monitoring: http://ipm-hpc.sourceforge.net/
[22] https://pop-coe.eu/node/69
[23] A. Sarje, S. Song, D. Jacobsen et al. Parallel Performance Optimizations on Unstructured Mesh-based Simulations. Procedia Computer Science 51, 2016-2025, 2015.
[24] J. Cornelis, S. Cools, and W. Vanroose, The communication-hiding Conjugate Gradient method with deep pipelines, Submitted to SIAM Journal on Scientific Computing, 2018.
[25] Min Si and Pavan Balaji, Process-based Asynchronous Progress Model for MPI Point-To-Point Communication, proceedings HPCC/SmartCity/DSS, 2017.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.