DOI: 10.14529/jsfi210407

High-performance Shallow Water Model for Use on Massively Parallel and Heterogeneous Computing Systems

Andrey V. Chaplygin[1], *Anatoly V. Gusev*[1,2,3], *Nikolay A. Diansky*[1,3,4]

This paper presents the shallow water model, formulated from the ocean general circulation sigma model INMOM (Institute of Numerical Mathematics Ocean Model). The shallow water model is based on software architecture, which separates the physics-related code from parallel implementation features, thereby simplifying the model's support and development. As an improvement of the two-dimensional domain decomposition method, we present the blocked-based decomposition proposing load-balanced and cache-friendly calculations on CPUs. We propose various hybrid parallel programming patterns in the shallow water model for effective calculation on massively parallel and heterogeneous computing systems and evaluate their scaling performances on the Lomonosov-2 supercomputer. We demonstrate that performance per a single grid point on GPUs dramatically decreases for small grid sizes starting from 2^{19} points per node, while performance on CPUs scales up to 2^{17} well. Although, calculations on GPUs outperform calculations on CPUs by a factor of 4.7 at 30 nodes using 60 GPUs and 360 CPU cores at 6100×4460 grid size. We demonstrate that overlapping kernel execution with data transfers on GPUs increases performance by 28%. Furthermore, we demonstrate the advantage of using the load-balancing method in the Azov Sea model on CPUs and GPUs.

Keywords: shallow water, supercomputer modeling, heterogeneous computing systems, MPI, OpenMP, CUDA.

Introduction

The current intensive development of climate models, particularly the ocean general circulation models, is associated primarily with the rapid development of computer technology. The emergence of teraflop and petaflop computing systems opened up the possibility of designing ocean models of high spatial resolution, which allows to describe meso- and submesoscales of eddy variability in the scope of long-term simulations. Today, most high-performance computing systems are heterogeneous, combining various computing processors, clearly seen from the TOP 500 list of most powerful supercomputers in the world. Such systems can generally consist of a large number of processors of various types. Nowadays, the main direction of developing heterogeneous systems is the joint use of multi-core Central Processing Units (CPUs) and massively parallel accelerators, such as Graphics Processing Units (GPUs). Supercomputer technology is rapidly developing in Russia, and the development trend is similar to the world one – this can be seen from the TOP 50 list of most powerful supercomputers in the Commonwealth of Independent States (CIS). The most powerful supercomputers of Russia, for example, "Lomonosov-2" at Lomonosov Moscow State University, are heterogeneous computing systems. Creating a model that effectively uses the resources of such heterogeneous computing systems is a complex and relevant problem nowadays [1, 13]. The variety of computing systems leads to more complex parallel programming patterns and challenges porting software for efficient usage.

The shallow water equation set is a key component in ocean general circulation models, which is difficult enough to resolve. This system of equations in ocean models is obtained for

[1]Marchuk Institute of Numerical Mathematics of the Russian Academy of Sciences, Moscow, Russia
[2]P.P. Shirshov Institute of Oceanology of the Russian Academy of Sciences, Moscow, Russia
[3]N.N. Zubov State Oceanographic Institute, Moscow, Russia
[4]Lomonosov Moscow State University, Moscow, Russia

barotropic adaptation by vertically integrating three-dimensional momentum and continuity equations. Due to the high speed of the external gravity waves, the solution of the barotropic adaptation in ocean models performs with a time step smaller by one or two orders of magnitude than the time step for solving three-dimensional equations [2]. Therefore the demanding time for solving the shallow water equation set is a significant part of the total time spent for the complete equation set of ocean hydrothermodynamics. We take the system of shallow water equations as our starting point for evaluating various parallel methods and approaches useful for ocean models. This paper considers the system of shallow water equations in the form presented in the ocean general circulation sigma model INMOM (Institute of Numerical Mathematics Ocean Model). The INMOM model is being developed at the INM RAS (Marchuk Institute of Numerical Mathematics of the Russian Academy Sciences). For more than a decade and a half, the model has been used as the oceanic block of the climate model INMCM (Institute of Numerical Mathematical Climate Model). This coupled model is so far the only representative from Russia at various stages of the international project for comparing climate models CMIP (Coupled Model Intercomparison Project), conducted under the auspices of IPCC (Intergovernmental Panel on Climate Change) [10]. The model is completely written in the Fortran 90/95 programming language. The shallow water model has been formulated that can be used both as a program block of the sigma ocean model INMOM and independently, for example, to calculate tsunami waves, tides, and wind surges [4, 6].

This paper presents a new software architecture of the shallow water model, based on the separation of concerns, which involves using various hybrid parallel programming patterns. Software architecture separates the physics-related code of the model from features of parallel implementation, thereby simplifying the support and development of the entire software package. Our approach is influenced by the PSyKAl approach, which is also based on the separation of concerns [3]. In contrast with the PSyKAl approach, our software architecture preserves the original structure of loops in computational kernels at the lowest software architecture level and adds loops over block data structures at the intermediate parallel level. That allows us to implement the following approaches:

- load-balancing and cache-friendly calculation on CPUs;
- utilizing various parallel approaches on GPUs such as overlapping kernel execution with data transfer, load-balancing, and effective calculation on computing nodes with multiple GPUs per node.

Many atmospheric and oceanic general circulation models use the uniform domain decomposition method as the baseline of parallel implementation [7, 16]. However, due to land in most ocean areas, a block-based decomposition using non-uniform partitions and load-balancing methods is more efficient. There are several implementations of block-based decomposition in ocean models. For example, the parallel version of the finite element model of the Arctic Ocean (FEMAO) uses the logical mask of wet points marking computational points for each CPU core and uses data arrays shared by all blocks [17]. In contrast, Parallel Ocean Program (POP) [20] and High Resolution Operational Model for the Baltic Sea (HIROMB) [24] allocate separate data arrays for each block and organize computations in blocks, which is more cache-friendly on CPUs. We present a block-based decomposition implemented using derived data types containing blocks that are allocated separately and distribute data among processing units. A load-balancing method using the Hilbert space-filling curves is also presented. Our novelty is an organization of computations in blocks both on CPUs and GPUs.

We present various hybrid parallel programming patterns for use on massively parallel and heterogeneous computing systems. A pure MPI and hybrid MPI-OpenMP are presented as calculation patterns on CPUs. Hybrid approaches for calculations on CPUs have recently become increasingly relevant and are used in many hydrodynamic models [1, 8, 21]. Our model uses the task-based hybrid MPI-OpenMP approach, which is more efficient compared to the widespread vector-based hybrid MPI-OpenMP approach in ocean models [6]. General-purpose computing on GPUs is becoming increasingly popular for climate modeling too. There are examples of successful porting of atmosphere and ocean models on GPUs, including shallow water models [12, 18, 25]. We present three hybrid parallel programming patterns for calculations on GPUs: hybrid MPI-CUDA, hybrid asynchronous MPI-OpenMP-CUDA, and multi GPUs per node MPI-OpenMP-CUDA calculation patterns for effective use on heterogeneous computing systems. Our novelty is using block-based decomposition on GPUs to achieve overlapping kernel execution with data transfer, calculations with load-balancing, and effective calculation on computing nodes with multiple GPUs per node. The code for GPUs is written using native CUDA Fortran syntax instead of CUDA C common in ocean modeling.

1. Description of the Shallow Water Model

The model considered in this paper is based on a system of nonlinear shallow water equations, which is written in an arbitrary orthogonal coordinate system in the following form:

$$\begin{gathered}\frac{\partial r_x r_y h u}{\partial t} + T_u(u,v) - F_u(u,v) - h r_x r_y l v + r_y h g \frac{\partial \zeta}{\partial x} = RHS_u, \\ \frac{\partial r_x r_y h v}{\partial t} + T_v(u,v) - F_v(u,v) + h r_x r_y l u + r_x h g \frac{\partial \zeta}{\partial y} = RHS_v, \\ \frac{\partial h}{\partial t} + \frac{1}{r_x r_y}\left(\frac{\partial u r_y h}{\partial x} + \frac{\partial v r_x h}{\partial y}\right) = 0,\end{gathered} \tag{1}$$

where r_x, r_y are the Lamé metric coefficients that arise while writing a system of equations in an arbitrary orthogonal coordinate system; u, v are the components of the depth-averaged horizontal velocity vector; l is the Coriolis parameter; g is the free-fall acceleration; ζ is a deviation of the sea surface height from its undisturbed state; $h = H + \zeta$ is the total depth of the ocean; H is the depth of the ocean at a rest state describing bottom topography.

The transport operators T_u, T_v are written in the divergent form:

$$\begin{aligned}T_u(u,v,h) &= \frac{\partial h r_y u u}{\partial x} + \frac{\partial h r_x v u}{\partial y} - h\left(v\frac{\partial r_y}{\partial x} - u\frac{\partial r_x}{\partial y}\right)v, \\ T_v(u,v,h) &= \frac{\partial h r_y u v}{\partial x} + \frac{\partial h r_x v v}{\partial y} + h\left(v\frac{\partial r_y}{\partial x} - u\frac{\partial r_x}{\partial y}\right)u.\end{aligned} \tag{2}$$

The viscosity operators F_u, F_v are written as the divergence of the stress tensor:

$$\begin{aligned}F_u(u,v) &= \frac{1}{r_y}\frac{\partial}{\partial x}\left(r_y^2 K D_T h\right) + \frac{1}{r_x}\frac{\partial}{\partial y}\left(r_x^2 K D_S h\right), \\ F_v(u,v) &= -\frac{1}{r_x}\frac{\partial}{\partial y}\left(r_x^2 K D_T h\right) + \frac{1}{r_y}\frac{\partial}{\partial x}\left(r_y^2 K D_S h\right),\end{aligned} \tag{3}$$

where K is the viscosity coefficient, and D_T and D_S are tension and shear components of the stress tensor:

$$D_T = \frac{r_y}{r_x}\frac{\partial}{\partial x}\left(\frac{u}{r_y}\right) - \frac{r_x}{r_y}\frac{\partial}{\partial y}\left(\frac{v}{r_x}\right),$$
$$D_S = \frac{r_x}{r_y}\frac{\partial}{\partial y}\left(\frac{u}{r_x}\right) + \frac{r_y}{r_x}\frac{\partial}{\partial x}\left(\frac{v}{r_y}\right). \quad (4)$$

The gradients of atmospheric pressure and wind friction stress are generally calculated at the right-hand side of equations RHS_u, RHS_v. At the solid boundary, no normal flow and free slip are set as velocity boundary conditions.

In the form (1)–(4), the nonlinear shallow water equations are presented in the ocean general circulation sigma model INMOM as vertically integrated momentum and continuity equations in order to resolve fast barotropic gravitational waves at a separate stage with minimum computational efforts [9]. The system of equations (1)–(4) is solved using numerical methods on the traditional Arakawa 'C' structured grid. The second-order numerical schemes on the structured grid and the explicit first-order 'leapfrog' scheme are used as spatial and temporal discretization schemes in the model, respectively. Due to the explicit time scheme, our computational method is matrix-free. More details about the form of writing a system of nonlinear shallow water equations and their numerical implementation can be found in papers [4, 9]. The shallow water model simulates extreme surges in the Azov Sea well: numerical results match ocean model results and actual observations [11]. Also, the model simulates the 2011 tsunami in Japan, which led to the Fukushima disaster, and numerical results match observations [5].

2. Software Architecture

A new software architecture for the shallow water model has been developed based on the separation of concerns. This software architecture divides program code into three layers. The lowest layer contains all subroutines required to calculate nonlinear shallow water equations, so-called computational model kernels. The highest layer is responsible for calling computational kernels and describes the time cycle of the model at a relatively high level without the knowledge of parallel data structures and parallel methods used in the model. The intermediate layer between the first two is responsible for parallel methods and approaches used in the model. That separation allows flexibly configuring the model for various computing systems without changing physics-related parts of the program.

A three-layer software architecture based on the separation of concerns has proven itself in the shallow water model of the ocean model NEMO (Nucleus for European Modeling of the Ocean) [3]. The researchers plan to implement such software architecture in the whole ocean model NEMO by 2022 [15].

As mentioned before, the shallow water model is completely written in the Fortran language introducing its specifics into the software architecture. Modules, derived types (classes), interfaces, and macros are extensively used in the code. We remind that the code for GPUs is written using native CUDA Fortran syntax. We describe each program layer of the model.

2.1. Kernel Layer

The Kernel layer contains all computational subroutines, so-called model kernels. The original non-parallelized program is a set of exactly such subroutines, which are the model's baseline. In total, there are about 15 model kernels in the shallow water model. The model kernel is a sub-

routine that consists of grid variables calculations without data synchronization inside. It means that if the subroutine has several loops over grid points and data synchronizations between them, it must be split into several subroutines (model kernels) without data synchronization inside. The Interface layer is responsible for data synchronization between model kernel calls and will be discussed later.

In the case of using CPUs for computing, the model kernel is a two-dimensional loop over grid points that updates values at grid points by numerical schemes. At this level, we work with ordinary two-dimensional arrays. Figure 1a shows a general view of the model kernel for calculation on CPUs, where nx_start, nx_end, ny_start, ny_end are boundaries of the subdomain; bnd_x1, bnd_x2, bnd_y1, bnd_y2 are boundaries of the subdomain including halo points; var is a grid variable.

```
subroutine kernel(var)
  real(wp8), intent(inout) :: var(bnd_x1:bnd_x2, bnd_y1:bnd_y2)
  [...]
  do m = nx_start, nx_end
    do n = ny_start, ny_end
      var(m, n) = [...]
    enddo
  enddo
end subroutine
```

(a) Code for CPUs

```
attributes(global) subroutine kernel(var)
  real(wp8), intent(inout) :: var(bnd_x1:bnd_x2, bnd_y1:bnd_y2)
  [...]
  m = (blockIdx%x-1)*blockDim%x + threadIdx%x + (nx_start - 1) - 1
  n = (blockIdx%y-1)*blockDim%y + threadIdx%y + (ny_start - 1) - 1
  if (m <= nx_end + 1 .and. n <= ny_end + 1) then
    var(m, n) = [...]
  endif
end subroutine
```

(b) Code for GPUs

Figure 1. The Kernel layer of the shallow water model

In the case of using GPUs for computing, the model kernel is updating values at a single grid point instead of the usual two-dimensional loop over grid points for calculation on CPUs. Each point in the grid corresponds to its thread on GPU. The model kernel launching on GPU is performed by threads that cover all grid points. Figure 1b shows a general view of the model kernel for calculation on GPUs.

In that way, 15 model kernels have been implemented for calculations on both CPUs and GPUs.

2.2. Algorithm Layer

The Algorithm layer establishes the order of calling model kernels and is the highest software architecture layer. The main time cycle of the model is described at this level. At this level, we

work with abstract data structures, such as the ocean_type class, which includes the main grid variables of shallow water equations. Figure 2 shows a code example of this layer. In this example, the sea level calculation kernel (kernel_ssh) is called first, then the velocity component calculation kernel (kernel_uv). Special envoke procedure, part of the Interface, calls model kernels and will be discussed later.

```
[...]
type(ocean_type), target :: ocean_data
procedure(empty_kernel), pointer :: kernel_ssh, kernel_uv
[...]
call envoke(ocean_data%ssh, kernel_ssh)
call envoke(ocean_data%u, ocean_data%v, kernel_uv)
[...]
```

Figure 2. The Algorithm layer of the shallow water model

2.3. Interface Layer

The intermediate layer between the model kernel and the model kernel call is the Interface layer at which parallel methods and approaches are implemented. In particular, the block-based decomposition (Section 3.1), the load-balancing method (Section 3.1), the hybrid approach using MPI and OpenMP technologies (Section 3.2), hybrid approaches using MPI, OpenMP and CUDA technologies (Section 3.3) are implemented at the Interface layer. This layer also includes code for processors synchronization, halo swaps, data transfers between CPU and GPU. The Kernel layer and the Algorithm layer contain a description of physical processes in the shallow water model. At the same time, nothing is known at these layers about features of parallel implementation hiding from them in the Interface layer. The Interface layer allows configuring the model to any target computing system flexibly. At this layer, we work with specific parallel data types, for example, the data 2D_real 8_type class, which contains distributed data across blocks and processors. In the case of using GPUs for computing, parallel data types contain symmetrical data in GPU and procedures for synchronizing this data between CPU and GPU.

Figure 3 shows the interface implementation for calculating model kernels on CPUs using a single envoke subroutine. This subroutine calls the model kernel for each data block and then synchronizes OpenMP threads and MPI processes (see Section 3.2 for more information about this approach).

Figure 3 shows the interface implementation for calculating model kernels on GPUs using a single envoke subroutine. This subroutine calls the model kernel for each data block corresponding to its GPU using a special CUDA syntax. The subroutine performs data synchronization between CPU and GPU. OpenMP threads and MPI processes are synchronized too (see Section 3.3 for more information about this approach).

3. Parallel Methods and Approaches

The following parallel methods and approaches have been implemented in the shallow water model based on the described software architecture: the block-based decomposition with load-balancing; the hybrid approach using MPI and OpenMP; hybrid approaches using MPI,

```
subroutine envoke(var, kernel)
  type(data2D_real8_type) :: var
  [...]
  !$omp do private(k) schedule(static, 1)
  do k = 1, blocks
    call kernel(var%block(k)%host)
  enddo
  !$omp end do nowait
  call sync_halo
end subroutine
```

(a) Interface for CPUs

```
subroutine envoke(var, kernel)
  type(data2D_real8_type) :: var
  [...]
  !$omp do private(k) schedule(static, 1)
  do k = 1, blocks
    cudaSetDevice(k-1)
    call kernel<<<tGrid, tBlock>>> (var%block(k)%device)
  enddo
  !$omp end do nowait
  call copy_device_to_host
  call sync_halo
  call copy_host_to_device
end subroutine
```

(b) Interface for GPUs

Figure 3. The Interface layer of the shallow water model

OpenMP and CUDA for computing on heterogeneous computing systems. Further, we describe each parallel method and approach in the model.

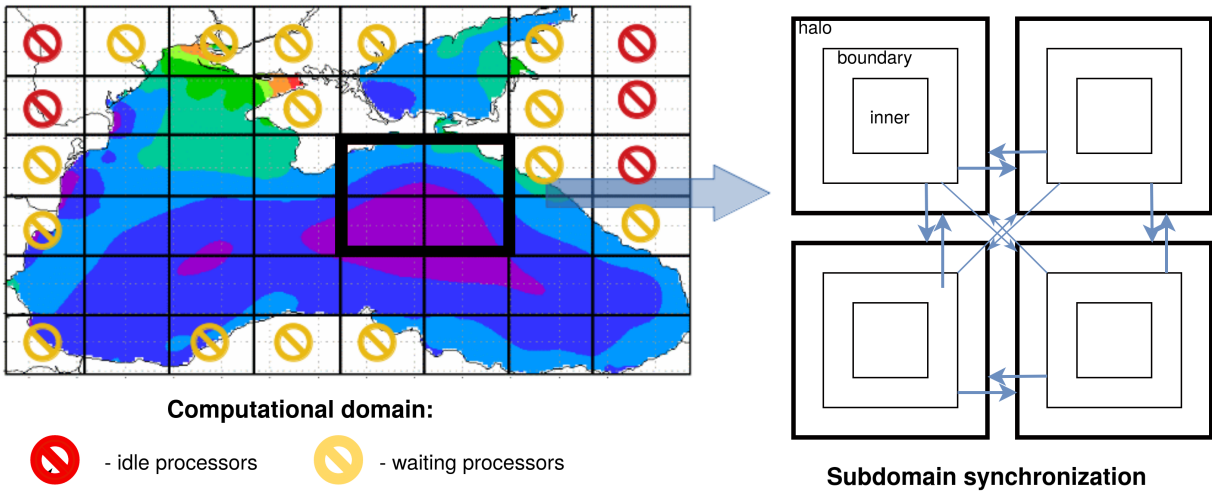


Figure 4. The uniform domain decomposition method and synchronization of processors

3.1. Block-based Decomposition and Load-balancing Method

The shallow water model uses a two-dimensional domain decomposition method as the primary parallelization method. The initial computational domain is divided into subdomains, and each processor is assigned its subdomain. Each subdomain has extra boundaries with halo points which exchanges boundary data with a neighboring subdomain. Processors are synchronized, and halo points are updated using MPI technology before each subdomain calculation. The most common and easily implemented domain decomposition method is the method of uniform subdivision into rectangular subdomains. Figure 4 demonstrates this method and the synchronization mechanism of processors with halo points updating. The figure also shows that the uniform domain decomposition method leads to workload imbalances among participating processors due to land points. Some subdomains contain only land points; some subdomains are more than half-filled with land points. Thus, some processors are idle in computing leading to performance losses. Due to islands and coasts in most ocean areas, load-balancing is an exceptionally urgent task for ocean models, particularly shallow water models [4].

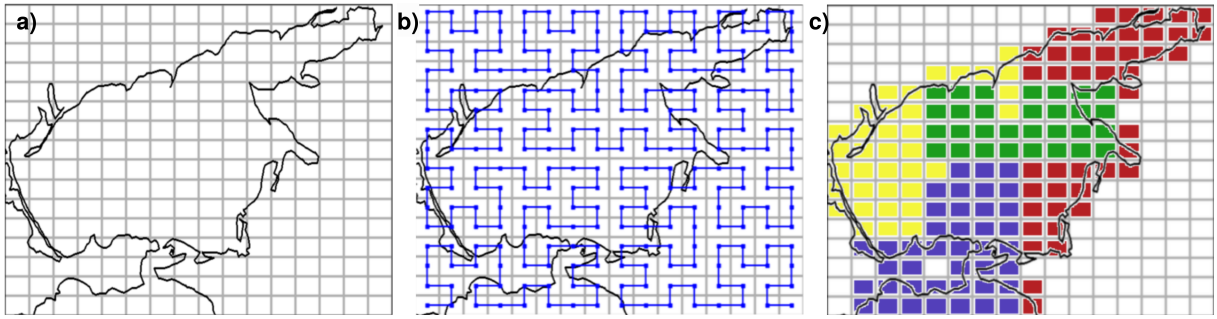


Figure 5. Load-balancing method: a) The domain is uniformly divided into blocks; b) The Hilbert curve is formed; c) Subdomains are formed along the Hilbert curve

Therefore, an improved method of subdivision into subdomains, so-called block-based decomposition, has been implemented in the model. This method uniformly divides the computational domain into rectangular blocks of small size. Then, subdomains are formed along the Hilbert space-filling curves to have approximately the same amount of workload. Land blocks are excluded from subdomains and further calculations. Each processor is assigned a certain number of blocks, which form its computational subdomain. Figure 5 clearly shows the steps of the described algorithm. All calculations in the model are organized in blocks, and each block has extra boundaries with halo points, as shown in Fig. 6. Block halo points are updated before each calculation. If a neighbor block is located on the same processor, we copy boundary values without calling the MPI library. For halo points exchanges with blocks on other processors, we use asynchronous MPI calls.

The block-based decomposition has another advantage in addition to the load-balanced distribution. It is efficient memory management while computing on CPUs, and one can get a performance increase on CPUs due to better cache behavior of smaller blocks. For hydrodynamic models, this property is essential because most of them are strongly memory-bound, so memory management is critical to the model's efficiency and performance scaling [22].

The shallow water model employs the block-based decomposition for calculations on GPUs as well as CPUs. Calculations on GPUs are also organized in blocks, and synchronizations occur in the same way as it is done for calculations on CPUs, with the addition of halo points transfer

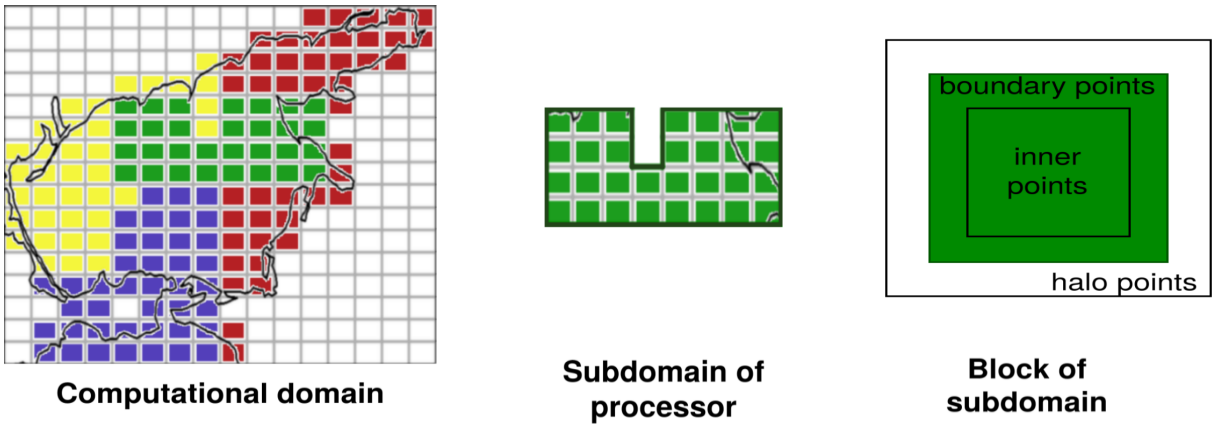


Figure 6. The block-based decomposition

between CPU and GPU. The block-based decomposition allows organizing asynchronous data transfer between CPU-GPU and overlapping calculations with memory copying and communications, leading in turn to improved performance on GPUs (see Section 3.3).

However, the block-based decomposition has a disadvantage: overheads for copying block boundary values during synchronization. Previous work [6] showed that effective work with cache memory compensates for copying overheads during synchronization for calculation on CPUs with small blocks. Therefore this disadvantage can be considered insignificant. This disadvantage becomes significant for calculations on GPUs since copying block boundary values must be carried out between CPU and GPU. Nevertheless, as it will be shown in Section 4, it is possible to improve performance using the block-based decomposition on GPUs for large computational areas due to the overlapping computations with synchronizations.

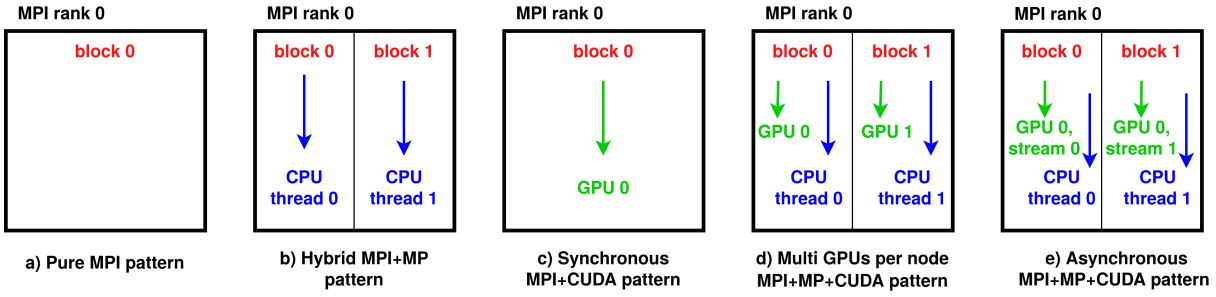


Figure 7. Parallel programming patterns implemented in the shallow water model

3.2. Hybrid Approach Using MPI and OpenMP for Use on Multiprocessor Computing Systems

Data synchronization between processors is a bottleneck in the shallow water model on multiprocessor computing systems. With an increase in the number of computing nodes, synchronization overheads increase due to the high load on the communication network. It is possible to reduce the load on the communication network using OpenMP for parallelization on shared memory inside a node. It is a so-called hybrid approach. The pure MPI approach creates a

separate MPI process for each core on a node, whereas the hybrid MPI + OpenMP approach creates only one MPI process for each node and separate threads for each core.

In the shallow water model, the task-based hybrid MPI-OpenMP approach has been implemented, which distributes subdomains (blocks from block-based decomposition) across OpenMP threads as shown in Fig. 7 compared to the pure MPI approach. Blocks are first distributed across MPI processes using the load-balancing method. Then, blocks are distributed across available threads within the MPI process, ensuring a uniform computational workload per thread. The previous work [6] has showed that this approach has the advantage compared to the widespread vector-based MPI-OpenMP hybrid approach, in which OpenMP is used only for parallelizing two-dimensional loops over subdomains. The performance of the implemented task-based hybrid approach is twice as high as the vector-based hybrid approach when calculating the model on multiple computing nodes. The previous work has also showed the advantage in performance of the task-based hybrid approach over the pure MPI approach.

3.3. Hybrid Approaches Using MPI, OpenMP and CUDA for Use on Heterogeneous Computing Systems

In the shallow water model, calculation on GPUs has been fully supported using CUDA technology. For this purpose, we have adapted 15 model kernels to calculations on GPUs. We have modified the Interface layer of the software architecture to utilize various calculation patterns on GPUs in the model. As mentioned earlier, the model kernel is a single grid point calculation on GPUs by a single thread, and it is launched by threads entirely covering the computational domain. It is necessary to note two essential details in model kernels implementation for calculation on GPUs. The first is that double precision is used everywhere in calculations, and, second is the lack of memory optimizations. The last means that no specific data placement optimizations on GPUs are implemented in model kernels; in particular, shared and texture memories are not used. All memory accesses in model kernels occur immediately to the GPU's global memory. Modern generations of GPUs are less sensitive compared to the older ones to data placement optimization, mostly due to improvements of global memory caches, as shown in [14]. The authors of this paper have considered various benchmark applications, including computational fluid dynamics solver, and showed that using different memory optimizations on modern generations of GPUs (Pascal, Volta) overall does not produce as much speedup as older ones. However, these results should be considered only as part of an overall picture. Furthermore, we have implemented model kernels for calculations on GPUs with minimal effort, and adaptation on GPUs can be further automated using macros, as done in work [3].

The implementation on GPUs has been adapted to support the block-based decomposition in the shallow water model, which has proven itself on CPUs. Due to the block-based decomposition, it is possible to balance a workload of computations on GPUs. Three parallel programming patterns have been implemented for calculations on GPUs: the synchronous MPI-CUDA pattern, the asynchronous MPI-OpenMP-CUDA pattern, the multi GPUs per node MPI-OpenMP-CUDA pattern. Note that all patterns have not been challenging to implement in the software architecture based on the separation of concerns. All code modifications have been implemented at the Interface layer without affecting the Kernel and the Algorithm layers. We describe each of the patterns in detail.

3.3.1. Synchronous MPI-CUDA calculation pattern

In this approach, each MPI process is assigned a subdomain containing only one block and a single GPU, as shown in Fig. 7. The subdomain calculation is performed entirely on GPU using CUDA. After model kernel's calculation on GPUs, processors synchronization occurs as follows:

1. Boundary points of the subdomain are transferred from GPU to CPU synchronously, meaning blocking data transfers are used.
2. MPI processes are synchronized, and halo points of each subdomain are updated. MPI synchronization is performed entirely on the CPU.
3. The updated halo points are transferred back from CPU to GPU. Data transfer is still synchronous.

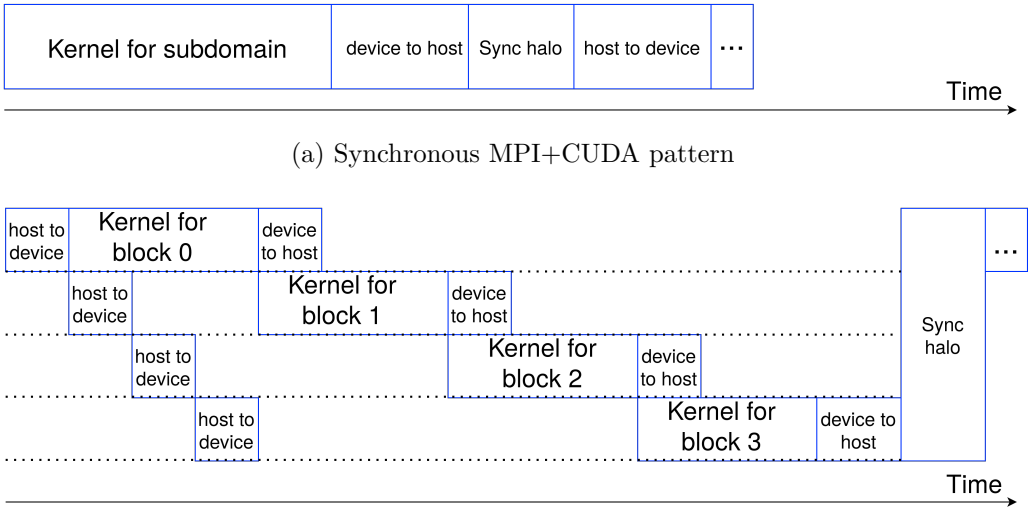
This pattern supports calculations on multiple GPUs assuming only one block per MPI process and does not support the block-based decomposition. Thus, there is no load-balancing of calculations on GPUs using this pattern.

3.3.2. Asynchronous MPI-OpenMP-CUDA calculation pattern

In this pattern, the block-based decomposition is supported for calculations on GPUs. Each MPI process is assigned a subdomain containing multiple blocks, and OpenMP threads and CUDA streams are created for each subdomain's block in the MPI process, as shown in Fig. 7. The model kernel is launched for calculation on GPU for each subdomain's block as follows:

1. Each OpenMP thread asynchronously launches the model kernel on GPU for the subdomain's block. Launching is performed in the CUDA stream corresponding to the OpenMP thread.
2. Each OpenMP thread asynchronously transfers boundary points from GPU to CPU of the subdomain's block. Data transfer is performed in the CUDA stream corresponding to the OpenMP thread.
3. All CUDA streams and OpenMP threads are synchronized.
4. MPI processes are synchronized, and halo points of each subdomain's block are updated. MPI synchronization is performed entirely on the CPU.
5. Each OpenMP thread asynchronously transfers back halo points from CPU to GPU of the subdomain's block. Data transfer is performed in the CUDA stream corresponding to the OpenMP thread.

Since modern GPUs have separate control elements for execution kernels and data transfers, this calculation pattern organizes asynchronous data transfers and overlaps kernel execution on GPU with data transfers between CPU and GPU using different CUDA streams. Figure 8 shows steps of this calculation pattern in comparison with the synchronous MPI-CUDA pattern schematically. It can be seen that data transfers overlap with kernel execution in time for the asynchronous MPI-OpenMP-CUDA pattern. This calculation pattern is completely hybrid and designed for more efficient calculation on computing nodes with one GPU per node compared to the synchronous MPI-CUDA pattern.



(a) Synchronous MPI+CUDA pattern

(b) Asynchronous MPI+OpenMP+CUDA pattern

Figure 8. The synchronous MPI-CUDA and the asynchronous MPI-OpenMP-CUDA patterns

3.3.3. Multi GPUs per node MPI-OpenMP-CUDA calculation pattern

This pattern also supports the block-based decomposition for calculations on GPUs, but differently than the asynchronous MPI-OpenMP-CUDA pattern. Each MPI process is assigned a number of blocks and OpenMP threads as many as available GPUs on a node. Accordingly, every GPU managed by a single OpenMP thread contains a single block subdomain on a node. OpenMP threads independently launch CUDA kernels, allowing efficient use of every GPU available on a node. Synchronizations are organized as in the MPI-CUDA pattern, but with synchronization of GPUs on a node and block boundaries points gathering for synchronization of MPI processes. The approach is completely hybrid and designed to calculate on computing nodes with multiple GPUs per node.

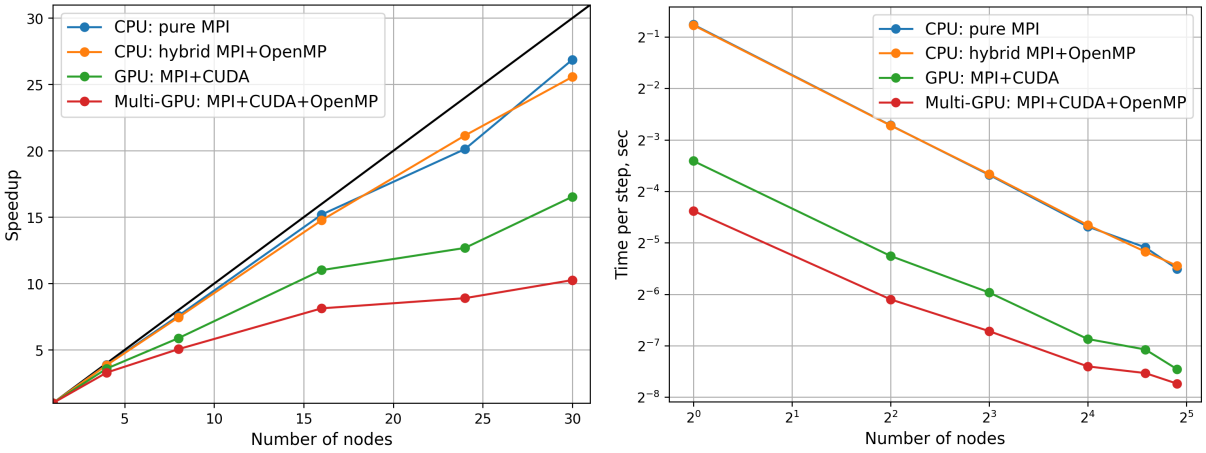
4. Results and Discussions

Our experiments were performed on the Lomonosov-2 supercomputer at Lomonosov Moscow State University [23], which is a completely heterogeneous computing system and is one of the most high-performance supercomputers in Russia today according to the TOP500 list. We performed our numerical experiments on the Pascal and Volta sections of the supercomputer, which contain modern GPUs on nodes. The Pascal section includes 160 nodes with one Intel Xeon Gold 6126 2.60GHz CPU (12 cores) and two Nvidia Tesla P100 GPUs; the Volta section includes 16 nodes with Intel Xeon Gold 6126 2.60GHz CPU (12 cores) and two Nvidia Tesla V100 GPUs. We compiled the software code using the Nvidia HPC Fortran compiler (PGI compiler), which supports native CUDA in Fortran and has recently become free. We used the optimization option -fast and libraries Open MPI v3.1.5, CUDA 11.1 for compilation.

4.1. The Box Test

The first series of experiments were performed for the Box test. This test represents the computational domain without land points and with constant sea depth. We set a gaussian water level elevation as the initial condition, and there were no wind forcing. We made this test

to demonstrate the performance scaling of the shallow water model without workload imbalances on processors. We evaluated the model's performance at different grid sizes of the computational domain. We chose computational grids corresponding to the Azov Sea grids with a resolution of 250 meters and 62.5 meters: 1525 × 1115 points and 6100 × 4460 points. We performed model simulations for one model day, with 86400 model steps in total.



(a) Speedup

(b) Time per model step (in logarithmic scale)

Figure 9. Performance scaling on the Pascal section for the Box test with 6100 × 4460 grid size

Figure 9 demonstrates the performance scaling of various calculation patterns on CPUs and GPUs at 6100 × 4460 grid size. Demonstrated results on the figure were obtained using 30 nodes of the supercomputer's Pascal section. The pure MPI and the hybrid MPI-OpenMP calculation patterns on CPUs demonstrate close to linear (black line on the figure) scaling up to 30 nodes (360 cores in total). However, the hybrid MPI-OpenMP calculation pattern failed to outperform the pure MPI pattern on this supercomputer. We assume that using 30 nodes is not enough to see the advantages of the hybrid calculation pattern over the pure MPI due to the low load on the communication network. The multi GPUs per node MPI-OpenMP-CUDA pattern allows performing calculations on 60 GPUs using 30 nodes and demonstrates the best calculation time due to the efficient use of all computing resources on a node. The multi (two exactly) GPUs per node pattern have twice better performance up to 16 nodes than one GPU per node CUDA-MPI pattern, but then the performance difference decreases due to small subdomain size per GPU. Calculation on a single GPU outperforms any calculation pattern on a single CPU by a factor of 6.3. Although the performance scaling on GPUs is worse than on CPUs, calculations on GPUs still outperform calculations on CPUs by a factor of 4.7 at 30 nodes using 60 GPUs and 360 CPU cores. We also compared the performance scaling on CPUs and GPUs at 1525 × 1115 grid size. Figure 10 shows times per a single grid point for calculations on CPUs and GPUs running 1525 × 1115 and 6100 × 4460 grid sizes. Performance per a single grid point on GPUs dramatically decreases after 2^{19} points per node, while performance on CPUs scales up to 2^{17} well. Accordingly, GPUs are much more sensitive to the grid size than CPUs in two aspects. First, communication overheads, including data transfers between CPU and GPU, can exceed the computation time and become a bottleneck in GPUs' performance. Second, small subdomains lead to better cache behavior for calculation only on CPUs but cannot saturate execution and entirely hide memory latencies on GPUs.

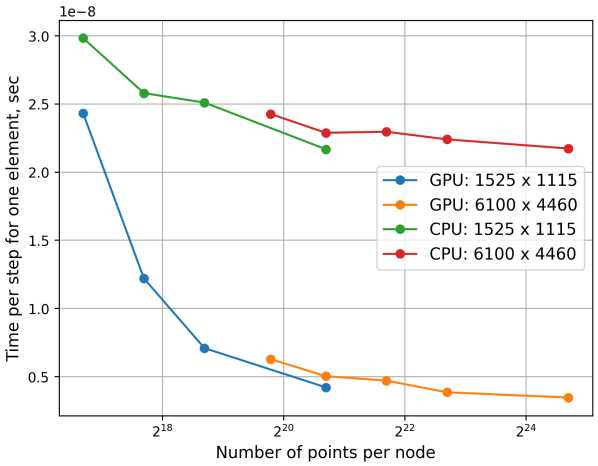


Figure 10. Time per single grid point (in logarithmic scale). The Pascal section of the supercomputer Lomonosov-2 was used

To further investigate performance scaling on GPUs, we tested the asynchronous MPI-OpenMP-CUDA calculation pattern, which overlaps data transfers with kernel execution. Experiments at 6100 × 4460 grid size were performed on the supercomputer's Volta section, including Nvidia Tesla V100 GPUs on nodes. Figure 11 demonstrates performances using a different number of blocks in the block-based decomposition on a single GPU. We see that the asynchronous calculation pattern on GPUs outperforms by 17% the synchronous calculation pattern due to kernel execution and data transfer overlapping. Figure 12 shows performance scaling of the asynchronous calculation pattern on GPUs using an optimal number of blocks (8 blocks per GPU as shown in Fig. 11) compared to the synchronous calculation pattern on GPUs. This experiment shows that the asynchronous pattern is better scaled up to 8 nodes and 28% faster on 8 GPUs than the synchronous calculation pattern. Also, this experiment clearly shows that overlapping kernel execution with data transfer compensates for overheads of copying blocks' boundary values during synchronization in the block-based decomposition on GPUs. However, this statement is valid only for sufficiently large computational domains.

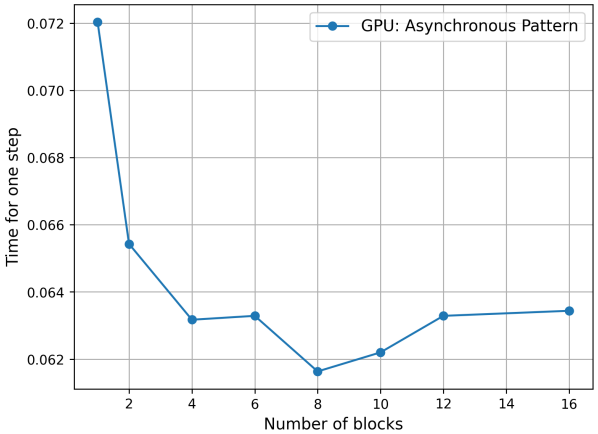
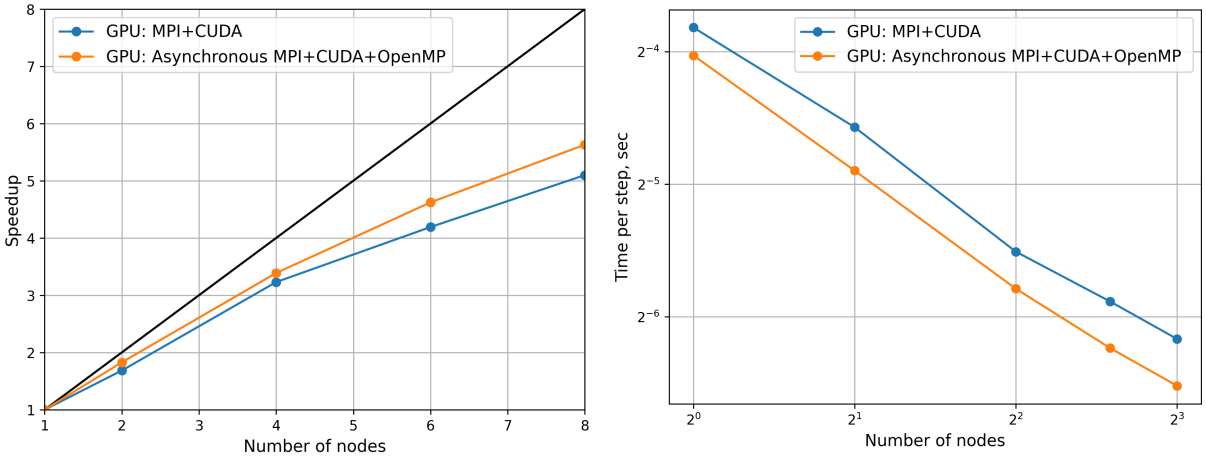


Figure 11. Time per model step (in logarithmic scale) on a single V100 GPU for the Box test with 6100 × 4460 grid size



(a) Speedup

(b) Time per model step (in logarithmic scale)

Figure 12. Performance scaling on V100 GPUs for the Box test with 6100 × 4460 grid size

4.2. The Azov Sea Test

The Azov Sea is a convenient region to test shallow water models because its dynamics and circulation can be described well by two-dimensional numerical models, thanks to small depth [19]. The computational domain of the Azov Sea has a relatively large number of land points. Figure 5 shows that more than half of the blocks is entirely land when dividing the domain into small blocks. Thus, the load-balancing method will be especially relevant here. The second series of experiments were performed for the Azov Sea to evaluate the performance scaling of the shallow water model with workload imbalances on processors and demonstrate the advantage of using the load-balancing method for calculations on CPUs and GPUs. We choose different spatial resolutions for the test: 250 meters (1525 × 1115 grid size) and 62.5 meters (6088 × 4448 grid size). It should be noted that a high resolution of the Azov Sea model is required to more correctly describe coastal flow currents and sea level changes, which are required for practical purposes, for example, for calculating level fluctuations in ports of the Azov Sea. Simulations were performed for one model day, with 86400 model steps in total.

The LB metric is responsible for balancing the partitioning in terms of computing load on processors and is defined as follows. Suppose that the partition occurs into k subdomains for p processors, then LB is defined as:

$$LB(k,p) = \frac{\max_{1\le i\le k} W_i}{\frac{1}{p}\sum_{i=1}^{k} W_i},$$

where $\max_{1\le i\le k} W_i$ – is the maximum workload of the i-th subdomain, $\sum_{i=1}^{k} W_i$ – is the full workload of the entire computational domain. The workload is computed differently for calculations on CPUs and GPUs. For CPUs, the workload of the subdomain is a sum of "wet" points in the subdomain. However, for GPUs, the workload of the subdomain is a sum of any points ("dry" and "wet"). Due to branch divergence, performance on GPUs does not drop with increasing the proportion of "wet" points, unlike on CPUs, and we take it into account. So, for calculation on GPUs, we uniformly distribute blocks from the block-based decomposition across GPUs, excluding entirely land blocks. This metric shows the ratio of the maximum subdomain's

workload to the optimal workload. The value $LB = 1$ corresponds to a perfectly load-balanced partition.

As mentioned early, the block-based decomposition divides the domain into small blocks and distributes them across available processors. On the one hand, using a small number of blocks in a partition leads to a high LB metric and unbalanced subdomains of processors. On the other hand, using a large number of blocks in a partition leads to high overheads of copying blocks' boundary values during processors synchronization, which is especially true for calculations on GPUs since we must transfer data between CPU and GPU for each block. Therefore, the number of blocks k in the partition is chosen adaptively for distribution across p processors according to the following criterion:

$$k(p) = \min_{n=0,1,\ldots} \{4^n \; s.t. \; LB(4^n, p) - LB(4^{n+1}, p) < 0.15\}. \tag{5}$$

That criterion means that optimal blocks partition is a minimum partition (4^n blocks) for which more fine partitions (4^{n+1} blocks and more) do not significantly reduce the value of the LB metric. Table 1 demonstrates optimal number of blocks for calculation on CPUs according to the described criterion; the optimal number of blocks is highlighted in the table.

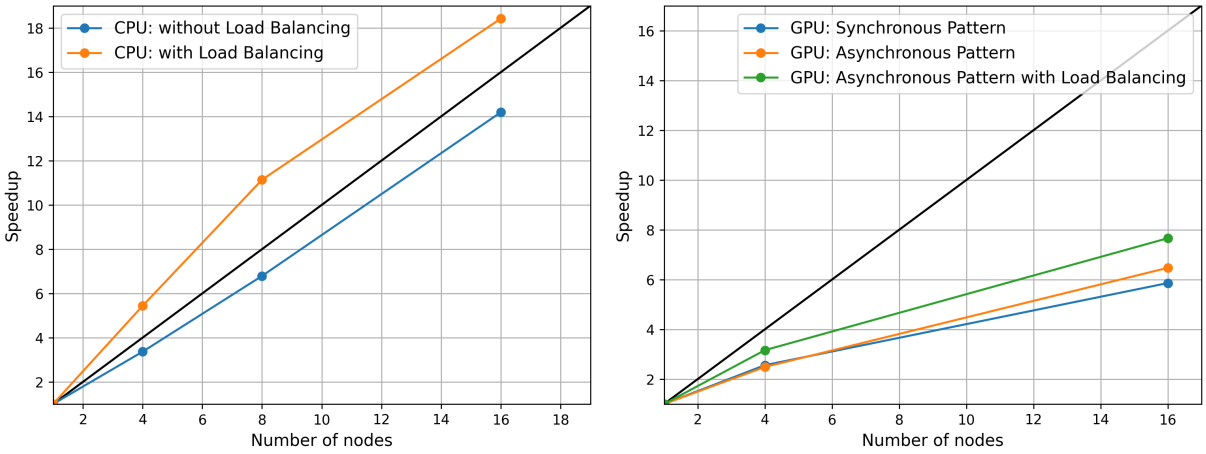
Table 1. LB metrics for the Azov Sea with 250 meters resolution

Processes	LB for 256 blocks	LB for 1024 blocks	LB for 4096 blocks
48	1.371	**1.045**	1.012
96	1.802	**1.154**	1.022
192	–	1.385	**1.070**

We tested the Azov Sea on the Pascal section of the Lomonosov-2 supercomputer. Figure 13 demonstrates performance scalings of the model with load-balancing compared to the model without load-balancing on CPUs at 1525 × 1115 grid size and on GPUs at 6088 × 4448 grid size. We considered only the pure MPI calculation pattern on CPUs and compared the asynchronous calculation pattern to the synchronous pattern on GPUs. On one node, the model was calculated without load-balancing. We used an optimal number of blocks according to criteria 5. For calculations on CPUs, Tab. 1 shows an optimal number of blocks; for GPUs, we used 64 blocks for 4 nodes and 256 blocks for 16 nodes. The figure shows that load-balancing has a significant impact on CPU computing performance: the model with load-balancing outperforms by 30% the model without load-balancing at 16 nodes and scales superlinearly due to better cache behavior of small blocks. For calculation on GPUs, it can be seen that the asynchronous pattern with load-balancing outperforms the synchronous and asynchronous patterns without load-balancing by 30% and 18% respectively at 16 nodes. Still, performance on GPUs, even with the higher resolution of the Azov Sea, scales worse than on CPUs. As mentioned before, GPUs are much more sensitive to the problem size than CPUs (see Fig. 10). That fact is especially relevant here because, with increasing nodes, the number of points per GPU dramatically decreases due to the presence of land points in the computational domain.

Conclusions

In this paper, we present the three-layer software architecture of the shallow water model based on the separation of concerns. The software architecture separates the physics-related code



(a) Speedup of CPU version for 250m resolution (b) Speedup of GPU version for 62.5m resolution

Figure 13. Performance scaling of the Azov Sea model. The Pascal section of the supercomputer Lomonosov-2 was used

from features of parallel implementation, simplifying the model's support and development. We present the blocked-based decomposition to improve the two-dimensional domain decomposition method proposing load-balanced and cache-friendly calculations on CPUs. We support the block-based decomposition for calculations on GPUs proposing overlapping kernel execution with data transfer. We present various hybrid parallel programming patterns for use on massively parallel and heterogeneous computing systems. The pure MPI and the hybrid task-based MPI-OpenMP are presented as calculation patterns on CPUs. We develop three hybrid parallel programming patterns for calculations on GPUs. The synchronous MPI-CUDA pattern supports calculations on multiple GPUs assuming only one block per MPI process and does not employ block-based decomposition. The asynchronous MPI-OpenMP-CUDA pattern overlaps kernel execution with data transfers to more effective calculation on nodes with one GPU per node than the synchronous MPI-CUDA pattern. Lastly, the multi GPUs per node MPI-OpenMP-CUDA pattern is designed to calculate on nodes with multiple GPUs per node.

We test the shallow water model on the Lomonosov-2 supercomputer at Lomonosov Moscow State University. First, we evaluate different calculation patterns' scaling performances on CPUs and GPUs at the computational domain without land points. We demonstrate that performance per a single grid point on GPUs dramatically decreases after 2^{19} points per node while performance on CPUs scales up to 2^{17} well. Although, calculations on GPUs outperform calculations on CPUs by a factor of 4.7 at 30 nodes using 360 CPU cores and 60 GPUs at 6100×4460 grid size. We demonstrate that the asynchronous MPI-OpenMP-CUDA pattern is better scaled up to 8 nodes and 28% faster on 8 GPUs than the synchronous calculation pattern. Second, we demonstrate the advantage of using the load-balancing method in the Azov Sea model. We show that the load-balancing significantly impacts computing performances: calculations with load-balancing outperform by 30% calculations without load-balancing on both CPUs and GPUs at 16 nodes.

The considered shallow water model is formulated from a barotropic solver of the ocean general circulation sigma model INMOM. This research challenges us to extend current work to the three-dimensional ocean model INMOM on heterogeneous supercomputers.

Acknowledgements

The reported study was funded by RFBR, project number 20-31-90109. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

1. Afzal, A., Ansari, Z., Faizabadi, A.R., Ramis, M.K.: Parallelization Strategies for Computational Fluid Dynamics Software: State of the Art Review. Archives of Computational Methods in Engineering 24, 337–363 (2017). https://doi.org/10.1007/s11831-016-9165-4

2. Shchepetkin, A.F., McWilliams, J.C.: The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model. Ocean Modelling 9(4), 347–404 (2005). https://doi.org/10.1016/j.ocemod.2004.08.002

3. Porter, A.R., Appleyard, J., Ashworth, M., et al.: Portable multi- and many-core performance for finite-difference or finite-element codes – application to the free-surface component of NEMO (NEMOLite2D 1.0). Geosci. Model Dev. 11, 3447–3464 (2018). https://doi.org/10.5194/gmd-11-3447-2018

4. Chaplygin, A.V., Dianskii, N.A., Gusev, A.V.: Load balancing using Hilbert space-filling curves for parallel shallow water simulations. Num. Meth. Prog. 20:1, 75–87 (2019). https://doi.org/10.26089/NumMet.v20r108

5. Chaplygin, A.V., Diansky, N.A., Gusev, A.V.: Parallel Modeling of Nonlinear Shallow Water Equation. In: Proc. 60th All-Russia Conf. on Applied Mathematics and Informatics, Moscow Institute of Physics and Technology, Dolgoprudny, Russia, November 20-26, 2017. pp. 192–194. Moscow Inst. Phys. Technol., Dolgoprudny (2017)

6. Chaplygin, A.V., Gusev, A.V.: Shallow Water Model Using a Hybrid MPI/OpenMP Parallel Programming. Problems of Informatics 1, 65–82 (2021). https://doi.org/10.24411/2073-0667-2021-10006

7. Christidis, Z.: Performance and Scaling of WRF on Three Different Parallel Supercomputers. In: Kunkel, J., Ludwig, T. (eds.) High Performance Computing. ISC High Performance 2015. Lecture Notes in Computer Science, vol. 9137, pp. 514–528. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20119-1_37

8. Akhmetova, D., Iakymchuk, R., Ekeberg, O., Laure, E.: Performance Study of Multithreaded MPI and OpenMP Tasking in a Large Scientific Code. 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 756–765. IEEE (2017) https://doi.org/10.1109/IPDPSW.2017.128

9. Diansky, N.A.: Ocean circulation modelling and research of its response to short-term and long-term atmospheric forcing. Fizmatlit, Moscow (2013)

10. Volodin, E.M., Diansky, N.A., Gusev, A.V.: Simulation and Prediction of Climate Changes in the 19th to 21st Centuries with the Institute of Numerical Mathematics, Russian Academy of Sciences, Model of the Earths Climate System. Izv., Atmos. Ocean. Phys. 49(4), 347–366 (2013)

11. Fomin, V.V., Diansky, N.A.: Simulation of Extreme Surges in the Taganrog Bay with Atmosphere and Ocean Circulation Models. Russ. Meteorol. Hydrol. 43, 843–851 (2018). https://doi.org/10.3103/S1068373918120051

12. Fu, H., Gan, L., Yang, C., et al.: Solving global shallow water equations on heterogeneous supercomputers. PLoS ONE 12(3), e0172583 (2017). https://doi.org/10.1371/journal.pone.0172583

13. Lawrence, B.N., Rezny, M., Budich, R., et al.: Crossing the chasm: how to develop weather and climate models for next generation computers? Geosci. Model Dev. 11, 1799–1821 (2018). https://doi.org/10.5194/gmd-11-1799-2018

14. Bari, M.S., Stoltzfus, L., Lin, P., et al.: Is Data Placement Optimization Still Relevant on Newer GPUs? 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Dallas, TX, USA, 2018. pp. 83–96. IEEE, 2018. https://doi.org/10.1109/PMBS.2018.8641666

15. NEMO Consortium. NEMO development strategy Version 2: 2018-2022. https://www.nemo-ocean.eu/wp-content/uploads/NEMO_Development_Strategy_Version2_2018-2022.pdf (2018)

16. Tintó, O., Acosta, M., Castrillo, M., et al.: Optimizing domain decomposition in an ocean model: the case of NEMO. Procedia Computer Science 108, 776–785 (2017). https://doi.org/10.1016/j.procs.2017.05.257

17. Perezhogin, P., Chernov, I., Iakovlev, N.: Advanced parallel implementation of the coupled oceanice model FEMAO (version 2.0) with load balancing. Geosci. Model Dev. 14, 843–857 (2021). https://doi.org/10.5194/gmd-14-843-2021

18. Reguly, I.Z., Giles, D., Gopinathan, D., et al.: The VOLNA-OP2 tsunami code (version 1.5). Geosci. Model Dev. 11, 4621–4635 (2018). https://doi.org/10.5194/gmd-11-4621-2018

19. Saburin, D.S., Elizarova, T.G.: Modelling the Azov Sea circulation and extreme surges in 2013-2014 using the regularized shallow water equations. Russian Journal of Numerical Analysis and Mathematical Modelling 33(3), 173–185 (2018). https://doi.org/10.1515/rnam-2018-0015

20. Smith, R., Jones, P., Briegleb, B., et al.: The parallel ocean program (POP) reference manual: ocean component of the Community Climate System Model (CCSM) and Community Earth System Model (CESM). http://www.cesm.ucar.edu/models/cesm1.0/pop2/doc/sci/POPRefManual.pdf

21. Liu, T., Zhuang, Y., Tian, M., et al.: Parallel Implementation and Optimization of Regional Ocean Modeling System (ROMS) Based on Sunway SW26010 Many-Core Processor. IEEE Access 7, 146170–146182 (2019). https://doi.org/10.1109/ACCESS.2019.2944922

22. van Werkhoven, B., Maassen, J., Kliphuis, M., et al.: A distributed computing approach to improve the performance of the parallel ocean program. Geosci. Model Dev. 7, 267–281 (2014). https://doi.org/10.5194/gmd-7-267-2014

23. Voevodin, Vl., Antonov, A., Nikitenko, D., et al.: Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community. Supercomputing Frontiers and Innovations 6(2), 4–11 (2019). https://doi.org/10.14529/jsfi190201

24. Wilhelmsson, T.: Parallelization of the HIROMB ocean model. https://pdfs.semanticscholar.org/ee95/be1a6bb90becdc31c84f83c343ca8daf5bdc.pdf

25. Xu, S., Huang, X., Oey, L.-Y., et al.: POM.gpu-v1.0: a GPU-based Princeton Ocean Model. Geosci. Model Dev. 8, 2815–2827 (2015). https://doi.org/10.5194/gmd-8-2815-2015