High-level framework for solving systems of the PDEs on distributed systems

Yevhen Pankevych
Faculty of Applied Sciences
Ukrainian Catholic University
Kozelnytska St 2a, L'viv, Ukraine
y.pankevych@ucu.edu.ua

Oleg Farenyuk
Institute for Condensed Matter Physics
of the National Academy of Sciences of Ukraine
Svientsitskii St 1, Lviv, Ukraine;
indrekis@icmp.lviv.ua
Faculty of Applied Sciences,
Ukrainian Catholic University
Kozelnytska St 2a, L'viv, Ukraine

***Abstract*—This work aims to develop a framework for the high-level and user-transparent distribution of calculations for solving systems of partial differential equations. Framework provides a simple high-level application programming interface and automatically partitions problems into sub-problems that are then executed on MPI-based clusters. Disposing of CUDA accelerators is supported. The management of processes and accelerators is performed by the framework using the MPI services. The performance of the developed system was analyzed, the influence of the data distribution scheme on performance was investigated, and the performance of the CPU and GPU modes of the framework was compared. Computational fluid dynamics problems on the 2D unstructured mesh were used as a benchmark task for the framework. Linear acceleration on a number of available CPUs is observed, with several super-linear cases.**

***Index Terms*—MPI, CUDA, high-performance computing, C++, computational fluid dynamics, CFD, Finite Volume Method, FVM**

I. Introduction

The constant increase in the performance of modern computers allows for performing more and more complex computing tasks. However, currently, the growth of performance of a single CPU is limited, and most progress is archived by increasing the number of available CPUs [1], motivating to search for more human-friendly methods in utilizing them for solving problems [2].

There are several similar frameworks. OneAPI Threading Building Blocks (oneTBB) is an excellent high-level framework for efficient parallel computing [3]. However, oneAPI works only with a single multiprocessor system, though heterogeneous – no provisioning for the distributed systems is provided. It is also relatively low-level from the mathematical point of view – the framework does not provide mathematical abstractions and algorithms, which are important for solving PDE systems. There are also dedicated packages for computational fluid dynamics (CFD) or for solving general systems of differential equations. One of the biggest and most well-known CFD frameworks, OpenFOAM [4], provides a large number of tools useful for solving PDEs, supports MPI-based distributed computations, and uses unstructured 3D mesh. Currently, it provides limited GPU and multi-GPU support, mainly through additional modules and third-party solutions. Additionally, though being a great framework, it has a rather steep learning curve. Similar frameworks are an active field of research, to name a few: Manapy [5] allows solving finite volume PDE on an unstructured mesh using MPI but does not support accelerators such as CUDA; in [6], a multi-GPU using MPI for OpenACC-based communications is proposed, and in [7], similar work is done using native CUDA implementation.

This work describes a framework developed for automatic distributed computing based on MPI, with an optional ability to CUDA accelerators, for solving computational fluid dynamics (CFD) problems. This class of problems was chosen because of its broad scope of application – medicine, geology, construction, chemistry, physical research, etc [8]. Within the framework, a scheme for the automatic distribution of work, a scheme of communications, a mode of operation using one or several GPUs, etc., were developed. The framework provides the user with a high-level API to specify the problem conditions, while parallelization and CUDA acceleration are transparent to the user.

The performance of the developed system was investigated for several typical use cases.

II. The sample problem

This framework focuses on solving parabolic and hyperbolic equations with time derivatives. A general form of second-order linear PDE for the function of two variables $u(x, y)$ can be written as:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0, \quad (1)$$

where $u_{ab} = \frac{\partial^2 u}{\partial a \partial b}$ and $u_a = \frac{\partial u}{\partial a}$. When $B^2 - 4AC > 0$, the equation is hyperbolic, and when $B^2 - 4AC = 0$ – parabolic.

The CFD problem described by the compressible Euler's equations [9], which are hyperbolic quasi-linear equations, were chosen for performance measurements and testing. Their conservative form is used for calculations:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho v_x\\ \rho v_y\\ \rho e\end{pmatrix}+\frac{\partial}{\partial x}\begin{pmatrix}\rho v_x\\ \rho v_x^2+P\\ \rho v_x v_y\\ (\rho e+P)v_x\end{pmatrix}+\frac{\partial}{\partial y}\begin{pmatrix}\rho v_y\\ \rho v_y v_x\\ \rho v_y^2+P\\ (\rho e+P)v_y\end{pmatrix}=0, \tag{2}$$

where ρ – density, v_x, v_y – velocity, P – pressure, γ – the ideal gas adiabatic index, e – total energy.

The Finite Volume Method (FVM) [10], which uses the integral form of these equations, on a two-dimensional unstructured mesh [11] composed of arbitrary quadrilaterals, was used to solve the problem. The cell-centered finite volume scheme was used. It defines the volumes as the cells of the mesh, limited by their faces. A 2D quadrangle unstructured mesh was used [11]. The central-upwind scheme was used to interpolate values on the interfaces:

$$\phi_i = \phi_C + \nabla\phi_C \cdot \mathbf{d}_{Ci} \tag{3}$$

where ϕ_C – value on the cell center, $\nabla\phi_C$ – gradient of the variable at this cell, $\mathbf{d}_{Ci}$ – vector in direction to the interface center from node center.

Gradients are reconstructed using the Green-Gauss method. The system of PDEs is solved using an explicit time-step scheme [12]. Its stability is controlled using the Courant-Friedrichs-Lewy (CFL) [13] condition.

III. The Framework description

A. Framework goals

The framework goal is to solve numerically user-specified PDE systems. For this, the framework should: work with the problem geometry (mesh) provided by the user, create the necessary entities on the mesh and use them to formulate a system of equations, perform simulation, visualize, and save the results. In addition, the framework should support distribution and CUDA accelerators as transparently as possible.

The FVM method of solving systems of PDEs, as well as other similar methods – finite elements methods, and finite differences methods, are highly parallel and suitable for parallel solving. So a developed framework was intended to preserve this property while providing high-level, flexible, and general API. Additionally, flexibly changing implementations and extending them was implemented – one possible future extension would be using oneAPI [14].

B. API

To use the framework, the user should perform the following sequence of actions.

- Initialize the framework by calling

```
CFDArcoGlobalInit::initialize(
    argc, argv, skip_history_flag);
```

- Create a mesh and set the geometry. There are several ways to specify the geometry: reading the geometry file; manual creation of vertices, edges, and nodes by adding corresponding objects to mesh instance; creation of the basic geometry of a uniform square mesh. Reading mesh by:

```
auto mesh = read_mesh(file_path);
```

Creating geometry mesh:

```
mesh.init_basic_internals();
```

After creating the geometry, the user should generate all the necessary structures for the current geometry by calling the corresponding method:

```
mesh.compute();
```

- Next stage is decomposing and distributing the mesh:

```
CFDArcoGlobalInit::
  make_node_distribution(
    mesh, DistributionStrategy,
    priorities);
```

Linear (naive approach) and Cluster distribution strategies are supported. Also, a vector of priorities that sets the relative parts of computation each domain should perform can be passed.

- If necessary, initialize CUDA mode:

```
CFDArcoGlobalInit::enable_cuda(mesh,
    num_of_cuda_nodes);
```

- Create variables on meshes:

```
auto var = Variable(mesh,
  initial_values,
  boundary_fn,
  cuda_boundary_fn,
  str_variable_name);
```

- Pass the initial conditions in the form of an Eigen::Vector with values for each mesh node, as well as a boundary condition wrapped in std::function. This function receives mesh and variable data for each cell and returns updated data. Several predefined boundary conditions are already implemented in the framework. A user could provide a CUDA version of this function to improve performance in CUDA mode, but it is optional.
- Create a dt variable, which specifies timestep and its updates:

```
auto dt = DT(mesh,
    update_policy_fn,
    cuda_update_policy_fn,
    arguments);
```

- Specify a system of equations using the usual operators: +, -, *, /, as well as differentiation functions: d1dx(), d1dy(), d2dx(), d2dx(). Currently, the left part may contain the actual variable or its first and second time derivatives. Constants and value vectors can also be used. Finally, the user specifies the time derivative. Example of specification:

```
equation_system = {
 {d1t(v1),'=',d1dx(v2)*d1dy(v3)},
 {d1t(v2),'=',d1dx(v4*v1)-d1dy(v1)},
}
```

- Create a problem object and run the simulation:

```
auto equation = Equation(timesteps_num);
equation.evaluate(all_vars,
    equation_system, &dt,
    show_progress_bar,
    vars_to_store_history);
```

- If necessary, visualize or save data after the simulation.

```
init_store_history_stepping(
    vars_to_store_history,
    mesh);
// ... simulation ...
finalize_history_stepping();
```

C. Framework core

To support the high-level interface, the framework builds a directed acyclic graph (tree) for the mathematical equations provided by the user. Operations act as nodes of a tree, while operands act as edges. This method is ideologically similar to using Template Expressions [15] for mathematical C++ libraries, such as Eigen [16]. This approach allows the user to modify the problem flexibly while retaining control over the computational complexity and allows mesh-based partitioning and parallel execution transparently.

There are two approaches available for mesh distribution on available computers. The first is a naive distribution depending on the number of cells of the boundary between nodes – it uses the linear index of the cell in the array of cells. The second is the use of a clustering algorithm to take into account the locality of cells and divide the mesh into local subdomains. This algorithm uses a modified k-means with constraints described in [17]. The main modification is that the described algorithm uses a random selection of the initial centroids, while in the proposed solution, the centroids are selected depending on the linear indices for potentially faster convergence. Clustering for complex geometry can reduce the number of communications, in contrast to the simple division of data without taking into account their locality.

Distributed computations are implemented by using Message Passing Interface (MPI) [18]. This allows integrating the solution into most modern cluster systems easily. The framework uses Eigen matrix library [16], and performs calculations using vector operations provided by the library, so calculations are vectorized – SIMD [19] is used[1], and calculations are optimized for efficient cache usage.

The framework supports the optional usage of the CUDA accelerators [20]. In this mode, the process will use the CUDA accelerator available on the node to minimize host calculation time. Most of the possible calculations will be performed on the accelerator, and if not explicitly requested, for example for saving the simulation history, intermediate data would not be copied to the host at all. When using CUDA, each cell of the mesh is processed by a separate CUDA thread, preserving the great scalability of the problem.

In the presence of several GPUs work optionally can be distributed between them. In the current implementation, it is allowed only when accelerators are installed on different nodes. This provides the ability to simulate problems that are limited by the amount of memory of a single GPU, as well as speed up the execution of the simulation in general.

[1]Though, the impact of it was not thoroughly measured in our work.

IV. Benchmarks

A. Methodology

The performance measurements were conducted using the three setups, representing typical use cases:

- distributed system performance using CPU only,
- single node system using a single CUDA GPU,
- distributed system with a single CUDA GPU per node.

Additionally, the impact of the clustering algorithm on system performance was measured.

To account for the impact of uncontrollable factors[2] [21], all experiments were carried out five times, and the corresponding average values and standard deviation were calculated. For CUDA GPU experiments, unless otherwise specified, the mode of operation with complete avoidance of memory transfers from the device to the host is selected to obtain and compare the peak performance. Since the framework was tested using AWS machines, testing was divided into several stages in order to avoid switching between burst and normal modes of nodes used.

B. Tests setup

The main measurements of the framework performance were carried out on a cluster of 8 AWS t3a.xlarge machines, each with four vCPUs and 16 GB of RAM. To compare the performance of GPU and CPU modes of the framework operations, a node with AMD Ryzen 7 6800H with 16 GB of DDR5 4800 MHz was used for reference CPU performance and Nvidia RTX 3060 (28 streaming multiprocessors, 3584 CUDA cores, 6 Gb RAM) with the same CPU and host RAM size. To test the multi-GPU behavior, two nodes with Nvidia T4s (40 streaming multiprocessors, 2560 CUDA cores, 16 GB RAM) and four vCPU + 16 GB of RAM were used.

C. Results

1) CPU only distributed benchmark: The experiments were performed for a uniform square mesh with the clustering algorithm disabled. The behaviour of the framework with different configurations of mesh sizes has been studied. The results are presented in the Fig. 1.

The calculations show a linear speedup when the number of nodes increases. There is also a small super-linear speedup

[2]Such as changes in the clock frequency of processors, uncontrolled number of context switching, ISR execution, etc.

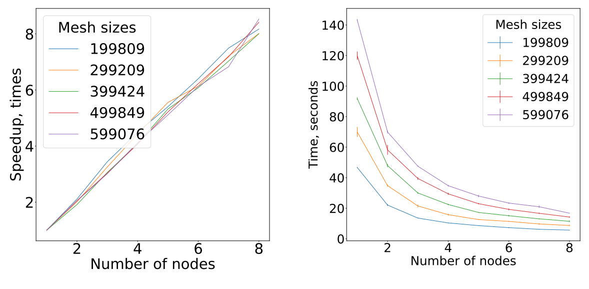


Fig. 1. The speedup effect wrt. scaling the number of active nodes in the cluster used for the task execution varying the mesh sizes: (a) Speedup ratio; (b) Mean execution time measured in seconds, (with additional bars showing the std. for the five-trials for each experiment).

when using a large number of nodes ($\approx$ 8.3 for eight nodes compared to one node), which is most likely due to better use of caches because of reduced problem size per processor. This shows a high level of framework scalability – it allows the expansion of the cluster used while maintaining high performance. Fig. 1 (b) also shows that the performance results are stable and representative – the standard deviation of the execution did not exceed 2.2 seconds or 3.3%.

2) Single node system using a single CUDA GPU benchmark: The benchmark was run in CUDA mode for a single CUDA GPU, with all calculations performed on the GPU while the single used CPU was responsible for system initialization and management. Additionally, effect of the saving the system evolution, which causes excessive memory copies between the accelerator and the host, was studied. Results of the test for the different mesh sizes are presented in Fig. 2.

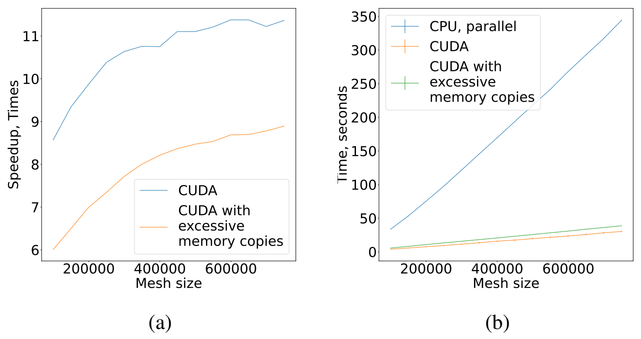


Fig. 2. The speedup effect from the usage of CUDA wrt. the mesh size. Experiment varying the usage of CUDA acceleration with and without excessive memory copies or CPU for the task execution: (a) Speedup ratio; (b) Mean execution time, (bars showing the std. for the five trials for each experiment are less than the line width).

The results show an increasing relative speedup for CUDA compared to the parallel CPU-only approach. A speedup of more than ten times was observed. On average, copying from memory adds 30-40% to the time spent compared to the non-copying version. However, as the problem size increases, the impact of these overheads decreases (from $\approx 40\%$ to $\approx 28\%$). The high performance of the CUDA mode was expected due to the good fit of the problem to a large number of available CUDA cores, but the main conclusion is that the framework efficiently manages using GPU accelerators both for problems without full-memory copying and with it. The performance measurement results are highly stable and reproducible – time fluctuations between experiments do not exceed 2%.

3) Distributed system with a single CUDA GPU per node: The test was conducted using two GPUs, each on a remote node in the cluster. To compare the results, similar tests were also conducted for one of these cards, but with the mode of memory copying between the host and the device enabled. This is necessary due to the fact that when using the two GPUs, copying is unavoidable – it is necessary to exchange the edge values of each of the domains, so for the comparability of the experiments for one card, similar copying was also carried out.

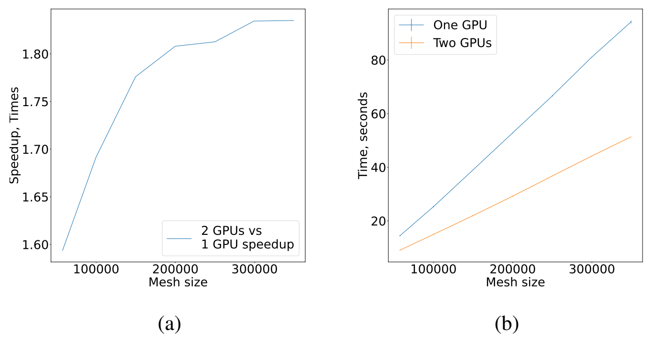
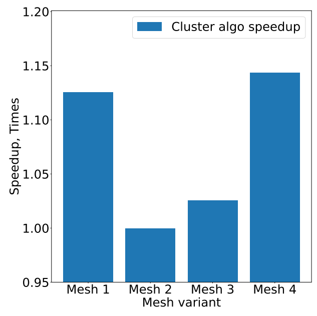


Fig. 3. The speedup effect from the usage of two GPUs wrt. the mesh size. Experiment varying the usage of single-GPU or multi-GPU (two GPUs) for compute acceleration: (a) Speedup ratio; (b) Mean execution time, (bars showing the std. for the five trials for each experiment are less than the line width).

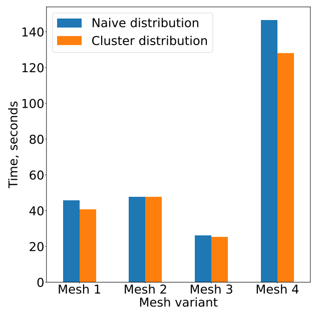
Results are presented in Fig. 3. There is an increase in speedup when the size of the problem increases, due to the decreasing weight of communications compared to calculations. The speedup reaches the value of x1.83, which is close enough to linear speedup when performed on two cards. Thus, the framework scales well enough on several accelerators, even in a distributed environment.

4) Clustering algorithm impact: This test was carried out for meshes with different geometries, comparing the performance for dividing the mesh into domains by the clustering algorithm and by the naive approach, described in III-C. The test was performed using an 8-node cluster to maximize the impact of the distribution scheme. Fig. 4 shows the time and speedup for this approach.

Results show that the division into domains with accounting for locality has a certain advantage, which can be explained by the reduction of the size of communication between nodes. However, the effect of this algorithm is not very stable – it highly depends on the shape of the mesh and varies between 0% and 12% – for some mesh types, naive distribution is good enough.



(a)



(b)

Fig. 4. The speedup effect from the naive distribution algorithm wrt. the different mesh variants. Experiment with varying the naive or clustering distribution approaches on the 8-node cluster: (a) Speedup ratio; (b) Mean execution time.

V. Conclusions

A framework for automatic distributed parallel computations for solving systems of PDEs was developed, and its properties were investigated. The framework allows optional utilization of CUDA accelerators.

Performance measurements for the model problem showed that the framework preserves close to linear scalability of the numerical solving of systems of linear PDEs both while using CPU and CUDA GPUs.

Future work could include providing additional computational backends, more thorough performance assessments including the microarchitecture benchmarks, writing the documentation, providing additional examples, and so on.

The code of the framework and examples of its use can be found in the repository on GitHub [22].

References

[1] K. Rupp, *42 years of microprocessor trend data*, 2022. [Online]. Available: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[2] W. Spataro, G. Trunfio, and G. Sirakoulis, "High performance computing in modelling and simulation," *International Journal of High Performance Computing Applications*, vol. 31, Jun. 2015. DOI: 10.1177/1094342015584473.

[3] I. Corporation, *Oneapi threading building blocks*, 2023. [Online]. Available: https://github.com/oneapi-src/oneTBB.

[4] O. Ltd, *OpenFOAM*, https://openfoam.org/, 2004.

[5] I. Kissami and A. Ratnani, "Manapy: MPI-based framework for solving partial differential equations using finite-volume on unstructured-grid," *ArXiv*, vol. abs/2203.00925, 2022.

[6] W. Xue, C. Jackson, and C. Roy, "An improved framework of GPU computing for CFD applications on structured grids using OpenACC," *Journal of Parallel and Distributed Computing*, vol. 156, Jun. 2021. DOI: 10.1016/j.jpdc.2021.05.010.

[7] J. Lai, H. Yu, Z. Tian, and H. Li, "Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters," *Scientific Programming*, vol. 2020, p. 8 862 123, Sep. 2020, ISSN: 1058-9244. DOI: 10.1155/2020/8862123.

[8] R. Raman, Y. Dewang, and J. Raghuwanshi, "A review on applications of computational fluid dynamics," vol. 2, Jul. 2018.

[9] J. K. Hunter, "An introduction to the incompressible euler equations," p. 25, Sep. 2006.

[10] J. Peiro and S. Sherwin, "Finite difference, finite element and finite volume methods for partial differential equations," in Jan. 2005, pp. 2415–2446. DOI: 10.1007/978-1-4020-3286-8_127.

[11] A. Lintermann, "Computational meshing for cfd simulations," in Oct. 2020, pp. 85–115, ISBN: 978-981-15-6715-5. DOI: 10.1007/978-981-15-6716-2_6.

[12] J. FERZIGER and M. Peric, *Computational Methods for Fluid Dynamics / J.H. Ferziger, M. Peric*. Jan. 2002, vol. 3, ISBN: 978-3-540-42074-3. DOI: 10.1007/978-3-642-56026-2.

[13] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967. DOI: 10.1147/rd.112.0215.

[14] I. Corporation, *oneAPI*, https://www.oneapi.io/, 2020.

[15] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[16] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[17] N. Ganganath, C.-T. Cheng, and C. K. Tse, "Data clustering with cluster size constraints using a modified k-means algorithm," in *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2014, pp. 158–161. DOI: 10.1109/CyberC.2014.36.

[18] L. Clarke, I. Glendinning, and R. Hempel, "The MPI message passing interface standard," *Int. J. Supercomput. Appl.*, vol. 8, May 1996. DOI: 10.1007/978-3-0348-8534-8_21.

[19] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[20] NVIDIA, P. Vingelmann, and F. H. Fitzek, *CUDA, release: 10.2.89*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[21] A. Sultanov, M. Protsyk, M. Kuzyshyn, D. Omelkina, V. Shevchuk, and O. Farenyuk, "A statistics-based performance testing methodology: A case study for the i/o bound tasks," in *2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT)*, 2022, pp. 486–489. DOI: 10.1109/CSIT56902.2022.10000626.

[22] Y. Pankevych, *CFDArco: A distributed framework for CDF problems with CUDA support*, 2023. [Online]. Available: https://github.com/yewhenp/cfdARCO.