The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-–9 June 2022, Oslo, Norway

DEVELOPMENT OF A LOW-LEVEL, ALGEBRA-BASED LIBRARY TO PROVIDE PLATFORM PORTABILITY ON HYBRID SUPERCOMPUTERS

Xavier Álvarez-Farré[1], Àdel Alsalti-Baldellou[1,2], Guillem Colomer[1], Andrey Gorobets[3], Assensi Oliva[1] and F. Xavier Trias[1]

[1] Heat and Mass Transfer Technological Center, Technical University of Catalonia,
Carrer de Colom 11, 08222 Terrassa (Barcelona), Spain; www.cttc.upc.edu
{xavier.alvarez.farre, adel.alsalti, guillem.colomer, asensio.oliva, francesc.xavier.trias}@upc.edu

[2] TermoFluids SL
Carrer de Magí Colet 8, 08204 Sabadell (Barcelona), Spain; www.termofluids.com

[3] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences,
Miusskaya Sq. 4, 125047 Moscow, Russia
andrey.gorobets@gmail.com

Key words: Platform portability, Heterogeneous computing, Hybrid supercomputer, Algebraic kernels, MPI + OpenMP + OpenCL

Abstract.

Continuous enhancement in hardware technologies enables scientific computing to advance incessantly and reach further aims. Since the start of the global race for exascale high-performance computing, massively-parallel devices of various architectures have been incorporated into the newest supercomputers, leading to an increasing hybridization of compute nodes. In this context of accelerated innovation, software portability and efficiency become crucial. Traditionally, scientific computing software development using mesh methods is based on calculations in iterative stencil loops over a discretized geometry—the mesh. Despite being intuitive and versatile, the interdependency between algorithms and their computational implementations in stencil applications usually results in a large number of subroutines and introduces an inevitable complexity when it comes to portability and sustainability. An alternative is to break the interdependency between the algorithm and its implementation, and then to cast the calculations into a minimalist set of kernels. Algebra-based implementations rely on a reduced set of basic linear algebra subroutines, which simplifies the deployment of software in hybrid computing systems. In this work, we tackle the development of a fully-portable, algebraic library that can be coupled beneath other high-level, algebra-oriented framework. Namely, this library provides platform portability in the simplest possible manner (*i.e.*, the user develops applications in a purely sequential style). Internally, algebraic objects are distributed among computing devices using a multilevel decomposition approach. Data exchanges between computing units or between nodes are hidden by a multithreaded overlapping scheme.

1 INTRODUCTION

In solving major challenges, scientific computing relies on high-performance computing (HPC) systems, also known as supercomputers. Many algorithms employed in scientific computing have a very low arithmetic intensity (AI), which is the ratio of computing work in floating-point operations (flop) to memory traffic in bytes. Hence numerical simulation codes are usually memory-bounded, making processors suffer from severe data starvation [1, 2, 3]. To top it off, the calculations often result in irregular, non-coalescing memory access patterns, reducing the memory access efficiency. Ironically, the memory bandwidth of computing hardware grows much slower than its peak performance, aggravating the problem. All this motivates the introduction of new parallel architectures with faster and more complex memory configurations into HPC systems. A quick look at the world's fastest supercomputers today [4] reveals the huge variety of hardware architectures and system configurations competing in the race for exascale computing.

The divergence in hardware architectures started back in 2004 when the increase in core's complexity and clock speeds reached a plateau, and therefore the demands for more computing power had to be met by other means [2]. Rapidly, systems based on multicore processors or multi-socket configurations seized the top supercomputers list, adding additional levels of parallelism. The default message-passing interface (MPI) parallel models assuming data equidistance between processes ceased to be valid. Hybrid approaches combining MPI and open multi-processing (OpenMP) appeared in response, although they could easily end up delivering even worse performance on complex non-uniform memory access (NUMA) configurations [5, 6]. To top it off, the divergence intensified in 2008 when devices of various architectures introducing completely different parallel paradigms came into play, such as many integrated cores (MICs) or graphics processing units (GPUs). The stream processing (SP) parallel paradigm must be met to deal with such devices. Ever since, the scientific computing community has been facing significant challenges.

The use of GPUs in scientific computing is nowadays rather mature, and there are many successful examples in the literature [7, 8, 9, 10, 11]. For instance, the early GPU implementations in [12], extended in [13], proved to be two orders of magnitude faster than its central processing unit (CPU) counterpart. Moreover, the solution of two-phase flows on multi-GPU systems [14] was not only faster but also more energy-efficient. An example of a GPU porting of an open-source Navier–Stokes solver, the AFiD code, is found in [15]. Further examples of multi-GPU simulations of supersonic and hypersonic flows can be found in [16]. One of the most impressive GPU-based simulations is found in [17], after [18], on the solution of turbulent flows, reporting a sustained performance of 13.7Pflop/s.

Traditionally, scientific computing software development using mesh methods is based on calculations in iterative stencil loops (ISL) over a discretized geometry—the mesh. In this work, this implementation approach is referred to as *stencil* or *stencil-based*. Despite being intuitive and versatile, the interdependency between algorithms and their computational implementations in stencil applications usually results in a large number of subroutines and introduces an inevitable complexity when it comes to portability and sustainability [19].

Implementing new physics or numerical methods in a stencil-based framework or its specialization for different mesh types usually requires the design of new computing subroutines and data structures. This is the main drawback of such an approach because the effort is not

necessarily accumulative and thus reduces the software's sustainability. Some authors propose domain-specific tools to address this, generalizing the stencil computations for specific fields. For instance, a framework that automatically translates stencil functions written in C++ to both CPU and GPU codes is proposed in [20]. However, these generalizations are still heavily restricted by the shape of the stencil they target.

An alternative to stencil implementations is to break the aforementioned interdependency between algorithm and implementation so that the calculations are cast into a minimalist set of universal kernels. In other words, the idea is to use the classical ISL just for data building and leave the calculations to a reduced set of basic operations; in this way, legacy codes can be used and maintained indefinitely as preprocessing tools, and the calculation engines become easy to port and optimize.

By casting discrete operators and mesh functions into sparse matrices and vectors, it is shown that all the calculations in a typical computational fluid dynamics (CFD) algorithm for the direct numerical simulation (DNS) and large-eddy simulation (LES) of incompressible turbulent flows boil down to the following basic linear algebra subroutines: sparse matrix-vector product (SpMV), linear combination of vectors (axpy) and dot product (dot) [21, 22, 6, 23]. From now on, we refer to this implementation based on algebraic subroutines as *algebraic* or *algebra-based*. In this algebraic approach, the kernel code shrinks to dozens of lines; the portability becomes natural, and maintaining OpenMP, open computing language (OpenCL), or compute unified device architecture (CUDA) implementations takes minor effort. Besides, standard libraries optimized for particular architectures (*e.g.*, cuSPARSE [24], clSPARSE [25]) can be easily linked in addition to specialized in-house implementations. A similar approach is found in PyFR [18], where the majority of operations are cast in terms of matrix-matrix multiplications linking with appropriate BLAS libraries. In the context of the DNS, the preconditioned conjugate gradient (CG) method following such an algebraic approach was implemented in [26], and its potential was exploited in [27] to perform petascale CFD simulations. Using an algebra-based formulation provided robust, portable, and optimized implementations in all cases. Consequently, the design of algebra-based algorithms for its use in massively parallel architectures seems a smart strategy for the efficient solution of both industrial and academic scale problems.

2 OVERVIEW OF MODERN SUPERCOMPUTERS

In general, hybrid supercomputers consist of multiple nodes interconnected via a high-bandwidth network (see Figure 1). An efficient distributed-memory (DM) multiple instruction, multiple data (MIMD) parallelization capable of hiding the inter-node communication overhead is required to engage the nodes of an HPC system. The MPI standard is typically used at this level. In turn, hybrid nodes combine several computing units of various architectures that feature different parallel paradigms. Indeed, there are still many powerful *traditional* supercomputers based on single-processor nodes. However, these are considered just a particular case of the more general hybrid nodes described from now on.

The CPU, the so-called *host*, consists of a pool of cores packed into NUMA nodes: different CPU sockets, even groups of cores within sockets, with separate memory banks and controllers. The non-uniform memory access allows for faster access to local memory at the expense of slower access to remote memory. A fine-tuned, NUMA-aware shared-memory (SM) MIMD paral-

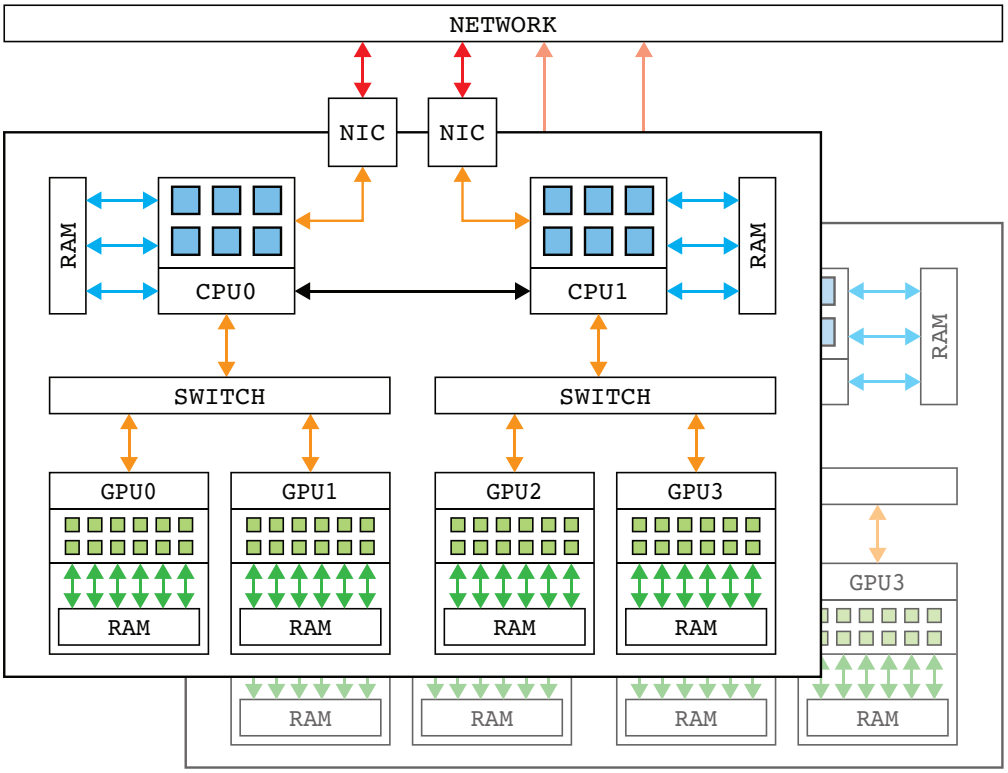

Figure 1: Example of a hybrid supercomputer consisting of two six-core CPUs and four GPUs.

lelization is required to engage all the CPU cores, ensuring thread affinity and local memory access. Moreover, modern manycore CPUs integrate dozens of cores and allow for simultaneous multithreading (SMT), from two threads per core in Intel CPUs up to eight in IBM ones. In most cases, SMT is intended for hiding memory latency, thus increasing the throughput delivered per core. The OpenMP interface is a common choice for multithreaded programming on CPUs, despite that it offers very little facility to express information about data locality or data movement. Special emphasis is also given to the increasingly larger vector registers such as advanced vector extensions (AVX), which introduce the single instruction, multiple data (SIMD) paradigm.

On the other hand, massively-parallel coprocessors, also known as *accelerators*, integrate an on-chip memory space separated from the host. Such devices are independent processors in their own right, although they are not intended for general-purpose programming. Indeed, coprocessors usually feature a limited instruction set focused on accelerating specific tasks and thus have to be driven by the host processor. For this reason, it is common to find the terms *master* and *slave* in the literature referring to such a workflow. Nowadays, the GPU is the most common accelerator, and algorithms must be compatible with the SP paradigm to deal with such processors efficiently. There are two common choices for implementing SP algorithms: the vendor-locked platform from NVIDIA, CUDA, and the open-source application programming interface (API) developed by Khronos, OpenCL. While the latter is a highly portable option compatible with virtually any hardware device, CUDA is geared towards NVIDIA GPUs only. However, in NVIDIA environments, it is usually preferred because it is easier to master and, in

most cases, delivers higher performance.

Supercomputers' topology, the way computing units and nodes are interconnected, is also paramount in code development and decision making. In computer architecture, any communication system that transfers data between components inside a node or between nodes is called a bus. Different buses are used depending on which components connect, and their specifications can vary in orders of magnitude. Complex topologies introduce an important non-uniform input-output access (NUIOA) factor because I/O devices and accelerators are directly connected to single processor sockets, and hence accessing such devices from remote sockets results slower.

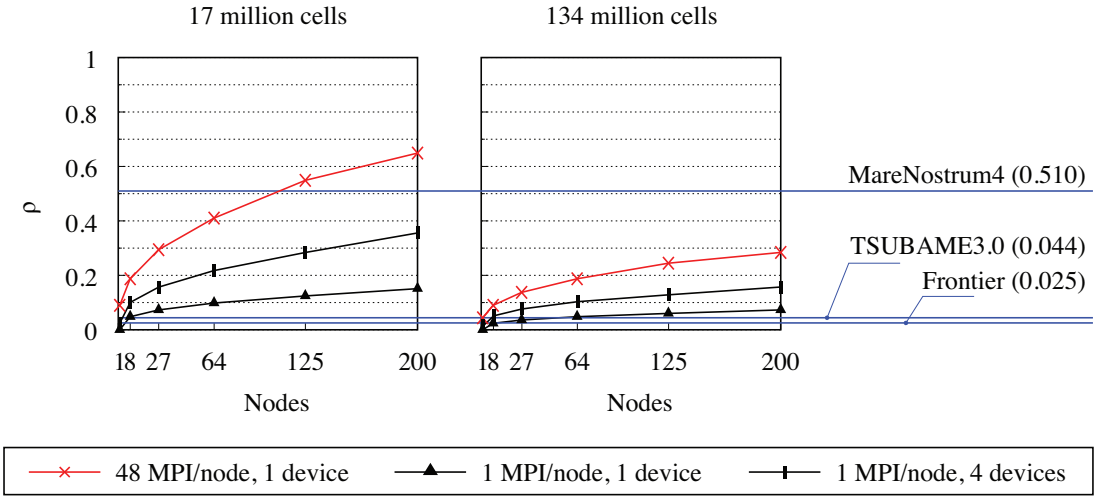

Figure 2: Ratio of halo to inner elements, ρ, with respect to different mesh partitioning approaches considering a Laplacian matrix derived from the symmetry-preserving discretization [28] on a hex-dominant unstructured mesh. Blue lines represent the theoretical maximum ratio supported by different supercomputers.

Unfortunately, the efforts to increase the memory bandwidth of compute nodes have strengthened another bottleneck. Hitherto, the B/F ratio was the major enemy of most scientific computing applications; now, the network to main memory balance (the N/B ratio) is also very harmful, specially in large-scale DM computing. Namely, the N/B ratio in a traditional CPU-based system such as MareNostrum 4 is 12.5 to 256GB/s. An application must access the main memory at least 20 times more than the network to be able to scale. This means that the SpMV considering a Laplacian matrix derived from the symmetry-preserving discretization [28] on a hex-dominant unstructured mesh (*i.e.*, seven non-zeros per row) supports a maximum of 50% halo elements. In Frontier this value falls to 2.5%. On the other hand, the distribution of a 134 million cells mesh among 8 processes results in 2.3% halo elements. Therefore, future implementations must minimize the total amount of halo in the system. Implementations relying on one or multiple MPI process per device can be considered obsolete in favor of a single MPI process per node, as shown in Figure 2. Furthermore, optimized NUMA-aware, multithreaded data exchange protocols are required.

3 THE HPC^2 FRAMEWORK

3.1 Motivation

Let us assume there is a framework that provides us with discrete differential operators (matrices) and fields (vectors), without going into detail about the numerical method used to discretize. Matrices are provided in standard compressed sparse row (CSR) format, while vectors are given as one-dimensional arrays. No workload distribution or partitioning is regarded yet. Following the nomenclature in [28] consider, for instance, the evaluation of the heat flux as

$$\boldsymbol{q}_s = -k\mathsf{G}\boldsymbol{T}_c. \tag{1}$$

Presuming that the discrete gradient operator and temperature field are given, we want to write Equation 1 in our computer program as

```
1 q = - k*G*T;
```

where `T` is the discrete temperature field, built as an element of the vector space the gradient maps from, `G` is the discrete gradient operator, `k` is the thermal conductivity and `q` is the discrete heat flux, built as an element of the vector space the gradient maps to.

The hierarchical parallel code for high-performance computing (HPC^2) project aims at computing any algebra-based model on, say, both a laptop and a hybrid supercomputer without changing any line of the code. Namely, it would be interesting for a research laboratory to execute a large-scale DNS of turbulent flow on a massively parallel supercomputer (*e.g.*, via a PRACE project on MareNostrum 4) and then to compute multiple overnight industrial simulations on different GPU-accelerated nodes. Thus, such a framework must rely on data structures and kernels that are appropriately managed at a lower-level code block, a black box that is simply configured through command-line parameters.

3.2 Portable implementation model

In a nutshell, our portable implementation model introduces three types of objects: *actuator*, *container* and *shaper*. A shaper is some sort of instructions manual that allows transforming any initial sequential input into a set of hierarchical partitions enabled for parallel processing. Containers are the data storage objects that stock such hierarchical partitions. Finally, the actuator is the object that provides with methods and functions to firstly create shapes and then manipulate and operate containers. This portable implementation model will be described throughout the section.

The solution we propose is depicted in Figure 3. Namely, there are four objects conceived for developers and four for users. The set of objects intended for users are nothing but high-level handlers or *wrappers* that contain lists of their low-level counterparts. Recall that the hierarchical parallel implementation of HPC^2 is designed to work with multilevel partitions as described in [6]. Therefore, each high-level object operates with the required number of nth-level partitions on the backs of the users. In other words, no specification of the parallel environment or configuration is required in deploying an algorithm using HPC^2 library (the user can use HPC^2 objects in a way as simple as `std::vector`). This is illustrated in Listing 1 following the example in Equation 1.

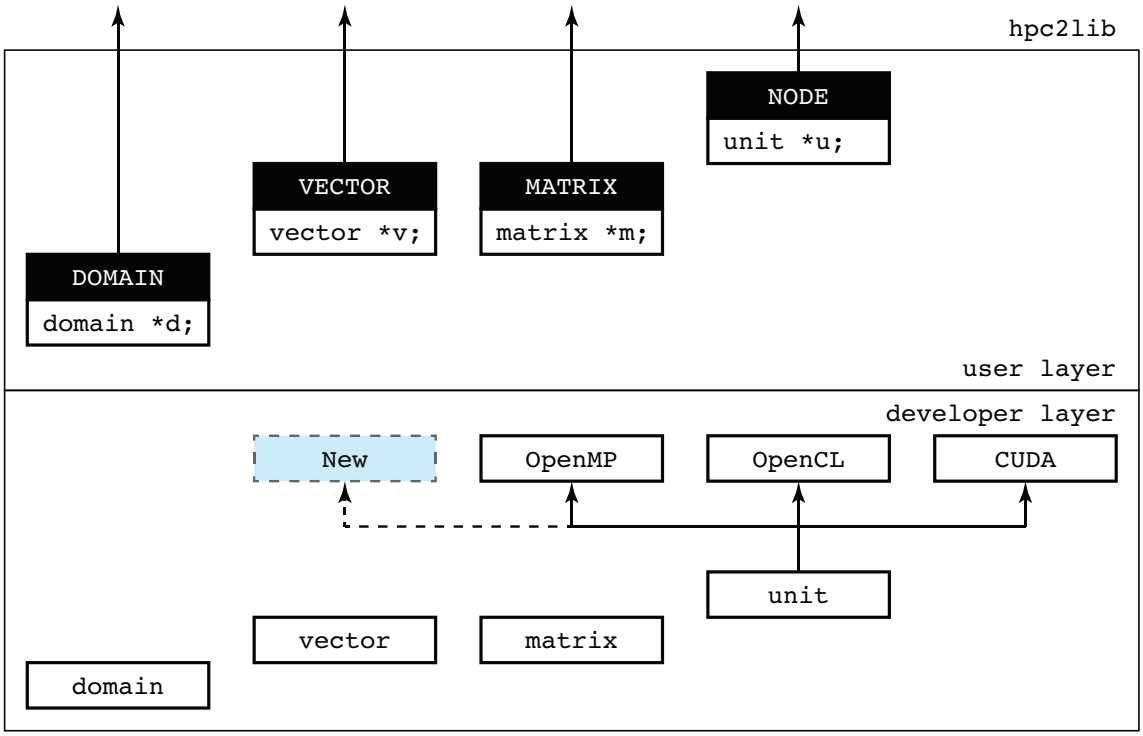

Figure 3: Diagram of classes in our pure virtual approach for managing platform portability.

Listing 1: Heat flux computation using hpc2lib.

```
#include "hpc2lib.h"

/* NODE object is a singleton and is accessed through a pointer */
NODE *Node;

int main(int argc, char** argv){
    /* initialize MPI stuff */
    int prov, req = MPI_THREAD_MULTIPLE;
    MPI_Init_thread(&argc, &argv, req, &prov);

    /* initialize NODE with command line arguments */
    Node->Init(&argc, &argv);

    /* initialize HPC2 objects using plain sequential data in binary files */
    DOMAIN Cells, Faces;
    Node->CreateDomain(Cells, Faces, "input_G.bin");

    VECTOR q = Node->BuildVector(Faces, 0.0);
    VECTOR T = Node->BuildVector(Cells, "input_T.bin");
    MATRIX G = Node->BuildMatrix(Cells, Faces, "input_G.bin");

    /* compute the gradient */
    double k = 0.598;
    Node->SpMV(G, T, q, -k);

    MPI_Finalize();
}
```

The code in the example above (Listing 1) can be compiled and executed on virtually any modern computing environment. By way of example, Listing 2 shows three command-line configurations to execute the simulation on: 1) 200 nodes with 48 cores each, 2) a fat node with 4 NVIDIA GPUs, 3) two hybrid nodes with one 14-core CPU and one AMD GPU each.

Listing 2: Execution of an application with HPC^2 on different parallel, potentially hybrid systems. Only command-line parameters change.

```
1 mpirun -np 200 ./heat -devices=1 -imp=openmp -thr=48
2 mpirun -np 1 ./heat -devices=4 -imp=cuda,cuda,cuda,cuda
3 mpirun -np 2 ./heat -devices=2 -imp=openmp,opencl -thr=13,1 -wgt=68,288
```

The parameter `devices` determines the number of *virtual* devices to be used, that is, the number of second-level partitions. We specify virtual devices to highlight that a physical computing unit or device can be assigned to multiple virtual devices and, therefore, to multiple second-level partitions. This is particularly interesting in units with device queues such as GPUs. Then, at least one processor thread is assigned to each virtual device. It is possible to assign multiple threads to a virtual device using the parameter `thr`, which is particularly necessary in multi-core units. To deal with load balancing, the parameter `wgt` determines the relative workload for each virtual device. Last, but not least, the parameter `imp` determines the implementation, or backend, assigned to a virtual device. In this portable implementation, all architecture-specific implementations is encapsulated in a single, pure virtual class (the actuator).

Unit and node

The node is the actuator provided to the user. As a singleton class, only one instance exists during the execution; and it always exist [29]. Node must be initialized at the beginning of the program using command-line parameters (line 3 of Listing 1) to determine the number of virtual devices, their implementation and relative weight, or the number of threads that operate. Then, if the user wants to create a domain, he uses the node (line 16 of Listing 1); if he wants to build vectors or matrices using plain binary files, he uses the node (lines 18–20 of Listing 1); if he wants to execute kernel, he uses the node (line 24 of Listing 1). At a glance it may seem a sequential application, but a powerful hierarchical parallel implementation is managed in the background.

In this approach, all the architecture-specific implementation is encapsulated in a single low-level, pure virtual class, the virtual unit, which can be specialized for different architectures. Currently, there are three different implementations of virtual unit: OpenMP, OpenCL, and CUDA. Each implementation is designed to allocate and operate second-level partitions of vectors and matrices. In each execution, the node will generate as many derived instances of virtual units as requested by the parameters, one per second-level partition. If a new architecture or parallel paradigm comes into play, the implementation of a new virtual unit is sufficient to port the entire numerical simulation framework.

Vector and matrix

This pair of objects are containers used to store discrete mesh functions and operators. Vectors must be bound to a domain so that the user-given plain data, the input binary file, is transformed and distributed accordingly among the required compute nodes and units. Moreover, matrices

are bound to two domains representing the input and output vector spaces. The former is used to renumber column indices, while the latter is used to transform and distribute matrix rows among the hardware.

There are multiple sparse matrix storage formats deployed (*e.g.*, the diagonal format, the standard CSR, or the ELLPACK [30] and its variants [31]), and more can be added easily.

Domain

This object describes a (computational) vector space, and contains the information required to transform such space from the user-given plain or sequential form, the input binary file, into the HPC^2-based multilevel form. It is required for allocating and operating containers, which are vector and matrix. Therefore, the domain is the shaper that describes the size of second- and third- level partitions of the vector space and the size and offset of its subsets. Besides, it contains maps to reorder the data back and forth, and to perform the required data exchanges in DM parallelization.

4 CONCLUSIONS

A portable implementation model of a low-level, algebra-based library has been proposed. It is based on three types of objects: actuator, container, and shaper. In this approach, all the architecture-specific implementation is encapsulated in a single low-level, pure virtual class, the virtual unit, which can be specialized for different architectures. It has been shown that implementing a new virtual unit is sufficient to port the entire numerical simulation framework.

On the other hand, the architecture-agnostic implementation model presented in this work allows users of higher-level, algebra-oriented frameworks developing codes in a purely sequential fancy, while still executing simulations on the most modern hybrid supercomputers.

ACKNOWLEDGEMENTS

The work of X.Á.F, À.A.B, A.O., and F.X.T. has been financially supported by the following R+D projects: RETOtwin (PDC2021-120970-I00), given by MCIN/AEI/10.13039/501100011033 and European Union Next Generation EU/PRTR, FusionCAT (001-P-001722), given by *Generalitat de Catalunya* RIS3CAT-FEDER. X. Á. F. has also been supported by a predoctoral contract (2019FI_B2-00076) by the Government of Catalonia. À.A.B has also been supported by the predoctoral grants DIN2018-010061 and 2019-DI-90, given by MCIN/AEI/10.13039/501100011033 and the Catalan Agency for Management of University and Research Grants (AGAUR), respectively. The work of A. G. has been funded by the Russian Science Foundation, project 19-11-00299. The studies of this work have been carried out using computational resources of the Barcelona Supercomputing Center (IM-2020-3-0030 and IM-2022-1-0015). The authors thankfully acknowledge these institutions.

REFERENCES

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[2] Peter M. Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science Engineering*, 15(6):16–26, 2013.

[3] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

[4] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 list – November 2021. `https://www.top500.org/lists/top500/list/2021/11/`, November 2021.

[5] Xavier Álvarez-Farré, Andrey Gorobets, F. Xavier Trias, and Assensi Oliva. NUMA-aware strategies for the heterogeneous execution of SpMV on modern supercomputers. In *World Congress in Computational Mechanics and ECCOMAS Congress*, January 2021.

[6] Xavier Álvarez-Farré, Andrey Gorobets, and F. Xavier Trias. A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers. *Computers & Fluids*, 214:104768, 2021.

[7] Joshua Romero, Jacob Crabill, Jerry E. Watkins, Freddie D. Witherden, and Antony Jameson. ZEFR: A GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method. *Computer Physics Communications*, 250:107169, 2020.

[8] Fei Jiang, Kazuki Matsumura, Junji Ohgi, and Xian Chen. A GPU-accelerated fluid–structure-interaction solver developed by coupling finite element and lattice Boltzmann methods. *Computer Physics Communications*, 259:107661, 2021.

[9] Seiya Watanabe and Takayuki Aoki. Large-scale flow simulations using lattice boltzmann method with AMR following free-surface on multiple GPUs. *Computer Physics Communications*, 264:107871, 2021.

[10] Sanghyun Ha, Junshin Park, and Donghyun You. A multi-GPU method for ADI-based fractional-step integration of incompressible Navier–Stokes equations. *Computer Physics Communications*, 265:107999, 2021.

[11] Andrey Gorobets and Pavel Bakhvalov. Heterogeneous CPU+GPU parallelization for high-accuracy scale-resolving simulations of compressible turbulent flows on hybrid supercomputers. *Computer Physics Communications*, 271(108231), 2022.

[12] Akinori Yamanaka, Takayuki Aoki, Satoi Ogawa, and Tomohiro Takaki. GPU-accelerated phase-field simulation of dendritic solidification in a binary alloy. *Journal of Crystal Growth*, 318(1):40–45, 2011.

[13] Shinji Sakane, Tomohiro Takaki, Munekazu Ohno, Yasushi Shibuta, and Takayuki Aoki. Acceleration of phase-field lattice Boltzmann simulation of dendrite growth with thermosolutal convection by the multi-GPUs parallel computation with multiple mesh and time step method. *Modelling and Simulation in Materials Science and Engineering*, 27(5):054004, jul 2019.

[14] Peter Zaspel and Michael Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Computers & Fluids*, 80(1):356–364, 2013.

[15] Xiaojue Zhu, Everett Phillips, Vamsi Spandan, John Donners, Gregory Ruetsch, Joshua Romero, Rodolfo Ostilla-Mónico, Yantao Yang, Detlef Lohse, Roberto Verzicco, Massimiliano Fatica, and Richard J.A.M. Stevens. AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters. *Computer Physics Communications*, 229:199–210, aug 2018.

[16] Aleksey N. Bocharov, Nikolay M. Evstigneev, Pavel V. Petrovskiy, Oleg I. Ryabkov, and Igor O. Teplyakov. Implicit method for the solution of supersonic and hypersonic 3D flow problems with lower-upper symmetric-Gauss-Seidel preconditioner on multiple graphics processing units. *Journal of Computational Physics*, 406:109189, apr 2020.

[17] Peter Vincent, Freddie Witherden, Brian Vermeire, Jin Seok Park, and Arvind Iyer. Towards green aviation with Python at petascale. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, nov 2016. IEEE.

[18] Freddie Witherden, Antony M. Farrington, and Peter E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, nov 2014.

[19] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 177–186, New York, NY, USA, jun 2015. ACM.

[20] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. High-productivity framework for large-scale GPU/CPU stencil applications. *Procedia Computer Science*, 80:1646–1657, 2016.

[21] Guillermo Oyarzun, Ricard Borrell, Andrey Gorobets, and Assensi Oliva. Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers. *International Journal of Computational Fluid Dynamics*, 31(9):396–411, 2017.

[22] Xavier Álvarez-Farré, Andrey Gorobets, F. Xavier Trias, Ricard Borrell, and Guillermo Oyarzún. HPC^2–A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD. *Computers & Fluids*, 173:285–292, 2018.

[23] Nicolás Valle, Xavier Álvarez-Farré, Andrey Gorobets, Jesús Castro, Assensi Oliva, and F. Xavier Trias. On the implementation of flux limiters in algebraic frameworks. *Computer Physics Communications*, 271(108230), 2022.

[24] cuSPARSE: The API reference guide for cuSPARSE, the CUDA sparse matrix library. Technical Report March, NVIDIA Corporation, 2020.

[25] Joseph L. Greathouse, Kent Knox, Jakub Poła, Kiran Varaganti, and Mayank Daga. clSPARSE: A vendor-optimized open-source sparse BLAS library. In *IWOCL '16: Proceedings of the 4th International Workshop on OpenCL*, New York, NY, USA, apr 2016. ACM.

[26] Guillermo Oyarzun, Ricard Borrell, Andrey Gorobets, and Assensi Oliva. MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids*, 92:244–252, mar 2014.

[27] Ricard Borrell, Jorge Chiva, Oriol Lehmkuhl, Guillermo Oyarzun, Ivette Rodríguez, and Assensi Oliva. Optimising the Termofluids CFD code for petascale simulations. *International Journal of Computational Fluid Dynamics*, 30(6):425–430, jul 2016.

[28] F. Xavier Trias, Oriol Lehmkuhl, Assensi Oliva, Carles David Pérez-Segarra, and R. W. C. P. Verstappen. Symmetry-preserving discretization of Navier–Stokes equations on collocated unstructured meshes. *Journal of Computational Physics*, 258:246–267, 2014.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2009.

[30] David R. Kincaid, Thomas C. Oppe, and David M. Young. ITPACKV 2D user's guide. Technical report, Center for Numerical Analysis, University of Texas, 1989.

[31] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.*, 43(4), jan 2017.