

GridapDistributed: a massively parallel finite element toolbox in Julia

Santiago Badia^{1, 2}, Alberto F. Martín^{*1}, and Francesc Verdugo²

¹ School of Mathematics, Monash University, Clayton, Victoria, 3800, Australia. ² Centre Internacional de Mètodes Numèrics en Enginyeria, Esteve Terrades 5, E-08860 Castelldefels, Spain.

Summary

GridapDistributed is a registered Julia (Bezanson et al., 2017) software package which provides fully-parallel distributed memory data structures for the Finite Element (FE) numerical solution of mathematical models of physical processes governed by (systems of) Partial Differential Equations (PDEs) on parallel computers, from multi-core CPU desktop computers, to HPC clusters and supercomputers. These distributed data structures are designed to mirror as far as possible their counterparts in the Gridap (Badia & Verdugo, 2020) Julia software package, while implementing/leveraging most of their abstract interfaces (see (Verdugo & Badia, 2021) for a detailed overview of the software design of Gridap). As a result, sequential Julia scripts written in the high-level Application Programming Interface (API) of Gridap can be used verbatim up to minor adjustments in a parallel distributed memory context using GridapDistributed. This equips end-users with a tool for the development of simulation codes able to solve real-world application problems on the vast amount of computational resources available at HPC clusters and supercomputers while using a highly expressive, compact syntax, that resembles mathematical notation. This is indeed one of the main advantages of GridapDistributed and a major design goal that we pursue.

In order to scale FE simulations to large core counts, the mesh used to discretize the computational domain on which the PDE is posed must be partitioned (distributed) among the parallel tasks such that each of these only holds a local portion of the global mesh. The same requirement applies to the rest of data structures in the FE simulation pipeline, i.e., FE space, linear system, solvers, data output, etc. The local portion of each task is composed by a set of cells that it owns, i.e., the *local cells* of the task, and a set of off-processor cells (owned by remote processors) which are in touch with its local cells, i.e., the *ghost cells* of the task. (Badia et al., 2020) This overlapped mesh partition is used by GridapDistributed, among others, to exchange data among nearest neighbours, and to glue together global Degrees of Freedom (DoFs) which are sitting on the interface among subdomains. Following this design principle, GridapDistributed provides scalable parallel data structures for simple grid handling (in particular, Cartesian-like meshes of arbitrary-dimensional, topologically n-cube domains), FE spaces setup, and distributed linear system assembly. It is in our future plans to provide highly scalable linear and nonlinear solvers tailored for the FE discretization of PDEs (e.g., linear and nonlinear geometric multigrid and domain decomposition preconditioners). In the meantime, however, GridapDistributed can be combined with other Julia packages in order to realize the full potential required in real-world applications. These packages and their relation with GridapDistributed are overviewed in the next section.




Building blocks and composability

Figure 1 depicts the relation among GridapDistributed and other packages in the Julia package ecosystem. The interaction of GridapDistributed and its dependencies is mainly designed

^{*}Corresponding author.

DOI: [DOIunavailable](#)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Pending Editor](#) 

Reviewers:

- [@Pending Reviewers](#)

Submitted: N/A

Published: N/A

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

In partnership with



This article and software are linked with research article DOI [10.3847/xxxxx](#), published in the Journal of Open Source Software.

with separation of concerns in mind towards high composability and modularity. On the one hand, Gridap provides a rich set of abstract types/interfaces suitable for the FE solution of PDEs (see (Verdugo & Badia, 2021) for more details). It also provides realizations (implementations) of these abstractions tailored to serial/multi-threaded computing environments. GridapDistributed **implements** these abstractions for parallel distributed-memory computing environments. To this end, GridapDistributed also leverages (**uses**) the serial realizations in Gridap and associated methods to handle the local portion on each parallel task. (See Figure 1 arrow labels.) On the other hand, GridapDistributed relies on PartitionedArrays (Verdugo, 2021) in order to handle the parallel execution model (e.g., message-passing via the Message Passing Interface (MPI) (Message Passing Interface Forum, 2021)), global data distribution layout, and communication among tasks. PartitionedArrays also provides a parallel implementation of partitioned global linear systems (i.e., linear algebra vectors and sparse matrices) as needed in grid-based numerical simulations. While PartitionedArrays is a stand-alone package, segregated from GridapDistributed, it was designed with parallel FE packages such as GridapDistributed in mind. We cover the main abstractions and design principles behind PartitionedArrays in the sequel.

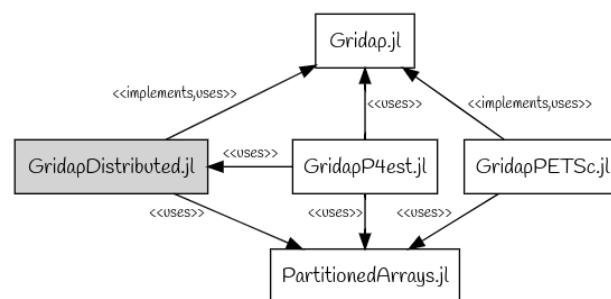


Figure 1: GridapDistributed and its relation to other packages in the Julia package ecosystem. In this diagram, each rectangle represents a Julia package, while the (directed) arrows represent relations (dependencies) among packages. Both the direction of the arrow and the label attached to the arrows are used to denote the nature of the relation. Thus, e.g., GridapDistributed depends on Gridap and PartitionedArrays, and GridapPETSc depends on Gridap and PartitionedArrays. Note that, in the diagram, the arrow direction is relevant, e.g., GridapPETSc depends on GridapDistributed but not conversely.

As mentioned above, PartitionedArrays is Julia software package which provides a set of key tools for the distributed-memory parallelization of grid-based discretization methods, such as Finite Difference (FD), Finite Volume (FV) or FE methods. Figure 2 presents the main software abstractions in PartitionedArrays and their relations. With a central role in PartitionedArrays, we have the AbstractBackend and AbstractPData abstract Julia types. The former is a software abstraction for a group of parallel tasks and the communication layer that orchestrates their concurrent execution. The latter is a low-level abstract type (i.e., users are not exposed to it) representing data partitioned over several chunks or parts (e.g., a global mesh partitioned into several local meshes). The main rationale behind these data types is that they allow to implement parallel algorithms in a generic way (e.g., those associated to the PRange, PVector and PSparseMatrix types, to be covered later) independently of the underlying hardware/message-passing programming model.

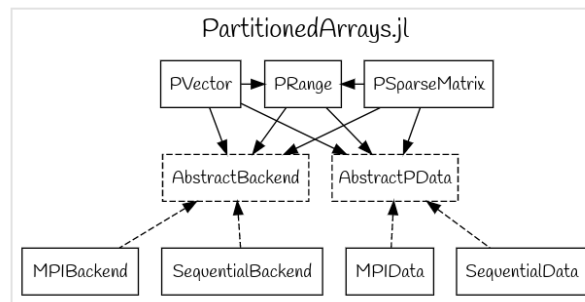


Figure 2: Main software abstractions in PartitionedArrays and their relations. Dashed boxes represent abstract data types, while regular ones, concrete data types. Dashed arrows represent “implements” relationships (i.e., the type at the tail of the arrow implements the abstract interface associated to the abstract type at the head of the arrow), while regular arrows “uses” relationships (i.e., the type at the tail of the arrow uses the type and associated methods at the head of the arrow).

At present, PartitionedArrays (v0.2.8) provides two backends for running generic parallel algorithms, namely SequentialBackend and MPIBackend, and their associated SequentialData and MPIData, resp., AbstractPData implementations (see Figure 2). With SequentialBackend, the parallel data is split into chunks, and stored in a conventional (sequential) Julia interactive session (typically in an Array). The parallel tasks in the algorithm are executed one after the other. Note that SequentialBackend does not mean to distribute the data into a single part. The data indeed can be split into an arbitrary number of parts. With MPIBackend, chunks of parallel data and parallel tasks are mapped to different MPI processes. End-users applications are to be executed in MPI mode, e.g., `mpirun -n 4 julia --project=. simulation.jl`. Using SequentialBackend, GridapDistributed users can run their parallel applications during development stage using a standard Julia session, so that they can leverage very useful tools in the Julia development workflow like Revise and Debugger. This is even more crucial given the current lack of generalized support of Julia in parallel debugging tools. Once the code is robust with SequentialBackend, it can be automatically deployed in a supercomputer switching to MPIBackend. The library is designed such that other backends like a ThreadedBackend (for parallel multi-threaded execution), DistributedBackend (for the Julia Distributed programming model), or MPIXBackend (for hybrid MPI+X models) may be added in the future. On top of these abstractions, PartitionedArrays provides the PRange, PVector and PSparseMatrix data types. PRange implements the AbstractUnitRange Julia interface, and it represents an overlapping partition of a range of global identifiers, e.g., cells in a mesh, DoFs in a FE space, rows/columns in a matrix, or entries in a vector. Using the methods of PRange one, e.g., may retrieve data on ghost cells or DoFs using a nearest neighbour communication. This type is used to describe the parallel data layout of rows and cols in PVector and PSparseMatrix objects, which represent, respectively, a global vector partitioned into overlapping chunks, and a global sparse matrix into overlapping chunks of rows.

As mentioned earlier, GridapDistributed offers a built-in Cartesian-like mesh generator, and does not provide, by now, built-in highly scalable solvers. To address this, as required by real-world applications, one can combine GridapDistributed with GridapP4est (Martin, 2021) and GridapPETSc (Verdugo et al., 2021) (see Figure 1). The former provides a mesh data structure that leverages the p4est library as highly scalable mesh generation engine (Burstedde et al., 2011). This engine can mesh domains that can be expressed as a forest of adaptive octrees. The latter enables the usage of the highly scalable solvers (e.g., algebraic multigrid) in the PETSc library (Balay et al., 2021) to be combined with GridapDistributed.

Parallel scaling benchmark

Figure 3 reports the strong (left) and weak scaling (right) of GridapDistributed when applied to an standard elliptic benchmark PDE problem, namely the 3D Poisson problem. In strong form this problem reads: find u such that $-\nabla \cdot (\kappa \nabla u) = f$ in $\Omega = [0, 1]^3$, with $u = u_D$ on Γ_D (Dirichlet boundary) and $\partial_n u = g_N$ on Γ_N (Neumann Boundary); \mathbf{n} is the outward unit normal to Γ_N . The domain was discretized using the built-in Cartesian-like mesh generator in GridapDistributed. The code was run on the NCI@Gadi Australian supercomputer (3024 nodes, 2x 24-core Intel Xeon Scalable *Cascade Lake* cores and 192 GB of RAM per node) with Julia 1.7 and OpenMPI 4.1.2. For the strong scaling test, we used a fixed **global** problem size resulting from the trilinear FE discretization of the domain using a 300x300x300 hexaedra mesh (26.7 MDoFs) and we scaled the number of cores up to 21.9K cores. For the weak scaling test, we used a fixed **local** problem size of 32x32x32 hexaedra, and we scaled the number of cores up to 16.5K cores. A global problem size of 0.54 billion DoFs was solved for this number of cores. The reported wall clock time includes: (1) Mesh generation; (2) Generation of global FE space; (3) Assembly of distributed linear system; (4) Interpolation of a manufactured solution; (5) Computation of the residual (includes a matrix-vector product) and its norm. Note that the linear solver time (GAMG built-in solver in PETSc) was not included in the total computation time as it is actually external to GridapDistributed.

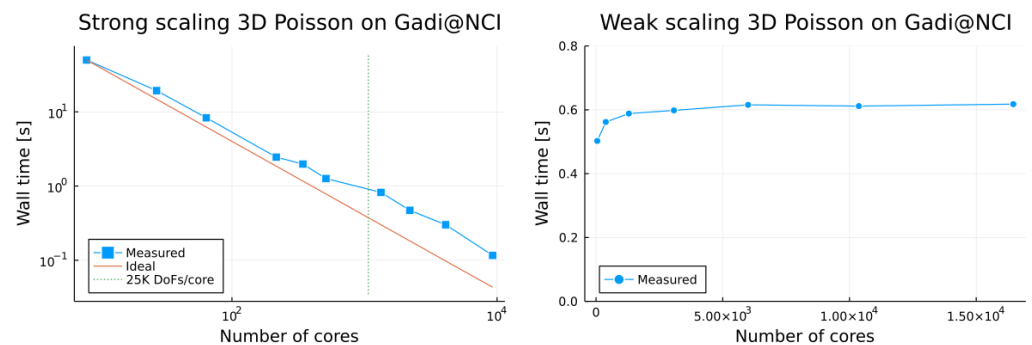


Figure 3: Strong (left) and weak (right) scaling of GridapDistributed when applied to 3D Poisson problem on the Australian Gadi@NCI supercomputer.

Figure 3 shows, on the one hand, an efficient reduction of computation times with increasing number of cores, even far beyond a relatively small load of 25K DoFs per CPU core. On the other hand, an asymptotically constant time-to-solution (i.e., perfect weak scaling) when the number of cores is increased in the same proportion of global problem size with a local problem size of 32x32x32 trilinear FEs.

Demo application

To highlight the ability of GridapDistributed and associated packages (see Figure 1) to tackle real-world problems, and the potential behind its composable architecture, we consider a demo application with interest in the geophysical fluid dynamics community. This application solves the so-called non-linear rotating shallow water equations on the sphere, i.e., a surface PDE posed on a two-dimensional manifold immersed in three-dimensional space. This complex system of PDEs describes the dynamics of a single incompressible thin layer of constant density fluid with a free surface under rotational effects. It is often used as a test bed for horizontal discretisations with application to numerical weather prediction and ocean modelling. We in particular considered the synthetic benchmark proposed in (Galewsky et al., 2016), which is characterized by its ability to generate a complex and realistic flow.

For the geometrical discretization of the sphere, the software uses the so-called cubed sphere mesh (Ronchi et al., 1996), which was implemented using GridapP4est. The spatial discretization of the equations relies on GridapDistributed to build a **compatible** set of FE spaces (Gibson et al., 2019) for the system unknowns (fluid velocity, fluid depth, potential vorticity and mass flux) grounded on Raviart-Thomas and Lagrangian FEs defined on the manifold (Rognes et al., 2013). Compatible FEs are a kind of advanced discretization techniques that enable the balanced forces of the continuous equations to be accurately captured in the computational model. The spatial discretization is stabilized using the so-called Anticipated Potential Vorticity Method (APVM) (Rognes et al., 2013). Time integration is based on a implicit trapezoidal rule, and thus a fully-coupled nonlinear problem has to be solved at each time step. In order to solve this nonlinear problem, we leveraged a Newton-GMRES solver preconditioned with an algebraic preconditioner provided by GridapPETSc (on top of PETSc 3.16). The *exact* jacobian of the shallow water system was computed/assembled at each nonlinear iteration.

Figure 4 shows the magnitude of the vorticity field after 6.5 simulation days (left) and the results of a strong scaling study of the model on the Australian Gadi@NCI supercomputer (right). The reported times correspond to the *total* wall time of the first 10 time integration steps; these were the only ones (out of 3600 time steps, i.e., 20 simulation days with a time step size of 480 secs.) that we could afford running for all points in the plot due to limited computational budget reasons. We considered two different problem sizes, corresponding to 256x256 and 512x512 quadrilaterals/panel cubed sphere meshes, resp. Overall, Figure 4 confirms a remarkable ability of the ecosystem of Julia packages at hand to efficiently reduce computation times with increasing number of CPU cores for a complex, real-world computational model.

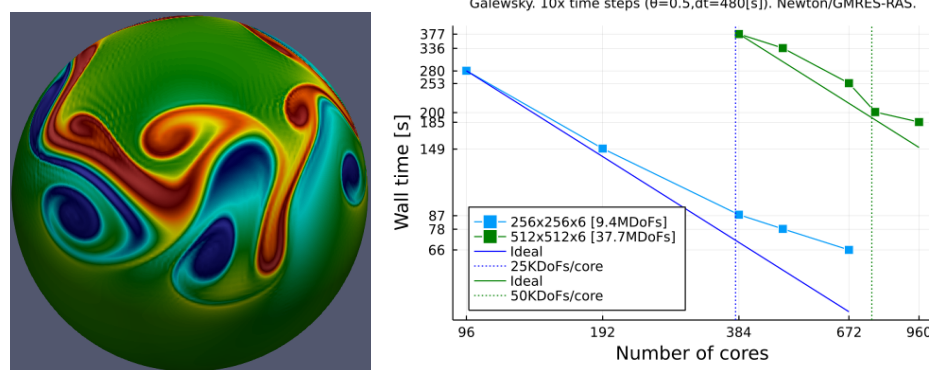


Figure 4: Magnitude of the vorticity field after 6.5 simulation days with a coarser 48x48 quadrilaterals/panel cube sphere mesh (left) and strong scaling (right) of the non-linear rotating shallow water equations solver on the Australian Gadi@NCI supercomputer.

Acknowledgements

This research was partially funded by the Australian Government through the Australian Research Council (project number DP210103092), the European Commission under the FET-HPC ExaQute project (Grant agreement ID: 800898) within the Horizon 2020 Framework Programme and the project RTI2018-096898-B-I00 from the “FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación.” F. Verdugo acknowledges support from the Spanish Ministry of Economy and Competitiveness through the “Severo Ochoa Programme for Centers of Excellence in R&D (CEX2018-000797-S).” This work was also supported by computational resources provided by the Australian Government through NCI under the National Computational Merit Allocation Scheme (NCMAS).

References

- Badia, S., Martín, A. F., Neiva, E., & Verdugo, F. (2020). A Generic Finite Element Framework on Parallel Tree-Based Adaptive Meshes. *https://doi.org/10.1137/20m1328786*, 42(6), C436–C468. <https://doi.org/10.1137/20M1328786>
- Badia, S., & Verdugo, F. (2020). Gridap: An extensible Finite Element toolbox in Julia. *Journal of Open Source Software*, 5(52), 2520. <https://doi.org/10.21105/JOSS.02520>
- Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., ... Zhang, J. (2021). *PETSc/TAO users manual* (ANL-21/39 - Revision 3.16). Argonne National Laboratory.
- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A Fresh Approach to Numerical Computing. *http://Dx.doi.org/10.1137/141000671*, 59(1), 65–98. <https://doi.org/10.1137/141000671>
- Burstedde, C., Wilcox, L. C., & Ghattas, O. (2011). p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *http://Dx.doi.org/10.1137/100791634*, 33(3), 1103–1133. <https://doi.org/10.1137/100791634>
- Galewsky, J., Scott, R. K., & Polvani, L. M. (2016). An initial-value problem for testing numerical models of the global shallow-water equations. *http://Dx.doi.org/10.3402/Tellusa.v56i5.14436*, 56(5), 429–440. <https://doi.org/10.3402/TELLUSA.V56I5.14436>
- Gibson, T. H., McRae, A. T. T., Cotter, C. J., Mitchell, L., & Ham, D. A. (2019). *Compatible Finite Element Methods for Geophysical Flows*. <https://doi.org/10.1007/978-3-030-23957-2>
- Martin, A. F. (2021). GridapP4est. In *GitHub repository*. GitHub. <https://github.com/gridap/GridapP4est.jl>
- Message Passing Interface Forum. (2021). *MPI: A message-passing interface standard version 4.0*. <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- Rognes, M. E., Ham, D. A., Cotter, C. J., & McRae, A. T. T. (2013). Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2. *Geoscientific Model Development*, 6(6), 2099–2119. <https://doi.org/10.5194/GMD-6-2099-2013>
- Ronchi, C., Iacono, R., & Paolucci, P. S. (1996). The “Cubed Sphere”: A New Method for the Solution of Partial Differential Equations in Spherical Geometry. *Journal of Computational Physics*, 124(1), 93–114. <https://doi.org/10.1006/JCPH.1996.0047>
- Verdugo, F. (2021). PartitionedArrays. In *GitHub repository*. GitHub. <https://github.com/fverdugo/PartitionedArrays.jl>
- Verdugo, F., & Badia, S. (2021). The software design of Gridap: a Finite Element package based on the Julia JIT compiler. *ArXiv e-Prints*. <https://arxiv.org/abs/2109.12818>
- Verdugo, F., Sande, V., & Martin, A. F. (2021). GridapPETSc. In *GitHub repository*. GitHub. <https://github.com/gridap/GridapPETSc.jl>