




GridapDistributed: a massively parallel finite element toolbox in Julia

Santiago Badia ^{1,2}, Alberto F. Martín ^{1¶}, and Francesc Verdugo ²

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. ¶ Corresponding author

DOI: [10.21105/joss.04157](https://doi.org/10.21105/joss.04157)

Software

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

Editor: [Kevin M. Moerman](#) 

Reviewers:

- [@PetrKryslUCSD](#)
- [@Leticia-maria](#)
- [@jedbrown](#)

Submitted: 18 January 2022

Published: 09 June 2022

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](https://doi.org/10.3847/xxxxx), published in the Journal of Open Source Software.

Summary and statement of need

The ever-increasing demand for resolution and accuracy in mathematical models of physical processes governed by systems of Partial Differential Equations (PDEs) can only be addressed using fully-parallel advanced numerical discretization methods and scalable solution methods, thus able to exploit the vast amount of computational resources in state-of-the-art supercomputers. To this end, GridapDistributed is a registered Julia ([Bezanson et al., 2017](#)) software package which provides fully-parallel distributed memory data structures and associated methods for the Finite Element (FE) numerical solution of PDEs on parallel computers. Thus, it can be run on multi-core CPU desktop computers at small scales, as well as on HPC clusters and supercomputers at medium/large scales. The data structures in GridapDistributed 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 Francesc Verdugo & Badia ([2022](#)) 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 massively parallel 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 neighbors, 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 and associated methods 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 matrix-free 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.

There are a number of high quality open source parallel finite element packages available in

the literature. Some examples are deal.II (Arndt et al., 2021), libMesh (Kirk et al., 2006), MFEM (Anderson et al., 2021), FEMPAR (Badia et al., 2017), FEniCS (Logg et al., 2012), or FreeFEM++ (Hecht, 2012), to name a few. All these packages have their own set of features, potentials, and limitations. Among these, FEniCS and FreeFEM++ are perhaps the closest ones in scope and spirit to the packages in the Gridap ecosystem. A hallmark of Gridap ecosystem packages compared to FreeFEM++ and FEniCS is that a very expressive and compact (yet efficient) syntax is transformed into low-level code using the Julia JIT compiler and thus they do not need a sophisticated compiler of variational forms nor a more intricate workflow (e.g., a Python front-end and a C/C++ back-end).

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 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 Francesc Verdugo & Badia (2022) 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 (F. 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 an stand-alone package, segregated from GridapDistributed, it was designed with parallel FE packages such as GridapDistributed in mind. In any case, GridapDistributed is designed so that a different distributed linear algebra library from PartitionedArrays might be used as well, as far as it is able to provide the same functionality.

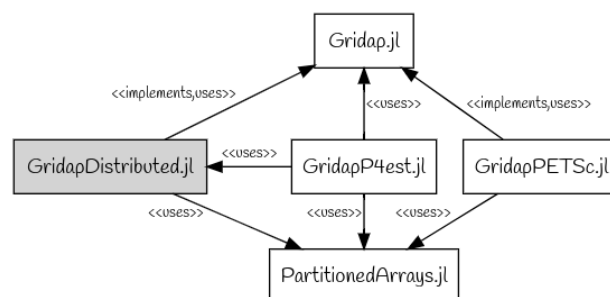


Figure 1: GridapDistributed and its relation to other packages in the Julia package ecosystem. In this diagram, each rectangle represents a Julia package, while the (directed) arrows represent relations (dependencies) among packages. Both the direction of the arrow and the label attached to the arrows 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., GridapP4est depends on GridapDistributed but not conversely.

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 (F. 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.

Usage example

In order to confirm our previous claims on expressiveness, conciseness and productivity (e.g., a very small number of lines of code), the example Julia script below illustrates how one may use GridapDistributed in order to solve, in parallel, a 2D Poisson problem defined on the unit square. (In order to fully understand the code snippet, familiarity with the high level API of Gridap is assumed.) The domain is discretized using the parallel Cartesian-like mesh generator built-in in GridapDistributed. The only minimal burden posed on the programmer versus Gridap is a call to the `prun` function of `PartitionedArrays` right at the beginning of the program. With this function, the programmer sets up the `PartitionedArrays` communication backend (i.e., MPI communication backend in the example), specifies the number of parts and their layout (i.e., (2,2) 2D layout in the example), and provides a function (using Julia do-block syntax for function arguments in the example) to be run on each part. This function is equivalent to a sequential Gridap script, except for the `CartesianDiscreteModel` call, which, in GridapDistributed, also requires the `parts` argument passed back by the `prun` function. In a typical cluster environment, this example would be executed on 4 MPI tasks from a terminal as `mpirun -n 4 julia --project=. example.jl`.

```
using Gridap
using GridapDistributed
using PartitionedArrays
partition = (2,2)
prun(mpi,partition) do parts
    domain = (0,1,0,1)
    mesh_partition = (4,4)
    model = CartesianDiscreteModel(parts,domain,mesh_partition)
    order = 2
    u((x,y)) = (x+y)^order
    f(x) = -Δ(u,x)
    reffe = ReferenceFE(lagrangian,Float64,order)
    V = TestFESpace(model,reffe,dirichlet_tags="boundary")
    U = TrialFESpace(u,V)
    Ω = Triangulation(model)
    dΩ = Measure(Ω,2*order)
    a(u,v) = ∫( ∇(v)⋅∇(u) )dΩ
    l(v) = ∫( v*f )dΩ
    op = AffineFEOperator(a,l,U,V)
    uh = solve(op)
    writevtk(Ω,"results",cellfields=["uh"=>uh,"grad_uh"=>∇(uh)])
end
```

Parallel scaling benchmark

Figure 2 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.

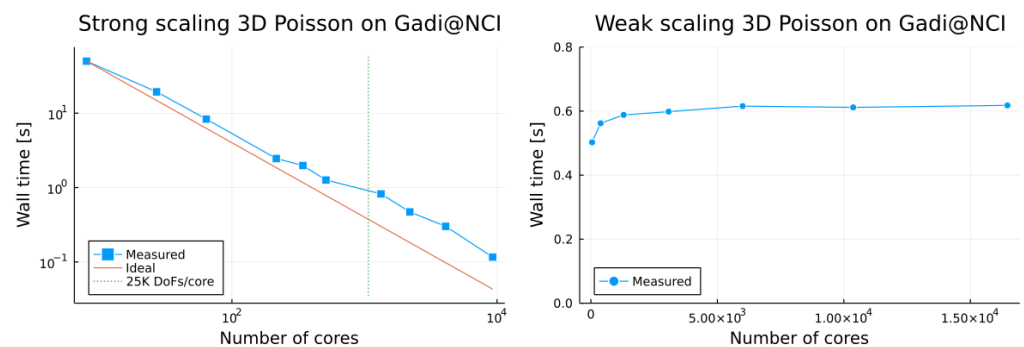


Figure 2: Strong (left) and weak (right) scaling of GridapDistributed when applied to 3D Poisson problem on the Australian Gadi@NCI supercomputer.

Figure 2 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 advanced discretization techniques that preserve at the discrete level physical properties of the continuous equations. In order to stabilize the spatial discretization we use the most standard stabilization method in the geophysical flows literature, namely the so-called Anticipated Potential Vorticity Method (APVM) (Rognes et al.,

2013). We stress that other stabilisation techniques, e.g., Streamline Upwind Petrov–Galerkin (SUPG)-like methods, have also been implemented with these tools (Lee et al., 2022). Time integration is based on a fully-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 3 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 spurious ringing artifacts in the magnitude of the vorticity field are well-known in the APVM method at coarse resolutions and can be corrected using a more effective stabilization method, such as, e.g., SUPG-like stabilization (Lee et al., 2022). 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. We stress that the time discretization is fully implicit. Thus we can afford larger time step sizes than with explicit methods. Besides, the purpose of the experiment is to evaluate the scalability of the framework, and not necessarily to obtain physically meaningful simulation results. Overall, Figure 3 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.

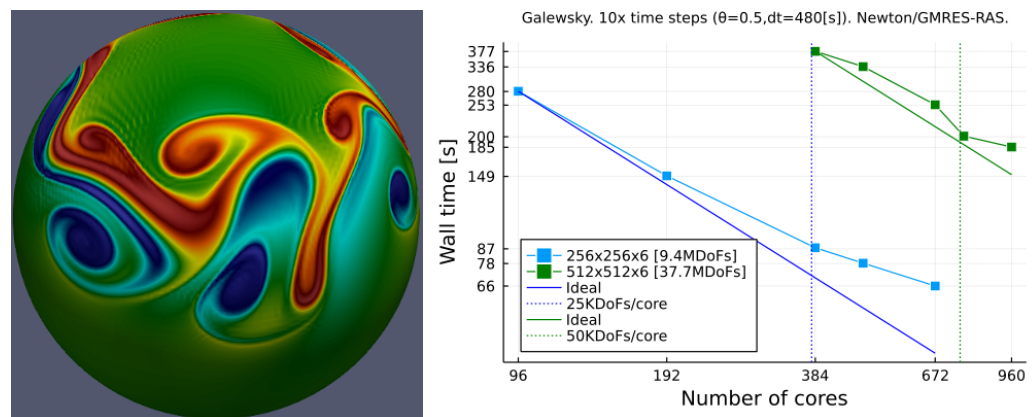


Figure 3: Magnitude of the vorticity field after 6.5 simulation days with a coarser 48x48 quadrilaterals/panel cubed sphere mesh (left) and strong scaling (right) of the non-linear rotating shallow water equations solver on the Australian Gadi@NCI supercomputer.

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 Program and the project RTI2018-096898-B-I00 from the “FEDER/Ministerio de Ciencia e Innovación (MCIN) – Agencia Estatal de Investigación (AEI)”. F. Verdugo acknowledges support from the “Severo Ochoa Program for Centers of Excellence in R&D (2019-2023)” under the grant CEX2018-000797-S funded by MCIN/AEI/10.13039/501100011033. This work was also supported by computational resources provided by the Australian Government through NCI under the National Computational Merit Allocation Scheme (NCMAS).

References

- Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Dobrev, J. C. V., Dudouit, Y., Fisher, A., Kolev, Tz., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., & Zampini, S. (2021). MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, 81, 42–74. <https://doi.org/10.1016/j.camwa.2020.06.009>
- Arndt, D., Bangerth, W., Blais, B., Fehling, M., Gassmüller, R., Heister, T., Heltai, L., Köcher, U., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.-P., Proell, S., Simon, K., Turcksin, B., Wells, D., & Zhang, J. (2021). The deal.II library, version 9.3. *Journal of Numerical Mathematics*, 29(3), 171–186. <https://doi.org/10.1515/jnma-2021-0081>
- Badia, S., Martín, A. F., Neiva, E., & Verdugo, F. (2020). A generic finite element framework on parallel tree-based adaptive meshes. *SIAM Journal on Scientific Computing*, 42(6), C436–C468. <https://doi.org/10.1137/20M1328786>
- Badia, S., Martín, A. F., & Principe, J. (2017). FEMPAR: An object-oriented parallel finite element framework. *Archives of Computational Methods in Engineering*, 25(2), 195–271. <https://doi.org/10.1007/s11831-017-9244-1>
- 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. *SIAM Review*, 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. *SIAM Journal on Scientific Computing*, 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. *Tellus A: Dynamic Meteorology and Oceanography*, 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*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-23957-2>
- Hecht, F. (2012). New development in FreeFem++. *J. Numer. Math.*, 20(3–4), 251–265. <https://doi.org/10.1515/jnum-2012-0013>
- Kirk, B. S., Peterson, J. W., Stogner, R. H., & Carey, G. F. (2006). libMesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4), 237–254. <https://doi.org/10.1007/s00366-006-0049-3>
- Lee, D., Martín, A. F., Bladwell, C., & Badia, S. (2022). *A comparison of variational upwinding schemes for geophysical fluids, and their application to potential enstrophy conserving discretisations in space and time*. arXiv. <https://doi.org/10.48550/ARXIV.2203.04629>
- Logg, A., Mardal, K.-A., & Wells, G. (Eds.). (2012). *Automated solution of differential equations by the finite element method*. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-642-23099-8>
- 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, Francisc, & Badia, S. (2022). The software design of Gridap: A finite element package based on the Julia JIT compiler. *Computer Physics Communications*, 276, 108341. <https://doi.org/10.1016/j.cpc.2022.108341>
- Verdugo, F., Sande, V., & Martin, A. F. (2021). GridapPETSc. In *GitHub repository*. GitHub. <https://github.com/gridap/GridapPETSc.jl>