18.337 Midterm/Final Project: Parallelized solution of 3D finite element problems in Julia

Corbin Foucart

1 Problem statement

The focus of this project is to leverage the power of Julia and parallel computing to more efficiently solve large-scale 3D finite element ocean equations, augmenting a a pre-existing research code. I omit the details of the Hybridizable Discontinuous Galerkin (HDG) scheme used to solve these equations; the interested reader may consult [3]. The important background to the problem can be summarized as:

1. The existing code is written entirely in python.
2. Profiling at the start of the project revealed that the memory and speed bottleneck of the code is due to solving a large linear system during time marching, as is typical in finite element schemes.
3. The pre-existing code solves these linear systems using an LU factorization routine called from `scipy`.
4. We would like to do better, in terms of both memory and computation time.

2 LU factorization and memory limitations

For 3D finite element problems, node memory becomes a limiting factor in computation when LU factorization is used as a solution method for the global linear solve. This is because in memory, a sparse linear system will generally contain $\mathcal{O}(n)$ entries, whereas an LU factorization may be dense, requiring storage of $\mathcal{O}\left(n^2\right)$ entries, without special treatment. For the purposes of benchmarking, we solve a representative 3D finite element PDE (Advection-diffusion-reaction equation, see eq. (1))[1]. Figure 1 depicts the benchmarking results of the HDG solver, run on a single compute node. In this case, the grid resolution (number of degrees of freedom) of the 3D problem was increased until the compute node was unable to meet the memory requirements of the problem, which ended the test. We can glean from the figure that the memory requirements can make such problems intractable far before computation time does so. Note that the last successful test before memory failure takes less than 3 hours to complete, well within the realm of acceptable comuputation time. Further, we see that as the linear system grows, we have that the LU factorization begins to dominate the computation time, compared with the nominal 10 timesteps, each of which applied the factorized matrix in order to solve the global system in the benchmarking problem.

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\mu \nabla \phi) - \nabla \cdot (\boldsymbol{v}\phi) + R \tag{1}$$

Furthermore, for problems in which the geometry (finite element mesh) is unsteady in time, a linear solve is necessary at every timestep, and therefore an LU factorization is not needed. However, direct linear solves will still require storage of $\mathcal{O}\left(n^2\right)$ entries. Therefore, the solution of large-scale ocean problems necessitates the

[1] The primary motivation for solving an ADR equation rather than the incompressible Navier-Stokes equations is that solving the INS uses a projection method to decouple the pressure and velocity solves, leading to a series of linear systems, which must be solved in a particular order. The benchmarking of such a system is unnecessarily complicated, so we use an ADR equation. It is important to note that all the methods developed herein can be immediately used instead to solve the INS for ocean problems.

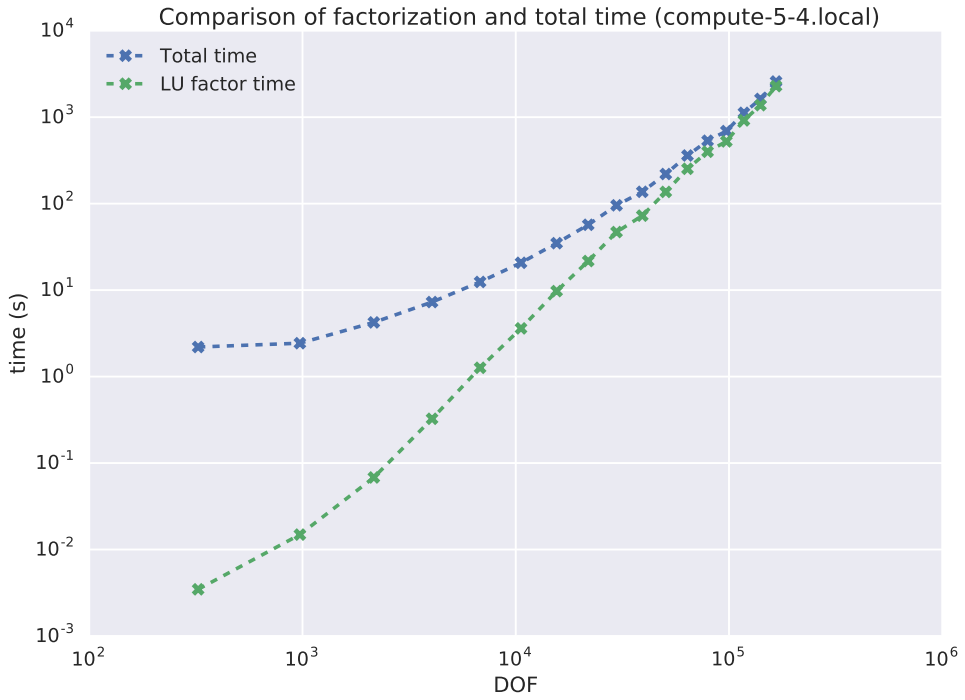


Figure 1: LU factorization as a bottleneck, 16 GB node

use of iterative solvers for the global linear system. For 3D problems so large that the memory requirements exceed that of even a high-performance compute node, the linear system can not be stored in its entirety, and the iterative solution must be computed in a matrix-free manner, and ideally, parallelized manner.

3 Linking a Python code to Julia

4 Efficiency and performance of linear solvers

4.1 boundary treatment

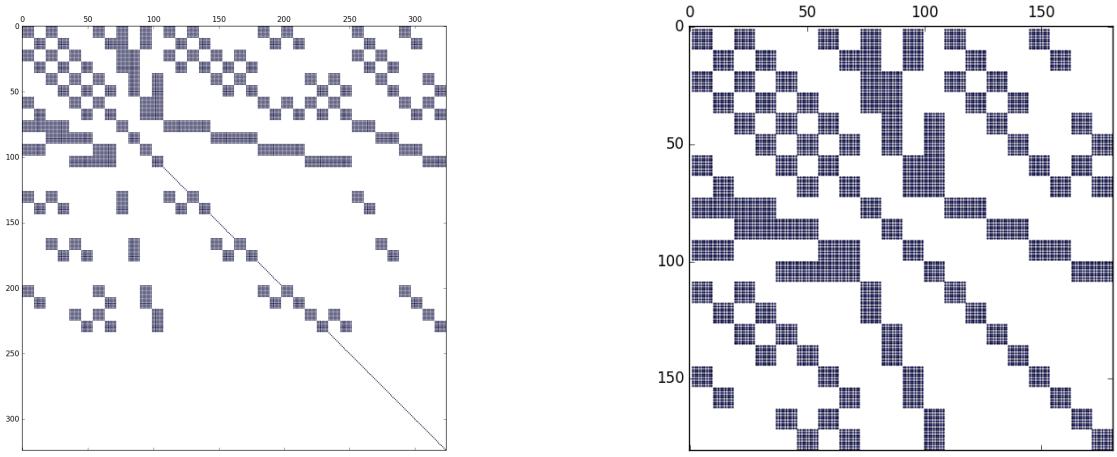
The entries corresponding to the dirichlet boundary conditions in the global flux matrix 'break' the symmetric positive definite quality of the global flux matrix. It is computationally attractive to formulate a mode of solution which retains the SPD quality, so that conjugate gradient methods can be applied. It turns out that it is very important to take advantage of this SPD information, since our results indicate that more general iterative solvers perform much worse. In some sense, we have the following block structure:

$$A = \left[\begin{array}{c|c} A^* & D \\ \hline 0 & I \end{array}\right], \qquad \left[\begin{array}{c|c} A^* & D \\ \hline 0 & I \end{array}\right] \left[\begin{array}{c} \lambda^* \\ \hline \lambda_D \end{array}\right] = \left[\begin{array}{c} b^* \\ \hline b_D \end{array}\right] \tag{2}$$

In principle, we can simply solve:

$$\left.\begin{array}{l} A^*\lambda^* + D\lambda_D = b^* \\ I\lambda_D = b_D \end{array}\right\} \Rightarrow A^*\lambda^* = b^* - Db_d \tag{3}$$

which is a smaller linear system, but also has the added benefit that A^* is SPD. This follows from the bilinearity and coercivity of the diffusion operator in equation (1. In principle, this rearrangement is much more complicated, since A does not start in the block structure as written above, see 2(a). I will refer to this



(a) after `mk_A`

(b) Dirichlet data removed

Figure 2: Global flux matrix

problem as 'compressing' the global matrix. Ultimately, we wish instaead to solve the 'compressed' linear system, see ,figure 2(b). Some approaches to handle this:

4.1.1 brute force compression and solution of resulting linear system

The following is an algorithm which keeps the sparse representation of A, $\boldsymbol{b}$ is the RHS, and r is a list containing the rows / columns to remove formed during the existing `mk_A` routine. N is the size of A.

Algorithm 1 Communicating Dirichlet data to RHS

1: $\boldsymbol{b}^* \leftarrow \boldsymbol{b}$
2: **for** $i \in r$
3: $\quad \boldsymbol{rC} \leftarrow A$`[:, i]` ▷ index into column of A
4: $\quad \boldsymbol{rC}[i] \leftarrow 0$
5: $\quad \boldsymbol{b}^* \leftarrow \boldsymbol{b}^* - b_i * \boldsymbol{rC}$
6: **end for**

At the end of this algorithm, we have the original A matrix as in fig 2(a), but the right hand side now contains the correct data for the compressed linear system in the non-Dirichlet indicies, and the unmodified solution in the Dirichlet entries. As an aside, efficient implementations should operate on this vector directly to avoid having to copy the boundary data after the linear solve. The next step is to remove all of the data corresponding to Dirichlet rows and columns from A. We can take advantage of Julia's flexibility to write a compression method for the `SparseMatrixCSC` type.

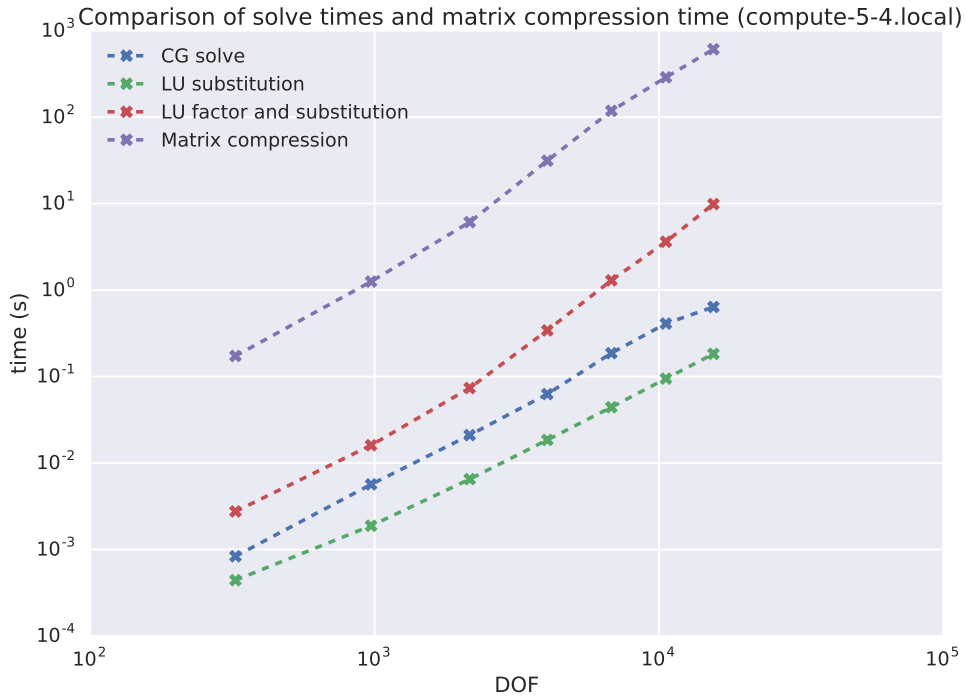


Figure 3: Inefficiency of compressing a sparse matrix

Algorithm 2 Brute force removal of Dirichlet rows and columns from A (sparse, CSC)

for $i \in r$
  select range of column pointers corresponding to column i
  delete column pointers
  decrement remaining column pointers
  for $k \in [1, N-$ columns removed$]$
    find rows greater than i ▷ rows stored sorted, so this is a binary search
    remove found rows
    downshift matrix values by rows removed
  end for
end for

Note that this algorithm is not readily parallelizable, since all workers would be operating on the same data structure, resulting in a race condition. Coordination between workers would also be inefficient due to having to downshift the data, affecting any operation 'upstream'. Overall, compressing the sparse matrix is iniefficient. Profiling demonstrates that the large number of calls to `deleteat!` are expensive (even more so than the binary search) when "compressing" the CSC matrix, since they shift every subsequent element of the array down. This leads to enormous memory allocations, and is extremely inefficient in terms of space and computational time; this is clear from the benchmarking test in figure 3.

4.1.2 SPD solve without matrix compression

It is possible to leverage the multiple dispatch design of Julia in order to avoid this inefficiency. Instead of performing any type of matrix compression, we can simply leave the matrix as is, move the boundary information to the right hand side, and perform an iterative solve on the non-boundary rows and columns. In other words, we ignore the entries which break the symmetry. Namely, we can define a new `HollowSparseMatrix`

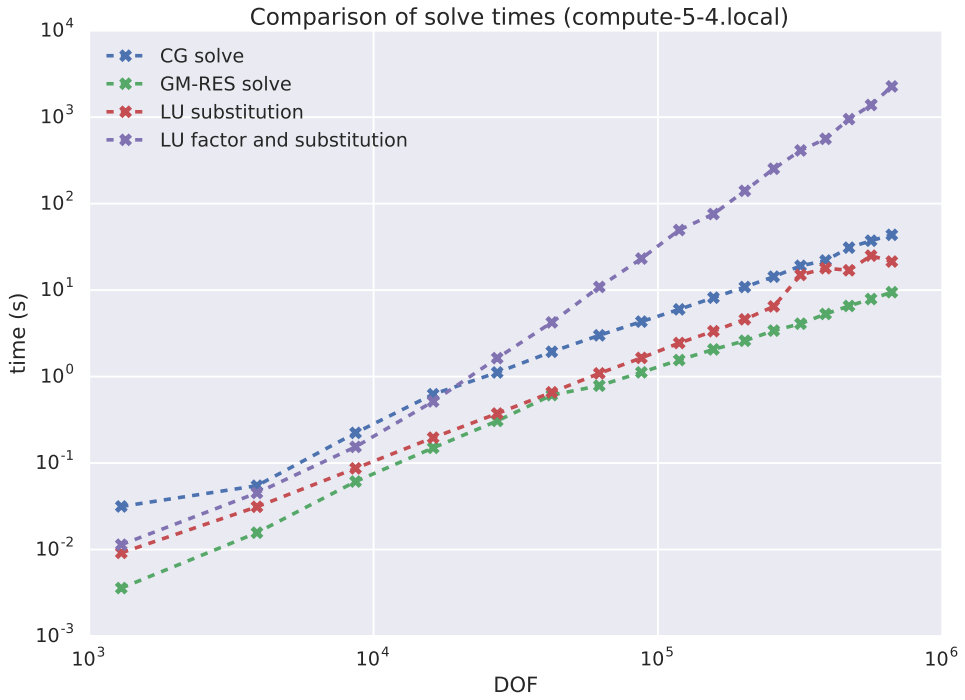


Figure 4: Performance until LU memory limit, 16 GB node

immutable type that takes the original matrix data, along with a list of rows/columns on which it should operate. We overload the $*$ operator for matrix-vector multiplication to reflect this change, and can call the same CG routine that we would call on an SPD matrix. Note the power of multiple dispatch here! Since a CG method needs only to be able to evaluate a residual and to compute the linear transformation $A(x)$, as long as each is defined for another representation of the linear transformation, we can use the same CG code. In figures 4 and 5, we can see that performance of the CG solve is reasonable once the inefficient compression technique is no longer needed.

Figures 4 and 5 represent a series of benchmarking runs, on a small and large compute node repsectively. The runs ended when the node was unable to accomodate the memory requirements of the LU factorization. On both the smaller and larger, the iterative methods perform better than the LU computation in both space and time as the problem size grows. Upon first glance, it also seems that GM-RES outperforms CG. Figure 6 demontrates this to not be the case; the GM-RES does not converge to the specified tolerance. It is somewhat vindicating that the CG method performs better, given the effort to preserve the SPD quality of the global matrix. Further, the iterative methods continue to perform well up to about 10^8 DOF, at which point, the initialization of the python code (not even the linear solve!) becomes the limiting factor in terms of memory. Even so, the switch the iterative methods allows consideration of problems of 2 orders of magnitude higher than previously.

4.2 a brief aside on preconditioners

Choosing a preconditioning matrix for a linear system is more of an art than a science, and goes beyond the scope of this project. However, since implementation of some basic preconditioners was straightforward, figures 7(a) and 7(b) show the distribution of iterations and convergence times sampled over 1000 time steps. The results illustrate the capricious nature of preconditioners: although the SOR preconditioner converges in the fewest number of iterations, it is by far the slowest, due to the additional cost of multiplication with a non-diagonal matrix per iteration.

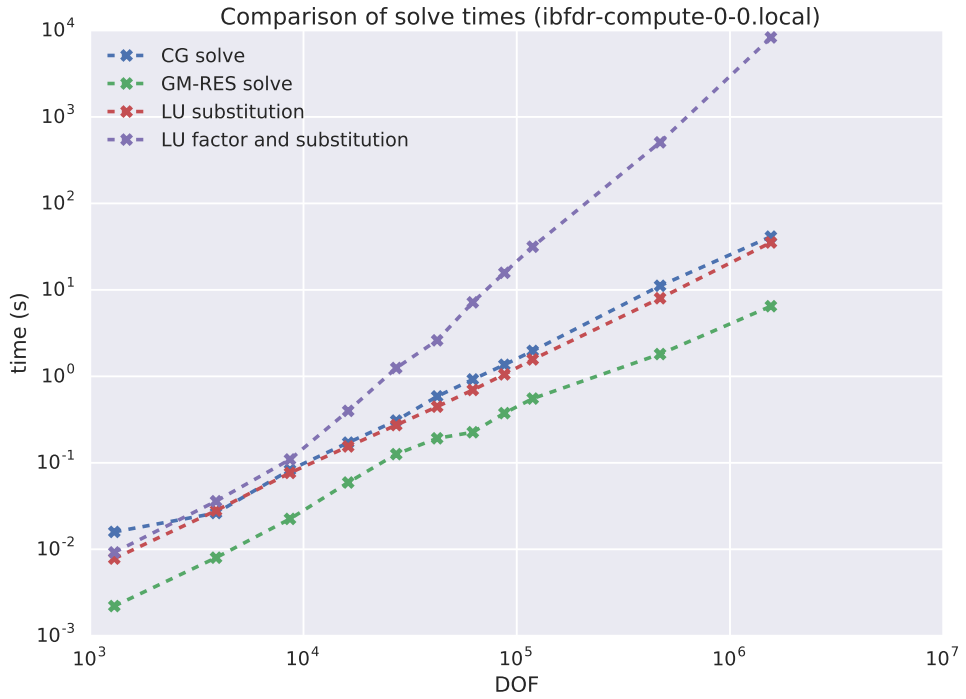


Figure 5: Performance until LU memory limit, 256 GB node

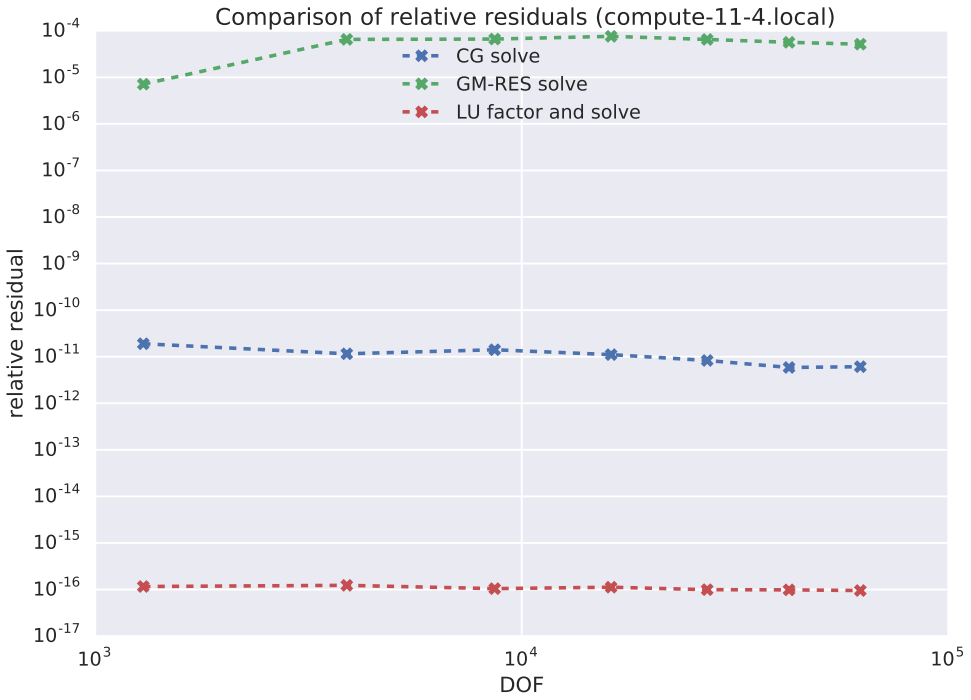
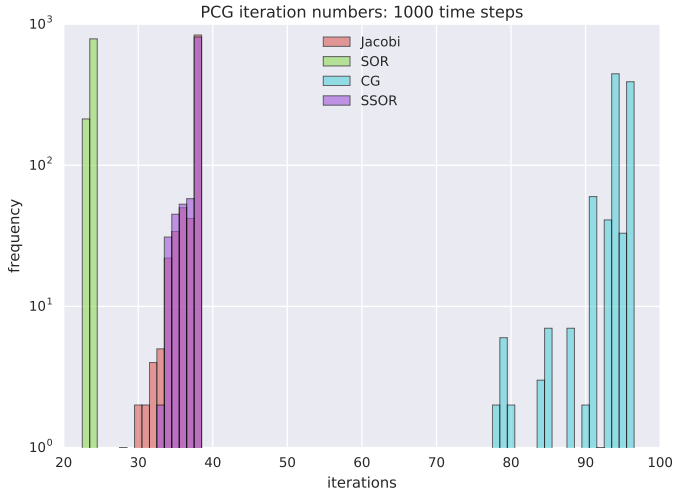
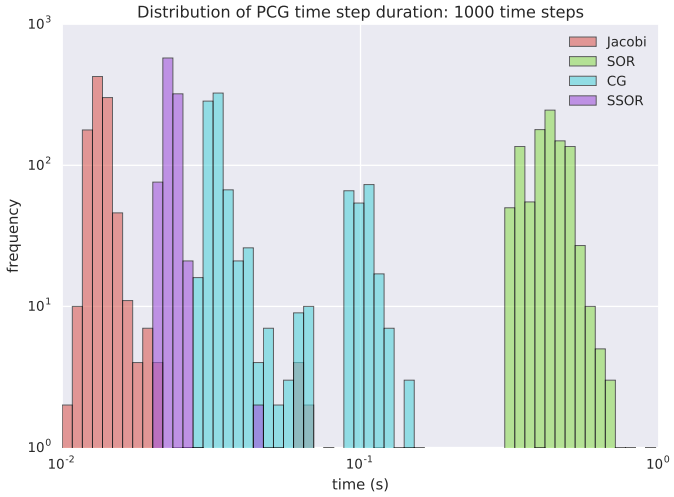


Figure 6: Comparison of relative residuals $\frac{\|b-Ax\|}{\|b\|}$



(a) Number of iterations



(b) Iteration times

4.3 multi-core paralellization of the linear solver

We can make use of Julia's native `SharedArray` type, along with corresponding operator overloading with a parallel implementation of the `HollowSparseMatrix` type in order to distribute the matrix multiplication computations over the available cores of a given compute node. Of course, this initial setup involves some computational overhead, and we see in figure 7 see that utilizing a multi-core approach is only performant for ADR problems involving greater than 10^5 degrees of freedom. We are most interested in performance of the CG solver, given that the GMRES solver exhibited undesirable convergence properties as discussed earlier. It is worth noting that the `scipy` LU factorization and substitution routines which are discussed earlier (and were benchmarked against *serial* iterative methods) already make use of multiple cores via the BLAS libraries called underneath the python implementation.

This result was the main focus of this work and indeed allows our research to consider orders of magnitude larger 3D problems than previously possible.

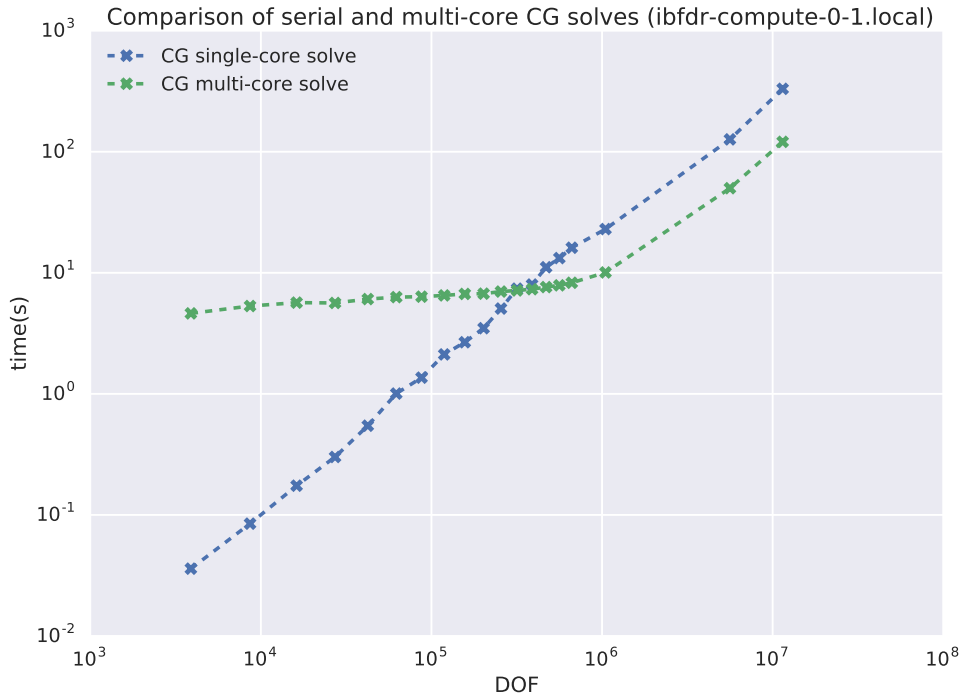


Figure 7: Comparison of single core and multi-core solves, 24 cores

References

[1] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[2] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[3] M. P. Ueckermann and P. F. J. Lermusiaux. Hybridizable discontinuous Galerkin projection methods for Navier–Stokes and Boussinesq equations. *Journal of Computational Physics*, 306:390–421, 2015.