CS598 Final Project Report

Malachi Phillips

12/20/2018

Introduction

The numerical solution of partial differential equations (PDEs) is becoming more ubiquitous in science and engineering applications. The clock speed of modern computers, meanwhile, has not increased. Therefore, there is a need for performant, parallel PDE solution methods. One such class of methods, known as Schwarz methods, offers a method of decomposing a problem into several smaller computations, each of which may be done independent of the other. Firedrake is a performant implementation of the unified form language (UFL) domain-specific language (DSL) for solving finite element (FE) problems of the FEniCS project. However, Firedrake has little support in expressing domain decomposition techniques, especially given that subdomain implementation remains an open issue within Firedrake. Herein, a miniature DSL for expressing domain decomposition methods directly into Firedrake is proposed. Although this proves possible, there is an extensive amount of modification still required in order to make this abstraction performant.

Methodology

Schwarz-like domain decomposition methods were originally introduced in 1870 in order to demonstrate the existence of the solution of Poisson's equation on a domain comprised of a rectangle and a disk overlapping [7], see figure 1.

In particular, this project is concerned with providing an implementation of alternating Schwarz, additive Schwarz, and P.L. Lion's algorithm, which is essentially an additive Schwarz term that includes a Robin boundary condition at the interface between two subdomains. This last method is especially significant, as it can provide iteration convergence without the need for overlapping subdomains, while the former two have convergence rates that are tied to the amount of overlap between subdomains.

In order to implement any sort of Schwarz-like domain decomposition technique inside Firedrake, first some syntax for specifying the proper subdomain and interfaces between subdomains is useful. However, any sort of modification to the unified form lanauge (UFL) represents a significant challenge in implementation, as it affects upstream sources such as Firedrake and COFFEE, and is therefore not a valid path forward [2]. Rather, this work proposes to instead implement a source-to-source translation. From a description of the

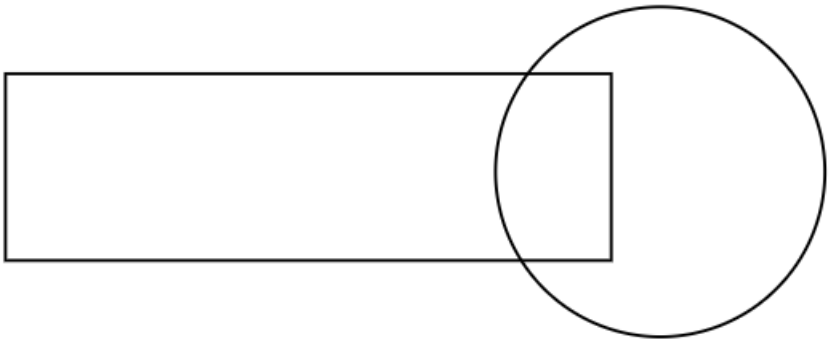

Figure 1: Original domain decomposition proposed by Schwarz.

domain decomposition technique, the relevant subdomains, and the relevant interface identifiers, appropriate Firedrake code is generated in place in order to imeplement the domain decomposition method.

Within Firedrake, subdomain creation and submesh creation still remains an open problem (for more information, see: https://github.com/firedrakeproject/firedrake/issues/723 and https://github.com/firedrakeproject/firedrake/issues/1350). In order to mitigate this issue, Firedrake does, however, support specifying subdomain and interface identifiers that are built into the Gmsh [6]. Therefore, with enough care, the meshes may be generated with the subdomain decomposition already baked into the mesh. This, however, represents its own challenges in mesh generation and graph partitioning, none of which are the focus of this project. Therefore, these aspects are entirely ignored, with exception to the particular problem of interest used in this project, overlapping rectangular subdomains.

The syntax for the mini-DSL proposed herein is demonstrated in Figure 2. In particular, the input program takes as input a Python program with anything a programmer wants. However, between the labels `BEGIN PROGRAM HERE` and `END PROGRAM HERE`, the mini-DSL is allowed to reason about source-to-source translation into Firedrake. In particular, only a few keywords are provided in this DSL. `mesh_name ~ <Python variable for Firedrake mesh>` specifies the name of the corresponding `Mesh` variable in the Firedrake script. `form ~ <UFL signifying weak form of PDE>` specifies the weak form of the PDE, which will be used as a template to generate the individual blocks corresponding to each subdomain to be solved. `rhs ~ <UFL signifying the right-hand side vector>` similarly signifies the template for which the right-hand side of the system is formed. `space_variable ~ <Python variable for Firedrake spatial coordinates>` specifies the name of the corresponding `SpatialCoordinates` variable in the Firedrake script. `functionSpace ~ <Python variable for Firedrake function space>` specifies the name of the corresponding `FunctionSpace` variable in the Firedrake script. `solution ~ <Name>` specifies the output variable name, and need not, although it may, match any of the previously prescribed names. `dirichletBCs={...}` specifies a mapping in the form of a Python dictionary between the subdomain exterior surface identifier and the value to be applied as a Dirichlet condition. `domains={...}` specifies a mapping in the form of a Python dictionary between the subdomain names, signified as a String, with the subdomain identifiers, which must be present in the Gmsh file. Lastly, `interfaces={...}` is a mapping from the interface name to the line or interface identifier in the Gmsh file. Interface names are Strings of the form d{first domain}n{second domain}, where {first domain} and {second domain} are both valid domains in the previously mentioned `domains={...}` mapping and specify two subdomains that do, in fact, share a boundary. The values in this Python dictionary are, however, just the identifiers corresponding to the Gmsh interface between the two subdomains.

Consider the following two subdomain case detailed in 3. Subdomain one, which herein is referred to as `O1`, has interfaces one, two, three, and four. Only one, two, and three have homogeneous Dirichlet boundary conditions employed on them. Similarly, subdomain two, which herein is referred to as `O2`, has interfaces five, six, seven, and eight. Only five, seven, and eight have homogeneous Dirichlet boundary conditions. Notice that interface four refers to the `dO1nO2` interface, while interface six refers to the `dO2nO1` interface. Unfortunately, the programmer must explicitly list out all of these mappings inside the `domains`, `dirichletBCs` and `interfaces` inside the mini-DSL language in order for the correct Firedrake syntax to be generated. Furthermore, these interface and subdomain identifiers must also be specified inside the Gmsh files themselves in order for the system to work. However, at this point, a given form may be compiled into the correct Firedrake code. For example, the weak form needed for solving the Poisson equation would result in:

```
form ~ (dot(grad(u),grad(v)))*dx
```

which, for the two subdomain case shown in figure 3, would result in the compiled code

```
a1 = dot(grad(u),grad(v))*dx(1)
a2 = dot(grad(u),grad(v))*dx(2)
```

with some other complicating details for selecting that the solution of `a1` need be only applied to subdomain one, and the solution of `a2` need be only applied to subdomain two. For more complicated details over how this is possible inside Firedrake, please consult with the GitHub for this project: https://github.com/MalachiTimothyPhillips/dd-firedrake.

```
from firedrake import *
mesh=Mesh(...)
V=FunctionSpace(mesh, "CG", 1)
u=TrialFunction(V)
v=TestFunction(V)
f=Function(V)
x,y=SpatialCoordinate(mesh)
f.interpolate(sin(2*pi*x)*sin(2*pi*y))
par={'ksp_type':'preonly','pc_type':'lu'}
myBCSurfaces=[...]
myBCs=[]
for surface in myBCSurfaces:
    myBCs.append(DirichletBC(V,0,surface))
BEGIN PROGRAM HERE
mesh_name ~ mesh
form ~ (dot(grad(u), grad(v))) * dx
rhs ~ f * v * dx
space_variable ~ coord
functionSpace ~ V
dirichletBCs={...}
# domain number to id mapping
domains={...}
# d_Oi_n_Oj->interface id mapping
interfaces={...}
solution~u
solver_settings ~ solver_parameters=par
END PROGRAM HERE
```

Figure 2: Syntax for mini-DSL expressing transformation from a description of the domain decomposition technique (default: alternating Schwarz) into the appropriate Firedrake syntax.

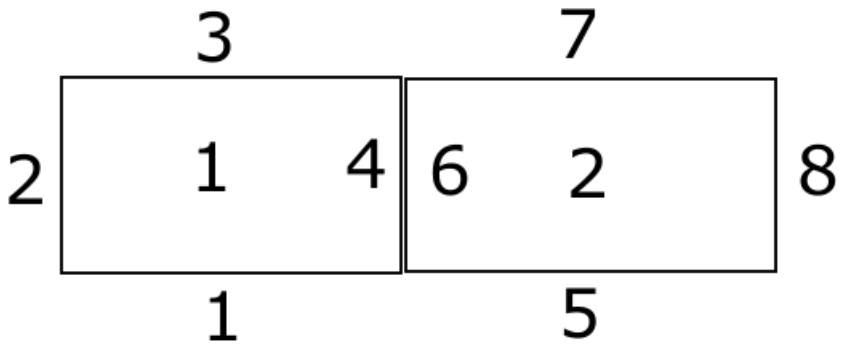

Figure 3: Two subdomain example case, demonstrating what the user must provide to the mini-DSL in order to generate Firedrake code.

Unfortunately, expressing the domain decomposition in terms of a Robin boundary condition necessary for implementing P.L. Lion's algorithm actually requires some amount of modification to the weak form itself (see: https://github.com/firedrakeproject/firedrake/issues/735). In particular, for the Poisson equation, $-\nabla^2 u = f$, subject to the Robin boundary condition $u + \nabla u \cdot n = g$ on Γ, requires editing the weak from of the equation into the following

```
dot(grad(u), grad(v)) * dx - dot(grad(u),n)*v*ds == f*v*dx
```

Substituting the boundary condition for the surface integral into the equation yields the final form

```
dot(grad(u),grad(v))*dx + u*v*ds == f*v*dx-g*v*ds
```

Although handling the substitution of `dx` into the appropriate `dx(subdomain id)` and `ds` into the appropriate `ds(interface id)` is already handled by the mini-DSL itself, there is simply no way of completely automating the process of transforming the weak form of an equation for the standard Dirichlet case into the corresponding weak form for expressing Robin boundary conditions across different subdomain interactions. Although the programmer does not need to explicitly write out the couplings for this weak form in the P.L. Lion's case, she nevertheless must write the weak form in an appropriate way to leverage the Robin boundary conditions. This is simply a limitation of the current mini-DSL.

Results

All the analysis below and results are reported on the basis of a toy sample problem, which consists of generating a two-dimensional square domain on $[0,1]x[0,1]$ with different number of evenly-overlapping rectangular subdomains. In particular, the four subdomain case, which consists of overlapping subdomains is shown in 4. The PDE of interest is Poisson's Equation, $-\nabla^2 u = f$.

As a mini-DSL aimed at performing source-to-source translations from some description of the domain decomposition into the resulting Firedrake, it seems fitting to first measure the ability of this mini-DSL to succinctly express even the most complicated domain decomposition schemes without sinking too much programmer productivity into writing out the domain decomposition explicitly. Figure 5 measures the resulting lines of code required in order to express the appropriate domain decomposition scheme in both the mini-DSL and inside Firedrake itself for the overlapping squares case, see figure 4.

Although lines of code is not necessarily the best measure of programmer productivity (Bill Gates once said: "Measuring programming progress by lines of code is like measuring aircraft building progress by weight"),

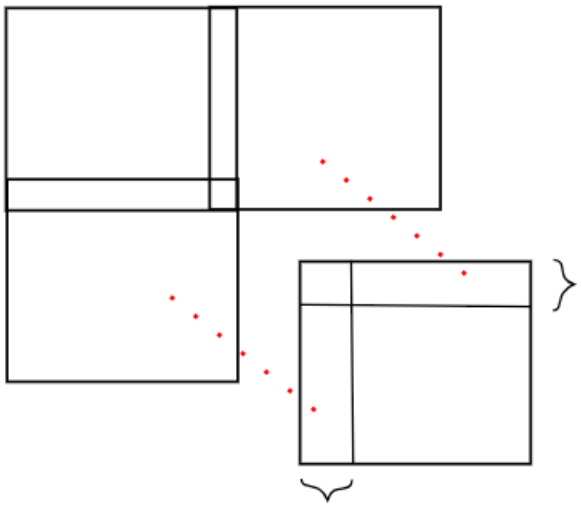

Figure 4: Four subdomain overlapping case. A constant thickness δx is used for the overlap size in all dimensions.

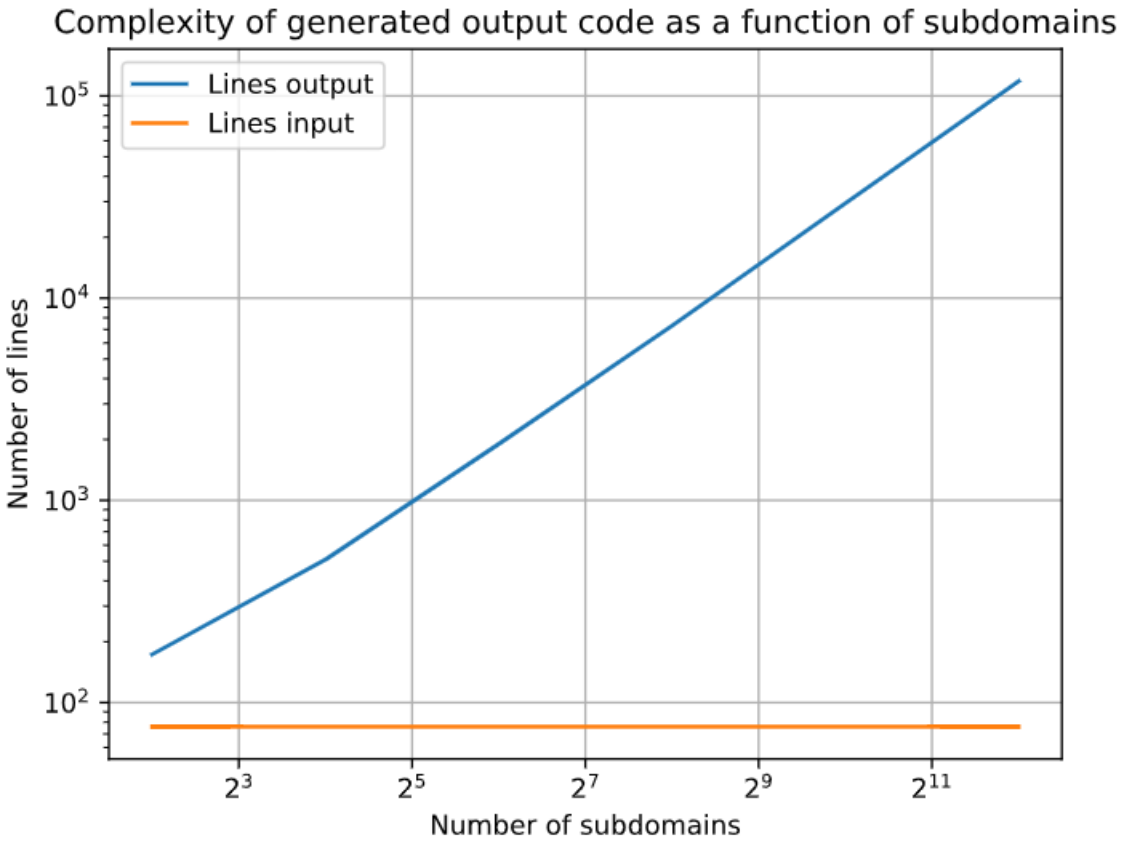

Figure 5: Lines of code measured as a function of the number of subdomains. Orange line represents the number of lines needed to write the domain decomposition in the mini-DSL, while the blue line refers to the output code in Firedrake.

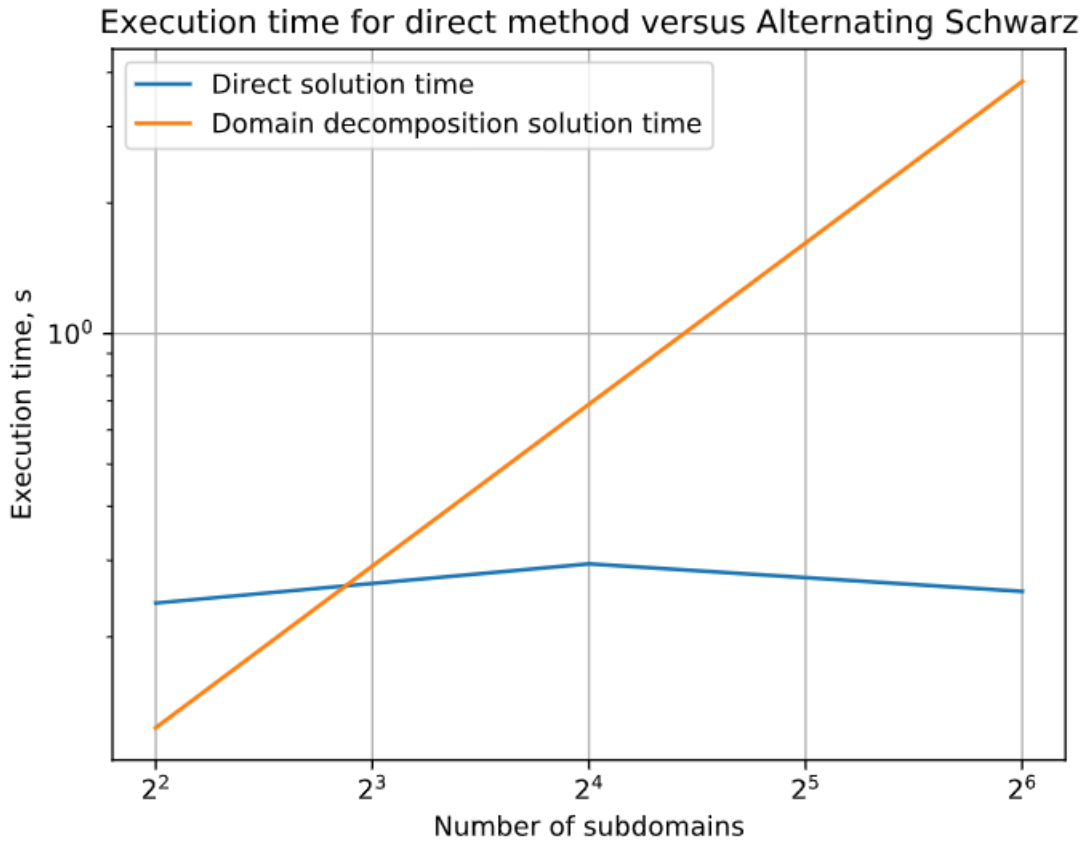

Figure 6: Scaling with respect to the number of subdomains for the direct solution versus the alternating Schwarz algorithm.

nevertheless the evidence presented in 5 demonstrates that this mini-DSL is a sufficiently compact way of expressing relatively complicated domain decomposition schemes. However, performance results are also of great importance to determine whether bothering with domain decomposition in Firedrake is even wortwhile.

Performance results are benchmarked on the University of Illinois porter system, which features 16 Intel(R) Xeon(R) CPU E5-2630 v3 which run at a clock speed of 2.4 GHz. As a first experiment, this project aims to see at how overall solution time (which is taken to include only the time to solution of a linear solve, and completely disregards the amount of time it takes Firedrake to compile the system itself) scales with respect to the number of subdomains in a given problem, see figure 6. In particular, this is conducted using overlapping subdomains similar to that shown in 4. A direct LU decomposition is used to in both the direct solution case and domain decomposition case.

From figure 6, it is apparent that scaling with respect to the number of subdomains does not necessarily provide an advantage over solving the PDE using Firedrake itself. However, in the four overlapping subdomain case, the solution is able to converge in as few as five iterations, making it possible to run ever-so-slightly faster than the direct Firedrake solution. This promises that, given a smart enough domain decomposition technique, it may be possible to outperform Firedrake without needing to improve on the system provided in this project. However, scaling out to many subdomains provides no benefit, a phenomena which has been well knwon in domain decomposition circles since its inception [3].

Since the four subdomain case may offer some promise, scaling with respect to the number of elements for both the direct solve and the domain decomposition technique is performed. The results are summarized in figure 7. In particular, once increasing the problem size, the total execution time needed in order to solve

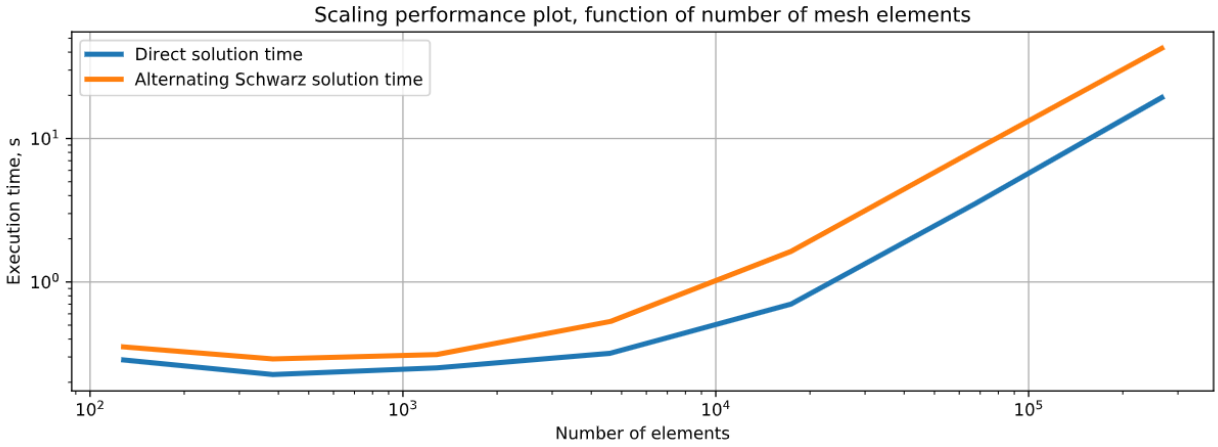

Figure 7: Scaling with respect to the number of elements for the direct solution verus the alternating Schwarz algorithm on a four subdomain, 2x2 overlapping grid of subdomains.

both using the direct and domain decomposition methods increase with respect to the number of elements. However, once scaling past the toy problem first presented in figure 6, the four subdomain decomposition does not represent any saving with respect to execution time, as shown in figure 7. Therefore, this project, in its current state, does not offer a significant speed up advantage over using Firedrake for this toy problem. However, it should be noted that both the direct time and subdomain cases only differ by some constant multiplier, and exhibit the same asymptotics with respect to the number of elements. Therefore, it is not difficult to imagine that it may be possible to tweak the results in this project to arrive at a more performant domain decomposition DSL for Firedrake.

Conclusion

The mini-DSL implemented as part of this project is evaluated on two main points: (1) its ability to offer an abstraction for writing domain decomposition in Firedrake, and (2) its ability to improve the overall time to solution of Firedrake for a given test problem, say Poisson's Equation. As shown in figure 5, this abstraction is able to succinctly express domain decomposition techniques in a few lines of code that would otherwise take many thousands of lines to write in Firedrake itself. Therefore, this mini-DSL exceeds in its first objective. However, with respect to the second objective, this domain decomposition abstraction is only able to outperform Firedrake on particular instance: four overlapping subdomains on a particularly small, toy problem. It is shown that, once scaling with respect to the number of elements for this toy problem, that this mini-DSL remains no longer performant. However, asymptotically, both the direct method and domain decomposition method represent the same time complexity. Therefore, it might be possible for the results of this project to be extended in the future in order to make some performant mini-DSL in Firedrake for expressing domain decomposition techniques.

References

[1] Rathgeber, Florian, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. "Firedrake: Automating the Finite Element Method by Composing Abstractions." ACM Trans. Math. Softw. 43 (3): 24:1-24:27. https://doi.org/10.1145/2998441.

[2]R. C. Kirby and L. Mitchell, "Solver composition across the PDE/linear algebra barrier," SIAM Journal on Scientific Computing, vol. 40, no. 1, pp. C76-C98, Jan. 2018.

[3]D. E. Keyes, How Scalable is Domain Decomposition in Practice? .

[4]R. C. Kirby and L. Mitchell, "Code generation for generally mapped finite elements," arXiv:1808.05513 [cs, math], Aug. 2018.

[5]L. F. Pavarino, "Additive Schwarz methods for thep-version finite element method," Numerische Mathematik, vol. 66, no. 1, pp. 493-515, Dec. 1993.

[6] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering 79(11), pp. 1309-1331, 2009.

[7] Gander, Martin J. "Schwarz methods over the course of time." Electron. Trans. Numer. Anal 31, no. 5 (2008): 228-255.