Geosci. Model Dev. Discuss.,
https://doi.org/10.5194/gmd-2019-86-AC2, 2019

Geoscientific Model Development Discussions

Interactive comment on "Slate: extending Firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond" *by* Thomas H. Gibson et al.

Thomas H. Gibson et al.

t.gibson15@imperial.ac.uk

Received and published: 28 October 2019

The authors would like to sincerely thank the referee for the careful review and constructive comments for improving the manuscript.

1) "How is the `.inv()` command translated to generated code? Are you using LU/Cholesky or directly inverting the matrix? It is clear the former for the `A.solve()` command as the decomposition is passed, but not for `.inv()`."

This is a good question, and we will make this more clear in the updated version of the paper. As mentioned on page 6, Slate expressions are transformed into C++ code

using Eigen as the main linear algebra interface. By default, `A.inv()` gets translated to the corresponding Eigen call: `.inverse()`, which uses LU with partial pivoting for general matrices. It is also the default behavior of `A.solve(B)`. A complete list of factorization strategies compatible with the current version of Slate are documented on Eigen's public webpage (`https://eigen.tuxfamily.org/dox/`), under the category: "Dense linear problems and decompositions." An example of how one would specify a factorization type directly is shown in Listing 1, lines 28 and 30.

2) "How would you deal in practice with non-linear problems solved using, e.g., a Newton-Krylov method? Is it possible to compose SLEPc and the Preconditioned Krylov Solvers? How would one setup SLEPc to call the code to recover the internal/local variables between Newton iterations?"

If you mean "can we differentiate through Slate expressions?" for problems with local non-linear systems, this is not currently supported. However, one can imagine extending the Slate abstraction to support non-linear solvers. The question then becomes: "how do we generate code for this?" This is a topic of on-going discussion within the Firedrake project, and therefore, something like this may very well become part of the core Slate abstraction.

For global non-linear problems, we don't need to do anything extra in Slate. One can use the technology provided in Section 4 of the paper together with PETSc/SLEPc non-linear solvers. Standard Newton, for example, requires the solution of the linearized Jacobian system for the linear updates. Each Newton iteration, therefore, requires a linear solve. In Firedrake, the Jacobian is constructed by differentiating the non-linear PDE (expressed in UFL), which generates yet another UFL expression for the Jacobian. The result is then used as the operator for the linear solver (in PETSc, these are "KSP"s). Hybridization or static condensation can then be used by specifying the correct solver options. In practice, a single Newton iteration would look very similar to the Picard method described in Section 5.2. By design, all preconditioners presented in Section 4 are entirely composable with the PETSc

library and can be applied in the usual way, even when nested inside of an outer non-linear method (e.g. `snes_type newtonls`, `ksp_type preonly`, `pc_type python`, `pc_python_type firedrake.SCPC`). This follows from the framework developed in (Kirby & Mitchell, 2018). We shall stress this more clearly in the revised manuscript.

3) "What does the generated Eigen code look like?"

A .zip file is attached as a supplement (`static_condensation.zip`), which provides an entire generated C++ code and a PDF displaying just the kernel for static condensation (all generated from Slate). This was generated by an example taken directly from the shallow water test case in Section 5.2.

Slate's linear algebra compiler generates templated C++ code conforming to PyOP2's application program interface, described in (Rathgeber et al., 2012). Typically, the generated kernel will take an output tensor, any coefficients, and possibly external data as arguments. The body of the kernel will make appropriate function calls to local assembly kernels generated by Firedrake's form compiler TSFC (Homolya et al., 2018). Local data structures (element matrices/vectors) are derived types of Eigen's Matrix class (`https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html`), and populated by the output of the local assembly kernels. Once all local data structures are filled, the dense linear algebra operations are performed and populated into the output tensor. The result is then passed onto Firedrake's global assembler (PyOP2).

We have discussed the possibility of providing a code listing in the paper, which would show some generated C++. We came to the conclusion that it would be unhelpful at best and, at worst, distracting away from the central message of the paper, which is the Slate abstraction itself.

4) "It is not totally clear how the output from TSFC is fed into the linear algebra compiler. Do you call the TSFC kernel, get the complete cell tensor, split it and then perform the

dense linear algebra operations? Or is everything 'interleaved' into one single cell tensor kernel by the linear algebra compiler? Or do you call multiple TSFC kernels, one for each sub-block and then perform the dense linear algebra operations?"

TSFC splits an assembly kernel for a mixed operator into separate kernels for each sub-block of the local tensor. Then the single kernel responsible for performing the dense linear algebra will first need to make multiple function calls to gather each local contribution. We shall make this more clear in Section 2 of the paper.

5) "The high-level problem setup in sections 1 and 2 is pretty terse. I don't think someone who has some knowledge of FEM but is not a real subject expert could get through this section. Some of the wording is quite heavy on jargon too, e.g. global data structure = sparse matrix? I appreciate you don't have time to do a deep-dive into FE, but a bit more text and some pointers to more detailed explanations (other Firedrake papers?) would be useful."

This is a very fair point. We will improve the sections to make the content more accessible to a general reader. In particular, we will steer away from too much jargon (or give concrete definitions) and provide appropriate references for the more technical details.

6) "Small notational comments: * You do not mention the bold symbol = vector function convention.

* Your convention of non-bold capitals being (local? - not sure) discrete linear operators (matrices and vectors) is also not mentioned, although K (the finite element cell) would break this 'convention'. I'd also note that K is used twice for different objects (the finite element cell and a matrix in eq.76) It would probably aid readability if it was possible to distinguish between matrices and vectors, global and local, especially when the operations become more complex in the later sections. There also seem to be (global? - not sure) discrete linear operators in bold."

We shall clarify our notation throughout the paper to better distinguish between local

and global tensors and their ranks (matrix vs vector). We will also avoid repeated use of particular characters, as you have pointed out (K the cell vs K the element tensor).

References

Kirby, Robert C., and Lawrence Mitchell. “Solver composition across the PDE/linear algebra barrier.” SIAM Journal on Scientific Computing 40.1 (2018): C76-C98.

Rathgeber, Florian, et al. “PyOP2: A high-level framework for performance-portable simulations on unstructured meshes.” 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. IEEE (2012).

Homolya, Miklós, et al. "TSFC: a structure-preserving form compiler." SIAM Journal on Scientific Computing 40.3 (2018): C401-C428.

Please also note the supplement to this comment:
https://www.geosci-model-dev-discuss.net/gmd-2019-86/gmd-2019-86-AC2-supplement.zip

---

Interactive comment on Geosci. Model Dev. Discuss., https://doi.org/10.5194/gmd-2019-86, 2019.