Finch: Domain Specific Language and Code Generation for Finite Element and Finite Volume in Julia

Eric Heisler, Aadesh Deshmukh, and Hari Sundar

School of Computing, University of Utah, Salt Lake City, Utah, USA
`eric.heisler@utah.edu`, `u1369232@utah.edu`, `hari.sundar@utah.edu`

Abstract. We introduce Finch, a Julia-based domain specific language (DSL) for solving partial differential equations in a discretization agnostic way, currently including finite element and finite volume methods. A key focus is code generation for various internal or external software targets. Internal targets use a modular set of tools in Julia providing a direct solution within the framework. In contrast, external code generation produces a set of code files to be compiled and run with external libraries or frameworks. Examples include a `matlab` target, for smaller problems or prototyping, or `C++/MPI` based targets for larger problems needing scalability. This allows us to take advantage of their capabilities without needlessly duplicating them, and provides options tailored to the needs of the domain scientist. The modular design of Finch allows ongoing development of these target modules resulting in a more extensible framework and a broader set of applications. The support for multiple discretizations, including finite element and finite volume methods, also contributes to this goal. Another focus of this project is complex systems containing a large set of coupled PDEs that could be challenging to efficiently code and optimize by hand, but that are relatively simple to specify using the DSL. In this paper we present the key features of Finch that set it apart from many other DSL options, and demonstrate the basic usage and current capabilities through examples.

Keywords: Domain specific language · Code generation · Finite element method · Finite volume method · Parallel computing · Julia.

1 Introduction

Solving partial differential equations (PDEs) numerically on a large scale involves a compromise between highly optimized code exploiting details of the problem or hardware, and extensible code that can be easily adapted to variations. Rapidly evolving technology and a shift to heterogeneous systems places a higher value on the latter, prompting a move away from hand-written code made by experts in high performance computing, to generated code produced through a high-level domain specific language (DSL). Another motivating factor is the realm of medium-scale problems where good performance is needed, but

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_9

the cost of developing optimal code may not be justified. At this scale it is up to domain scientists to develop their own software or piece it together from more general-purpose libraries. Finally, the choice of discretization method, like finite element(FE) or finite volume(FV), is significant in multiphysics systems where different aspects of the system are better handled by different methods.

In response, numerous DSLs for solving PDEs have been developed. On one end of the spectrum are high-level options such as Matlab Toolboxes and Comsol. They are general-purpose and don't require a high level of programming skill. As a trade-off, they lack customizability. The low-level code is often, by design, hidden from the user and difficult to modify.

At the opposite end are lower-level libraries such as Nektar++[4] and deal.II[3] providing customizable components optimized for a specific purpose. They require more programming input and skill from the user. This also makes it harder to modify the code for variations, resulting in many of the limitations of hand-written code.

This work aims for a middle-ground, where most of the programming input is handled within the scope of a moderately high-level DSL while allowing low-level customization and in some cases direct code modification. Some options in this realm include Fenics[2] and Firedrake[25] for finite element methods, OpenFOAM[10] for finite volume methods, Devito[20] for finite difference methods, and many others focused on a specific type of problem or technique. There are also tools in Julia including DifferentialEquations.jl[24] which provides a broad environment of ordinary differential equation solvers with a Julia interface.

This work introduces Finch, a DSL for solving PDEs. The framework aims to be discretization agnostic, and currently supports finite element and finite volume methods. The goal is to enable a domain scientist to create efficient code for problems ranging from small scale simulations on a laptop computer, to larger systems requiring scalability on modern supercomputers. Two key ideas to achieving this goal are a modular software design and generation for external software frameworks.

Rather than depending on a single, general-purpose code, a set of modules are used to grant the flexibility to adapt to problem requirements or resources. Some examples include various discretization methods such as FE, both CG and DG variants, and FV, as well as numerical tools such as PETSc's linear solvers, GPU based options, or matrix-free methods. The development of new modules opens up possibilities for optimization and new types of problems

Another strategy is the generation of code for various external software targets. This allows it to leverage the capabilities of existing software frameworks that are well suited to a type of problem. For example, the Dendro library[9, 26, 8] provides an adaptive octree framework that is suitable for very large scale problems using distributed memory parallel techniques. Manually writing code for this framework requires high programming proficiency and familiarity with the software. Finch provides a simpler interface to this resource while presenting the generated code to the user for modification or inspection. Another target is `C++` using the aMat[27] library which handles the mesh and data structure creation in Julia then utilizes a library of efficient parallel sparse matrix operations to compute the solution in an independent `C++` program. The diversity of code generation targets allows constructing a set of tools suiting a user's needs.

FINCH is written completely in Julia, which is easy to use and has speed comparable to low-level languages such as C[16]. Julia is growing in popularity as a serious scientific computing language. It allows a simplified, intuitive interface without resorting to external C/C++/Fortran libraries as is common with Python-based DSLs. The metaprogramming features and wide selection of libraries also make Julia a convenient choice for FINCH.

2 Related Work

DSLs can be found in some form for countless mathematical and computational tasks. Some examples with a similar purpose and interface include the Unified Form Language(UFL)[1] and FreeFEM[12] used to write variational forms of PDEs. Components corresponding to test functions, trial functions, and other values are combined in expressions representing volume or facet integrals of elements. Since FINCH was originally developed for FE, a similar design was chosen. The internal representation involves categorizing terms of the expression depending on type of integral and linear vs. bilinear forms. The Julia-based FE DSL MetaFEM[29] also involves writing a variational form expression, though with a very different grammar.

In contrast, FINCH is designed to accommodate more general types of expressions and does not assume a variational form. It also allows custom operator definitions that act on the symbolic tensor arrays of entities in the expression. For example, when using a FV method, specialized flux operators can be defined and included in the PDE expression.

A relevant FV DSL is used by OpenFOAM[21], which again involves components such as variables and coefficients in an expression resembling the mathematical notation. This works with a predefined set of operations and is designed specifically for types of problems that commonly use FV methods. There is no notion of variational forms.

It is worth noting some modules of Dune[6], such as Dune-fem are designed for both FE and FV methods, but these are low-level interfaces that are difficult to compare to the higher-level DSLs described here.

The other aspect is code generation where the internal representation becomes numerical code. There are many code generation techniques for FE. Some exploit tensor product construction for high order FE[28][22][15]. Others use the independent nature of Discontinuous Galerkin methods to utilize GPUs[5] or vectorization[17]. The FE software FEniCS utilizes the set of tools FFC[18] and Dolfin[19]. There are also options for FV[23] and FD[21], though perhaps less common than for FE.

The code generation modules used by FINCH are specific to their target, and employ a variety of techniques accordingly. The modular design allows selection of ideal techniques either by the user or automatically depending on the target software, hardware, or problem details.

3 Domain Specific Language

The goal of a mathematical DSL is to provide an interface that closely resembles the notation used by domain scientists while reducing extraneous details and syntax needed by the underlying programming language. Many DSLs accomplish this in an object-oriented way by creating classes representing mathematical objects with a set of intuitive operations. We have adopted a similar strategy in which the basic components of the equations, such as unknown variables, coefficients and test functions, are very basic objects that include an array of symbolic components. For example, a 3-dimensional vector quantity u would correspond to the array $[u_1, u_2, u_3]$. Common arithmetic and differential operations are defined for these objects, and users can define their own custom operators that act on these symbolic arrays. It is also possible to use these basic operations to build packages of specialized operators for a class of problems.

As an example, the following code creates a vector-valued unknown variable u, a known scalar coefficient k defined by a function of coordinates (x, y, z, t), and a vector test function v which belongs to the same function space as u.

```
u = variable("u", type=VECTOR)
coefficient("k", "sin(pi*x)*y*z")
testSymbol("v", type=VECTOR)
```

The differential equations are written in terms of these objects. When using FE, this is done by writing the weak form of the equation in residual form. Note that integration over the volume is implied, and surface integrals can be specified by wrapping those terms in `surf(...)`. Here is an example of specifying a Poisson equation.

Original PDE	$\nabla \cdot (a\nabla u) - f = 0$
Weak form	$-(a\nabla u, \nabla v) - (f, v) = 0$
FINCH input	`-a*dot(grad(u),grad(v)) - f*v`

When using FV, it is assumed that the equations are in a conservation form. The source and flux terms are given as input, and the time derivative of the variables is implied as shown in the following advection-reaction equation

PDE	$\int_V \frac{du}{dt} dx = \int_V g(u,x)dx - \int_{\partial V} \mathbf{f}(u,x) \cdot \mathbf{n} ds$
Source	$g(u,x) = ku$
Flux	$\mathbf{f}(u,x) = u\mathbf{b}$
FINCH input	`source(k*u)`
	`flux(u*b)`

In addition to the standard operators such as $*$, $-$, `dot` and `grad` used above, a user can define new operators to put in these expressions. For example, the flux shown will result in a central flux approximation. To use a custom flux, one could define the operator `myFlux(u,b)`, and substitute that for `u*b` in the expression above. Note that this definition could be either a symbolic manipulation of the array for $u * b$ or a numerical callback function when needed.

3.1 User Input

Typically a Julia script will be written for a particular problem, but it is also possible to work interactively. A set of functions or macros are used to a) Set up the configuration, b) Specify the mesh, entities and equations, and c) Process data for output. A variety of example scripts are in the repository[13].

As an example, the following commands will configure a 2D unstructured grid using a fourth-order polynomial function space based on Lobatto-Gauss nodes, and generate code for a target specified in the `external_target_module.jl` file.

```
generateFor("external_target_module.jl")
domain(2, grid=UNSTRUCTURED)
functionSpace(space=LEGENDRE, order=4)
nodeType(LOBATTO)
```

In contrast to this, a user who is content with the defaults could provide as little as `domain(2)`.

Problem specification should start with a mesh. There are some simple mesh generation options built in. For example, to construct a uniform 50×20 grid of quadrilateral elements in a unit square domain with a separate boundary ID for each face, use the command: `mesh(QUADMESH,elsperdim=[50,20],bids=4)`

For more practical problems, external mesh generating software can be used to create a mesh file that is then imported into FINCH. Currently the GMSH(`.msh`) and MEDIT(`.mesh`) formats are supported.

Separating boundary regions for additional boundary conditions is done with the command `addBoundaryID(BID, onBdry)` where `BID` is a number to be assigned to that region, and `onBdry` is a function or expression of (x, y, z) that is true within the desired region.

For distributed memory parallelism it is necessary to partition the mesh. This is done internally using METIS via the Julia library METIS_jll. This will be done automatically according to the number of processes available through MPI, but can be configured as desired.

After setting up the scenario, entities such as variables and coefficients are defined and expressions for the equations are input as described above. Using the command `solve(u)` will then either generate the code files for external targets or run the internal solver to produce a solution. Considering the internal route, the solution will now be found in the `u.values` array, and is available for post-processing, visualizing, or output in a number of formats such as binary data or VTK files.

Indexed Entities - Some problems involve a set of several quantities that share the same type of equation with different parameters. Similarly, one may try to solve an equation over a range of parameters. In these cases indexed variables and coefficients greatly simplify the way the problem is specified and present an opportunity to reorganize the code in a more optimal way. As an example, consider a set of unknown quantities $u_{i,j}$ and a corresponding set of coefficients k_i belonging to the same type of diffusion equation. For brevity the dependence on j is omitted, but could correspond, for example, to different boundary conditions.

$$\frac{d}{dt}u_{i,j} = k_i \mathit{\Delta} u_{i,j} \quad i = 1...20\ ,\ j = 1...40$$

It is cumbersome to individually write out the equations if there are many values of i and j. Rather, we can write one equation using indexed entities.

```
I  = index("I", range=[1,20])
J = index("J", range=[1,40])
u = variable("u", type=VAR_ARRAY, location=NODAL, index = [I,J])
k = coefficient("k",k_array,type=VAR_ARRAY,location=NODAL,index=I)
weakForm(u, "Dt(u[I,J]*v) + k[I]*dot(grad(u[I,J]),grad(v))")
assemblyLoops(u, [I, J, "elements"])
```

The last line describing assembly loops instructs the code generator to nest the assembly loops in this order. In some cases it may be more efficient to parallelize an outer index loop before the elemental loop. The user can arrange this as desired.

3.2 Symbolic Representation

After entering the expressions for the equations, they are transformed into an intermediate symbolic representation. The entity symbols are replaced with arrays of corresponding tensor components, as discussed above, and the operators are applied to ultimately create a set of symbolic expressions. These expressions go through processing stages to separate known and unknown terms, simplify them, and identify time dependent terms. The resulting symbolic terms are in the form of computational graphs, based on Julia `Expr` trees, containing symbolic entity objects. These graphs are what is eventually passed to the code generation utilities.

This simple chart illustrates the process using the weak form input for a 2D Poisson equation. The input expression starts at the top, symbols are substituted, operators are applied, the terms are partitioned into groups and computational graphs are built with symbolic entities.

```
                    -a*dot(grad(u), grad(v)) - f*v
                                  ↓
 -[_a_1]*dot_op(grad_op([_u_1]), grad_op([_v_1]))-[_f_1]*[_v_1]
                                  ↓
[-(_a_1*D_1__u_1*D_1__v_1 + _a_1*D_2__u_1*D_2__v_1)]+[-_f_1*_v_1]
                                  ↓
```

bilinear: `[-_a_1*D_1__u_1*D_1__v_1 - _a_1*D_2__u_1*D_2__v_1]`
linear: `[-_f_1 * _v_1]`
entities: `D_1__u_1` $= \frac{d}{dx}u_1$, `D_2__u_1` $= \frac{d}{dy}u_1$, etc.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_9

The entity is essentially a symbol, like `_u_`, along with it's component index on the right, `1`, and a collection of flags on the left, `D_1_`. The flags can have any value and will be interpreted by the relevant code generation module. For example, the flags `CELL1_` and `CELL2_` would be interpreted as values on respective sides of a face in a finite volume context.

4 Code Generation

The code generation step is where the process diverges. The details are specific to the generation target, but they essentially all perform the same two tasks. They must interpret the computational graph containing symbolic entities described above, generating their mathematical equivalent, and they must collect these calculations in a functional piece of code that performs the overall computation.

When designing a new target module, there are only three functions that must be provided. The first one, `get_external_language_elements`, provides basic language-specific info such as comment characters to aid with formatting. The second is `generate_external_code_layer`, which interprets the computational graph of the symbolic representation and generates code to perform the elemental calculations. The third function, `generate_external_files`, is responsible for creating all of the code files. It takes the elemental calculation from the second function and wraps it in the rest of the code to create a complete program including build files and instructions.

4.1 Elemental Computation

The elemental computation varies significantly with different targets, but to illustrate the process the 2D Poisson example from above will used as input. This type of problem will essentially need code for assembling elemental matrices and vectors embedded in an elemental loop. The elemental matrix will correspond to the bilinear terms,

```
-_a_1 * D_1__u_1 * D_1__v_1 - _a_1 * D_2__u_1 * D_2__v_1
```

Note that this symbolizes

$$\int_K \left(-a\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} - a\frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right) dK$$

When discretized into polynomial basis functions at Gaussian integration points, this becomes

$$A_{jk}u_j = \sum_j u_j \sum_i w_i J_i \left(-a_i * \phi_{ij,x}\phi_{ik,x} - a_i * \phi_{ij,y}\phi_{ik,y}\right)$$

Where w_i are quadrature weights, J_i are geometric factors, $\phi_{ij,x}$ are x-derivatives of the jth basis functions at the ith quadrature points. The inner i sum can be arranged as a matrix expression.

$$Q_x^T W Q_x + Q_y^T W Q_y$$

With W being a diagonal matrix combining weights, geometric factors, and a_i for each quadrature point. Q_x combines geometric factors with precomputed matrices Q_R that essentially contain the basis function derivatives,$\frac{\partial \phi}{\partial R}$, at the quadrature points in a reference element, but in practice it will be more sophisticated as it will include a transformation from a nodal basis into a modal one to benefit from better properties. For details on this, please refer to [14].

When the code generator encounters a term like
`_a_1 * D_1__u_1 * D_1__v_1` it will recognize the three factors as coefficient, unknown, and test function respectively and make the associations

`D_1__v_1` $\rightarrow Q_x^T$
`D_1__u_1` $\rightarrow Q_x$
`_a_1` $\rightarrow a_i$

and create code to perform those matrix operations. The way this calculation is implemented is up to the code generator, which provides opportunities for optimization. For example, when using uniform elements the Jacobian matrix only needs to be computed for one element and Q_x can be fully precomputed. Taking it one step further, if the coefficients in this term are also constant, the entire $Q_x^T W Q_x$ matrix can be precomputed.

Another opportunity for optimization depends on element type. For example, the Dendro target exclusively uses hexahedral elements and exploits their symmetry by using the tensor product of one-dimensional operators. This saves on both arithmetic and memory costs.

When designing a new target or when taking advantage of some new hardware, these elemental calculations can be optimized in a modular way that makes the transition easy.

4.2 Global computation

After handling the elemental computation, the next task is to combine these results into a global system. This is mainly where parallel strategies come into play. Since this typically involves looping over elements to assemble and solve a global linear system, the process can be parallelized using multithreading, distributed memory multiprocessing, and GPU techniques. Again, the details of this task may look completely different depending on the target and in many cases it is handled by the external software framework.

Note that when using FV the mathematics will be substantially different, but the overall structure of the computation is similar.

4.3 Modifying Generated Code

Advanced users may wish to inspect the generated code and make modifications by hand. In many cases there may be features of the problem that can be exploited for better performance that are not automatically included. For this purpose the elemental assembly code can be exported to a code file, modified as desired, and imported again to either run the calculation or generate the full code package for external targets. The commands for this are `exportCode` and `importCode`. Naturally, exporting should happen after the equations have been entered, and importing is done on a later run before solving.

Note that this code typically only contains the elemental assembly function. For even more control it is possible to also export and import the full assembly loop code for internal targets, but a good familiarity with the FINCH data structures is needed to take advantage of this. Similarly, since external targets are fully accessible as code files they can be modified as desired depending on the user's knowledge of the target software.

5 Performance Opportunities

Since one of the goals of FINCH is to take advantage of the capabilities of specialized external software, there are various strategies for parallelization, adaptivity, and efficient data structures available to achieve high performance. One example of this is the DENDRO target which offers distributed memory parallelism through MPI, adaptive mesh refinement, and proven large-scale scalability. It is ideal for problems that can benefit from very fine grained adaptive meshing, but is limiting in the possible domain geometry.

Another target is AMAT which is essentially a specialized linear algebra library providing very efficient algorithms for sparse linear systems. It also supports an assortment of parallelization strategies based on `MPI`, `OpenMP`, and GPU options.

The performance of both of these targets is explored in the Demonstrations section below.

5.1 Performance Within FINCH

The performance focus is not limited to external tools. The internal Julia targets can also make use of distributed and shared memory parallelism as well as efficient data organization options. When solving linear systems, the user can select a variety of tools beyond the defaults provided by Julia's LinearAlgebra package.

The simplest way to take advantage of these tools is with multithreading. FINCH automatically detects how many threads are available to the Julia instance and uses the native Julia package `Threads` to take advantage of this throughout the computation. To enable this feature a user simply needs to specify the number of threads when launching Julia. This is done with the argument `-t n` or `--threads n` to use `n` threads, or substitute `auto` in place of `n` to use the number of local CPU threads.

Distributed memory parallelism is provided by the Julia package `MPI.jl` which makes use of the system's available MPI implementation. Again, this is specified at launch using the system's MPI execution command. FINCH will detect how many processes are available and arrange the computation accordingly.

Partitioning is needed when using a distributed parallel strategy, and the most straightforward method is to partition the mesh evenly among the processes. This is accomplished using `Metis` through the `METIS_jll` package.

When using FV with higher order flux reconstructions several neighboring elements may be needed. This could potentially complicate the partitioning process because an irregular number of ghost elements would need to be maintained. To address this issue, partitioning is done on the initial course mesh with only nearest neighbor ghosts. Then the elements are refined in a consistent way depending on the flux order desired. The resulting finer mesh will include the needed number of ghosts in an efficient and reliable way. The refinement should be considered when planning a mesh utilizing this feature.

There are several choices when it comes to solving large, sparse linear systems. The default in Finch is provided by the LinearAlgebra package which utilizes BLAS and LAPACK. Another option is PETSc, interfaced through `PETSC.jl`, which provides better performance in distributed parallel environments as demonstrated below. There is also a matrix-free option for certain targets that is particularly useful for large-scale problems where the cost of assembling a global matrix is prohibitive.

5.2 Cache Optimization

In addition to parallel techniques, the organization of data structures and the elemental loop ordering can improve performance through more efficient cache use. A number of data organization options are available in FinchḞor example, a mesh from the built-in mesh generation utility provides elements that are ordered lexicographically. In order to improve spatial locality, the elements can be rearranged either into a space-filling curve, such as a Hilbert or Morton curve, or into tiles.

To aid with this development and potentially provide a means for automated tuning, Finch employs a cache simulator. Pycachesim[11] was chosen for this because it is light-weight and although it was developed for use in Python, the backend is written in C. The C library can be utilized directly by Finch to roughly characterize the cache performance of a particular problem setup on a specified cache hierarchy.

The cache simulator is essentially another target for code generation. Rather than performing the mathematical computation, the approximate sequence of memory accesses is fed into the simulator. At the end of the computation the cache statistics are recorded and analyzed. This presents Finch with a tool for tuning and measuring the effectiveness of changes in configuration.

6 Demonstration

The following example applications demonstrate some of the capabilities of Finch and illustrate the performance aspects of the various tools and code generation targets. Since external targets rely on the performance capabilities of the target framework, please refer to their respective documentation for a more rigorous analysis. For example, the Dendro framework has shown competitive scalability for large scale simulations[8][7].

6.1 Steady-state Advection-diffusion-reaction Equation

The following equation(1) is used to demonstrate a FE problem. We use several different sets of tools and compare them in terms of performance.

$$\nabla \cdot (D\nabla u) + \mathbf{s} \cdot \nabla u - cu = f \tag{1}$$

$$u(x \in \partial\Omega) = 0$$

$$\Omega = [0,1]^3, D = 1.1, c = 0.1, \mathbf{s} = (0.1, 0.1, 0.1)$$

Here all of the coefficients are given constant value, but we have intentionally generated them as functions of (x, y, z) to increase computational complexity. The motivation for this is to demonstrate the performance for a more practical problem while simplifying analysis with an exact solution. The function f was constructed such that u satisfies (2).

$$u(x, y, z) = \sin(3\pi x)\sin(2\pi y)\sin(\pi z) \tag{2}$$

The weak form expression provided to FINCH is

```
weakForm(u,
    "-D*dot(grad(u), grad(v)) + dot(s, grad(u))*v - c*u*v - f*v")
```

The discretization is continuous Galerkin with quadratic hexahedral elements.

Internal target With appropriate choice of mesh this is suitable for running on a typical computer, but for these tests we are using the Frontera supercomputer with dual socket Intel Xeon Platinum 8280 nodes having 56 cores. The execution time of different code generation targets and linear solvers were compared for a range of processor counts as shown in figure 1. For smaller problems running on only a few cores, the default Julia tools are an easy and viable option, though PETSc may be more efficient. The default method does not scale well in a distributed memory parallel context. For larger problems and many processors, PETSc and matrix-free are both good options. The figure shows that the matrix-free method is better when many processors are available. On the other hand, PETSc performed better for small process counts.

aMAT target The same problem was solved using the aMAT target. Code files, partitioned mesh data, reference element, and geometric factors were set up and exported from Julia. The code was compiled and run with the aMAT library using the precomputed data. aMAT provides options for assembling and solving the system including a direct PETSc solve or a hybrid matrix technique. MPI, OpenMP, and GPU tools are available. Figure 1(bottom left) compares the PETSc and hybrid versions based on MPI with the same hardware as above. Note that this execution time does not include the mesh creation and other setup. A comprehensive total would also include the compilation and file management time when using an external target, but they are omitted here.

DENDRO target Finally, the same problem was solved using the DENDRO target. Code files were generated in Julia then compiled using the DENDRO library. The resulting adaptively refined mesh produced by DENDRO contained

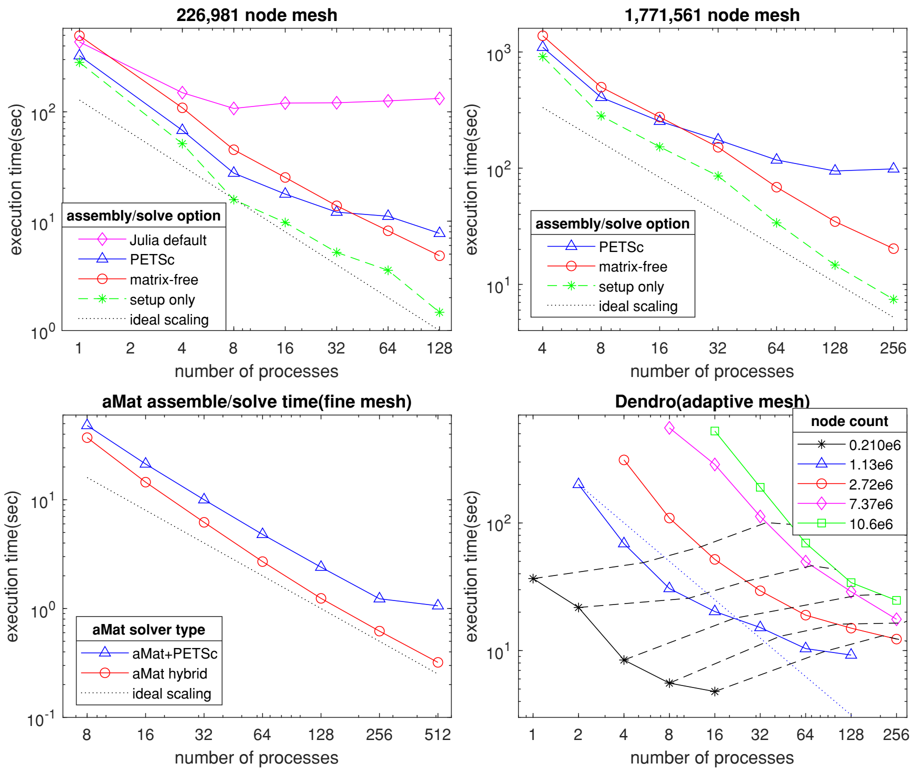

Fig. 1. *Top row:* Internal target execution time on a coarse(*top left*) and fine mesh(*top right*) with different options. "Setup only" excludes assembly/solve time. *Bottom left:* aMat's time on the fine mesh using PETSc and hybrid methods. Only assembly and solve time is included as setup is done separately within Finch. *Bottom right:* Dendro's execution time for several mesh sizes. Black dashed lines show interpolated weak scaling contours. The blue dotted line is an ideal scaling based on the blue curve.

between 0.21 million and 10.6 million nodes depending on input parameters. It was tested on the Notchpeak cluster at the University of Utah using two-socket Intel XeonSP Skylake nodes with 32 cores each. Figure 1(bottom right) demonstrates that the computation scales well for this problem on a fine mesh up to 256 processes.

This ability to quickly test a model in Julia before seamlessly transitioning to a more specialized external target is a key feature of Finch that can significantly speed up development time for complex multiphysics problems.

7 Conclusion

This paper presents Finch, a new DSL and code generation framework for PDEs. The modular design and support for external code generation targets provides versatility and allows the user to take advantage of evolving technology in terms of high performance software packages such as DENDRO, and hardware resources supporting multithreading, MPI, and GPUs. The discretization agnostic concept, currently including finite element and finite volume techniques, further expands the range of applications for which it is well suited. We demonstrate and compare the performance capability of several code generation targets and configurations.

Acknowledgements This work was funded by National Science Foundation grants 1808652 and 2008772. The computing resources on Frontera were through an allocation by the Texas Advanced Computing Center PHY20033.

References

1. Alnæs, M.S., Logg, A., Ølgaard, K.B., Rognes, M.E., Wells, G.N.: Unified form language: A domain-specific language for weak formulations of partial differential equations. ACM Trans. Math. Softw. **40**(2) (mar 2014). https://doi.org/10.1145/2566630
2. Alnaes, M.S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M.E., Wells, G.N.: The fenics project version 1.5. Archive of Numerical Software. **3**(100), 9–23 (2015). https://doi.org/10.11588/ans.2015.100.20553
3. Arndt, D., Bangerth, W., Blais, B., Clevenger, T.C., Fehling, M., Grayver, A.V., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.P., Rastak, R., Thomas, I., Turcksin, B., Wang, Z., Wells, D.: The `deal.II` library, version 9.2. Journal of Numerical Mathematics **28**(3), 131–146 (2020). https://doi.org/10.1515/jnma-2020-0043, https://dealii.org/deal92-preprint.pdf
4. Cantwell, C., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J.E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R., Sherwin, S.: Nektar++: An open-source spectral/hp element framework. Computer Physics Communications **192**, 205–219 (2015). https://doi.org/10.1016/j.cpc.2015.02.008
5. Dorozhinskii, R., Bader, M.: Seissol on distributed multi-gpu systems: Cuda code generation for the modal discontinuous galerkin method. In: The International Conference on High Performance Computing in Asia-Pacific Region. pp. 69–82. HPC Asia 2021, ACM Press, New York, NY (2021). https://doi.org/10.1145/3432261.3436753
6. Dune: Dune (2022), https://www.dune-project.org
7. Fernando, M., Neilsen, D., Lim, H., Hirschmann, E., Sundar, H.: Massively parallel simulations of binary black hole intermediate-mass-ratio inspirals. SIAM J. Sci. Comput. **42**(2), 97–138 (Apr 2019). https://doi.org/10.1137/18M1196972
8. Fernando, M., Neilsen, D., Sundar, H.: A scalable framework for adaptive computational general relativity on heterogeneous clusters. In: Proceedings of the ACM International Conference on Supercomputing. pp. 1–12. ICS'19, ACM Press, New York, NY (2019). https://doi.org/10.1145/3330345.3330346
9. Fernando, M., Sundar, H.: Dendro home page, 2020. URL https://octree.org
10. Foundation, T.O.: Openfoam (2022), https://openfoam.org
11. Hammer, J.: pycachesim: Python cache hierarchy simulator (2001), https://github.com/RRZE-HPC/pycachesim

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_9

12. Hecht, F.: New development in freefem++. J. Numer. Math. **20**(3-4), 251–265 (2012), https://freefem.org/
13. Heisler, E., Deshmukh, A., Sundar, H.: Finch code repository (2022), https://github.com/paralab/Finch
14. Hesthaven, J.S., Warburton, T.: Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications. Springer Verlag (2008)
15. Homolya, M., Kirby, R.C., Ham, D.A.: Exposing and exploiting structure: optimal code generation for high-order finite element methods (2017), https://arxiv.org/abs/1711.02473
16. JuliaLang.org: Julia benchmarks (2021), https://julialang.org/benchmarks
17. Kempf, D., Heß, R., Müthing, S., Bastian, P.: Automatic code generation for high-performance discontinuous galerkin methods on modern architectures. ACM Trans. Math. Software **47**(1), 1–31 (Dec 2020). https://doi.org/10.1145/3424144
18. Kirby, R.C., Logg, A.: A compiler for variational forms. ACM Trans. Math. Softw. **32**(3), 417–444 (sep 2006). https://doi.org/10.1145/1163641.1163644
19. Logg, A., Wells, G.N.: Dolfin: Automated finite element computing. ACM Trans. Math. Softw. **37**(2) (apr 2010). https://doi.org/10.1145/1731022.1731030
20. Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P.A., Herrmann, F.J., Velesko, P., Gorman, G.J.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geoscientific Model Development **12**(3), 1165–1187 (2019). https://doi.org/10.5194/gmd-12-1165-2019
21. Macià, S., Martínez-Ferrer, P.J., Mateo, S., Beltran, V., Ayguadé, E.: Assembling a high-productivity dsl for computational fluid dynamics. In: Proceedings of the Platform for Advanced Scientific Computing Conference. pp. 1–11. PASC '19, ACM Press, New York, NY (2019). https://doi.org/10.1145/3324989.3325721
22. McRae, A.T.T., Bercea, G.T., Mitchell, L., Ham, D.A., Cotter, C.J.: Automated generation and symbolic manipulation of tensor product finite elements. SIAM J. Sci. Comput. **38**(5), 25–47 (Oct 2016). https://doi.org/10.1137/15M1021167
23. Pietro, D.A.D., Gratien, J.M., Häberlein, F., Michel, A., Prud'homme, C.: Basic concepts to design a dsl for parallel finite volume applications: extended abstract. In: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing. pp. 1–12. POOSC '09, ACM Press, New York, NY (2009). https://doi.org/10.1145/1595655.1595658
24. Rackauckas, C., Nie, Q.: Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. Journal of Open Research Software **5**(1), 15 (2017)
25. Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., Mcrae, A.T.T., Bercea, G.T., Markall, G.R., , Kelly, P.H.J.: Firedrake: automating the finite element method by composing abstractions. ACM Trans. Math. Softw. **43**(3), 1–27 (2016). https://doi.org/10.1145/2998441
26. Sundar, H., Sampath, R., Biros, G.: Bottom-Up construction and 2:1 balance refinement of linear octrees in parallel. SIAM J. Sci. Comput. **30**(5), 2675–2708 (Jan 2008)
27. Tran, H., Sundar, H.: A scalable adaptive-matrix spmv for heterogeneous architectures. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. IPDPS'22, accepted for publication (2022)
28. Uphoff, C., Bader, M.: Yet another tensor toolbox for discontinuous galerkin methods and other applications. ACM Trans. Math. Software **46**(4), 1–40 (Oct 2020). https://doi.org/10.1145/3406835
29. Xie, J., Ehmann, K., Cao, J.: Metafem: A generic fem solver by meta-expressions (2021)