Journal of Computational Science 68 (2023) 101981

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Multi-discretization domain specific language and code generation for differential equations

Eric Heisler [a,*], Aadesh Deshmukh [a], Sandip Mazumder [b], Ponnuswamy Sadayappan [a], Hari Sundar [a]

[a] School of Computing, University of Utah, Salt Lake City, 84112, UT, USA
[b] Department of Mechanical and Aerospace Engineering, The Ohio State University, Columbus, 43210, OH, USA

ARTICLE INFO

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

ABSTRACT

Finch, a domain specific language and code generation framework for partial differential equations (PDEs), is demonstrated here to solve two classical problems: steady-state advection diffusion equation (single PDE) and the phonon Boltzmann transport equation (coupled PDEs). Both finite volume and finite element methods are explored. In addition to work presented at the 2022 International Conference on Computational Science (Heisler et al., 2022), we include recent developments for solving nonlinear equations using both automatic and symbolic differentiation, and demonstrate the capability for the Bratu (nonlinear Poisson) equation.

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 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, Mathematica, and Comsol. They are general-purpose and do not 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++[1] and deal.II [2] 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, when desired, direct code modification. Some options in this realm include Fenics [3] and Firedrake [4] for finite element methods, OpenFOAM [5] for finite volume methods, Devito [6] 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 [7] 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,

* Corresponding author.
E-mail addresses: eric.heisler@utah.edu (E. Heisler), u1369232@utah.edu (A. Deshmukh), mazumder.2@osu.edu (S. Mazumder), saday@cs.utah.edu (P. Sadayappan), hari@cs.utah.edu (H. Sundar).

https://doi.org/10.1016/j.jocs.2023.101981
Received 14 October 2022; Received in revised form 13 February 2023; Accepted 24 February 2023
Available online 2 March 2023

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 Finch to leverage the capabilities of existing software frameworks that are well suited to a type of problem. For example, the Dendro library [8–10] 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 library [11] 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 [12]. 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.

This paper is an extended version of work presented at the 2022 International Conference on Computational Science [13]. We include new capabilities for solving nonlinear equations (Sec. 6.3), a new intermediate representation that greatly improves the interface between the DSL and code generation modules (Sec. 4.2), and we explore a variety of example problems to demonstrate the range of Finch's capabilities (Sec. 8).

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) [14] and FreeFEM [15] 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 DSL MetaFEM [16] also involves writing a variational form expression, though with a 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 [17], 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.

Table 1 illustrates a rough feature comparison with these similar DSLs. One feature to note is generation of user-accessible and modifiable code to allow inspection and manual tuning, which none of the others provides.

It is worth noting some modules of Dune [18], 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 [19–21]. Others use the independent nature of Discontinuous Galerkin methods to utilize GPUs [22] or vectorization [23]. The FE software FEniCS utilizes the set of tools FFC [24] and Dolfin [25]. There are also options for FV [26] and FD [17], though perhaps less common than for FE.

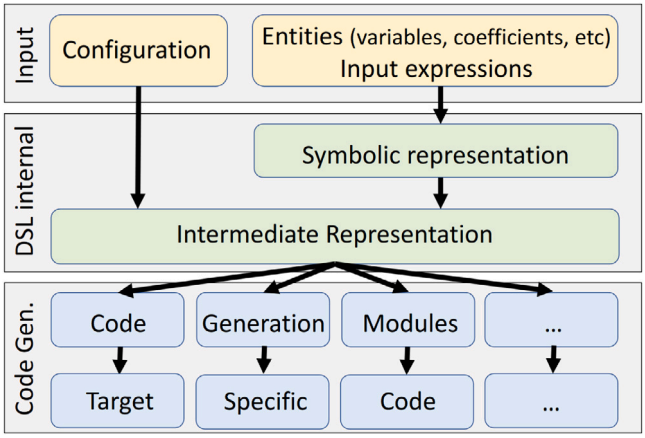


Fig. 1. Chart illustrating the steps from user input to target-specific code.

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. Fig. 1 outlines the process Finch uses to get from user input to generated code.

3. Domain specific language

The goal of the DSL is to provide a higher-level coding interface that closely resembles the form and notation used in a scientific or mathematical domain, while hiding extraneous programming details and syntax that is required by the underlying programming language. Many of the existing DSLs for differential equations accomplish this in an object-oriented way by creating classes representing symbolic mathematical objects combined with a set of common operations and rules [5,6,14–16]. The form of input is often designed to mimic a convention depending on discretization or equation type. We have adopted a similar strategy in which equations are entered in specific formats. When using an FE discretization, the equation is input in the weak form. On the other hand, when using a conservative FV discretization, a conservation form is used that is set equal to an implicit time derivative of the unknown variable. The examples below illustrate these types of input.

3.1. Input expressions

Input for FE discretization is done by writing the weak form of the equation as a residual expression set equal to zero. The volume integrals are implicit and surface integrals can be specified by wrapping those terms in `surface(...)`. Symbols for test functions must also be defined by the user and be present in each term. In general, after defining a set of variables $[u_1, u_2, \ldots]$ and test functions $[v_1, v_2, \ldots]$, the equation can be written in terms of these symbols using common arithmetic and differential operators. Each term of the expression must involve a test function to be consistent. The residual weak form including linear(F, G) and bilinear(f, g) terms would be expressed as

$$F(u_1, u_2, \ldots; v_1, v_2, \ldots) + f(v_1, v_2, \ldots) + \texttt{surface}(G(u_1, u_2, \ldots; v_1, v_2, \ldots) + g(v_1, v_2, \ldots)) = 0$$

This is illustrated by the following Poisson equation where $F(u, v) = -(a\nabla u, \nabla v)$, $f(v) = -(k, v)$, $G(u, v) = g(v) = 0$, and the convention $(u, v) = \int_\Omega u \cdot v dx$ is used.

Table 1
Feature comparison with similar high-level DSLs.

	Finch	FEniCS	MetaFEM	OpenFOAM
Finite element method	✓	✓	✓	×
Finite volume method	✓	×	×	✓
Unstructured mesh support	✓	✓	✓	✓
Can take advantage of structured meshes such as trees	✓	×	×	✓
Arbitrary equation specification	✓	✓	✓	×
Accessible/modifiable generated code	✓	×	×	×
Modular, swappable backends for linear algebra and parallel tools	✓	✓	×	×
Can provide complete solution from equation to output with no external steps	✓	✓	×	✓

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

A conservative FV discretization will take input in a conservation form in which the expression is assumed equal to a time derivative of the unknown variable. Integrals over a control volume or its surface are entered in the same way as in the weak form above. A conservation equation for variable u will have the form

$$\int_V \frac{du}{dt} dx = \int_V F(u) dx + \int_{\partial V} G(u) ds$$

The following advection–reaction equation illustrates this idea where $F(u) = f(u, x, t)$ is a source function to be integrated over the volume, and $G(u) = g(u, x, t)$ represents a flux to be integrated over the surface of the control volume.

PDE	$\int_V \frac{du}{dt} dx = \int_V f(u,x) dx - \int_{\partial V} \mathbf{g}(u,x) \cdot \mathbf{n} ds$ $f(u,x) = ku\ ,\ \mathbf{g}(u,x) = u\mathbf{b}$
Finch input	`k*u - surface(u*dot(b,normal()))`

The building blocks of these input expressions are the variables, coefficient functions, test functions, and operators. With the exception of an included set of operators, these components are defined by the user. Custom operators can also be defined and are discussed further below.

As an example, the following code creates a vector-valued unknown variable u, a known scalar coefficient k defined by a function of spacetime 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)
```

3.2. 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 one 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 for better performance. As an example, consider a set of unknown quantities $u_{i,j}$ and a corresponding set of coefficients k_i and c_j that exist in the same form of equation.

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

It would be very cumbersome to individually write out the large number of equations for different 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, index = [I,J])
k = coefficient("k",k_values,type=VAR_ARRAY,
index=I)
c = coefficient("c",c_values,type=VAR_ARRAY,
index=J)
weakForm(u, "Dt(u[I,J]*v) + k[I]*dot(grad(u[I,J]),
grad(v))
       + c[J]*u[I,J]*v")
assemblyLoops(u, [I, J, "elements"])
```

The generated code when using indexed entities is significantly different than when writing out a set of distinct equations. Since the form of the equations is the same, the assembly process can be described by a loop over indices, which opens up the possibility of more sophisticated parallel strategies. The last line in the example including `assemblyLoops` 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.

3.3. Symbolic operators

To understand the role of operators in this context, we need to describe the internal representation of these entities. When they are created, each of these pieces are assigned corresponding arrays of symbols. For example, a 3-dimensional vector quantity u would correspond to the array $[u_1, u_2, u_3]$. Operators take these arrays and generate new arrays of symbols or subexpressions. `dot(u,v)` will become the single-component (scalar) array $[u_1 * v_1 + u_2 * v_2 + u_3 * v_3]$.

It is important for the user to understand this because in addition to standard operators such as $*$, -, `dot` and `grad` used in the examples above, a user can define new operators to include in these expressions. For example, the conservation form expression for an advection equation may look like `surface(upwind(b,u))`. In this expression, the operator `upwind(b,u)` will result in $[u_{upwind} * (\mathbf{b} \cdot \mathbf{n})]$ where u_{upwind} is the value from the side of the face where $\mathbf{b} \cdot \mathbf{n}$ is positive. If a user wanted to define a new type of flux approximation labeled by the operator `myFlux(b,u)`, they could create a function, `myFluxFunction`, which would take the arrays represented by `u` and `b` and return some new array with the desired symbolic result. The command `customOperator(''myFlux'', myFluxFunction)` will add this new symbolic operator and allow it to be written in input expressions.

There are situations in which the desired operation is difficult or impossible to encode in this kind of symbolic expression. In that case a numerical callback function can be defined and used in the input as a symbolic function. During the code generation step, calls to this function will be created where needed.

3.4. Configuration

Typically a Julia script will be written for a particular problem, though 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.

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. For a complete listing and description of available configuration commands, see the documentation [27].

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

In contrast to this, a user who is content with the defaults could provide as little as `domain(2)`. There are a variety of example scripts included in the repository [28] that explore a range of configurations.

3.5. Mesh

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 rectangular elements in a unit square domain, and create 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.

Specifying new boundary regions for the purpose of defining boundary conditions can be 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 also be configured as desired.

3.6. Solve and post processing

After setting up the scenario as above, the command `solve(u)` will then either generate the code files for external targets or run the computation internally to produce a solution. Considering the internal route, the solution will now be found in the corresponding variable value arrays such as `u.values`. These results are available for post-processing, visualizing, or output in a number of formats such as binary data or VTK files.

If the target is external, there will be a directory including all code and build files along with basic instructions. Since one of the goals of Finch is to generate accessible code, the generated code is designed to be human readable and even include comments.

4. Intermediate representation

4.1. Symbolic level

After entering the input expressions, they are first transformed into a symbolic representation. The entity symbols are replaced with arrays of corresponding components, as discussed above, and the operators are applied to ultimately create a set of symbolic expressions. These expressions then go through several processing stages to separate known and unknown terms, identify time dependent terms, separate volume and surface integrals, and isolate nonlinear terms.

At this point the expressions can be manipulated to simplify or modify them for, for example, the type of time stepping scheme being used. This is also where symbolic differentiation can be used for handling nonlinear problems, as will be discussed further below. The resulting collections of symbolic terms are parsed into computational graphs, based on Julia `Expr` trees, containing symbolic entity objects.

Here we illustrate the process using the weak form input for a 2D Poisson equation.

```
-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 = d/dx u_1, D_2__u_1 = d/dy u_1, etc.
```

An entity in this context is essentially a symbol, like `_u_`, along with its 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 when using FV.

4.2. Finch IR

The symbolic representation produced by the process above is still very dependent on the discretization choice, time stepping scheme, and other configuration details. Before passing it to the code generation step, we want to encode all of that information into a description that will be independent of both the mathematical configuration and the code generation target, an Intermediate Representation (IR). The IR describes the computation at the level of pseudocode. This will make the creation of new target modules much simpler as they will only need to translate the pseudocode into their respective languages and in the context of any external software packages.

To illustrate the level of the IR, we will continue with the 2D Poisson example from above. Since this is a stationary FE problem, the code will essentially involve assembling a global linear system one element at a time and then solving for the unknown vector. For simplicity we will only look at the bilinear term that computes an elemental matrix.

`-_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 [29].

When building the IR for this computation with a term like `_a_1 * D_1__u_1 * D_1__v_1`, the three factors are identified as coefficient, unknown, and test function respectively, with the following associations.

$$\begin{aligned} \texttt{\_a\_1} &\rightarrow a_i \\ \texttt{D\_1\_\_u\_1} &\rightarrow Q_x \\ \texttt{D\_1\_\_v\_1} &\rightarrow Q_x^T \end{aligned}$$

The IR builder will also use knowledge of the configuration to optimize the calculation. For example, when using rectangular or simplex elements the Jacobian matrix can be reduced to a constant value. In the case of a uniform grid, it only needs to be computed for one element and Q_x can be 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 rather than doing it for each element. Other opportunities for optimization may depend on element type. The symmetry of high order hexahedral elements can be exploited by using the tensor product of one-dimensional operators. This saves on both arithmetic and memory costs.

The other optimization that can be done at the symbolic level is arithmetic manipulation. The symbolic software SymEngine is used to simplify expressions and combine terms on the where possible before parsing into the IR. It is important to note that no hardware or target-specific optimizations can be done at this level because the IR is intended to be independent of these.

The resulting IR for this part of the calculation will be an abstract linear algebra operation to construct $Q_x^T W Q_x$. Since different targets will have different ways of optimally evaluating this, such as differing data layouts and the use of accelerators, the IR is kept at this level and will need to be lowered when generating code.

5. 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 translate the IR described above into real code, and they must wrap the calculation in a complete program that includes setting up the mesh and other data structures, processing results, and anything required by external software packages. Build files, instructions, and other documentation will also be generated at this step. To allow easy addition of new targets, the code generation process has a modular design with a very simplified interface.

When designing a new target module, there are only two functions that must be provided to Finch. The first one, `get_external_language_elements`, provides basic language-specific info such as comment characters and file extensions to aid with formatting. The second is `generate_external_files`, which takes the IR and some configuration information, and is responsible for creating all of the code files.

This is also the part of the process where high performance computing strategies are incorporated. Again, the details of this task may look completely different depending on the target and in many cases they are handled by the external software libraries being utilized.

One example of the optimizations performed by the code generation step is organization of the quadrature loops to take advantage of vectorization. Since different target languages use different data organization strategies, such as Julia's column-major matrices as opposed to row-major used by C++, the linear algebra operations described by the IR are translated into different loop structures. However, more significant optimizations are often handled by target software, such as the adaptive refinement and load balancing performed by the external Dendro library

5.1. 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. All generated code files are, of course, accessible when using external targets, but they must also be written in a way that can be easily understood by the user. For this purpose the IR also includes the unusual feature of comment nodes. Rather than simply storing the computational steps, the IR contains comments that will be written into any generated files. This makes the result much easier to understand, facilitating code modification by hand.

When using the internal Julia target, the code is typically to be used immediately rather than being written to a file. If code inspection or modification are desired, the command `exportCode` will cause the generated Julia code for the solve portion of the computation to be written to a file. Note that this does not represent a stand-alone program, but simply the code that computes the solution based on existing data structures and utilities. This allows the exported code to be relatively compact and easy to understand.

After modifying the Julia code, the command `import` will read the code into Finch to be used in the solution. Naturally, exporting should happen after the entire problem has been specified, and importing is done on a later run before calling `solve`. Since the imported code can be freely modified without restriction, it is up to the user to ensure that the modified code is correct. No verification is performed by Finch to check correctness against a specified equation because this would restrict the ability to modify code as desired.

6. Features

6.1. Code generation targets

Finch is intended to provide a complete problem solving environment, from equation input to solution output. On the other hand, it is also designed to be used as a code creation tool, which takes a problem specification and generates a body of code to be run externally, perhaps in the context of some third-party software framework. There are significant benefits to both directions. A stand-alone solution makes model exploration and learning the software much more efficient. External targets provide flexibility and an array of high performance tools, without having to reinvent them or design Finch around them.

The default choice is to use the internal tools included in Finch. This solution will generate Julia code that can either be run directly or exported and imported as discussed in the previous section. This path can also make use of multiprocessing based on MPI or multithreading using Julia's threading utilities. This is discussed further below in the section on performance opportunities.

One of our goals is to take advantage of the capabilities of existing specialized libraries by generating code specifically for them. This enables the use of various strategies for parallelization, adaptivity, and efficient data structures to achieve high performance. The relatively simple, modular design of these external code generation targets allows adding support for new tools as they are developed or needed.

An 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. It is somewhat limiting in the possible domain geometry as it only uses hexahedral octrees, but since there are various targets available, a different choice can be selected when these limitations are unacceptable.

Another target is a C++ implementation using the aMat library 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-based options. The performance of both of these targets is explored in the Demonstrations section below.

There is also a simpler C++ target that uses MPI and PETSc, which is good for users who want to utilize multiprocessing and have access to the code in its entirety.

Going in a different direction, a Matlab target provides a very simplified option that is great for quick exploration of smaller-scale prototypes.

6.2. Multiple discretizations

In contrast to most of the software packages for solving PDEs, Finch aims to support various discretization methods. Since most methods involve partitioning a domain into small pieces we will refer to as elements, and assembling a global linear system to solve, a lot of the machinery can be reused and modified slightly to utilize a different discretization method. Currently FE and FV are both supported by Finch. The two methods can even be combined in the same solution by specifying different discretizations to be used for different variables. The way these techniques are merged is still a subject of current work.

One challenge is that different methods are designed to handle equations in different forms. As mentioned previously, the FE input is assumed to be in a weak form, while the FV input is assumed to be in a conservation form. Conveniently both of these forms involve a series of implied volume or surface integrals.

6.3. Nonlinear equations

Many problems of interest involve nonlinear PDEs that require more complicated techniques to find numerical solutions. Typically this involves linearizing the equations in some way that involves taking derivatives of the nonlinear terms. Some other software packages, such as Fenics [3], include options to easily do this by accepting nonlinear input equations and using Automatic Differentiation(AD). Another possibility is to use symbolic differentiation and restructure the equations. In either case, an iterative method will also need to be employed to find the solution.

Finch automatically detects nonlinearity while parsing the input equations, and can employ either AD or symbolic differentiation. The symbolic option makes use of SymEngine's tools. The equations are then restructured in the symbolic layer before building the IR by applying a first-order Taylor approximation to the nonlinear terms. The need for an iterative solution method is also signaled to the IR builder.

The input needed from the user is simply to specify the type of differentiation as well as tolerances for the iteration. As an example, consider the following weak form equation with a linear differential term and a second term including a nonlinear function $F(u)$.

$$-(a\nabla u, \nabla v) + (F(u), v) = 0$$

The linearized version of this equation below, derived from a first order Taylor approximation, requires the derivative $F' = \frac{\partial F}{\partial u}$ and the creation of an additional known variable u_0.

$$-(a\nabla u, \nabla v) + (F(u_0), v) + (u - u_0)(F'(u_0), v) = 0$$

Let us consider $F(u) = e^{Cu}$. The Finch script might include the following.

```
nonlinear(maxIters=100, relativeTol=1e-8,
      absoluteTol=1e-8, derivative="symbolic")
weakForm(u, "grad(u)*grad(v) + exp(C*u)*v")
```

Note that this specifies symbolic differentiation, $F' = Ce^{Cu}$. To instead use AD, the derivative parameter would be set to ``AD''. There are currently some limitations on the types of nonlinear equations that are possible, but we are working to expand the range of supported problems. Also, although this example illustrates an FE problem, the process is essentially the same for FV.

7. Performance opportunities

The performance benefit of utilizing external tools has been discussed in the sections above. This section will focus on features of the internal Finch target aimed at achieving good performance. There is a rich ecosystem of Julia packages providing high-performance tools that Finchcan take advantage of.

The internal Julia targets make use of distributed and shared memory parallelism as well as efficient data organization options. Also, when assembling and solving linear systems, the user can select a variety of tools beyond the defaults provided by Julia's LinearAlgebra package.

The simplest technique to take advantage of is multithreading. Finch automatically detects how many threads are available to the Julia instance and uses the native Julia package `Threads` to use 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. This does require a little more care in the part of the user when modifying generated code. Although one may be used to having direct access to solutions and variable values, when multiprocessing these resources will be distributed and may require extra communication to retrieve.

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. However, more complex sets of indexed equations may benefit from different partitioning schemes that partition among variable indices rather than, or in addition to, the mesh. This type of index partitioning was used to solve the phonon Boltzmann transport equation in [30].

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. This extra 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.

7.1. 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 FinchFor 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 simple cache simulator. CacheSim.jl [31], which is our Julia port of Pycachesim [32], was chosen for this because it is light-weight and is written natively in Julia allowing direct utilization by Finch. It can 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. At present, the cache simulator is used to inform the user of cache performance impacts to aid with tuning, but future work involves automatic tuning performed by Finch.

8. 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 [10,33]. Also note that these performance measurements are intended as a way to compare characteristics of different generation targets. Optimization of internal targets is ongoing and is not yet competitive with other mature DSLs, so at this time they are more suitable for small to medium scale problems.

The first two examples use the finite element method with the following configuration input. Note that many of these are default values that are not explicitly set in the input files.

```
floatDataType(Float64) # double precision
functionSpace(space=LEGENDRE, order=1)
nodeType(LOBATTO) # GLL nodes,
interpolated for Gaussian integration
```

The third example uses the finite volume method with the following configuration input.

```
finiteVolumeOrder(1) # linear flux reconstruction
timeStepper(EULER_EXPLICIT)
```

Other input parameters that are specific to the example are provided below. The full input code for these examples, as well as many others, can be found in the code repository [28]. The generated code for certain configurations are also provided.

8.1. Steady-state advection–diffusion–reaction equation

The following Eq. (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 Fig. 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. Fig. 2(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 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. Fig. 2(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.

8.2. Bratu equation (a nonlinear Poisson equation)

The Liouville–Bratu–Gelfand equation, or commonly Bratu Eq. (3), is a nonlinear equation that is relatively simple to analyze, yet exhibits interesting behavior. We use it here to demonstrate the use of Finch's nonlinear capabilities as well as the convenience of indexed entities.

$$\Delta^2 u + Ce^u = 0 \tag{3}$$

$$u(0) = u(1) = 0\ ,\ C \in [0, 3.5]$$

The interesting behavior of this problem is best seen by examining the solution for varying values of C and also using different initial guesses. The solution can converge to one of two distinct branches for a given value of C. Typically this would require rerunning the computation for each scenario separately, but using Finch's indexed variables and coefficients allows the problem to be easily defined for the whole set of configurations and the computation can be run in parallel.

We will define an indexed coefficient for C with 20 different values between 0 and 3.5 placed in the array `cvals`.

```
coefficient("C", cvals, type=VAR_ARRAY)
```

Since we want both upper and lower branch solutions for each C, we will create two indexed variables, u and w. They could be combined into a single variable indexed over both upper and lower values, but for simplicity we separate them here.

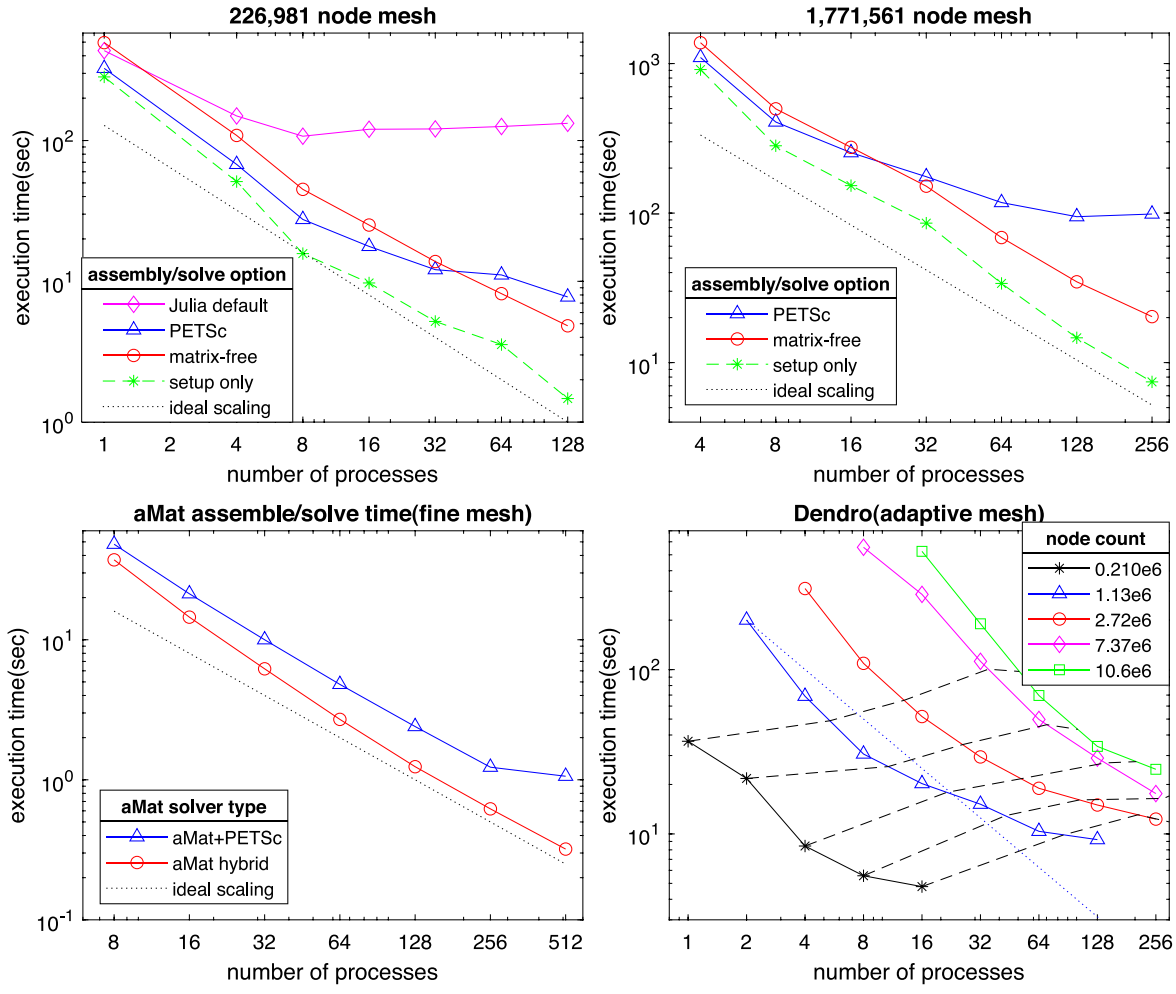


Fig. 2. *Top row:* Internal target execution time on a coarse(*top left*) and fine mesh(*top right*) with different options. "Setup only" excludes assembly/solve time. *Bottom left:* aMat's time on the fine mesh using PETSc and hybrid methods. Only assembly and solve time is included as setup is done 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.

```
ind = index("ind", range = [1,20])
u = variable("u", type=VAR_ARRAY, index=ind)
w = variable("w", type=VAR_ARRAY, index=ind)
```

Although Finch will automatically detect the nonlinearity and restructure the computation to linearize the equations, we still need to specify details for the iteration and the type of derivatives to use.

```
nonlinear(maxIters=100, relativeTol=1e-8,
absoluteTol=1e-8,
      derivative="symbolic")
```

After setting up the boundary conditions and initial guesses, the weak form expressions provided to Finch are

```
weakForm(u, "-dot(grad(u[ind]), grad(v)) + C[ind]
* exp(u[ind]) * v")
weakForm(w, "-dot(grad(w[ind]), grad(v)) + C[ind]
* exp(w[ind]) * v")
```

Solving this set of equations produces the result shown in Fig. 3. The characteristic behavior of the Bratu equation is correctly reproduced.

8.3. Phonon Boltzmann transport equation

The phonon Boltzmann Transport Equation(BTE) is used to model heat conduction in sub-micron scale semiconductor material [34], such as silicon. We use it here to demonstrate FV discretization and using indexed variables to explore different parallel strategies. When formulated in terms of phonon intensity, I, the BTE can be written as the following conservation equation:

$$\frac{\partial I}{\partial t} = \frac{I_0 - I}{\tau} - v_g \cdot \nabla I \tag{4}$$

with group velocity v_g, scattering time scale τ and equilibrium intensity I_0. The intensity also depends on wavevector direction and frequency. Using the FV method with control volume V and discretizing the directions and bands so that we are solving for $I_{d,b}$ for direction d and frequency band b, results in the following equation:

$$\int_V \frac{\partial I_{d,b}}{\partial t} dV = \int_V \frac{I_{0,b} - I_{d,b}}{\tau_b} dV - |v_g|_b \int_{\partial V} I_{d,b} \mathbf{s}_d \cdot \mathbf{n} dA \tag{5}$$

We have adopted the formulation and relation to temperature used in [35,36]. Please refer to those for more detailed descriptions of the model.

To input this problem in Finch, first the indices b for band and d for direction are defined, along with the indexed coefficients. Then the conservation form is entered as

```
conservationForm(I, "(Io[b] - I[d,b]) / beta[b] +
  surface(vg[b] * upwind([Sx[d];Sy[d]], I[d,b]))")
```

Note that an upwind approximation is used for the flux with direction vector `S = [Sx; Sy]`.

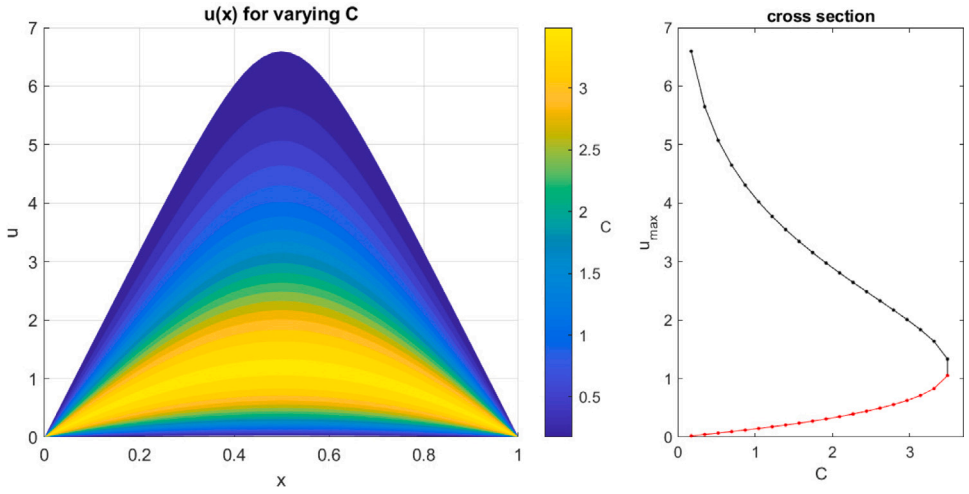


Fig. 3. *Left:* $u(x)$ for different values of $C \in [0, 3.5]$. Note the upper and lower regions of similar color representing the upper and lower branches. The branch depends on the initial guess for u. *Right:* A cross section through the central maximum showing the upper (black) and lower (red) branches for varying C.

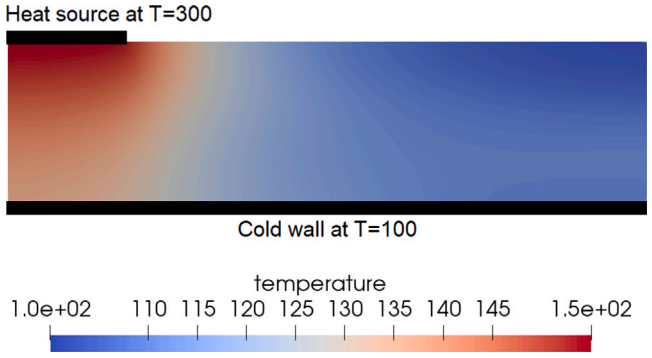


Fig. 4. Temperature in a 4 μm by 1 μm thin film after 5 ns. The hot and cold wall conditions are illustrated.

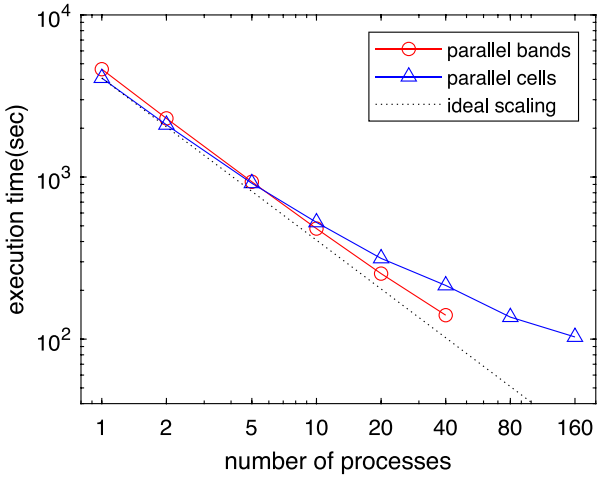


Fig. 5. Scaling for the BTE comparing band-based and cell-based parallel strategies. Note that 40 bands were used, which limits the band-based strategy to 40 processes.

We consider a 2-dimensional domain with one long wall held at a low temperature and a small portion of the opposing wall held at a high temperature. The remaining boundaries are assigned symmetry conditions and the initial temperature everywhere is equal to the cold wall. Fig. 4 illustrates the scenario and shows the temperature distribution after 5 ns.

Due to indirect coupling of frequency bands this is particularly well suited to parallelization of the band index. Recall that Finch allows a re-structuring of the assembly loops when using indexed variables with the command

```
assemblyLoops(I,[band,''elements'',direction])
```

where band and direction refer to their indices. Both band-based and cell-based strategies were compared and the scaling results are shown in Fig. 5. These computations were performed on the Notchpeak cluster at the University of Utah's Center for High Performance Computing using two-socket Intel XeonSP Cascadelake nodes with 40 cores each and 192 GB of memory. The discretization includes 16 directions and 40 frequency bands on a 60 × 15 uniform mesh of rectangular cells.

9. Conclusion

This paper presents Finch, a DSL and code generation framework for PDEs. One of the key features is the ability to generate code for various targets including an internal Julia target and several external targets. 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 aim to support different discretization methods, currently including finite element and finite volume techniques, further expands the range of applications for which it is well suited. We have demonstrated and compared the performance capability of several code generation targets and configurations.

Future work on Finch will involve (1)expanding the range of problems that can be solved, (2)developing new code generation targets as well as improving efficiency of existing ones, and (3)working with domain scientists to solve challenging and practical problems on a large scale. For the first point we are implementing robust methods for nonlinear problems and investigating the addition of new discretization methods. For the second point we are particularly interested in better utilizing different hardware resources such as GPUs and high performance architectures. This will involve developing targets that employ libraries that are both well suited to the problem type and hardware resources.

CRediT authorship contribution statement

Eric Heisler: Conceptualization, Software, Methodology, Investigation, Writing – original draft, Visualization. **Aadesh Deshmukh:** Software, Methodology. **Sandip Mazumder:** Conceptualization, Methodology, Writing – original draft. **Ponnuswamy Sadayappan:** Conceptualization, Methodology. **Hari Sundar:** Conceptualization, Methodology, Resources, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to the publicly available code repository is included in the paper.

Acknowledgments

This work was funded by National Science Foundation, United States grants 1808652, 2004236 and 2008772. The computing resources on Frontera were through an allocation by the Texas Advanced Computing Center PHY20033.

References

[1] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. Kirby, S. Sherwin, Nektar++: An open-source spectral/hp element framework, Comput. Phys. Comm. 192 (2015) 205–219, http://dx.doi.org/10.1016/j.cpc.2015.02.008.

[2] D. Arndt, W. Bangerth, B. Blais, T.C. Clevenger, M. Fehling, A.V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, R. Rastak, I. Thomas, B. Turcksin, Z. Wang, D. Wells, The `deal.II` library, version 9.2, J. Numer. Math. 28 (3) (2020) 131–146, http://dx.doi.org/10.1515/jnma-2020-0043, URL https://dealii.org/deal92-preprint.pdf.

[3] M.S. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, G.N. Wells, The FEniCS project version 1.5, Arch. Numer. Softw. 3 (100) (2015) 9–23, http://dx.doi.org/10.11588/ans.2015.100.20553.

[4] F. Rathgeber, D.A. Ham, L. Mitchell, M. Lange, F. Luporini, A.T.T. Mcrae, G.-T. Bercea, G.R. Markall, P.H.J. Kelly, Firedrake: automating the finite element method by composing abstractions, ACM Trans. Math. Software 43 (3) (2016) 1–27, http://dx.doi.org/10.1145/2998441.

[5] T.O. Foundation, OpenFOAM, 2022, URL https://openfoam.org.

[6] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P.A. Witte, F.J. Herrmann, P. Velesko, G.J. Gorman, Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration, Geosci. Model Dev. 12 (3) (2019) 1165–1187, http://dx.doi.org/10.5194/gmd-12-1165-2019.

[7] C. Rackauckas, Q. Nie, Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia, J. Open Res. Softw. 5 (1) (2017) 15.

[8] M. Fernando, H. Sundar, Dendro home page, 2020, URL https://Octree.Org.

[9] H. Sundar, R. Sampath, G. Biros, Bottom-Up construction and 2:1 balance refinement of linear octrees in parallel, SIAM J. Sci. Comput. 30 (5) (2008) 2675–2708.

[10] M. Fernando, D. Neilsen, H. Sundar, A scalable framework for Adaptive Computational General Relativity on Heterogeneous Clusters, in: Proceedings of the ACM International Conference on Supercomputing, ICS '19, ACM Press, New York, NY, 2019, pp. 1–12, http://dx.doi.org/10.1145/3330345.3330346.

[11] H.D. Tran, M. Fernando, K. Saurabh, B. Ganapathysubramanian, R.M. Kirby, H. Sundar, A scalable adaptive-matrix SPMV for heterogeneous architectures, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IPDPS '22, 2022, pp. 13–24, http://dx.doi.org/10.1109/IPDPS53621.2022.00011.

[12] JuliaLang.org, Julia benchmarks, 2021, URL https://julialang.org/benchmarks.

[13] E. Heisler, A. Deshmukh, H. Sundar, Finch: Domain specific language and code generation for finite element and finite volume in julia, in: D. Groen, C. de Mulatier, M. Paszynski, V.V. Krzhizhanovskaya, J.J. Dongarra, P.M.A. Sloot (Eds.), Computational Science – ICCS 2022, Springer International Publishing, Cham, 2022, pp. 118–132.

[14] M.S. Alnæs, A. Logg, K.B. Ølgaard, M.E. Rognes, G.N. Wells, Unified form language: A domain-specific language for weak formulations of partial differential equations, ACM Trans. Math. Software 40 (2) (2014) http://dx.doi.org/10.1145/2566630.

[15] F. Hecht, New development in FreeFem++, J. Numer. Math. 20 (3–4) (2012) 251–265, URL https://freefem.org/.

[16] J. Xie, K. Ehmann, J. Cao, MetaFEM: A generic FEM solver by meta-expressions, Comput. Methods Appl. Mech. Engrg. 394 (2022) 114907, http://dx.doi.org/10.1016/j.cma.2022.114907, URL https://www.sciencedirect.com/science/article/pii/S004578252200189X.

[17] S. Macià, P.J. Martínez-Ferrer, S. Mateo, V. Beltran, E. Ayguadé, Assembling a high-productivity DSL for computational fluid dynamics, in: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '19, ACM Press, New York, NY, 2019, pp. 1–11, http://dx.doi.org/10.1145/3324989.3325721.

[18] Dune, Dune, 2022, URL https://www.dune-project.org.

[19] C. Uphoff, M. Bader, Yet another tensor toolbox for discontinuous Galerkin methods and other applications, ACM Trans. Math. Softw. 46 (4) (2020) 1–40, http://dx.doi.org/10.1145/3406835.

[20] A.T.T. McRae, G.-T. Bercea, L. Mitchell, D.A. Ham, C.J. Cotter, Automated generation and symbolic manipulation of tensor product finite elements, SIAM J. Sci. Comput. 38 (5) (2016) 25–47, http://dx.doi.org/10.1137/15M1021167.

[21] M. Homolya, R.C. Kirby, D.A. Ham, Exposing and exploiting structure: optimal code generation for high-order finite element methods, 2017, URL https://arxiv.org/abs/1711.02473.

[22] R. Dorozhinskii, M. Bader, 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, in: HPC Asia 2021, ACM Press, New York, NY, 2021, pp. 69–82, http://dx.doi.org/10.1145/3432261.3436753.

[23] D. Kempf, R. Heß, S. Müthing, P. Bastian, Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures, ACM Trans. Math. Softw. 47 (1) (2020) 1–31, http://dx.doi.org/10.1145/3424144.

[24] R.C. Kirby, A. Logg, A compiler for variational forms, ACM Trans. Math. Software 32 (3) (2006) 417–444, http://dx.doi.org/10.1145/1163641.1163644.

[25] A. Logg, G.N. Wells, DOLFIN: Automated finite element computing, ACM Trans. Math. Software 37 (2) (2010) http://dx.doi.org/10.1145/1731022.1731030.

[26] D.A.D. Pietro, J.-M. Gratien, F. Häberlein, A. Michel, C. Prud'homme, 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, POOSC '09, ACM Press, New York, NY, 2009, pp. 1–12, http://dx.doi.org/10.1145/1595655.1595658.

[27] E. Heisler, A. Deshmukh, H. Sundar, Finch documentation, 2023, URL https://paralab.github.io/Finch/dev/.

[28] E. Heisler, A. Deshmukh, H. Sundar, Finch code repository, 2023, URL https://github.com/paralab/Finch.

[29] J.S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, Springer Verlag, 2008.

[30] E. Heisler, S. Saurav, A. Deshmukh, S. Mazumder, P. Sadayappan, H. Sundar, A Domain Specific Language Applied to Phonon Boltzmann Transport for Heat Conduction, ASME International Mechanical Engineering Congress and Exposition, Volume 8: Fluids Engineering; Heat Transfer and Thermal Engineering, 2022, http://dx.doi.org/10.1115/IMECE2022-95034.

[31] E. Heisler, CacheSim.jl code repository, 2023, URL https://github.com/paralab/CacheSim.jl.

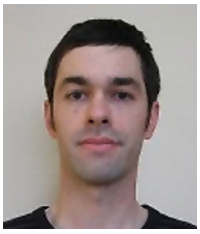
[32] J. Hammer, pycachesim: Python Cache Hierarchy Simulator, 2001, URL https://github.com/RRZE-HPC/pycachesim.

[33] M. Fernando, D. Neilsen, H. Lim, E. Hirschmann, H. Sundar, Massively parallel simulations of binary black hole intermediate-mass-ratio inspirals, SIAM J. Sci. Comput. 42 (2) (2019) 97–138, http://dx.doi.org/10.1137/18M1196972.

[34] A. Majumdar, C. Tien, F. Gemer, 'Microscale energy transport in solids, Microsc. Energy Transp. (1997) 3–93.

[35] S.A. Ali, G. Kollu, S. Mazumder, P. Sadayappan, A. Mittal, Large-scale parallel computation of the phonon Boltzmann Transport Equation, Int. J. Therm. Sci. 86 (2014) 341–351, http://dx.doi.org/10.1016/j.ijthermalsci.2014.07.019, URL https://www.sciencedirect.com/science/article/pii/S1290072914002233.

[36] S. Mazumder, Boltzmann transport equation based modeling of phonon heat conduction: progress and challenges, Ann. Rev. Heat Transfer 24 (2022).



Eric Heisler is a Ph.D. student at the University of Utah. He is developing tools for scientific computing including Finch.

Aadesh Deshmukh is a Master's student at the University of Utah. He is developing efficient software in Julia including Finch.

Sandip Mazumder is a Professor of Mechanical and Aerospace Engineering at The Ohio State University. His research explores computational fluid dynamics, heat transfer, and other nanoscale transport phenomena.

Ponnuswamy Sadayappan is a Professor in the School of Computing at the University of Utah. He is researching high-performance computing techniques for tensor computations.

Hari Sundar is an Associate Professor in the School of Computing at the University of Utah. He is researching high-performance, parallel computing techniques for scientific computing.