Wasatch: an Architecture-Proof Multiphysics Development Environment using a Domain Specific Language and Graph Theory

Tony Saad[a,1,*], James C. Sutherland[a,2]

[a]*Institute for Clean and Secure Energy, Department of Chemical Engineering. University of Utah, Salt Lake City, UT 84112, USA*

Abstract

To address the coding and software challenges of modern hybrid architectures, we propose an approach to multiphysics code development for high-performance computing. This approach is based on using a Domain Specific Language (DSL) in tandem with a directed acyclic graph (DAG) representation of the problem to be solved that allows runtime algorithm generation. When coupled with a large-scale parallel framework, the result is a portable development framework capable of executing on hybrid platforms and handling the challenges of multiphysics applications. We share our experience developing a code in such an environment - an effort that spans an interdisciplinary team of engineers and computer scientists.

Keywords: Domain specific language, Computational physics, Graph theory

1. Introduction

If one thing can be said about the recent development in computing hardware it is volatility. The changing landscape of hardware (multicore, GPU, *etc.*) poses a major challenge for developers of high-performance scientific computing (HPC) applications. Additionally, the problems being addressed by HPC are becoming increasingly complex, frequently characterized by large systems of coupled Partial Differential Equations (PDEs) with many different modeling options that each introduce additional coupling into the system. Such demands to handle multiphysics complexity add another layer of difficulty for both application developers and framework architects.

Our goal is to sustain active development and conduct fundamental and applied research amidst such a volatile landscape. We perceive the problem as having three key challenges:

- hardware complexity: characterized by writing code for new hardware and for different programming models (*e.g.* threads),

*Corresponding author.

Email addresses: tony.saad@chemeng.utah.edu (Tony Saad), james.sutherland@chemeng.utah.edu (James C. Sutherland)

URL: http://www.tonysaad.net (Tony Saad)

[1]Senior Computational Scientist.

[2]Associate Professor of Chemical Engineering.

- programming complexity: characterized by writing code to represent discrete mathematical operators and stencil calculations,
- multiphysics complexity: characterized by writing code that represents complex physical phenomena.

The goal is then to develop a computational framework that allows application developers to

- write architecture-agnostic code,
- write code that mimics the mathematical form it represents,
- easily address multiphysics complexity and its web of nontrivial data dependencies.

In what follows, we review the software environment that allows us to address the aforementioned challenges.

2. Addressing Hardware and Programming Complexity: Nebo

Hardware complexity is the challenge of developing code for many computing architectures such as CPUs and GPUs as well as different programming models such as Threads. To address this challenge we considered the concept of a Domain Specific Language (DSL) and developed an in-house DSL called Nebo [1].

Nebo is an embedded domain specific language (EDSL) designed to aid in the solution of partial differential equations (PDEs) on structured, uniform grids. Because Nebo is embedded in C++, it does not require two-phase compilation; rather, it uses template metaprogramming to allow the C++ compiler to transform the user-specified code into code that effectively targets CPU (including multicore) and GPU backends.

Nebo provides useful tools for defining discrete mathematical operators such as gradients, divergence, interpolants, filters, and boundary condition masks, and can be easily extended to support various discretization schemes (*e.g.* , different order of accuracy).

One of the many advantages of an EDSL is that it allows developers to write a single code but execute on multiple backends, such as CPUs and GPUs, as well as other programming models such as threads - all supported by Nebo.

Figure 1 shows the performance of assembling a generic scalar transport equation using Nebo compared to two other major codes at the University of Utah where untuned C++ nested loops are used. A speedup of up to `10x` is achieved for a grid size of 128^3. The comparisons were conducted on the same architecture (`2 x Intel Xeon 6-Core CPU E5-2620 @ 2.00GHz` with `15 MB L3 Cache`) using a single core and a single thread.

The threading performance of Nebo is shown in Fig. 2 where a standard scalar transport is assembled using various memory sizes and across a wide range of threads. A speedup of up to `12x` is achieved for the largest block size of 2^{18} bytes on 12 threads.

Nebo is currently being used in two major application codes at the University of Utah: Arches and Wasatch. Wasatch will be discussed at length in §5.

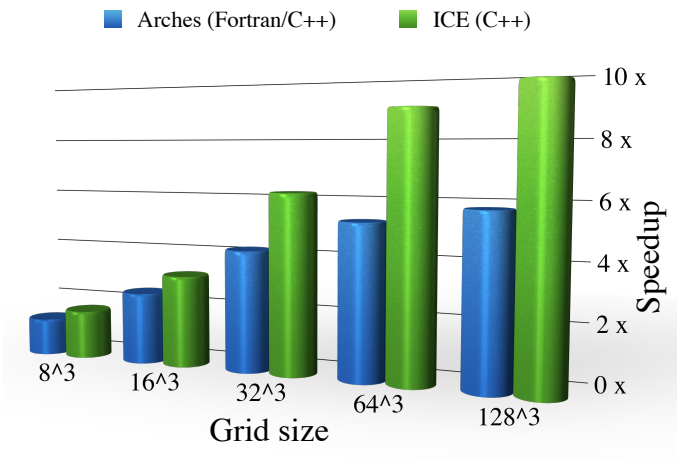


Figure 1: Nebo speedup vs untuned C++ nested loops for assembling a generic scalar equation. Tests were conducted on a single process for grid sizes ranging from 32^3 to 128^3 points.

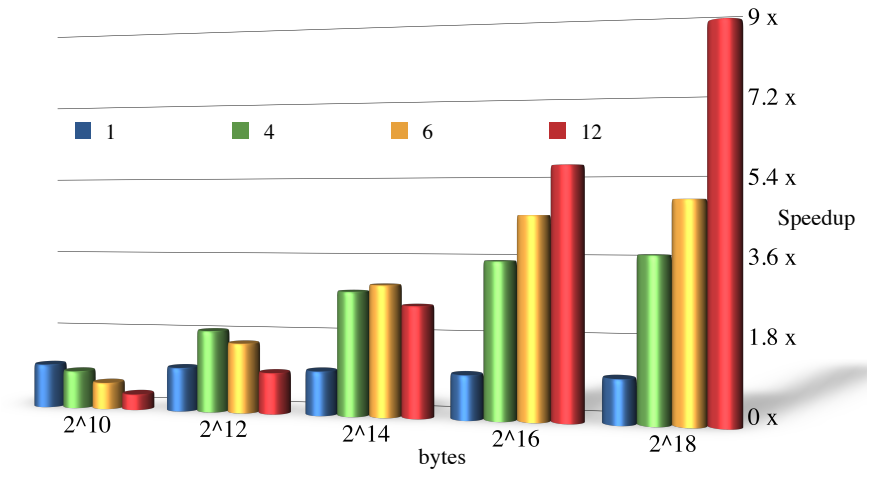


Figure 2: Nebo thread speedup for different memoy block sizes in bytes.

In the grand scheme of scalable computing, Nebo provides on-node fine-grained data parallelism at the grid loop level. It can be used within task-driven execution (discussed in §3) and within distributed-parallel frameworks (discussed in §4).

In addition to being platform-agnostic, Nebo provides a high-level, MATLAB-like interface that allows application developers to express the intended calculation in a form that resembles the mathematical model. The basic Nebo syntax consists of an overloaded bit shift operator separating a left-hand-side (LHS) and a right-hand-side (RHS) expression. The requirement for a Nebo assignment to compile is that the LHS and RHS expressions are of the same type. For example, if `a`, `b`, and `c` are all fields of the same type (*e.g.* , cell centered variables in a finite volume grid), then

```
a <<= b + c
```

computes the sum of `b` and `c`. Nebo also supports all the basic C++ mathematical functions such as sine and cosine along with all fundamental algebraic operations (`+`, `-`, `*`, and `/`). Nebo supports absolute values as well as inline conditional expressions. This advanced Nebo feature provides a powerful tool to assigning boundary conditions for example. A conditional statement simply looks like

```
a <<= cond( cond1, result1)
          ( cond2, result2)
          ...
          ( default );
```

where `cond1`, `cond2`, `result1`, `result2` ... `default` are all arbitrary Nebo expressions.

In addition, Nebo provides support for defining spatial fields on structured grids along with ghost cell and boundary mask information. Probably the most important feature of Nebo is type safety. Operations that result in inconsistent types will not compile. Nebo natively supports 17 field types which consist of four volumetric types corresponding to cell centered and x-, y-, and z-staggered fields as well as particle fields. Each volumetric type requires three face field types, namely, the x-, y-, and z-surfaces. Additional documentation on Nebo can be found at `https://software.crsim.utah.edu/software/`. The inner workings of Nebo have been discussed in [1].

2.1. Stencil Operations

One of the many pitfalls of programming a numerical method for partial differential equations is the coding of discrete mathematical operators (operators hereafter). These typically consist of gradients, divergence, filters, flux limiters, *etc*. A standard implementation of a discrete operator is accomplished via a triply nested `ijk`-loop with appropriate logic to get the fields to properly align and produce a result of the appropriate type and location on a grid. This is often exacerbated by the numerical scheme used and the presence of ghost cells. For example, finite volume approximations typically store fluxes at volume faces and scalars at cell centers. The list goes on, and if one is not cautious, it is easy to get caught up accessing corrupt memory and producing inconsistent, incorrect calculations. These challenges make up what we refer to as programming complexity.

Nebo provides tools for representing discrete mathematical operators such as gradient, divergence, and interpolant to list a few [2]. Operators are objects that consume one type of field

and produce another, depending on the stencil and type of discretization used. For example, an x-gradient operator typically consumes a cell-centered volumetric type and produces a field that lives on the x-surface of the cell-centered volumes. This can be written using Nebo syntax as

```
result <<= gradX(phi);
```

where `phi` is a cell centered field, `gradX` is a pointer to the gradient operator, and `result` is an x-surface field.

Operators can be chained (inlined) as long as they produce the correct field type. For example, the net contribution of the diffusive flux of a cell centered scalar is

```
result <<= divX( interpX(k) * gradX(phi) );
```

where, in this case, `result` is a cell centered field. Here, `k` is the diffusivity, `interpX` is an operator that interpolates `k` to an x-surface location, and `divX` is a divergence operator. This is allowed here because the product `interpX(k) * gradX(phi)` produces and x-surface field while `divX` consumes an x-surface field and produces a cell-centered one.

In addition, Nebo can handle ghost cells automatically. Once a grid variable is declared and allocated with an appropriate number of ghost cells, all subsequent operations using Nebo assignments will automatically handle ghost cells. This includes annotating fields to include the number of valid ghost cells in each direction, since ghost cells can be invalidated by application of stencil operators. Note that ghost cell exchanges must happen outside of Nebo. As will be shown in §4, this exchange takes place in Uintah.

For a list of supported fields, operators, and other details, the reader is referred to the Nebo website located at: https://software.crsim.utah.edu/software. Note that Nebo is distributed under a library called SpatialOps.

3. Addressing Multiphysics Complexity: ExprLib

The traditional approach to designing computational physics software relies almost entirely on specific algorithms and is usually tailored to meet the requirements of the application at hand. At the outset, the code is executed in a systematic order that is specified *a priori*. This code sequence is determined by the model specifics that are being used. Choosing a different model requires an entirely different series of steps that often necessitate modification of the code. Codes that are based on this model become complex and rigid when modified. This is what we refer to as multiphysics complexity.

Multiphysics complexity is caused by a focus on the algorithm or the flow of data. This flow of data represents a high level process and requires particular specification of task execution. To reduce code complexity and rigidity, Notz *et al.* [3] introduced the concept of automated algorithm generation for multiphysics simulations. Their software model is centered on exposing and utilizing low level data dependencies instead of high level algorithmic data flow. The outcome constitutes a shift in focus from the algorithm (which implies depencency among data) to explicitly exposing the dependencies among data (which implies an algorithmic ordering).

Automatic algorithm generation is accomplished by first decomposing a differential equation into basic expressions (*e.g.* convection or diffusion). Subsequently, these expressions are mapped

as nodes in a directed acyclic graph (DAG) that exposes the network of data requirements. In the last step, the graph is analyzed using graph theory to extract an appropriate solution algorithm. The DAG multiphysics approach provides an avenue for automated algorithm generation and execution by exposing task-level parallelism.

Our implementation of the multiphysics DAG approaches consists of a C++ library called ExprLib. ExprLib provides the API for application developers to write physics as fundamental expressions on a graph. ExprLib also provides the interface necessary to execute the graph using explicit time integration schemes. Furthermore, we make extensive use of C++ templates [4, 5] in our design to allow for the portability required by modern computational codes.

3.1. The DAG Multiphysics Approach

In this section, we provide a summary of the directed acyclic graph multiphysics approach of Notz *et al.* . [3] and illustrate its main characteristics in generalizing the concept of solution of partial differential equations. To make our exposition of the theory tangible, we make use of a simple example. Consider the calculation of the diffusive flux of enthalpy in the reactive flow simulation of multicomponent gases. For such a system, the total internal energy equation is

$$\frac{\partial \rho e_0}{\partial t} + \nabla \cdot (\rho e_0 \mathbf{u}) = -\nabla \cdot \mathbf{J}_h - \nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u} + p\mathbf{u}) \,, \tag{1}$$

where ρ is the density, e_0 is the total internal energy per unit mass, $\mathbf{u}$ is the velocity field, p is the pressure, and $\boldsymbol{\tau}$ is the stress tensor. In Eq. (1), the diffusive flux of enthalpy is given as

$$\mathbf{J}_h = -\lambda \nabla T + \sum_{i=1}^{n_s} h_i \mathbf{J}_i. \tag{2}$$

where λ is thermal conductivity of the gas mixture, T is the mixture temperature, h_i and $\mathbf{J}_i$ correspond to the enthalpy and mass diffusion of species i, respectively, and n_s is the total number of species.

A variety of models can be assumed for the thermal conductivity λ and the species mass diffusion $\mathbf{J}_i$, depending the physical situation at hand. The complete specification of a particular model is not essential for the construction of a graph; the dependencies among expressions are all that is needed in order to construct the integration scheme [3]. Details of the functional forms of any of the vertices are hidden. Therefore, every expression can be thought of as a single software component with inputs (its dependencies) and outputs (the field(s) it computes).

For this example, we will consider two models. The first corresponds to constant properties and is given by

$$\begin{cases} \lambda & = \lambda_0 = \text{const.} \\ \mathbf{J}_i & = D_i \nabla Y_i \\ h_i & = h_i(T) \end{cases}, \tag{3}$$

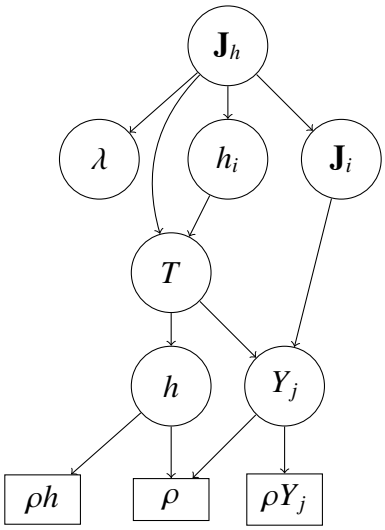


Figure 3: Expression graph for the diffusion model with constant properties model Eq. (3).

The second model depends on the thermodynamic state of the mixture and is specified via

$$\begin{cases} \lambda & = \lambda\left(T, p, Y_i\right) \\ \mathbf{J}_i & = \sum_{j=1}^{n_s} D_{ij}\left(T, p, Y_k\right) \nabla Y_j - D_i^T\left(T, p, Y_k\right) \nabla T \ . \\ h_i & = h_i\left(T\right) \end{cases} \tag{4}$$

Here, Y_i and D_i are the mass fraction and the mixture averaged diffusion coefficient for species i, respectively. D_{ij} represent the multicomponent Fickian diffusion coefficients while the factor D_i^T is associated with thermal diffusion of species i.

The DAG approach for multiphysics simulations is founded on the decomposition of partial differential equations into atomistic entities called expressions that expose data dependencies [3]. An expression is a fundamental entity in a differential equation and is usually associated with a physical mechanism or process. For example, in Eq. (1), the diffusive flux $\mathbf{J}_h$ is represented by an expression. In turn, expressions may depend on other expressions and so on. For instance, using the constant properties model given by Eq. (3), $\mathbf{J}_h$ depends on the thermal conductivity λ, the diffusion coefficient D_i, and the temperature T. This dependency can be easily mapped into a graph as illustrated in Fig. 3.

By representing a differential equation on a DAG, one simplifies the process of solving a PDE by converting it into a series of related tasks that can be analyzed using graph theory. The dependencies exposed by the graph are the only requirement for implementing new models. For example, if one uses the model given by Eq. (4), that model may be easily inserted into the graph with no algorithmic modification to the code. This is illustrated in Fig. 4 where new dependencies have been easily exposed. In this sense, a multiphysics code can be thought of as a collection of physical models that are turned on or off based on dependencies.

Execution is handled by a scheduler that traverses the DAG and manages memory required by

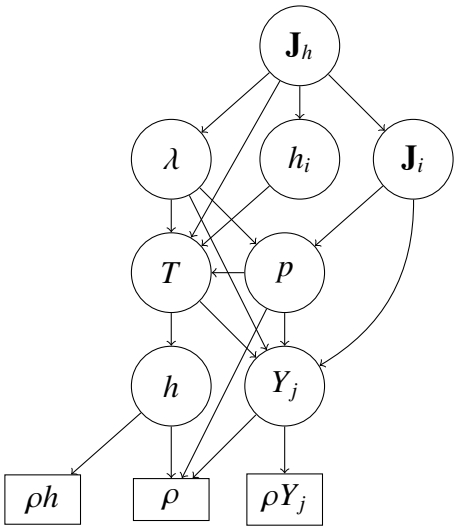


Figure 4: Expression graph for the diffusion model with species dependent properties Eq. (4).

each node as well as execution of each node in a multithreaded environment. Memory management includes asynchronous host-device transfers when accelerators such as Xeon Phi or NVIDIA GPUs. Each node in the graph computes one or more quantities, which other nodes in the graph have read-only access to. A given quantity can be available on multiple devices: host memory, and multiple accelerator card memories, and the graph scheduler ensures that required transfers occur between memory to maintain coherency as necessary.

In general, an expression requires the application developer to expose three essential ingredients: (a) the data copmuted by the expression, (b) the data required by the expression, and (c) a method for updating and manipulating the supplied data.

Once a graph is constructed, a reverse topological sorting algorithm [6] may be used to automatically generate the ordering of task execution. Once the an expression graph is constructed and sorted, an algorithm is defined and the code is executed.

4. Runtime Parallelism: Uintah

So far we have discussed the technologies that were developed to address the three challenges presented at the begining of the paper. Namely, Nebo for hardware and programming complexity and ExprLib for multiphysics complexity. What remains is to put all of these together in a massively parallel runtime environment. For this we use the Uintah computational framework [7].

The Uintah Computational Framework (UCF) is a massively-parallel software infrastructure designed to solve partial differential equations (PDEs) [8, 9] such as those arising in fluid dynamics, reacting flows, and combustion [10]. Uintah provides a set of tools that scientists and engineers can use to develop physics-based utilities (called components) to simulate real-world phenomena. To that end, several components have been developed over the years to leverage the scalability and parallel features of Uintah. These range from tools to simulate material crack propagation to buoyancy-driven flows and fully compressible flows.

Uintah is a task-based parallel environment that provides a variety of HPC services. It provides several schedulers for coarse grained automatic parallelization and inference of communication patterns. Several load balancers are also provided to handle load-balancing at runtime. In addition, Uintah provides parallel input/output (IO) and domain decomposition. One of the strengths of Uintah is its task-based approach.

A Uintah task consists of a callback function with a specification of data dependencies (requires) and outputs (computes). This is akin to the DAG approach discussed in §3. For each depencency, the developer specifies the number of ghost cells required. This in turn helps Uintah handle load-balancing and infer communication patterns efficiently. If a tasks requires ghost cells, then Uintah will trigger the approprirate MPI communication to provide the ghost cell values to the fields that need it.

Uintah tasks are similar to the concepts implemented in ExprLib (§3). However, as will be shown in §5, ExprLib allows one to aggregate several smaller tasks and expose them as a single Uintah task. Not only does this lead to memory optimizations via dynamic allocation and memory pools, it also allows application developers to have control over the granularity of graphs. The latter is left at the discretion of the application developer who has the choice to either expose every single (ExprLib) expression as a separate Uintah task or lump as many expressions as possible into larger Uintah tasks. In this sense, Uintah is responsible for coarse-grained parallelism [11] while ExprLib is responsible for fine-grained, task decomposition, and data parallelism through Nebo. Additional information about Uintah can be found at `http://www.uintah.utah.edu`.

5. Putting it all Together: Wasatch

Wasatch is a finite volume multiphysics application code that builds on top of Uintah, ExprLib, and Nebo. Wasatch currently supports the following types of physics:

- Incompressible constant density Navier-Stokes solver
- Large Eddy Simulation (LES) with four turbulence models: Constant Smagorinsky, Dynamic Smagorinsky, WALE, and Vreman
- Arbitrarily complex geometries using stairstepping
- Arbitrarily complex boundaries and boundary conditions
- Low-Mach variable density flows
- First, second, and third order Runge-Kutta Strong Stability Preserving temporal integration
- Lagrangian particle transport
- Lagrangian particle boundary conditions (inlets and walls)
- Eulerian representation of particles via the Quadratrue Method of Moments (QMOM) for solving population balance equations

- Modeling solids precipitation in aqueous solutions
- Arbitrarily complex scalar transport - advection, diffusion, reaction, and sources

All of the aforementioned physics were developed using ExprLib and Nebo. For its runtime parallelism, Wasatch uses Uintah's task scheduling via a `TaskInterface` class. The `TaskInterface` acts as a Uintah wrapper of ExprLib graphs. In other words, once an ExprLib graph is generated, it is wrapped as a Uintah callback task. Once a callback is complete (or about to start), Uintah is responsible for updating ghost cell information. Here's how things work in practice:

- Parse input file
- Trigger appropriate transport equations
- Transport equations will trigger appropriate ExprLib root expressions to be constructed
- ExprLib generates multiphysics directed acyclic graph at runtime
- Graph is wrapped as a Uintah task using the `TaskInterface` class
- Uintah task execution starts, which triggers the ExprLib graph execution
- ExprLib graph is executed by visiting every node in the graph and executing Nebo expressions

Uintah is therefore responsible for MPI-level parallelism, IO, and global scheduling. Wasatch is responsible for the communication between ExprLib graphs and Uintah as well as for implementing all the relevant physics. ExprLib and Nebo are responsible for fine-grained, task-level data parallelism. In this formulation, ExprLib aggregates a number of small tasks and exposes them to Uintah as one or more tasks. Data that live *within* the ExprLib graph are generally not accessible to Uintah unless they are marked for IO. On the other hand, data at the *boundaries* of a graph are generally accessible to Uintah and are shared between Uintah, ExprLib, and Nebo. A graphical illustration of the how Wasatch and Uintah operate is shown in Fig. 5.

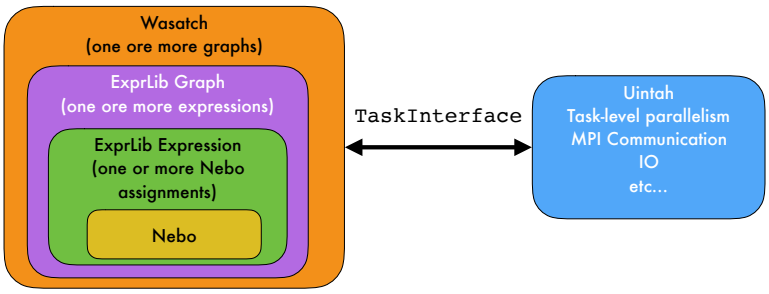


Figure 5: A graphical illustration of the relationship between Nebo, ExprLib, Wasatch, and Uintah. Nebo constitutes the core of every expression. Expressions are used to describe various physics and are then assembled as an ExprLib DAG that represents a PDE. One ore more ExprLib graphs are created by Wasatch and then exposed to Uintah via the `TaskInterface` class.

An example graph of a Wasatch simulation of a constant-density incompressible flow problem is shown in Fig. 6. Each node on the graph represents a different portion of the physics involved in an incompressible flow simulation. The various colors correspond to the ability of a certain node to be executed on GPU or not. While most expressions can be executed on GPUs, the linear solver[3] is not GPU-ready (*i.e.* pressure equation), hence the light gray indicating a CPU-only node.

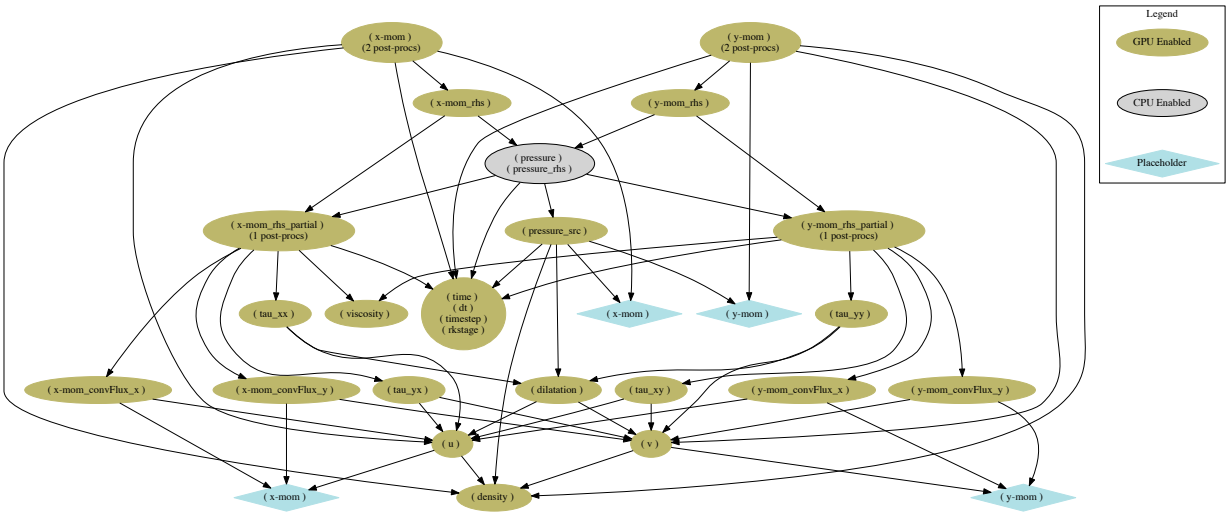


Figure 6: Directed acyclic graph from a Wasatch simulation of an incompressible flow solving the constant-density Navier-Stokes equations.

Next, we evaluate the scalability of Wasatch which has been tested against several applications on CPU and GPU clusters. In what follows, we present (1) CPU, (2) single-node GPU, and (3) GPU-cluster scalability.

5.1. CPU Scalability

Starting with CPU scalability, Fig. 7 shows the weak scaling of Wasatch on the three-dimensional Taylor-Green vortex - a standard, non-trivial fluid dynamics simulation that represents the decay of a vortex into fine turbulent structures. The scalability is tested for a fixed problem size per MPI process across a range of MPI processes up to 256,000 processors. We find that Wasatch accomplishes the best scalability for the largest problem size of 128^3 grid points per processor. Note that the loss of scalability on other problem sizes is due to the overhead of the linear solver which is required to enforce mass conservation in this case.

5.2. GPU Scalability

Next, we test Wasatch's GPU capabilities on a suite of scalar transport problems ranging from linear to nonlinear fully coupled systems of transport equations. The target equation is a generic

[3]The Hypre [12] linear solver is used in Wasatch. The coefficient matrix for the linear solver is assembled using the `HYPRE_STRUCTMATRIX` interface.

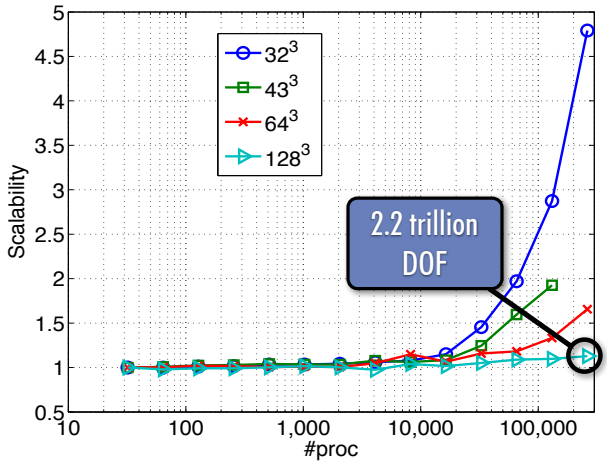


Figure 7: Wasatch CPU weak scaling on Titan for the three-dimensional Taylor-Green vortex using the Hypre linear solver for up to 256,000 processors and various patch sizes ranging from 32^3 to 128^3.

scalar transport equation. This equation describes the temporal and spatial evolution of a quantity, ϕ, subject to a velocity field $\mathbf{u}$, diffusivity Γ, and source term S_i

$$\frac{\partial \phi_i}{\partial t} = -\nabla \cdot \mathbf{u}\phi_i - \nabla \cdot \Gamma \nabla \phi_i + S_i; \quad i = 1, \cdots, N \tag{5}$$

where the subscript i stands for the i^{th} generic scalar and N is the total number of equations solved. One typically solves an arbitrary number of these equations; hence the use of the subscript i.

Coupling between the various equations can be accomplished via the source term, S_i. In an attempt to represent a wide range of physics, we choose three types of source terms

$$S_i = \begin{cases} 0 & \text{No source terms} \\ e^{\phi_i} & \text{Uncoupled source terms} \\ \sum_j e^{\phi_j}; \;\; j = 1, \cdots, N & \text{Coupled source terms} \end{cases} \tag{6}$$

Note that these are meant to mimic the cost of Arrhenius kinetics calculations, and do not represent a physically realistic model for reaction.

5.2.1. On-Node GPU Speedup

In this case, a total number of 30 equations is solved on a single compute node with a single GPU. A total of 10 timesteps are taken. The mean time per timestep is then used to compare the timings on the GPU and a single processor CPU simultion. The GPU speedup is reported in Fig. 8. A speedup of up to 50x is achieved on the most complex problems as well as the largest patch sizes. The results are consistent with the nature of the source terms used where maximum speedup is achieved for the most pointwise-oriented calculations.

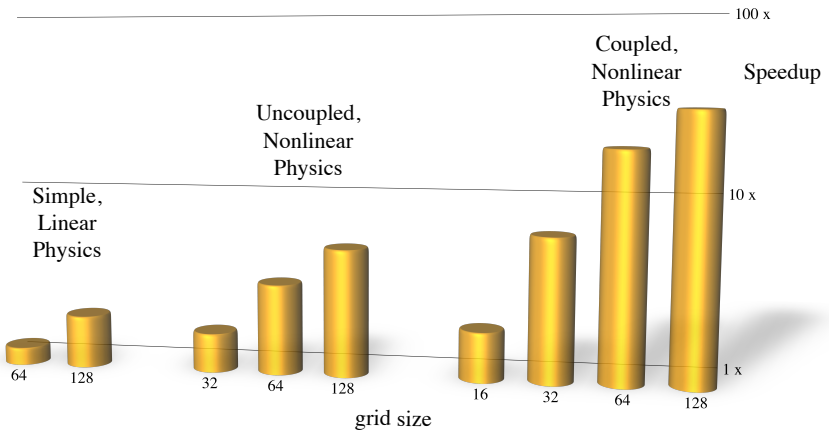


Figure 8: Wasatch On-Node GPU speedup for a variety of scalar transport problems ranging from linear uncoupled systems of PDEs to fully coupled.

5.2.2. GPU-Cluster Speedup

Similar to the on-node GPU speedup, the same problem was repeated on the Titan supercomputer for up to 12,800 GPUs. Results from weak scaling and speedup are shown for the various physics and number of equations, covering 10 and 20 equations, respectively. Results are shown in figures 9 through 11.

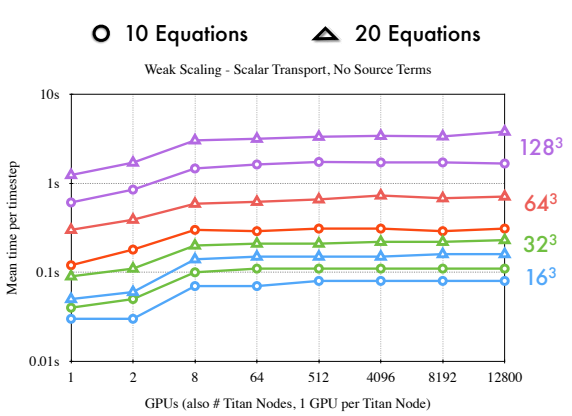
Fig. 9 shows the weak scaling and speedup results when solving the system of equations without any source terms. While decent weak scaling is achieved, acceptable speedups are visible for cases with more than 64^3 in resolution. Similar behavior is observed when solving the system of equations with a simple uncoupled source term shown in Fig. 10. Finally, for the fully coupled case shown in Fig. 11, speedups greater than one are visible at patch resolutions of 32^3 and upward to more than 20 times speedup for a grid resolution of 128^3 along with 20 equations.

These results are again consistent with the nature of the source terms being used. The cost of MPI communication between GPU nodes is evident in the reduced speedup when compared to the on-node results where a speedup of `50x` was accomplished for the most computationally demanding case.

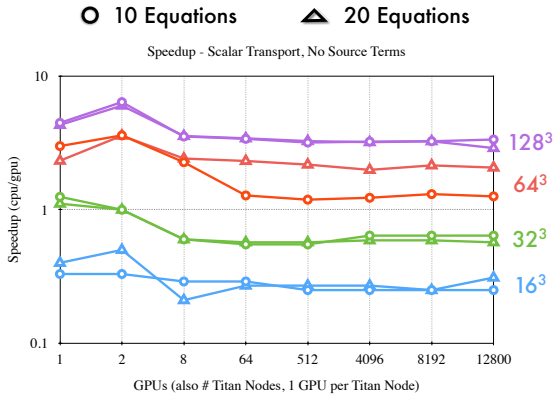
6. Related Work

Many of the ideas presented in this paper can be found in the literature. The concept of a DSL has been adopted by a variety of teams to address the challenges of modern computing architectures. A comparison between Nebo and similar DSLs has been conducted by Earl *et al.* in [1]. In that paper, Nebo is compared to POOMA [13], Pochoir [14], Liszt [15], and PotiMesh [16].

Another DSL that is worth mentioning is Nabla [17]. Nabla translates source files into C, C++, and CUDA code and thus requires two-stage compilation. It provides support for job scheduling that take input data and provide output. Upon compilation, a graph is generated by the nabla compiler for what is called a Nabla component. A composition of several Nabla components

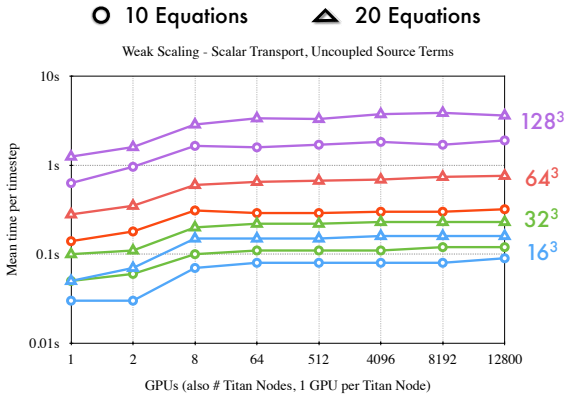


(a) GPU Mean time per timestep

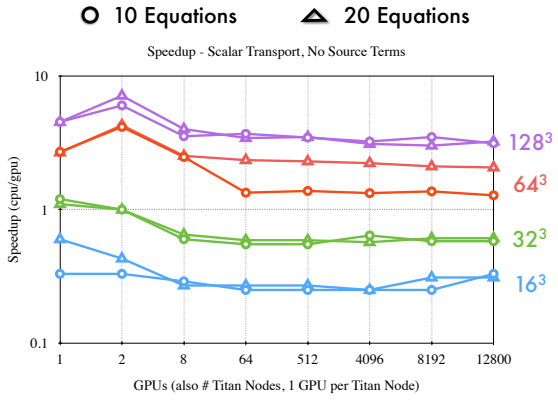


(b) Speedup measured as the ratio of the Mean time per timestep of the GPU over that of the CPU

Figure 9: Mean time per timestep and speedup for 10 (circle) and 20 (triangle) equations without source terms for up to 12,800 GPUs on Titan. Results are shown for different patch sizes ranging from 16^3 to 128^3

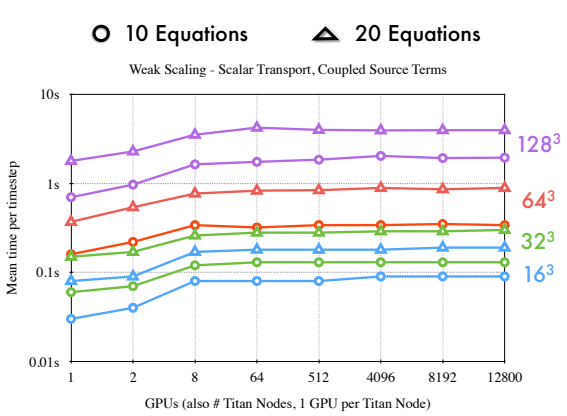


(a) GPU Mean time per timestep

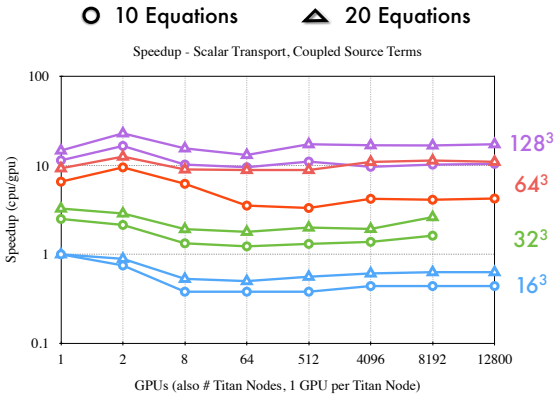


(b) Speedup measured as the ratio of the Mean time per timestep of the GPU over that of the CPU

Figure 10: Mean time per timestep and speedup for 10 (circle) and 20 (triangle) equations with uncoupled source terms for up to 12,800 GPUs on Titan. Results are shown for different patch sizes ranging from 16^3 to 128^3



(a) GPU Mean time per timestep



(b) Speedup measured as the ratio of the Mean time per timestep of the GPU over that of the CPU

Figure 11: Mean time per timestep and speedup for 10 (circle) and 20 (triangle) equations with fully coupled source terms for up to 12,800 GPUs on Titan. Results are shown for different patch sizes ranging from 16^3 to 128^3

produces a multi-physics application. One can think of this approach as a combination of ExprLib and Nebo.

There are also DSL alternatives to tackle the hardware complexity problem. Notable mentions are Kokkos [18, 19] from Sandia National Laboratories and RAJA [20] from Lawrence Livermore National Laboratories. Both of these employ C++ to provide basic portable, parallel constructs such as `foreach`. These are relatively low-level compared to Nebo, which provides a higher-level abstraction.

In terms of the DAG approach, the foundation of ExprLib is based on the work of Notz *et al.* [3, 21]. Notable alternatives to ExprLib have been used and implemented in SIERRA [22, 23] and Aria [24], Charon, and Phalanx as part of the Trilinos project [25].

7. Conclusions

In this paper, we discussed an approach to dealing with the volatile landscape of large-scale hybrid-computing software development. Our approach consisted of tackling three problems encountered by modern application developers, namely, (1) hardware complexity, (2) programming complexity, and (3) multiphysics complexity. We discussed how and Embedded Domain Specific Language (EDSL) such as Nebo can help address hardware complexity by allowing developers to write a single code but execute on multiple platforms. In addition, Nebo addresses programming complexity by providing support for a wide range of discrete stencil operators such as gradients, filters, *etc*. Finally, multiphysics complexity is addressed by using ExprLib, a C++ library capable to representing a time marching algorithm as a directed acyclic graph (DAG). These technologies are all put together to build Wasatch, a finite volume multiphysics code developed at the University of Utah. We discussed the capabilities and scaling properties of both Nebo and Wasatch. Results indicate that near ideal scalability is reached in some cases and very promising speedups

are obtained from the GPU and threaded backends.

Acknowledgments

The authors gratefully acknowledge support from the National Science Foundation PetaApps award 0904631 and Department of Energy award DE-SC0008998.

References

[1] C. Earl, M. Might, A. Bagusetty, J. C. Sutherland, Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations, Journal of Systems and Software (To Appear in 2016). doi:10.1016/j.jss.2016.01.023.

[2] J. C. Sutherland, T. Saad, The Discrete Operator Approach to the Numerical Solution of Partial Differential Equations, in: 20th AIAA Computational Fluid Dynamics Conference, Honolulu, Hawaii, USA, 2011, pp. AIAA–2011–3377.

[3] P. K. Notz, R. P. Pawlowski, J. C. Sutherland, Graph-Based Software Design for Managing Complexity and Enabling Concurrency in Multiphysics PDE Software, ACM Transactions on Mathematical Software 39 (1) (2012) 1–21. doi:10.1145/2382585.2382586.

[4] A. Alexandrescu, Modern C++ design, Vol. 98, Addison-Wesley Reading, MA, 2001.

[5] D. R. Musser, G. J. Derge, A. Saini, C++ Programming with the Standard Template Library, Addison–Wesley, 2001.

[6] A. B. Kahn, Topological sorting of large networks, Communications of the ACM 5 (11) (1962) 558–562. doi:10.1145/368996.369025.

[7] J. Luitjens, M. Berzins, Improving the performance of Uintah: A large-scale adaptive meshing computational framework, in: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–10.

[8] Q. Meng, J. Luitjens, M. Berzins, Dynamic task scheduling for the uintah framework, in: Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on, IEEE, 2010, pp. 1–10.

[9] M. Berzins, J. Schmidt, Q. Meng, A. Humphrey, Past, present and future scalability of the uintah software, in: Proceedings of the Extreme Scaling Workshop, University of Illinois at Urbana-Champaign, 2012, p. 6.

[10] J. Schmidt, M. Berzins, J. Thornock, T. Saad, J. Sutherland, Large scale parallel solution of incompressible flow problems using uintah and hypre, in: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, IEEE, 2013, pp. 458–465.

[11] M. Berzins, Q. Meng, J. Schmidt, J. C. Sutherland, Dag-based software frameworks for pdes, in: Euro-Par 2011: 17th International European Conference on Parallel and Distributed Computing, Springer, 2011, pp. 324–333.

[12] R. Falgout, U. Yang, *hypre*: A library of high performance preconditioners, Computational Science - ICCS 2002 (2002) 632–641.

[13] J. Reynders, The POOMA framework - a templated class library for parallel scientific computing., in: PPSC, 1997.

[14] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, C. E. Leiserson, The pochoir stencil compiler, in: Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, ACM, 2011, pp. 117–128.

[15] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a domain specific language for building portable mesh-based pde solvers, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 9.

[16] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. O. Odersky, K. Olukotun, Composition and reuse with compiled domain-specific languages, in: ECOOP 2013–Object-Oriented Programming, Springer, 2013, pp. 52–78.

[17] J.-S. Camier, Improving performance portability and exascale software productivity with the numerical programming language, in: Proceedings of the 3rd International Conference on Exascale Applications and Software, University of Edinburgh, 2015, pp. 126–131.
[18] H. C. Edwards, D. Sunderland, Kokkos array performance-portable manycore programming model, in: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, 2012, pp. 1–10.
[19] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, D. Beckingsale, A performance evaluation of Kokkos & RAJA using the tealeaf mini-app.
[20] R. Hornung, J. Keasler, The RAJA portability layer: overview and status, Lawrence Livermore National Laboratory, Livermore, USA.
[21] P. K. Notz, R. P. Pawlowski, J. C. Sutherland, Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software, ACM Transactions on Mathematical Software (TOMS) 39 (1) (2012) 1.
[22] H. C. Edwards, Managing complexity in massively parallel, adaptive, multiphysics applications, Engineering with Computers 22 (3-4) (2006) 135–155.
[23] J. R. Stewart, H. C. Edwards, A framework approach for developing parallel adaptive multiphysics applications, Finite Elements in Analysis and Design 40 (12) (2004) 1599–1617.
[24] P. K. Notz, S. R. Subia, M. M. Hopkins, H. K. Moffat, D. R. Noble, Aria 1.5 user manual, Sandia Report 2734 (2007) 2007.
URL http://prod.sandia.gov/techlib/access-control.cgi/2007/072734.pdf
[25] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al., An overview of the Trilinos project, ACM Transactions on Mathematical Software (TOMS) 31 (3) (2005) 397–423.