Wasatch: Addressing Multiphsyics and Hardware Complexity in a High-Performance Computing Environment

Tony Saad
Institute for Clean and Secure Energy
tony.saad@chemeng.utah.edu

James C. Sutherland
University of Utah
james.sutherland@chemeng.utah.edu

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 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 PDEs with many different modeling options that each introduce different nonlinear 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,
- programming complexity,
- multiphysics complexity.

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.

DEHPC '15, October 18–21, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

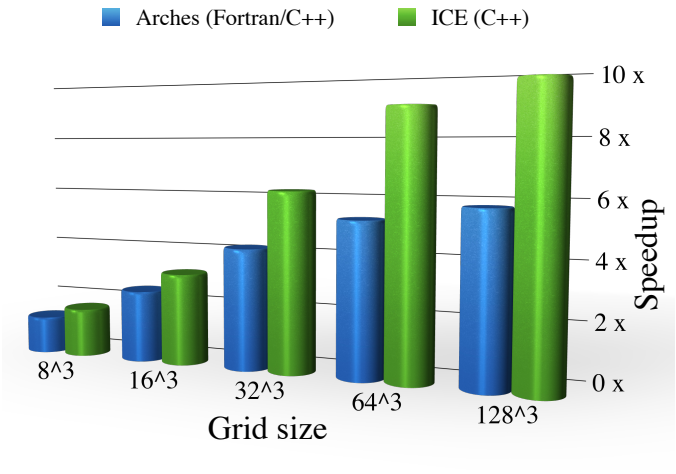

Figure 1. Nebo speedup vs untuned C++ nested loops for assembling a generic scalar equation. Tests were conducted on a single process with for grid sizes ranging from 32^3 to 128^3 points.

2. Addressing Hardware Complexity: Nebo

Nebo is an embedded domain specific language (EDSL) to aid in the solution of partial differential equations (PDEs) on structured, uniform grids. It provides useful 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).

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, threads, and GPUs - all supported by Nebo. In addition, one is able to fine-tune the backends leading to performance gains compared to traditional C++ loops. It was shown that Nebo outperforms standard (untuned) iterator or nested loops. This is shown in Fig. 1 where a generic scalar transport equation is assembled using Nebo and then compared to two other major codes at the Univesrity of Utah where untuned C++ nested loops are used. A speedup of up to 10x is achieved for a grid size of 128^3.

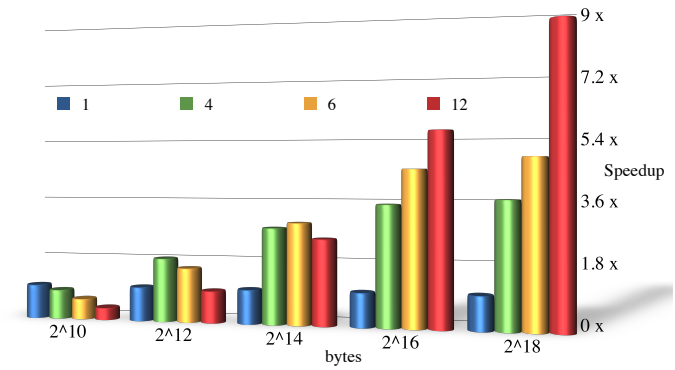

Figure 2. Nebo thread speedup for different memoy block sizes in bytes.

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.

In the grand scheme of scalable computing, Nebo provides fine-grained parallelism at the loop level as opposed to task-level parallelism, typically provided by a parallel framework.

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 overload bit shift operator separating a left-hand-side (LHS) and a right-hand-side (RHS) expressions. The conditions for a Nebo expression 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 supports all the basic C++ mathematical functions such as sine and cosine along wil 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... are all arbitrary Nebo expressions.

Next, we discuss SpatialOps, a C++ library that provides a structured-grid interface to Nebo along with stencil operations to compute derivatives, filters, and other mathematical and numerical operators.

3. Addressing Programming and Multiphysics Complexity: SpatialOps and ExprLib

One of the many pitfalls of programming numerical methods for partial differential equations are the operators. These typically consist of gradients, divergence, filters, flux limiters, etc... A typical implementation consists of 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. 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.

3.1 SpatialOps

SpatialOps stands for *Spatial Operators Library* and provides tools for representing discrete mathematical operators such as gradient, divergence, interpolant to list a few.[**?**] In addition, SpatialOps provides the support to define spatial fields on structured grids along with ghost cell and boundary mask information. It is designed using C++ and is built on top of Nebo and provides the interface for the application developer to access the entire Nebo functionality.

Probably the most important feature of SpatialOps is type safety. Operations that result in inconsistent types will not compile. SpatialOps currently supports 16 field types which consist of four volumetric types corresponding to cell centered and x-, y-, and z-staggered fields. Each volumetric type requires three face field types, namely, the x-, y-, and z-surfaces.

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 SpatialOps 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 diffusive flux of a cell centered scalar is

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

where, in this case, result is a cell centered field. This is allowed here because the div operator consumes an x-surface field and produces a cell-centered one.

For a list of supported fields, operators, and other details, the reader is referred to the SpatialOps website located at: https://software.crsim.utah.edu/jenkins/job/SpatialOps/doxygen/fields-meshes.html

3.2 ExprLib

Numerical simulation is a rapidly growing field in pure and applied sciences. The data delivered by a particular simulation provides fine grained information about the physical processes taking place in that system. This information is useful in verfying existing hypotheses and predicting performance characteristics. In a variety of cases, simulation aids in the discovery of new physical phenomena and also helps in the design of practical models. The ability to accomplish such high fidelity simulations necessitates the accurate and precise solution of nonlinear and coupled partial differential equations (PDEs). Typical high fidelity methods for transport phenomen include Direct Numerical Simulation, Large Eddy Simulation, Lattice Hydrodynamics, and Particle Dynamics.

The use of high fidelity methods requires advanced software design to address the software complexity and nonlinear coupling between governing equations. 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 modifcation of the code. Codes that are based on this model become complex and rigid when modified.

Code rigidity 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. [7] 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). 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 optimum solution algorithm. The DAG multiphysics approach provides an avenue for optimized code execution by uncovering fine grained parallelism.

In this article, we discuss an implementation of the DAG approach for use with highly parallel computations. Our implementation uses the C++ programming language [9] to construct a library capable of analyzing expressions and mapping them as nodes in a DAG. The library also provides the interface necessary to execute the graph using explicit time integration schemes. Furthermore, we make exclusive use of C++ templates [1, 6] in our design to allow for the portability required by modern computational codes.

3.3 The DAG Multiphysics Approach

In this section, we provide a summary of the directed acyclic graph multiphysics appraoch of Notz et al. [7] 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 multicompnent gases. For such a system, the total internal energy equation is

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

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

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

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 [7]. 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 and outputs.

For this example, we will consider two models. The first corresponds to constant properties while the second depends on thermodynamic state of the mixture. These are given by

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

and

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

Here, D_i is the mixture averaged diffusion coefficient for species i while D_{ij} represent the multicomponent Fickian diffusion coefficients. 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 [7]. 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 tree 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 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.

In general, an expression requires three essential ingredients: (a) the data supplied by the expression, (b) the data required by the expression, and (c) a method for updating and manipulating the supplied data. It is therefore essential to attach unique identification strings to each experession. These strings are called *Expression_Tags* and are supposed to be generic, non-specific to a particular implementation implementation.

Once the graph is constructed, a reverse topological sorting algorithm[3] may be used to automatically generate the ordering of task execution. Reverse Topological ordering of a directed acyclic graph is a linear ordering of its nodes such that each node is executed before all nodes to which its inbound nodes. Once the tree of expressions is constructed, an algorithm is defined and the code is executed.

3.4 Software Implementation

Our software implementaiton of the DAG multiphysics approach materializes in a C++ software library called ExprLib

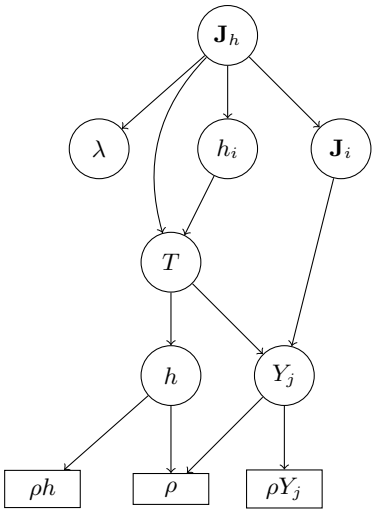

Figure 3. Expression graph for the diffusion model with constant properties model Eq. (3).

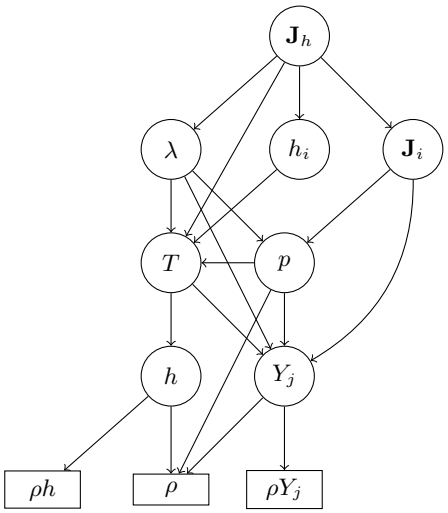

Figure 4. Expression graph for the diffusion model with species dependent properties Eq. (4).

The design of ExprLib aims at addressing three vital software design challenges: (a) facilitate dynamic transition in model form, (b) enable task parallelism, and (c) remove specific concern over flow of information in an algorithm. ExprLib exposes a simple interface to generate algorithms based on graph theory. Once the algorithm is generated, explicit time integration is used to obtain a solution.

The tools provided by ExprLib are centered around three essential components: an expression, a field manager, and a scheduler. The general concept behind using ExprLib is as follows. The user first writes code for expressions using the `Expression` base class and registers the expression with a factory. The user also indicates what quantities are needed. This may be through activating transport equations, etc., which trigger the requirement for the right-hand-side of the PDE to be computed. As the graph is recursively constructed through dependency discovery and analysis, only the dependencies among expressions are pulled from the expression registry and placed in the graph. The scheduler holds a graph and a field manager and controls the execution and storage/memory including required host-device transfers.

```
1  class MyExpression
2    : public Expr::Expression<FieldT>
3  { /* declare private variables such as fields,
       operators, etc. here */
4    MyExpression( /* class-specific arguments -
         typically Expr::Tag objects */ );
5  public:
6    class Builder : public Expr::
         ExpressionBuilder
7    {
8    public:
9      Builder(Expr::Tag result,
10             /* other arguments here */ );
11     Expr::ExpressionBase* build() const;
12
13   private:      /* add arguments here */
14   };
15   ~Test();
16   DECLARE_FIELD(FieldT, phi_)
17   DECLARE_FIELDS(SurfaceFieldT, f0_, f1_) ...
18   void bind_operators( const SpatialOps::
         OperatorDatabase& opDB );
19   void evaluate();
20 };
```

Listing 1. Skeleton of an expression base class.

3.4.1 The Expression Base Class

The `Expression` base class is central to the design of ExprLib. In general, an expression is built in two stages. In the first stage, the expression is registered in the `ExpressionFactory`, but may not be necessarily used in the graph. This occurs when, for example, the input file includes additional expressions that are not used by the governing equations. The second build stage for an expression takes place when it is referred to as a dependency. Therefore, only those expressions that are needed to construct the graph will actually be built.

Every expression is associated with a `Builder` class and must implement four methods. The `Builder` class is another base class in ExprLib and provides the necessary mechanism for the two-stage construction of an `Expression`. The four methods that an expression must implement provide the interface for specifying data dependencies and manipulation. The skeleton of an expression in ExprLib is shown in Listing 1.

3.4.2 Required Methods

There are two methods that an `Expression` must implement. These methods allow programmers to specify the dependencies, fields, operators, and data manipulation for an `Expression`.

- `bind_operators`: This method is also called prior to execution of an expression and allows the binding of operators that may be used by this expression. Operators are retrieved by their type from an `OperatorDatabase` object.
- `evaluate`: This method is used to populate the field that this expression computes. This is, for example, where the RHS of an equation can be calculated using the operators defined in the expression.

4. Runtime Parallelism: Uintah

The Uintah Computational Framework (UCF) [4] is a massively-parallel software infrastructure designed to solve partial differential equations (PDEs) [2, 5] such as those arising in fluid dynamics, reacting flows, and combustion [8]. 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. 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 code that builds on top of all the aforementioned technologies. 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
- Complex precipitation physics
- Arbitrarily complex scalar transport - advection, diffusion, reaction, and sources

All of the aforementioned physics was built on top of ExprLib, SpatialOps, and Nebo. For its runtime parallelism, Wasatch uses Uintah's interface via a TaskInterface class. Here's how things work:

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

An example graph for a Wasatch simulation of a constant-density incompressible flow problem is shown in Fig. 7. Each node on the graph represented 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 is not GPU-ready (i.e. pressure Poisson equation), hence the node corresponding to the pressure is shown in light gray indicating a CPU-only expression.

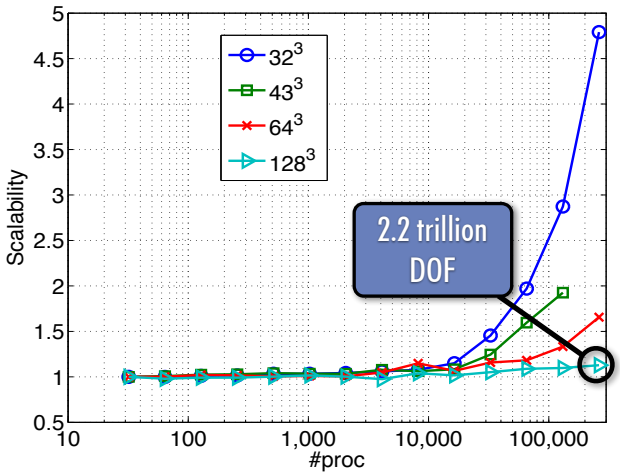

Figure 5. Wasatch weak scaling on Titan for the three-dimensional Taylor-Green vortex using the Hypre linear solver for up to 256,000 processors and various patch sizes ranging from 32^3 to 128^3.

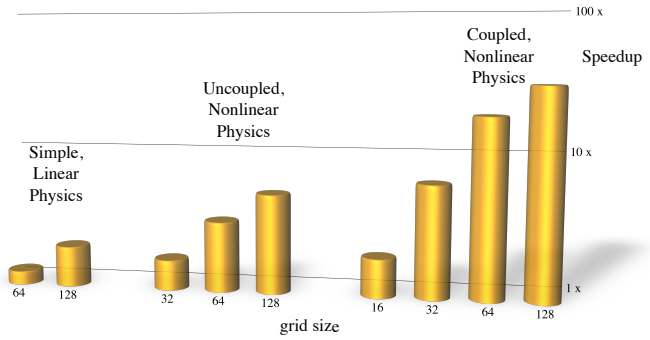

Figure 6. Wasatch GPU speedup for a variety of scalar transport problems ranging from linear uncoupled systems of PDEs to fully coupled.

The scalability of Wasatch has been tested against several applications. Starting with CPU scalability, Fig. 5 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.

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. A total number of 30 equations is solved and the GPU speedup is reported in Fig. 6. A speedup of up to 50x is achieved on the most complex problems as well as the largest patch sizes. This should give us an idea of what problems are worth using GPUs for in Wasatch.

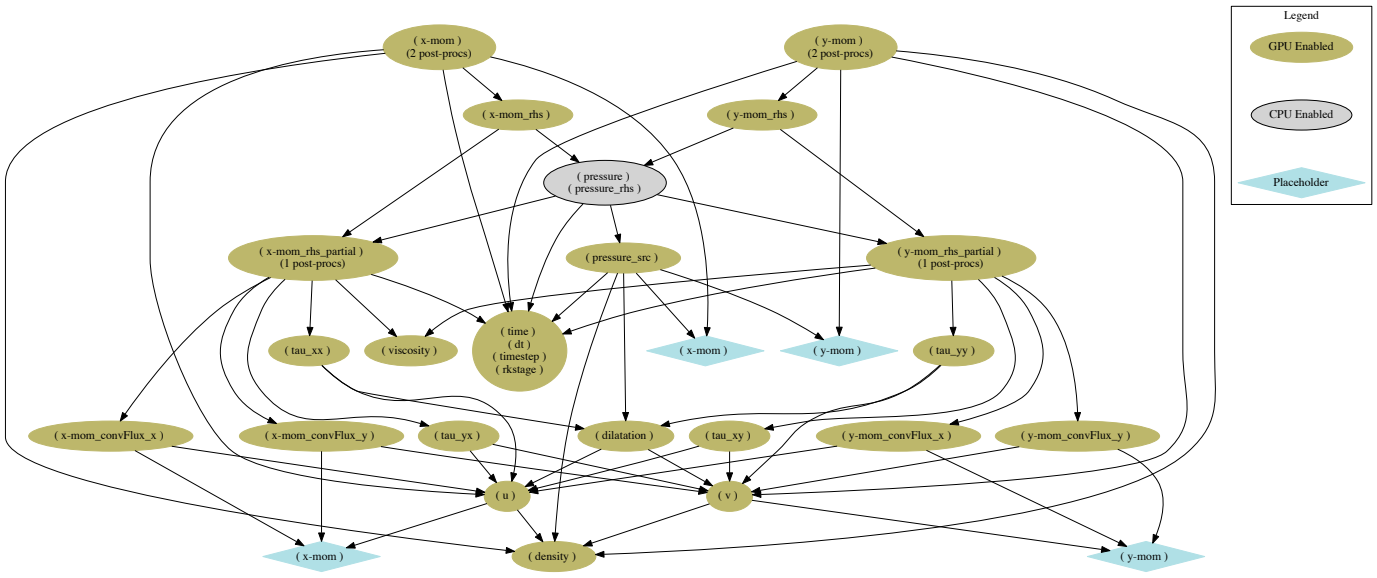

Figure 7. Directed acyclic graph from a Wasatch simulation of an incompressible flow solving the constant-density Navier-Stokes equations.

6. Conclusions

In this paper, we discussed our 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. To address programming complexity, we discussed SpatialOps, a C++ library that provides an interface to the Nebo EDSL and defines necessary operations for the discretization of PDEs on structured grids. 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

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375.

References

[1] A. Alexandrescu. *Modern C++ design*, volume 98. Addison-Wesley Reading, MA, 2001.

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

[3] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962. ISSN 00010782. . URL http://portal.acm.org/citation.cfm?doid=368996.369025.

[4] J. Luitjens and M. Berzins. Improving the performance of uintah: A large-scale adaptive meshing computational framework. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

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

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

[7] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software. volume IV, 2010.

[8] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and 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*, pages 458–465. IEEE, 2013.

[9] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.