Graph-Based Software Design for Managing Complexity and Enabling Concurrency in Multiphysics PDE Software

PATRICK K. NOTZ and ROGER P. PAWLOWSKI, Sandia National Laboratories
JAMES C. SUTHERLAND, University of Utah

Multiphysics simulation software is plagued by complexity stemming from nonlinearly coupled systems of Partial Differential Equations (PDEs). Such software typically supports many models, which may require different transport equations, constitutive laws, and equations of state. Strong coupling and a multiplicity of models leads to complex algorithms (i.e., the properly ordered sequence of steps to assemble a discretized set of coupled PDEs) and rigid software.

This work presents a design strategy that shifts focus away from high-level algorithmic concerns to low-level data dependencies. Mathematical expressions are represented as software objects that directly expose data dependencies. The entire system of expressions forms a directed acyclic graph and the high-level assembly algorithm is generated automatically through standard graph algorithms. This approach makes problems with complex dependencies entirely tractable, and removes virtually all logic from the algorithm itself. Changes are highly localized, allowing developers to implement models without detailed understanding of any algorithms (i.e., the overall assembly process). Furthermore, this approach complements existing MPI-based frameworks and can be implemented within them easily.

Finally, this approach enables algorithmic parallelization via threads. By exposing dependencies in the algorithm explicitly, thread-based parallelism is implemented through algorithm decomposition, providing a basis for exploiting parallelism independent from domain decomposition approaches.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**]: Data Abstraction; G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations; G.4 [**Mathematical Software**]: *Algorithm design and analysis*; G.4 [**Mathematical Software**]: *Parallel and vector implementations*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Scientific computing, object-oriented design, multiphysics, task graph

P. K. Notz acknowledges support from the DOE NNSA ASC Integrated Codes effort at Sandia National Laboratories under contract DE-AC04-94AL85000. R. P. Pawlowski acknowledges support from the DOE NNSA ASC Algorithms effort and the DOE Office of Science AMR program at Sandia National Laboratories under contract DE-AC04-94AL85000. J. C. Sutherland acknowledges support form the National Nuclear Security Administration under the Advanced Simulation and Computing program through DOE Research Grant DE-NA0000740 and by the National Science Foundation through grant no. PetaApps-0904631. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy' s National Nuclear Security Administration under contract DE-AC04-94AL85000.

Authors' addresses: P. K. Notz (corresponding author), Engineering Sciences Center, Sandia National Laboratories, PO Box 5800, MS 0836, Albuquerque, NM 87185-0836; email: pknotz@sandia.gov; R. P. Pawlowski, Computation, Computers and Math, Sandia National Laboratories, PO Box 5800, MS 0836, Albuquerque, NM 87185-0836; J. C. Sutherland, University of Utah, 50 South Central Campus Drive, Room 3290, Salt Lake City, UT 84112-1114.

DOI 10.1145/2382585.2382586 http://doi.acm.org/10.1145/2382585.2382586

ACM Reference Format:
Notz, P. K., Pawlowski, R. P., and Sutherland, J. C. 2012. Graph-Based software design for managing complexity and enabling concurrency in multiphysics PDE software. ACM Trans. Math. Softw. 39, 1, Article 1 (November 2012), 21 pages.
DOI = 10.1145/2382585.2382586 http://doi.acm.org/10.1145/2382585.2382586

1. INTRODUCTION

The last decade has produced tremendous advances in scientific computing capability, driven in large part by governmental investment in large-scale, massively parallel simulation, and the increasing availability/affordability of parallel computing hardware. Many large-scale scientific computing codes use a computational framework that provides parallel IO, data management, and MPI parallelization services. Examples of such frameworks include deal.II [Bangerth et al. 2007], libMesh [Kirk et al. 2006], SAMRAI [Hornung and Kohn 2002; Wissink et al. 2001], SIERRA [Stewart and Edwards 2004; Edwards 2006], Sundance [Long et al. 2010], Uintah [de St. Germain et al. 2000], and MOOSE [Gaston et al. 2009]. By abstracting data management and parallelization these frameworks enabled application programmers to focus on the algorithm implementation. In this context, an *algorithm* is the properly ordered sequence of steps to assemble a discretized set of coupled Partial Differential Equations (PDEs).

As simulation science matures, the complexity of the problems that simulation is applied to naturally increases, leading to a wide range of physical regimes that one would like to address, potentially within a single simulation. This increased complexity leads to a significant software engineering challenge.

Multiscale, multiphysics simulations rely on hierarchical models, which typically have limited ranges of validity. As a simulation realizes different physical regimes, different models may be appropriate. In principle, a simulation could dynamically detect when a model is approaching the limits of its applicability, select a more suitable model, and transition to the new model (see, for example, Oden et al. [2006]). However, dynamically transitioning from one model to another may require significant and intrusive changes such as: addition and/or removal of transport equations, modification of constitutive relationships, modification/addition/removal of source terms, modification of numerical algorithms, etc. Thus, model adaptivity requires significant software flexibility.

The primary source of complexity in traditional software implementations is due to a focus on algorithmic issues (e.g., flow of data) rather than data dependencies. In this article, we present an approach for software engineering that exposes dependencies among operations on data that allows for automated organization of the operation sequence. This design paradigm requires programmers to explicitly expose data dependencies in their software, but relieves them from understanding the complex interdependencies inherently present in multiphysics software. Instead, standard techniques from graph theory [Sedgewick 2002; Sinnen 2007] are used to handle the complex interdependencies.

A number of projects have explored complexity issues associated with multiphysics simulation. One popular approach is to define an Abstract Syntax Language (ASL) that allows users to specify a complete simulation in terms of the finite element weak form. For example, the Sundance project [Long 2003; Long et al. 2010] employs a sophisticated symbolic frontend using operator overloading of a base expression class. The user implements a weak form finite element description in either python or C++ and a dependency graph of expression objects is built internally from the symbolic operations. The FEniCS project [FEniCS 2010; Logg 2007] specifies an ASL called

the Unified Form Language (UFL) [Alnaes et al. 2011] that is preprocessed by a form compiler [Kirby and Logg 2006; Logg 2009] to generate highly efficient C++ code for the complete assembly process. An extensive comparison of a number of additional PDE assembly libraries can be found in Logg [2007].

Our proposed model is an alternative design to the symbolic formulations given before. Our "data centric" concept is slanted heavily towards complex multiphysics problems in a production environment where simple interfaces for analysts, flexible models/data structures, and the integration of nontrivial Third-Party Libraries (TPLs) are paramount. For example, certain ASL implementations require that all models be implemented in the domain language and thus rule out the ability to integrate TPLs that might have a complex formulation such as requiring a nested local linear solve (e.g., the Chemkin library [Kee et al. 2000]) or even a global nonlinear solve (e.g., a nonlinear elimination [Young et al. 2003]). Not only could the operations be impossible to describe using the ASL, but additionally the data structure of the TPL most likely will be incompatible with data structures used by the symbolic framework. Many TPL models represent a major investment manpower to develop, verify, and validate and should not be ruled out by the finite element engine. Our specification allows users to choose their own data model and, most importantly, directly exposes the data model to the users, allowing for integration of difficult TPLs even if inefficient copying is needed for data structure compatibility. We note that not all of the symbolic approaches rule out the preceding operations (in fact the Sundance code is quite capable in this area), but we are merely pointing out that our system is focused on making this easier from an implementation standpoint.

Another advantage of the data-centric approach is that writing directly to the data structures allows the user to control the level of granularity in the assembly process. If using an ASL, then the user must rely on the underlying engine to produce efficient code which depends on the particular ASL implementation. This allows greater flexibility at the cost of development time.

A final advantage deals with the ease of debugging. One great property of ASL is that the process can typically analyze the model formulation and identify efficiency improvements by restructuring the equation operators in ways that are not obvious to users. This, however, can make the code difficult to analyze since the operations no longer may resemble the original system due to the reordering.

The "data centric" system does have drawbacks. One issue is the loss of the view of the global system. This information can be recovered by traversing the graph operations. Another issue is development time. A symbolic frontend such as FEniCS or Sundance can significantly reduce development time since the ASL is usually quite compact and efficient. Finally, the "data centric" approach may introduce overhead with traversing the operations of the dependency graph. The Sundance code also uses an internal dependency graph and has demonstrated exceptional performance using this design by implementing a workset concept [Long et al. 2010]. All operations in a workset are executed on a block of similar finite elements such that the overhead of graph traversal is negligible compared to the work done within a node. The "data centric" approach follows this paradigm.

The goal of this article is to discuss an important aspect of handling complexity in multiphysics software design. Section 2 introduces the concepts of the design using a simple multispecies flux as an example. Section 3 expands the concept to a more complex implementation for Computational Fluid Dynamics (CFD). Section 4 presents additional benefits derived from a graph-based assembly process including sensitivity calculations and (multithreaded) algorithmic parallelism. In Section 5 we draw conclusions.

The software development approach described here has been used in the finite element production codes SIERRA/Thermal-Fluids [Edwards 2006; Stewart and Edwards 2004] and Charon at Sandia National Laboratories since 2003. The core capability has been implemented in the open-source Phalanx package [Pawlowski 2010] in the Trilinos project [Heroux et al. 2005; Trilinos 2011] and the ExprLib library under development at the University of Utah. The approach has been instrumental in developing both explicit and implicit, finite element and finite volume applications for a number of physics. Published results using this approach include general transport/reaction systems [Lin et al. 2010], turbulent CFD [Punati et al. 2011], magnetohydrodynamics [Shadid et al. 2010], multiphase chemically reacting flows [Musson et al. 2009], and semiconductor drift diffusion modeling [Lin et al. 2009].

2. A NEW PARADIGM FOR SOFTWARE DEVELOPMENT

Consider a simple example of the energy diffusive flux, comprised of the Fourier heat flux and (in a multispecies system) the species enthalpy diffusive flux,

$$\boldsymbol{q} = -k\nabla T + \sum_{i=1}^{n_s} h_i \boldsymbol{J}_i. \tag{1}$$

If we were to implement the evaluation of this term using traditional approaches, we would further need to know expressions for the thermal conductivity (k), species enthalpies (h_i), temperature (T), and species diffusive fluxes ($\boldsymbol{J}_i$) in terms of other auxiliary or solution variables so that these quantities could be ordered for calculation prior to calculating $\boldsymbol{q}$. However, even for this simple example, numerous challenges arise. For example, k may be selected from among a number of models and this may only be known at runtime. Therefore, at the point where $\boldsymbol{q}$ is constructed, we do not know the precise dependency on other variables. Furthermore, the species enthalpy diffusive flux term is only relevant for multispecies simulations and may take various forms depending on the constitutive model employed for $\boldsymbol{J}_i$.

What we would like is an approach to software design that allows maximal flexibility in the selection of various models while minimizing the burden on developers in maintaining and modifying such models. To accomplish this, we adopt a software design that directly exposes dependencies between data. At the heart of this design is an *Expression*, which supplies the following functionality.

(1) *Advertises its direct dependencies*. For the example in Eq. (1), the Expression for $\boldsymbol{q}$ would indicate that it required Expressions for k, T, h_i, and $\boldsymbol{J}_i$.
(2) *Binds fields that are required for calculation*. When fields are available for read/write, this method can be called on an Expression to allow the Expression to resolve memory for the fields it writes and reads.
(3) *Calculates the value for the Expression*. Again, in the case of Eq. (1), it would simply calculate the discrete values of $\boldsymbol{q}$ over the portion of the mesh that the fields were defined on, having obtained pointers to the appropriate fields in step 2.

By directly providing the functionality enumerated before, we can automatically construct algorithms (the properly ordered sequence of steps to calculate all required quantities/Expressions). This can be done by providing a few more abstractions.

(1) A *registry* where all Expressions that may be required for the calculation are placed. This registry allows registration of Expressions and provides a way to obtain fully constructed Expressions via a *tag*, typically implemented as a string.

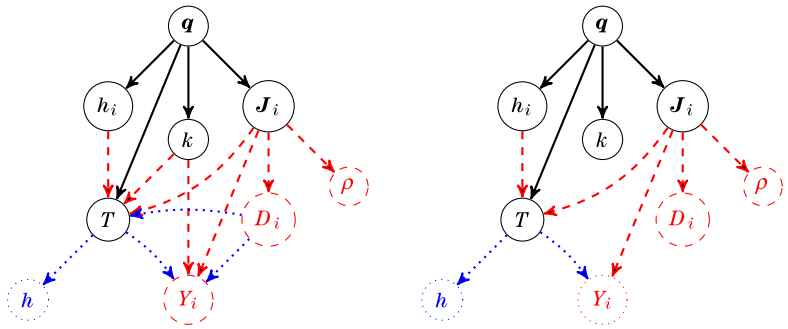

Fig. 1. Example dependency graph for the first three dependency layers in the calculation of the enthalpy diffusive flux given by Eq. (1). Solid lines are direct dependencies advertised by the Expression for $\boldsymbol{q}$ itself whereas dashed and dotted lines are dependencies discovered by following out-edges and recursively adding Expressions to the graph.

The registry thus serves as a mechanism to either set or request the Expression required to compute a given quantity.

(2) A mechanism to build the Directed Acyclic Graph (DAG) representing the dependency among Expressions. Once the user defines one or more quantities that are desired, these can be resolved from the registry and form the "root" nodes in a graph. Each node may then be queried to determine its dependencies, which are placed on the out-edges of the node. This process is repeated recursively until the "bottom" of the graph is reached where there are no further out-edges.

(3) A *scheduler* that traverses the graph to execute it. The execution graph may be obtained by inverting the dependency graph constructed in step 2. A node is scheduled for execution once all of its data-dependents (execution-parents) have completed. This allows significant flexibility in the ordering in which nodes are executed, and facilitates task-based parallelism, which will be discussed in more detail later.

Figure 1 shows the graph for the first two levels of dependencies on $\boldsymbol{q}$ for two scenarios. First (left graph), we consider $k(T, Y_i)$ and $D_i(T, Y_i)$, where Y_i are the species mass fractions, while the second (right graph), we consider constant k and D_i. In both situations we assume $\boldsymbol{J}_i = -\rho D_i \nabla Y_i$. The first level of dependencies is indicated by the solid black lines, the second level by dashed red lines, and the third level by dashed blue lines. The only thing that distinguishes the right and left graphs is the Expressions that are registered to calculate k and D_i. There is no other change required to any part of the software base.

There are several important implications of this approach relative to "traditional" programming models. First, although the data dependencies are well-defined, the execution ordering is not (in general) unique or known at compile time. The execution ordering is determined by a reverse topological sorting of the dependency graph (Figure 1) as well as the time that other Expressions take to execute. For example, in the left graph of Figure 1, either h_i (the mixture enthalpy) or Y_i could be executed first (or they could be executed concurrently in a multithreaded environment). After h_i and Y_i complete, T must be executed next (solving a nonlinear equation for T at each grid point), and represents a "serialization" point in the graph. Next, h_i, k, and D_i are all ready. However, for the right graph in Figure 1 h_i, Y_i, k, and D_i may all be executed concurrently. This clearly illustrates the algorithmic flexibility afforded by the graph approach.

Since the algorithm for assembling the overall set of discretized equations is deduced from the union of data dependencies, the application programmer need not specify it. This eliminates the need for complex logic and provides an entirely generic method of generating an assembly routine for complex multiphysics simulations. A hallmark of complex, multiscale simulations is the multitude of models that may be employed, with each model potentially requiring its own transport equations and employing various constitutive and state relationships. The graph-based approach described here directly facilitates such complex simulation endeavors by hiding the algorithmic complexity from the programmer. Indeed, when implementing the model for $\boldsymbol{q}$, the programmer need not be aware that there is a dependency on Y_i or D_i. These dependencies are automatically discovered when the graph is created. This insulates the developer significantly from changes made elsewhere in the software base. Models for other terms may be changed to introduce new dependencies (nonlinear couplings) among terms in equations that may have not existed previously and this will be automatically detected and handled.

A second important implication of the graph-based approach is that the programmer need only be aware of *one* level of dependencies (the *direct* dependencies of a given Expression). For example, in Figure 1, the Expression for T does not know that it is required by h_i, $\boldsymbol{q}$, etc.

A third implication of the graph-based approach is that, since the dependencies are explicitly declared in the code, it is straightforward to automatically construct parallel assembly algorithms at runtime (see Section 4.2). By prioritizing execution (e.g., using a priority-based scheduler) in a parallel execution environment, a more optimal ordering can be determined by measuring and utilizing the computational cost of each Expression (slower Expressions have higher priority) and the connectivity of the Expression within the DAG. This is the subject of Section 4.2.2.

In short, the proposed approach directly exposes the dependencies among data and extracts an appropriate algorithm from them, whereas traditional approaches directly expose the algorithm.

2.1. Implementation Considerations

2.1.1. Expressing Dependencies. As noted earlier, the essential aspect of the design is the specification of data provided by an Expression and the information required to compute that data. Thus, a necessary component of the design is a means of naming the data associated with Expressions. This requires that the names be *generic* and independent of a particular implementation. For example, if one chose to use simple string names, one would refer to k in the preceding example as "`thermal conductivity`" and not "`constant thermal conductivity`" or "`variable thermal conductivity`". By using the same name for all variants of an Expression, one can simply register the appropriate variant in the registry for use in the calculation.

2.1.2. On the Granularity of Expressions. The choice of what to include in an Expression involves several trade-offs. In this section, we review some of the consequences of the design and highlight the trade-offs that are typically encountered.

Each Expression constitutes an atomic level of computation in the system and produces some intermediate data that are used in the assembly process. A finer-grained decomposition of the model results in more Expressions for a given system and, hence, larger memory requirements[1].

[1]For this reason, Expressions are typically evaluated in multiple passes over subsets of the elements or cells in the analysis.

Moreover, since each Expression must be fully evaluated before its dependents, there are separate assembly loops for each Expression in the system. Consequently, a finer-grained decomposition results in more inner assembly loops and can adversely affect runtime performance.

An additional benefit of larger, course-grained Expressions is that they tend to make it easier to verify through source-code inspection that the target system of PDEs is are being correctly assembled. This is especially true for domain experts who may be extending the software but are less familiar with the details of the software design and implementation.

Expressions serve the purpose of providing extension points in the software design. Any implementation or *model* may be substituted for any other so long as the data produced conforms to the expected pattern, for example, scalar or vector fields. For this reason, fine-grain Expressions provide the maximum flexibility in composing different problems and offer the greatest opportunity for reuse.

Smaller Expressions also act as a unit of reuse within the software. This can take the form of runtime reuse, such as the Expression for $\nabla \boldsymbol{v}$ which is used by several Expressions in the example given before. Alternatively, this can take the form of reuse of the code that implements the Expression, such as one class that computes the gradient of any nodal vector field and is configured for a particular field during instantiation.

Additional benefits of smaller, fine-grained Expressions are that they tend to be easier to write, easier to read, and tend to have smaller defect rates. Fine-grained Expressions also tend to be easier to unit-test and debug. A larger number of Expressions may also produce more efficient parallel evaluation (see Section 4.2).

2.2. Interfacing with Existing Computational Frameworks

As mentioned in the Introduction, many existing computational frameworks handle domain decomposition and data parallelism via MPI. The proposed approach here is both complimentary and orthogonal to such frameworks, and can be implemented within them. During the problem setup phase the graph is constructed. The required fields can be identified directly from the graph and advertised to the framework. During the execution phase, once memory is available for each of the required fields, each Expression may be allowed to bind memory for the fields it reads/writes. Then each Expression may be executed in the required ordering as dictated by the graph and implemented by the scheduler.

3. A PRACTICAL EXAMPLE: CFD

In this section, we provide a more detailed example drawn from computational fluid dynamics in a finite-element context. Consider the steady-state solution of a coupled thermal-fluid problem composed of the incompressible Navier-Stokes and energy conservation equations [Bird et al. 2007],

$$\begin{aligned} \rho C_p \boldsymbol{v} \cdot \nabla T &= -\nabla \cdot \boldsymbol{q} + H_v \\ \rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} &= \nabla \cdot \boldsymbol{\sigma} \\ \nabla \cdot \boldsymbol{v} &= 0. \end{aligned} \tag{2}$$

Here, ρ is density, C_p is the specific heat, T is temperature, $\boldsymbol{v}$ is velocity, $\boldsymbol{q}$ is the diffusive flux of energy (discussed previously), H_v is a volumetric heat source, and $\boldsymbol{\sigma}$ is the fluid stress tensor. For this example, we consider the unknown solution variables to be T, $\boldsymbol{v}$, and pressure p.

Even this simple system of equations illustrates the couplings and variabilities of multiphysics simulations. Variabilities enter through the choices of material models for ρ and C_p, constitutive equations for $\boldsymbol{q}$ and $\boldsymbol{\sigma}$, and a multitude of conceivable source terms H_v. Couplings appear already in the preceding equations ($\boldsymbol{v}$ is in all equations) and additional couplings typically appear through material properties and constitutive equations that are functions of the unknown variables.

To further this example, we consider the finite element solution of (2) (see, for example, Donea and Huerta [2003]). We start with the weighted residual form which, after integration by parts and ignoring boundary contributions for the sake of simplicity, yields

$$\begin{aligned} R_T^i &= \int_\Omega \left[\left(\rho C_p \boldsymbol{v} \cdot \nabla T - H_V \right) \phi_T^i - \boldsymbol{q} \cdot \nabla \phi_T^i \right] \mathrm{d}\Omega = 0 \\ R_{v_k}^i &= \int_\Omega \left[\rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} \phi_{\boldsymbol{v}}^i + \boldsymbol{\sigma} : \nabla \left(\phi_{\boldsymbol{v}}^i \boldsymbol{e}_k \right) \right] \mathrm{d}\Omega = 0 \\ R_p^i &= \int_\Omega \nabla \cdot \boldsymbol{v} \phi_p^i \, \mathrm{d}\Omega = 0. \end{aligned} \tag{3}$$

In (3) Ω, is the domain over which the problem is solved and ϕ_T^i, $\phi_{\boldsymbol{v}}^i$, and ϕ_p^i are finite-element basis functions for T, $\boldsymbol{v}$, and p, with coefficients T^i, $\boldsymbol{v}^i$, and p^i, that is,

$$T = \sum_{i=1}^{N_T} T^i \phi_T^i, \quad \boldsymbol{v} = \sum_{i=1}^{N_v} \boldsymbol{v}^i \phi_{\boldsymbol{v}}^i, \quad \text{and} \quad p = \sum_{i=1}^{N_p} p^i \phi_p^i. \tag{4}$$

The standard approach to performing the integrations in (3) uses a numerical quadrature with N_q pairs of weights and abscissas $\{w, \boldsymbol{\xi}\}$. The discrete form of (3) is then

$$\begin{aligned} \hat{R}_T^i &= \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left[\left(\rho C_p \boldsymbol{v} \cdot \nabla T - H_V \right) \phi_T^i - \boldsymbol{q} \cdot \nabla \phi_T^i \right] w_q |j| = 0, \\ \hat{R}_{v_k}^i &= \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left[\rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} \phi_{\boldsymbol{v}}^i + \boldsymbol{\sigma} : \nabla \left(\phi_{\boldsymbol{v}}^i \boldsymbol{e}_k \right) \right] w_q |j| = 0, \\ \hat{R}_p^i &= \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \nabla \cdot \boldsymbol{v} \phi_p^i w_q |j| = 0, \end{aligned} \tag{5}$$

where N_e is the number of elements in the domain and $|j|$ is the determinant of the Jacobian of transformation from the physical space to a reference space where integration is performed.

To close the equations in (5) we need to specify relationships for $\boldsymbol{q}$, $\boldsymbol{\sigma}$, ρ, C_p, and H_V. Here we choose

$$\begin{aligned} \boldsymbol{q} &= -k \nabla T, \\ \boldsymbol{\sigma} &= \boldsymbol{\tau} - p \boldsymbol{I}, \\ \boldsymbol{\tau} &= \mu \left(\nabla \boldsymbol{v} + \nabla \boldsymbol{v}^t \right), \\ H_V &= \boldsymbol{\tau} : \nabla \boldsymbol{v}, \\ \mu &= \mu_\circ e^{-E/T}, \end{aligned} \tag{6}$$

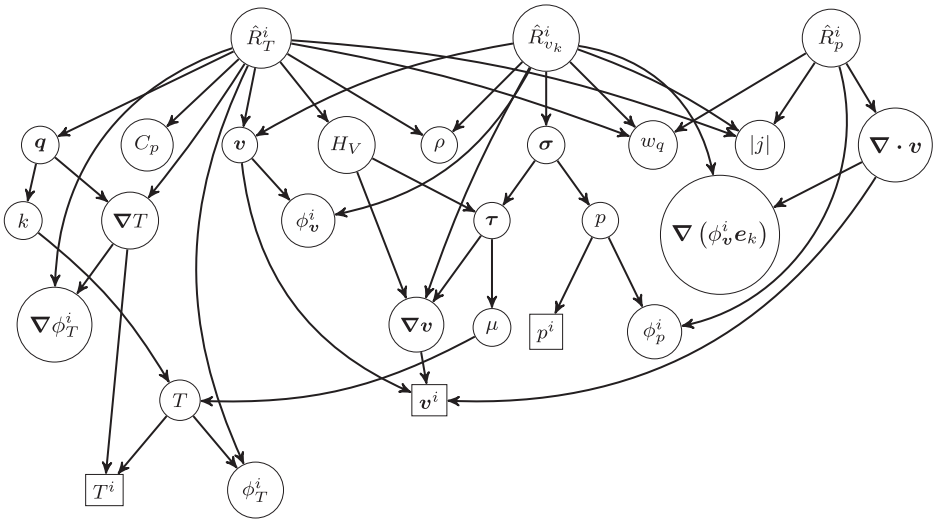

Fig. 2. Example Expression dependency graph for the system of PDEs in (5).

where μ is the viscosity, k is the thermal conductivity, and $\mu_\circ$ and E are coefficients of an Arrhenius viscosity function. For simplicity, we take k, ρ, and C_p to be constants.

3.1. Decomposing into Expressions

The next step is to decompose the discretized PDE system in (5) into a set of so-called Expressions with explicit dependencies. One possible decomposition is illustrated in Figure 2 where each node in the graph denotes an Expression and arrows point in the direction of dependencies, for example, $\hat{R}^i_T$ depends on $\boldsymbol{q}$, etc. In Figure 2 the Expressions for T^i, $\boldsymbol{v}^i$, and p^i are denoted by rectangles instead of circles to indicate that these are actual unknowns in the problem.

The critical observation to make is that, from the perspective of numerical algorithm implementation (i.e., the assembly sequence), the direct dependencies among Expression *data* are all that need to be provided/exposed to the scheduler to properly assemble the final system; the details of the functional form of any given Expression can be hidden from all other Expressions.

It is worth reemphasizing that in constructing the graph only the immediate "downstream" dependencies of an Expression are required and that only through the recursive construction of the final graph are nested dependencies revealed. For this example, we ultimately require evaluation of Eq. (5). We thus select these as the "roots" of our graph and then proceed to construct the graph by recursing through each Expression and obtaining its dependencies, then adding them to the graph and recursing again. The result of this process is the graph shown in Figure 2.

Note that upward dependencies are not required. For example, the Expression for T in Figure 2 does not know that it is required by several other Expressions.

Additionally, only direct (one-generation) downstream dependencies are required. For example, in Figure 2, $\boldsymbol{q}$ explicitly requires ∇T and k, but the Expression for $\boldsymbol{q}$ is unaware of the fact that k depends on T. If a different model for k was used, the only change would be from the node representing k downward.

In contrast with many traditional software structures, this insulates the programmer from the complicated dependencies that naturally arise in multiphysics problems.

3.2. Utilizing the Graph for Assembly

Given the DAG representation of the data dependencies, standard graph algorithms can be used to construct a suitable ordering for evaluating the Expressions (represented as nodes in the DAG) to consistently assemble the set of PDEs. For example, a reverse topological sort will produce an ordering of the Expressions where each Expression is guaranteed to appear after all of its dependent Expressions. In this example, one such suitable ordering is: ϕ_T^i, T^i, $\boldsymbol{v}^i$, $\phi_{\boldsymbol{v}}^i$, ϕ_p^i, p^i, C_p, ρ, $|j|$, $\nabla\left(\phi_{\boldsymbol{v}}^i \boldsymbol{e}_k\right)$, $\nabla \boldsymbol{v}$, $\boldsymbol{v}$, T, k, p, $\nabla \cdot \boldsymbol{v}$, μ, $\boldsymbol{\tau}$, $\nabla \phi_T^i$, ∇T, $\boldsymbol{q}$, H_V, $\boldsymbol{\sigma}$, w_q, $\hat{R}_T^i$, $\hat{R}_{v_k}^i$, $\hat{R}_p^i$. In a serial execution mode, these Expressions are evaluated sequentially.

4. ADDITIONAL BENEFITS FROM GRAPH INFORMATION

4.1. Computation of Newton Sensitivities

Complex, multiphysics simulations are often highly nonlinear and so it is common to use the Newton-Raphson method [Dennis and Schnabel 1983; Kelley 2003] for the solution of the resulting nonlinear system of algebraic equations that arise from the discretization. Application programmers often resort to matrix-free Newton-Krylov methods [Brown and Saad 1990; Knoll and Keyes 2004] or finite differencing techniques to avoid the complicated and error-prone task of writing the Newton sensitivities by hand. The graph-based design described in this article provides the means to compute exact, analytic sensitivities in a simple and manageable way. It also provides a clean way to combine analytic, finite differenced, and automatic [Aubert et al. 2001; Bartlett et al. 2006; Phipps et al. 2008; Rall 1981] sensitivities in a single calculation giving developers the liberty to choose the most appropriate or expedient approach on a per-Expression basis.

The existence of the DAG makes possible the runtime determination of which Newton sensitivities exist and how each Expression contributes to a sensitivity calculation. At the time of implementation, one doesn't need to know which degrees of freedom will be active during the simulation. Instead, sensitivity contributions are assembled while traversing the graph using partial derivatives and the chain rule.

For example, in the example system of PDEs presented in Section 3, a Newton-based solution method would require the sensitivity of the momentum equation $\hat{R}_{v,k}^i$ with respect to a temperature degree of freedom T_j. To write this term, $\frac{\partial \hat{R}_{v,k}^i}{\partial T_j}$, one would have to know how each term in the momentum equation depends on the temperature. If a model such as viscosity was swapped out for a different model that either added or removed a dependency on temperature, then the sensitivity calculation would have to account for this. Sensitivity calculations can quickly become interwoven with complex logic for switching between different models.

By using our proposed assembly algorithm, the DAG can be used to compute the sensitivities as it traverses the graph. In this case, the sensitivities that the user implements in an Expression are derivatives with respect to the direct dependencies on other Expression data.

By tracing the DAG, the sensitivity in our example is computed as

$$\frac{\partial \hat{R}_{v,k}^i}{\partial T_j} = \frac{\partial \hat{R}_{v_k}^i}{\partial \boldsymbol{\sigma}_{rs}} \frac{\partial \boldsymbol{\sigma}_{rs}}{\partial \boldsymbol{\tau}_{rs}} \frac{\partial \boldsymbol{\tau}_{rs}}{\partial \mu} \frac{\partial \mu}{\partial T} \frac{\partial T}{\partial T_j}. \tag{7}$$

In an Expression system, each Expression computes and stores its sensitivities using the chain rule so that dependent Expressions can use the accumulated values.

Furthering our example, the intermediate sensitivities could be computed and stored as

$$\begin{aligned}
\frac{\partial T}{\partial T_j} &\leftarrow \phi_T^j \\
\frac{\partial \mu}{\partial T_j} &\leftarrow \frac{\partial \mu}{\partial T}\frac{\partial T}{\partial T_j} = -\frac{E\mu_\circ}{T}e^{-E/T}\frac{\partial T}{\partial T_j} \\
\frac{\partial \boldsymbol{\tau}_{rs}}{\partial T_j} &\leftarrow \frac{\partial \boldsymbol{\tau}_{rs}}{\partial \mu}\frac{\partial \mu}{\partial T_j} = (\nabla \boldsymbol{v}_{rs} + \nabla \boldsymbol{v}_{sr})\frac{\partial \mu}{\partial T_j} \\
\frac{\partial \boldsymbol{\sigma}_{rs}}{\partial T_j} &\leftarrow \frac{\partial \boldsymbol{\sigma}_{rs}}{\partial \boldsymbol{\tau}_{rs}}\frac{\partial \boldsymbol{\tau}_{rs}}{\partial T_j} = \frac{\partial \boldsymbol{\tau}_{rs}}{\partial T_j} \\
\frac{\partial \hat{R}^i_{v,k}}{\partial T_j} &\leftarrow \frac{\partial \hat{R}^i_{v_k}}{\partial \boldsymbol{\sigma}_{rs}}\frac{\partial \boldsymbol{\sigma}_{rs}}{\partial T_j} = \sum_{q=1}^{N_q} \nabla\left(\phi_v^i \boldsymbol{e}_k\right)_{sr} w_q |j| \frac{\partial \boldsymbol{\sigma}_{rs}}{\partial T_j}.
\end{aligned} \tag{8}$$

In this form, the sensitivities for each Expression are composed of two contributions:

(1) the partial of the Expression with respect to the prerequisite and
(2) the partial of the prerequisite with respect to a degree of freedom.

The first contribution is valid for any degree of freedom and the second is merely numerical values stored in an array. Thus, the sensitivity of any Expression can be computed with respect to any degree of freedom using only the knowledge of the immediate prerequisites and the propagated values of the prerequisites' sensitivities. The user need only implement an evaluation of the first contribution (either analytically or by numerical approximation). Only the lowest-level Expressions representing the degrees of freedom require special treatment, that is, $\partial T/\partial T_j$ in this example. In practice, for each prerequisite Expression one simply iterates over the degrees of freedom that prerequisite is sensitive to, retrieves storage for the prerequisite's accumulated sensitivity, and computes and stores its own sensitivity. In general, sensitivities to a given degree of freedom may propagate through multiple prerequisites; for example, both k and ∇T depend on T_i in our example system. A natural consequence of this design is that as new physics are added to the code, the new sensitivities will appear in all existing Expressions automatically.

Each Expression in the system is responsible for evaluating its own sensitivities and simply storing them for access by the Expressions that depend upon it. Thus, each Expression has the opportunity to choose how to evaluate the sensitivities: handcoded analytic sensitivities (as in the right hand-side of (8)); finite differencing at the local (Expression) level; or through the use of automatic differentiation. This also gives implementers the liberty to deal with special situations, such as interfacing to third-party libraries or expensive database lookups, where computing the exact sensitivities may be expensive, undesirable, or impractical, without having to resort to finite difference or matrix-free algorithms for the entire system.

Computing sensitivities by finite difference can be performed efficiently with the use of the Expression graph. For instance, an algorithm can dynamically determine which degrees of freedom each Expression is sensitive to and compute only those finite differences. Moreover, finite differencing can be used to compute either the actual sensitivity, for example, $\partial \boldsymbol{\tau}_{rs}/\partial T_j$, or just the derivative with respect to the prerequisite Expression, for example, $\partial \boldsymbol{\tau}_{rs}/\partial \mu$. Finally, if one chooses to implement hand-coded sensitivities, a finite difference algorithm can be used to detect errors. This has been extremely useful in our experience because this approach is able to determine the precise Expression where the error is, typically down to a single line of code. For example,

using this technique, one can detect an error in $\partial \boldsymbol{\tau}_{rs}/\partial\mu$ rather than in the final Jacobian entry for $\partial \hat{\boldsymbol{R}}^i_{v_k}/\partial \boldsymbol{T}_j$.

4.2. Thread Parallelism via Algorithm Decomposition

Modern computing architectures offer shared memory paradigms within a distributed memory environment, where a single computational "node" may have many shared memory computational cores (currently on the order of 10s but expected to scale to 1000s [IESP 2010; Sarkar 2009]). Computational approaches that expose multiple opportunities for parallelism are needed to efficiently use these architectures. For example, when solving PDEs, the scientific community has traditionally relied upon domain decomposition approaches where the spatial and/or temporal domain is divided and balanced across distributed nodes. This approach has been very successful in the context of weak scaling (work per node is constant) where the addition of cores allows for more accurate discretizations using more cells/elements. However, for strong scaling (total work is constant) we may end up with more cores than available work. Other levels of parallelism need to be explored especially in this context of strong scaling. The changing hardware landscape will drive future applications to exploit shared memory where available.

The Expression graph identifies concurrency in the algorithm and can, therefore, be used to automatically evaluate individual Expressions in parallel on threads within a computational node. Furthermore, as the breadth of the graph increases the opportunity for thread parallelism likewise increases. One way this may happen is through the simultaneous solution of an increasing number of PDEs. However, as we demonstrate shortly, the increase in available concurrency is problem dependent.

Granularity is an important issue in parallelization. We consider two types of granularity in this context.

(1) *(Sub-)Domain granularity*. This refers to the size of the subdomain in a domain decomposition approach. This size is typically determined by considering the number of distributed memory computational nodes as well as the size of memory/cache on a local node/core. This can be changed as needed by the computational framework to optimize the workload per node/core.
(2) *Graph granularity*. The programmer determines graph granularity (see Section 2.1.2), trading complexity (and cost) of a single Expression for the amount of memory required to store variables associated with each Expression. Finer granularity in the graph exposes more opportunity for algorithmic parallelization.

We thus have two levels of parallelism exposed in the calculation: domain decomposition and graph/algorithm decomposition. These levels are independent of each other and can be used separately or together to optimize efficiency for a given problem and architecture. Specifically, domain decomposition would be determined by the number of available nodes as well as the cache size of the cores on each node. Once the domain has been decomposed, the graph associated with each subdomain can be executed in parallel on shared memory cores. The computational cost associated with a graph scales directly with the subdomain size, as each graph is executed over an entire subdomain. Depending on the memory requirements of the algorithm, the subdomains may be further decomposed into yet smaller domains or "worksets" so the memory required by the graph of Expressions fits entirely in cache.

4.2.1. Optimal Schedules, Upper Bounds on Speedup and Parallelizability. In this section, we analyze the concurrent evaluation of a graph of Expressions. In this context, we consider the evaluation of each Expression to be sequential, but multiple Expressions

may be evaluated concurrently as long as dependent Expressions are evaluated first. It is immediately clear that the extent to which a graph of Expressions may be evaluated concurrently is very much dependent on the problem being solved and implementation choices (see Section 2.1.2). Furthermore, the evaluation schedule, that is, the allocation of Expressions to processors[2] and the start times of each, cannot be determined a priori because the evaluation times of each Expression may change dynamically e.g., due to conditional branching). However, for a given graph one can assume Expression evaluation times and calculate the *schedule time*, namely the evaluation time of the entire graph, for various scheduling algorithms. Of particular interest here are the extreme cases of a single processor system and an infinite processor system. In practice, for a graph with $\mathcal{N}$ nodes the infinite processor schedule is the $\mathcal{N}$ processor schedule; that is, each node has its own processor. For the sake of these calculations, we assume all processors are of the same speed and there are no communication costs or other latencies.

The schedule time $\mathcal{T}_1$ for a single processor system is simply the sum of the execution times $\tau_e(n_i)$ for each node (Expression) n_i in the graph

$$\mathcal{T}_1 = \sum_{i=1}^{\mathcal{N}} \tau_e(n_i), \tag{9}$$

where $\mathcal{N}$ is the number of nodes in the graph.

In order to compute the schedule time $\mathcal{T}_\infty$ for the case of infinite number of processors, we must first define the data-ready, start, and finish times for each node in the graph (see, e.g., Sinnen [2007]). If execution of the graph is started at a time $t = 0$ then the start time $t_s(n_i)$ of node n_i is the nonnegative time when node n_i is started. The finish time $t_f(n_i)$ of node n_i is simply $t_f(n_i) = t_s(n_i) + \tau_e(n_i)$. The data-ready time $t_{dr}(n_i)$ for node n_i is the maximum finish time of all of the prerequisite Expressions for n_i.

$$t_{dr}(n_i) = \max_{n_j \in \text{prereq}(n_i)} t_f(n_j) \tag{10}$$

In the absence of communication costs, memory latency, or scheduling overhead, the start time of each node is simply the data-ready time, $t_s(n_i) = t_{dr}(n_i)$. Leaf nodes in the graph that have no prerequisite Expressions have a start time of zero. Note that t_s, t_f, and t_{dr} can be computed for every node in the graph if they are processed in a topological order. Once the finish times are known for each node in a graph $\mathcal{G}$, the schedule length for an infinite number of processors with no communication costs is simply the maximum finish time,

$$\mathcal{T}_\infty = \max_{n_i \in \mathcal{G}} t_f(n_i) \tag{11}$$

For a given number of processors N with schedule time $\mathcal{T}_N$ the speedup is defined as

$$\mathcal{S}_N = \frac{\mathcal{T}_1}{\mathcal{T}_N}. \tag{12}$$

Of particular interest is the speedup for the infinite processor case, which is the ratio of the two schedules given in (9) and (11).

$$\mathcal{S}_\infty = \frac{\mathcal{T}_1}{\mathcal{T}_\infty} \in [1, \mathcal{N}] \tag{13}$$

[2]In this context, we take *processor* to be an implementation-dependent execution mechanism such as a process or thread.

$\mathcal{S}_\infty$ is the upper bound of the speedup achievable with concurrency.

A measure of the *parallelizability* of a graph $\mathcal{G}$ with $\mathcal{N}$ nodes can be determined using Amdahl's law [Amdahl 1967], namely

$$\mathcal{P}(\mathcal{G}) = \frac{1 - 1/\mathcal{S}_\infty}{1 - 1/\mathcal{N}} \in [0, 1]. \tag{14}$$

In Section 4.2.4 we examine the upper bound $\mathcal{S}_\infty$ and parallelizability $\mathcal{P}$ for several real multiphysics analyses taken from our implementations.

The concept of a *task graph* is well established in the literature as a means of scheduling dependent tasks, in serial and parallel. See Sinnen [2007] for a survey on the subject of task graphs and scheduling on parallel systems. Like the dependency graphs introduced in Sections 2 and 3, task graphs are directed acyclic graphs. However, it is customary to draw task graphs with edges whose direction is given by the flow of information rather than dependencies. For example, if n_i and n_j are two nodes in a task graph and data is communicated from n_i to n_j then the edge e_{ij} points from n_i to n_j. Thus, scheduling algorithms and heuristics from the task graph literature may be readily applied to the Expression design presented in this work so long as one is careful to respect the reversed representation of the graph edges.

4.2.2. Scheduling Algorithms. For efficient parallel execution of an Expression graph on realistic machines with a limited number of cores, limited lowest-level cache, and memory latencies, one must weigh several issues.

— *Graph depth*. We may want to prioritize execution of Expressions that are deep within the graph above those that are shallow. For example, evaluation of the graph in Figure 2 would prioritize evaluation of ϕ_T^i first since it is at the maximum depth of 5 in the graph. The Expression for C_p could be evaluated at any time, but is of relatively low priority (its depth is only 1) so it can be used for backfilling if necessary.
— *Expression computational cost*. As execution proceeds, the cost of execution of a given Expression can be considered to increase priority for expensive Expressions.
— *Cache locality*. If the T Expression is evaluated on core "A" then it may be advantageous to evaluate ∇T on core "A" as well to improve cache reuse. Associating a core affinity to each Expression may be a way to achieve this.

Such considerations can be used to construct an efficient parallel dispatch algorithm that allows for backfilling to optimize parallel traversal of the graph. It is well known that the general problem of scheduling of task graphs, even for the case of static graphs with a priori known timings, is NP-complete [Ullman 1975]. In our applications, each node in the graph has unknown, dynamically changing timings and so we must resort to heuristic scheduling algorithms.

Currently, we employ an observer pattern [Gamma et al. 1994] to implement the parallel scheduling algorithm. Specifically, the graph is traversed and each node in the graph (corresponding to an Expression) subscribes to signals owned by each of its children (the nodes on each of its out-edges). Execution is started by loading all of the Expressions with no dependencies into a priority queue. As an Expression completes, it executes its signal, notifying all of its "parents." When an Expression receives signals from all of its children, it loads into the priority queue. This methodology ensures that all of the dependencies in the graph are met.

Table I. Relationship between Terms in Eq. (15) and Nodes in Figure 3

Term in equation (15)	$\frac{\partial y_i}{\partial t}$	$\Gamma_i \nabla \phi_i$	s_i	ϕ_i
Descriptor in Figure 3	`i_rhs`	`i_flux`	`i_src`	`var_i`

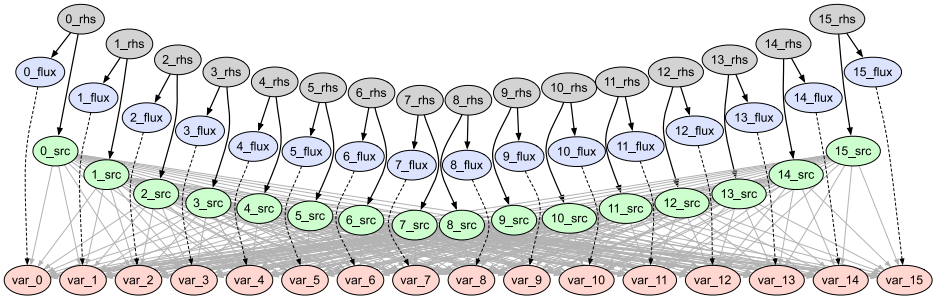

Fig. 3. Expression graph for the system of idealized parabolic (diffusion reaction) PDEs in (15) with $n = 16$.

Table II. Theoretical Parallelizability for Solving Eq. (15) for Various n

n	$\mathcal{P}$	$\mathcal{S}_\infty$
8	0.89	7.5
16	0.94	13.7
32	0.97	28.6

4.2.3. Scalability for an Idealized Case. To demonstrate the potential for exploiting the parallelism exposed by the Expression graph, we consider an idealized system of parabolic (diffusion reaction) PDEs given by

$$\frac{\partial y_i}{\partial t} = \nabla \cdot (\Gamma_i \nabla y_i) + s_i(y_1, \ldots, y_n) \quad i = 1 \ldots n, \tag{15}$$

where t is time, y_i is the species mass fraction, and Γ_i is a constant diffusion coefficient. Here, there is all-to-all coupling through the source term (s_i) which, for equation i, is a function of all y_j. In what follows, we consider the case for $n = 8$, 16, and 32 equations. The resulting Expression graph for $n = 16$ is shown in Figure 3, and Table I relates the terms in Eq. (15) to the nodes in the graph in Figure 3.

Following the analysis of Section 4.2.1, we obtain the upper bounds on speedup ($\mathcal{S}_\infty$) and parallelizability ($\mathcal{P}$) reported in Table II for various values of n. Figure 4 shows the speedup and parallel efficiency for solution of (15) with various n. Note that the entries in Table II correspond to using n cores for n equations, so that the most relevant point to compare between Table II and Figure 4 is $n = 8$ with 8 threads. There are several noteworthy results shown in Figure 4. First, the actual speedup does not approach the theoretical speedup for $n = 8$ where we expect $\mathcal{P} = 7.5$ and yet we obtain a speedup of approximately 3 on 8 threads. This certainly suggests room for improvement, and we suspect that setting thread affinity may improve scalability, since thread migration across cores will lead to inefficient cache behavior. Second, as n increases, the parallel efficiency increases. This is likely due to the increased opportunity to backfill and hide latency in the scheduling/queuing system.

In most practical applications, we employ MPI for large-scale parallelism, and graph-level parallelism will be employed at thread level within an MPI process. Figure 5 shows the strong scaling results for solving Eq. (15) with $n = 16$ using up to

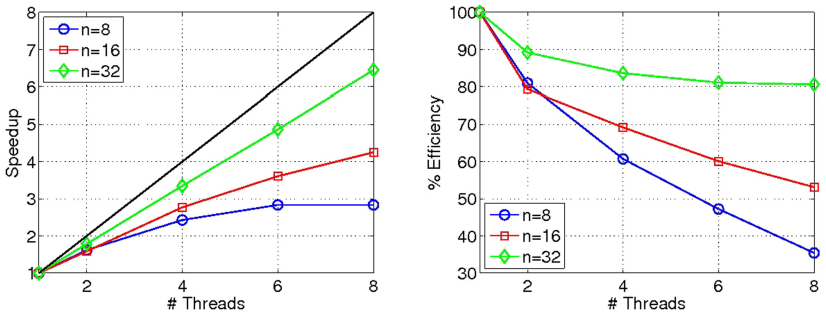

Fig. 4. Speedup (left) and parallel efficiency (right) for solution of Eq. (15) for various n. Theoretical values are given in Table II.

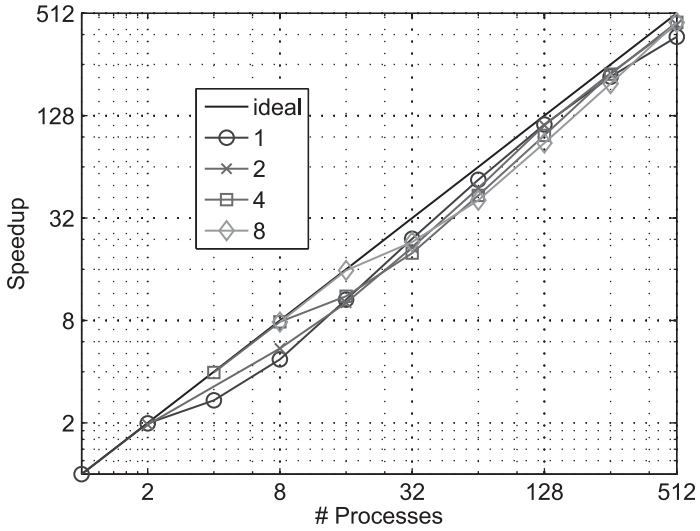

Fig. 5. Strong scaling speedup as a function of the total number of processes running. Each line corresponds to a different number of threads replacing some of the MPI processes.

512 processes. The algorithm was a fully explicit, Runge-Kutta time integrator[3] with a finite-volume discretization of (15).

These results were obtained on a machine with 8 cores per shared memory node. The circles indicate scalability using 512 MPI processes (run on 64 8-core nodes), where no concurrency in the graph is exploited. The diamonds indicate the results using 64 MPI processes with 8 threads per node, the maximum hardware concurrency available and processing the Expression graph with 8-way parallelism. As is evident in Figure 5, all of these demonstrate reasonable strong scaling. This demonstrates the viability of the Expression approach as a thread-based parallelization strategy that can be used in conjunction with traditional MPI-based domain decomposition techniques.

Figure 6 shows the parallel efficiency, obtained by dividing the speedup by the total number of processes. This figure illustrates the advantages of exposing independent parallelism (data parallel through domain decomposition and MPI and task parallel

[3]Note that for implicit time integrators, overall scalability will be influenced strongly by the scalability of the linear solver. Thus, for hybrid MPI/threaded approaches to be scalable in the context of implicit solvers, hybrid MPI/threaded linear solvers must be available.

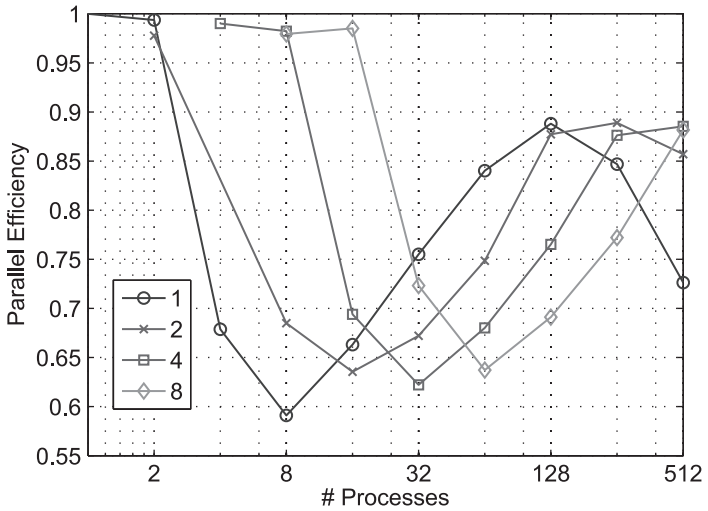

Fig. 6. Parallel efficiency (speedup/total no. processes) showing strong scaling as a function of the total number of processes running. Each line corresponds to a different number of threads replacing some of the MPI processes.

via the Expression graph). As we strong-scale this problem, cache behavior changes and optimal parallel efficiency is obtained by mixing data and task parallelism. For example, at 512 processes, one will obtain maximal parallel efficiency by using 8 threads whereas at 128 processes one should use straight MPI with one thread per node.

These results suggest the important role that cache behavior plays in achieving parallel scalability. Addressing these issues is beyond the scope of this article. However, the Expression graph approach offers a broad range of options for optimizing parallel performance by considering issues such as cache affinity, etc. For example, by setting thread affinity to specific cores, calculations of "downstream" nodes could be executed on the same core where their "parent(s)" were executed, allowing the fields to be consumed to remain in the cache where they were produced. Notably, the present approach allows such considerations to be made in a systematic, straightforward manner.

4.2.4. Scalability for Real-World Examples. As noted earlier, the structure of an Expression graph (number of Expressions and dependencies) and the computational cost of each Expression are implementation dependent. Nonetheless, we can extract representative graphs and Expression timings from various physics sets in order to understand, approximately, the speedups one can expect to see in real-world applications.

Table III shows some real-world example problems including the idealized case. The timings in these use cases are from simulations utilizing a Newton-Raphson method with analytic sensitivities. The timings, and hence speedup and parallelizability, may change with different solution algorithms or even performance tuning. Actual scalability studies for these cases were not performed; Table III shows the theoretical results.

The first example examined here comes from the solution of a three-dimensional (3D) supersonic flow of a viscous fluid with Sutherland's model for viscosity [Sutherland 1893]. A pseudo-transient solution procedure [Kelley and Keyes 1998] is used with an SUPG stabilized finite element solution [Hughes and Mallet 1986a] including a shock-capturing operator [Hughes and Mallet 1986b]. This problem involves three coupled PDEs (conservation of mass, momentum, and energy) and five degrees

Table III. Scalability and Parallelizability for Real-World Examples

Problem	$\mathcal{S}_\infty$	$\mathcal{P}$	# Unknowns
Idealized 16 Species Diffusion/Reaction System	13.7	0.94	16
3D Supersonic Flow	2.62	0.62	5
3D Heat Conduction	1.10	0.10	1
2D ALE Navier-Stokes with viscous heating	3.52	0.73	5

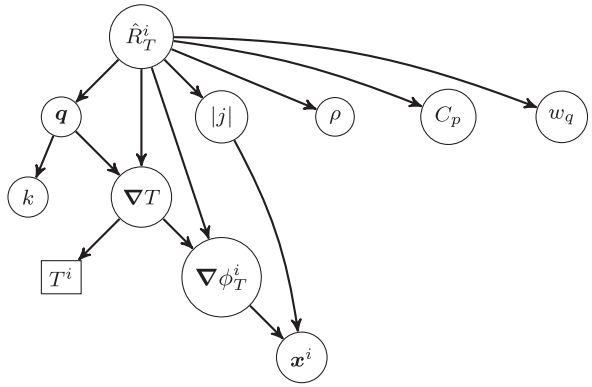

Fig. 7. Expression graph for example problem based on 3D heat conduction.

of freedom per mesh point (density, internal energy, and three components of momentum). In our implementation, the volumetric assembly operation results in an Expression graph with 91 Expressions.

The second problem analyzed simply involves the steady solution of a 3D heat conduction problem with a constant thermal conductivity on only a single degree of freedom, temperature, per mesh point. Consequently, the graph, whose structure is shown in Figure 7, contains only 11 Expressions and provides essentially no concurrency.

The final problem examined involves the solution of the 2D transient Navier-Stokes equations, transient heat conduction with convection and viscous heating, and an ALE moving mesh algorithm for free surface motion [Sackinger et al. 1996]. This problem contains five degrees of freedom per mesh point (two components of velocity, two components of mesh displacements, and pressure) and contains 60 Expressions.

The speedups observed here indicate that concurrent evaluation of Expression graphs may have only marginal performance benefit on many-core systems because these problems have little intrinsic algorithmic concurrency. This, together with the problem-dependent nature of the achievable speedup, indicate that this approach to concurrency will likely need to be used in concert with additional techniques to expose and exploit concurrency. For example, the aforementioned domain decomposition techniques can be used to parcel out subsets of the computational elements or cells to pools of threads which evaluate their own graph. On the other hand, as the complexity of the problem increases, with more physical processes being modeled (resulting in additional transport equations being solved), the breadth of the graph will also increase, providing more concurrency. Therefore, the Expression approach actually becomes more compelling as the problem complexity increases, not only from a parallelization standpoint, but from a software design and management standpoint as well.

As noted previously, the results from these use cases are dependent on many implementation choices. However, in our experience these are representative of the broad range of engineering applications commonly simulated using our codes. Two

notable exceptions are: (1) multicomponent reactive transport problems involving $O(10) - O(1000)$ independent species equations and coupled fluid flow and heat transfer (as shown in idealized form in Section 4.2.3) and (2) fully coupled discrete ordinate solution to radiative transport in participating media where the number of discrete ordinates ranges from $O(10) - O(100)$. This is because these situations result in broader graphs that provide more inherent concurrency.

5. CONCLUSIONS

We presented here a novel software design that addresses the inherent complexity of multiphysics software applications by decomposing the discretized PDE assembly process into a directed acyclic graph of Expressions. We summarized several advantages of the design as well as many of the trade-offs we have experienced in our own implementations.

By allowing the graph to determine the overall evaluation procedure, the complex logic associated with changing the PDE models is eliminated. The assembly process automatically orders the assembly procedure for efficiency and consistency. Models are evaluated once and the values are used by all dependent operators.

Expressions simplify model extension by allowing users to focus on the expression at hand instead of the overall system. Since Expressions only require direct dependencies, they do not require knowledge of the the entire equation set, discretization procedure, or the dependencies of the entire chain of Expressions. Expressions are small units of code, easy to write, easy to read, and easy to unit test in isolation. By only requiring direct dependencies, Expressions allow for exceptional code reuse across a wide range of applications.

Expressions can provide a path for algorithmic scalability. This capability is complementary and orthogonal to domain decomposition techniques. We have analyzed use cases from representative engineering simulations to show theoretical upper bounds on speedup for our current implementations. We have shown that scalability is highly dependent on the problem of interest, the choice in granularity of the graph, and the ability to avoid serialization points in the graph. The results shown here suggest that systems with large numbers of equations (i.e., species conservation equations for reacting flows) have the best chance for leveraging algorithmic scalability. Systems such as Navier-Stokes using our default implementations show minimal algorithmic scalability although the flexibility provided by the Expression system makes it an essential element for supporting this application.

Finally, Expressions provide a clean system to implement sensitivity evaluations as required by Newton-based nonlinear solution methods, optimization methods, and sensitivity analysis. By leveraging the chain rule and the dependency graph, one can automate sensitivity calculations, making the implementation trivial for users.

Ongoing work includes developing the framework to dynamically change models during a simulation, where one model suite may be substituted for another dynamically; adding or removing transport equations, modifying constitutive laws, etc., and automatically regenerating a new assembly algorithm. Additional future work includes the investigation of a similar task-graph-based design for managing higher-level algorithms for segregated solution strategies and loose coupling techniques. Finally, an exploration of merging the ideas demonstrated in this article with a symbolic frontend such as used in the Sundance code is planned.

ACKNOWLEDGMENTS

We wish to thank Mathew Hopkins for his contributions to the development of the original design.

REFERENCES

ALNAES, M. S., LOGG, A., OLGAARD, K. B., ROGNES, M. E., AND WELLS, G. N. 2011. Ufl: Unified form language. URL https://launchpad.net/ufl

AMDAHL, G. 1967. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conf. Proc. 30*, 483–485.

AUBERT, P., DI CÉSARÉ, N., AND PIRONNEAU, O. 2001. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Comp. Vis. Sci. 3*, 197–208.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2007. deal.II – A general purpose object oriented finite element library. *ACM Trans. Math. Softw. 33*, 4, 24/1–24/27.

BARTLETT, R. A., GAY, D. M., AND PHIPPS, E. T. 2006. Automatic differentiation of C++ codes for large-scale scientific computing. In *Proceedings of the International Conference on Computational Science (ICCS'06)*. V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., Lecture Notes in Computer Science, vol. 3994, Springer, 525–532.

BIRD, R. B., STEWART, W. E., AND LIGHTFOOT, B. N. 2007. *Transport Phenomena* 2 Ed. John Wiley & Sons.

BROWN, P. N. AND SAAD, Y. 1990. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Optim. 11*, 450–481.

DE ST. GERMAIN, J. D., MCCORQUODALE, J., PARKER, S., AND JOHNSON, C. 2000. Uintah: A massively parallel problem solving environment. In *Proceedings of the IEEE International Symposium on High Performance and Distributed Computing*. IEEE, 33–41.

DENNIS, JR., J. E. AND SCHNABEL, R. B. 1983. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ.

DONEA, J. AND HUERTA, A. 2003. *Finite Element Methhods for Flow Problems*. Wiley.

EDWARDS, C. H. 2006. Managing complexity in massively parallel, adaptive, multiphysics applications. *Engin. Comput. 22*, 3-4, 135–156.

FEniCS 2010. The FEnniCS Project. URL http://www.fenics.org

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

GASTON, D., NEWMAN, C., HANSEN, G., AND LEBRUN-GRANDIE, D. 2009. Moose: A parallel computational framework for coupled systems of nonlinear equations. *Nucl. Engin. Des. 239*, 10, 1768–1778.

HEROUX, M. A., BARTLETT, R. A., HOWLE, V. E., HOEKSTRA, R. J., HU, J. J., KOLDA, T. G., LEHOUCQ, R. B., LONG, K. R., PAWLOWSKI, R. P., PHIPPS, E. T., SALINGER, A. G., THORNQUIST, H. K., TUMINARO, R. S., WILLENBRING, J. M., WILLIAMS, A., AND STANLEY, K. S. 2005. An overview of the trilinos project. *ACM Trans. Math. Softw. 31*, 3, 397–423.

HORNUNG, R. AND KOHN, S. 2002. Managing application complexity in the samrai object-oriented framework. *Concurr. Comput. Pract. and Experi. 14*, 347–368.

HUGHES, T. J. R. AND MALLET, M. 1986a. A new finite element formulation for computational fluid dynamics: III. The generalized streamline operator for multidimensional advective-diffusive systems. *Comput. Meth. Appl. M. 58*, 305–328.

HUGHES, T. J. R. AND MALLET, M. 1986b. A new finite element formulation for computational fluid dynamics: IV. A discontinuity capturing operator for multidimensional advective-diffusive systems. *Comput. Meth. Appl. M. 58*, 329–336.

IESP. 2010. International exascale software project roadmap v1.0. Tech. rep., IESP. www.exascale.org

KEE, R. J., RUPLEY, F. M., MILLER, J. A., COLTRIN, M. E., GRCAR, J. F., MEEKS, E., MOFFAT, H. K., LUTZ, A. E., DIXON-LEWIS, G., SMOOKE, M. D., WARNATZ, J., EVANS, G. H., LARSON, R. S., MITCHELL, R. E., PETZOLD, L. R., REYNOLDS, W. C., CARACOTSIOS, M., STEWART, W. E., GLARBORG, P., WANG, C., AND ADIGUN, O. 2000. CHEMKIN Collection, Release 3.6. Reaction Design, Inc., San Diego, CA.

KELLEY, C. T. 2003. *Solving Nonlinear Equations with Newton's Method*. SIAM, Philadelphia, PA.

KELLEY, C. T. AND KEYES, D. E. 1998. Convergence analysis of pseudo-transient continuation. *SIAM J. Numer. Anal. 35*, 2, 508–523.

KIRBY, R. C. AND LOGG, A. 2006. A compiler for variational forms. *ACM Trans. Math. Softw. 32*, 3, 417–444.

KIRK, B. S., PETERSON, J. W., STOGNER, R. H., AND CAREY, G. F. 2006. `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engin. Comput. 22*, 3–4, 237–254. `http://dx.doi.org/10.1007/s00366-006-0049-3`

KNOLL, D. A. AND KEYES, D. E. 2004. Jacobian-Free Newton-Krylov methods: A survey of approaches and applications. *J. Comput. Phys. 193*, 2, 357–397.

LIN, P. T., SHADID, J. N., SALA, M., TUMINARO, R. S., HENNIGAN, G. L., AND HOEKSTRA, R. J. 2009. Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling. *J. Comput. Phys. 228*, 6250–6267.

LIN, P. T., SHADID, J. N., TUMINARO, R. S., SALA, M., HENNIGAN, G. L., AND PAWLOWSKI, R. P. 2010. A parallel fully-coupled algebraic multilevel preconditioner applied to multiphysics PDE applications: Drift-Diffusion, flow/transport/reaction, resistive MHD. *Int. J. Numer. Meth. Fluids 64*, 1148–1179.

LOGG, A. 2007. Automating the finite element method. *Arch. Comput. Meth. Engin. 14*, 2, 93–138.

LOGG, A. 2009. Efficient representation of computational meshes. *Int. J. Comput. Sci. Engin. 4*, 4, 283–295.

LONG, K. 2003. Sundance rapid prototyping tool for parallel PDE optimization. In *Large-Scale PDE-Constrained Optimization*, Lecture Notes in Computational Science and Engineering, vol. 30., 331–341.

LONG, K., KIRBY, R., AND VAN BLOEMEN WAANDERS, B. 2010. Unified embedded parallel finite element computations via software-based frechet differentiation. *SIAM J. Sci. Comput. 32*, 6, 3323–3351.

MUSSON, L. C., PAWLOWSKI, R. P., SALINGER, A. G., MADDEN, T. J., AND HEWETT, K. B. 2009. Multiphase reacting flow modeling of singlet oxygen generators for chemical oxygen iodine lasers. *Proc. SPIE*, vol. 7131, 71310J:1–71310J:8.

ODEN, J. T., PRUDHOMME, S., ROMKES, A., AND BAUMAN, P. T. 2006. Multiscale modeling of physical phenomena: Adaptive control of models. *SIAM J. Sci. Comput. 28*, 6, 2359–2389.

PAWLOWSKI, R. P. 2010. http://trilinos.sandia.gov/packages/phalanx/

PHIPPS, E., BARTLETT, R., GAY, D., AND HOEKSTRA, R. 2008. Large-Scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via AD. In *Advances in Automatic Differentiation*, C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J.Utke, Eds., Lecture Notes in Computational Science and Engineering.

PUNATI, N., SUTHERLAND, J., KERSTEIN, A., HAWKES, E., AND CHEN, J. 2011. An evaluation of the one-dimensional turbulence model: Comparison with direct numerical simulations of CO/H2 jets with extinction and reignition. *Proc. Combust. Inst. 33*, 1515–1522.

RALL, L. 1981. Automatic differentiation, techniques and applications. In Lecture Notes in Computer Science, vol. 120, Springer.

SACKINGER, P. A., SCHUNK, P. R., AND RAO, R. R. 1996. A Newton-Raphson pseudo-solid domain mapping technique for free and moving boundary problems: A finite element implementation. *J. Comput. Phys. 235*, 83–103.

SARKAR, V. 2009. Exascale software study: Software challenges in extreme scale systems. Tech. rep., DARPA/IPTO.

SEDGEWICK, R. 2002. *Algorithms in C++, Part 5: Graph Algorithms* 3rd Ed., Vol. 2, Addison Wesley.

SHADID, J. N., PAWLOWSKI, R. P., BANKS, J. W., CHACÓN, L., LIN, P. T., AND TUMINARO, R. S. 2010. Towards a scalable fully-implicit fully-coupled resistive MHD formulation with stabilized FE methods. *J. Comput. Phys. 229*, 20, 7649–7671.

SINNEN, O. 2007. *Task Scheduling for Parallel Systems*. Wiley Series on Parallel and Distributed Computing, John Wiley & Sons.

STEWART, J. R. AND EDWARDS, H. C. 2004. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements Anal. Des. 40*, 1599–1617.

SUTHERLAND, W. 1893. The viscosity of gases and molecular force. *Philosoph. Mag. 5*, 36, 507–531.

Trilinos 2011. The trilinos project. URL http://trilinos.sandia.gov

ULLMAN, J. 1975. NP-Complete scheduling problems. *J. Comput. Syst. Sci. 10*, 3 , 384–393.

WISSINK, A., HORNUNG, R., KOHN, S., SMITH, S., AND ELLIOTT, N. 2001. Large scale structured AMR calculations using the SAMRAI framework. Tech. rep. UCRL-JC-144755, Lawrence Livermore National Laboratory.

YOUNG, D. P., HUFFMAN, W. P., MELVIN, R. G., HILMES, C. L., AND JOHNSON, F. T. 2003. Nonlinear elimination in aerodynamic analysis and design. In *Large-Scale PDE-Constrained Optimization*, L. T. Biegler, O. Ghattas, and M. Heinkenschloss, Eds., Lecture Notes in Computational Science A, vol. 30., Springer, 17–43.

Received November 2010; revised December 2011; accepted January 2012