Loci: A Deductive Framework for Graph Based Algorithms *

Edward A. Luke

NSF Engineering Research Center
Mississippi State University
Box 9627, Mississippi State, MS 39762, USA
lush@erc.msstate.edu

Abstract. Modern computational fluid dynamics (CFD) software is complex. Often CFD simulations require complex geometries, flexible boundary conditions, multiple integrated computational models (for example, heat conduction, structural deformations, gas dynamics, etc.), as well as grid adaptation. As a result of this complexity, the correct implementation of numerical simulation components is actually less challenging than guaranteeing the correct coordination of complex component interactions. If one is to consider the development of CFD applications that reliably incorporate a broad selection of numerical models, then one must consider technologies that simplify, automate, and validate the numerical model coordination mechanisms. The Loci framework presented here addresses these issues by introducing a deductive framework for the coordination of numerical value classes constructed in C++.

1 Introduction

Recent developments in object-oriented numerics can roughly be classified into two broad branches: 1) the development of flexible high performance value classes such as MTL[1] and Blitz++[2], and 2) the development of application level frameworks or toolkits such as PetSc [3], POOMA II [4], and LPARX [5]. High performance value classes have significant reuse potential since their abstractions are generally built from well known and often used mathematical constructions such as vectors, matrices, tensors, and so on. On the other hand, application toolkits tend to be more specialized (for example, LPARX attacks AMR algorithms, while PetSc tends to be Krylov subspace centric). These two approaches are addressing fundamentally different problems: value classes reduce complexity associated with representing the mathematical structure (equations) of application components while application toolkits reduce complexity of integrating a diverse set of loosely related components.

The Loci[1] system is an application framework that seeks to reduce the complexity of assembling large-scale finite difference or finite element applications,

* This work was partially supported by NSF Cooperative agreement number ECD-8907070

1 The name Loci is derived from the framework's ability to compute the locus of application component specifications.

although it could be applied to any algorithm that is described with respect to a connectivity network or graph. The design of the Loci system recognizes that a significant portion of the complexity and bugs associated with developing large scale computational field simulations is derived from errors in control and data movement. In other words, a significant number of errors in these applications are caused by incorrect looping structures, improper calling sequences, or incorrect data movement. Many of these problems are subtle and result from gradual evolution of the application over time giving rise to inconsistencies between various application components. The Loci framework addresses these problems by automatically generating the control and data movement operations of an application from component specifications while guaranteeing a level of consistency between components. The approach taken is similar to Strand [6] except that it includes semantics of unstructured mesh computations in the rule specifications.

2 The Loci Data Model

In the Loci system, computational graphs are represented by collections of entities and collections of maps or connectivity lists. Entities represent sites where computations may occur in the graph. For example, the entities in a finite volume calculation may represent faces, cells, and nodes of the mesh. Maps may connect faces to their left and right cells, or cells to their nodes, and so forth. Values are bound to entities via the **store** construct which provides an injective mapping from entities to values. The **parameter** construct provides a singleton interface to value where a set of entities is mapped to a single value. Relationships between entities are provided by the **map** construct. The map construct can be composed with the store construct to provide an abstraction of indirection. The **constraint** construct, used to constrain computations to some subset of entities, provides an identity mapping over a given subset of entities. These constructs are illustrated in figure 1.

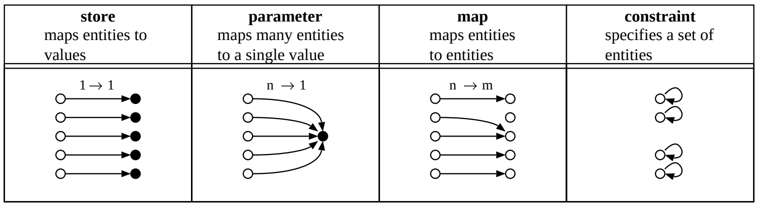

Fig. 1. Four Basic Database Constructs

These basic constructs are used to formulate a database of facts that describe the problem. Each fact provides information about some subset of entities, such as positions of nodes, or maps relating cells to nodes. Each of these facts is given an identifier that consists of a name, an iteration label, and an iteration offset. The iteration label corresponds to the nested iteration levels of a loop as in figure 2. Iteration labels form a partial order that is grounded at the stationary (constant) iteration level. Iteration offsets provide a means of accessing information at a previous iteration. The general form of a fact identifier is $\alpha\{\tau + \theta\}$ where α is the name, τ is the iteration identifier and θ is the iteration offset. For example, `pressure{n-1}` represents the identifier for pressure at the previous iteration of iterator n.

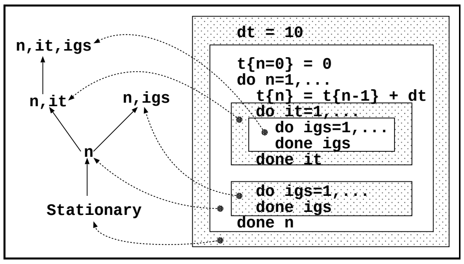

Fig. 2: Nested Iterations as a Hierarchy

3 Rule Specifications

In addition to a database of facts that includes the problem specification, a database of rules describes transformations that can be used to introduce new facts into the database. These rules correspond to fundamental computations involved in solution algorithms such as rules for evaluating areas of faces, or for solving equations of state. These rules are specified using text strings called rule signatures that describe the input stores, parameters, and maps required to perform a computation and the list of stores or parameters that it generates. Rule signatures are of the form `head <- body` where head consists of a list of variables that are generated by the application of the rule, while the body contains a list of variables that are accessed while performing the computation. The "`->`" operator is used to represent the application of a map in the access of a variable. For example, the rule signature of a computation that generates areas of faces might be given as `area<-face2nodes->position`, where "`face2nodes`" is a mapping that connects faces their defining nodes.

There are several classes of computations that are encountered in a typical unstructured grid computation. This section will identify the semantics of these computation rule classes.

3.1 Rule Constraints

Any rule that is specified can be constrained to only compute values for some subset of entities. This can be useful for specifying such things as boundary conditions as in `u<-inside->u,CONSTRAINT(Dirichlet)`. In addition to constraining rule applications, constraints also provide assertion semantics. A constraint implies that a rule must provide for every entity in the constraint. In many cases this can be used to automatically detect inconsistencies caused by incomplete information. For example, an improperly specified boundary condition yields a rule that cannot satisfy its constraint due to insufficient information.

3.2 Point-wise Rules

The most common rule in finite difference or finite element applications is the point-wise rule. The point-wise rule represents an entity by entity computation of values that are placed in the stores listed in its head. The computation specified in the rule is referentially transparent and local. The semantics of the point-wise rule application requires that an output variable can only define one value per entity. This is a variation of the single assignment semantic. If an ambiguous specification produces two rules that compute values for the same entity, it is flagged as an error during scheduling. Recursion is allowed in point-wise rules, provided that the single value per entity rule is not violated. Thus, recursion in point-wise rules is bounded to the number of entities in the simulation mesh. An example of a symmetric Gauss-Seidel method implemented through point-wise rule recursion will be illustrated in section 6.2.

3.3 Singleton Rules

A rule specification that consists exclusively of parameters for inputs and outputs is a singleton rule. Singleton rules specify a single computation that provides value for the parameters specified in its head. Like point-wise rules, these rules can not violate the uniqueness of parameter values. Any conflict of value assignment are reported as ambiguity errors during scheduling.

3.4 Reduction Rules

The Bird-Meertens formalism[7] provides a means of describing what are otherwise known as reduction operations[2]. In this abstraction, a reduction is described by a function composed of three components: a function that is applied to a set of values, an associative and commutative operator $\oplus$ that is defined on the type returned by the above mentioned function, and an identity element of operator $\oplus$, e. Thus a reduction, r, over values, $\{v_i | i \in [1, N]\}$, using function f and operator $\oplus$ is defined as

$$r = f(v_1) \oplus f(v_2) \oplus \cdots \oplus f(v_i) \oplus \cdots \oplus f(v_N). \tag{1}$$

When the reduction is evaluated using a left or right precedence rule, then a sequential evaluation is derived; however, the associative property of $\oplus$ allows for different parallel evaluation orders. For example, the set of values can be partitioned into subsets that can be evaluated concurrently as in

$$r = \{e \oplus f(v_1) \oplus f(v_2) \cdots \oplus f(v_p)\} \oplus \{e \oplus f(v_{p+1}) \oplus \cdots \oplus f(v_N)\}. \tag{2}$$

Parallel partitioning of reduction operations can be expressed when given three basic computational methods identified as unit, apply, and join as listed in table 1. The unit rule initializes a reduction variable to the identity element, the

[2] These reduction operations may be computations of global simulation parameters such as the maximum stable time-step, or of local accumulations such as summing mass contributions from cells to nodes.

apply rule "accumulates" results of a function application to a set of values, and the join operation "accumulates" the partial results. The algorithm for partitioning this reduction operation among parallel processors is accomplished by creating a copy of the reduction variable on each processor participating in the reduction. Each reduction variable is initialized to the identity, and then followed by the application of all apply rules that are in that processor's partition. Finally the partial results for each processor are reduced to the final result using join operations.

Table 1. Basic Reduction Steps

Rule Type	Function	Rule Signature
Unit Rule	$r_i^0 = e$	$r \leftarrow CONSTRAINT(v), UNIT(e)$
Apply Rule	$r_i^{j+1} = r_i^j \oplus f(v_i)$	$r \leftarrow r, v, APPLY(\oplus)$
Join Op	$r_i^{m+n} = r_i^m \oplus r_i^n$	Derived (No signature)

3.5 Iteration Rules

Iteration is defined by way of three types of rule specifications: build rules that construct the iteration, advance rules that advance the iteration, and collapse rules that terminate the iteration. This specification follows an analogy to the inductive proof in that build rules are analogous to an inductive base while advance rules are analogous to an inductive hypothesis.

For example, an iteration where a variable named `q` is iterated to a converged solution may be described by the following three rules: 1) a build rule of the form `q{n=0}<-ic`, 2) an advance rule similar to `q{n+1}<-q{n},dq{n}`, and 3) an iteration collapse rule `solution<-q{n},CONDITION(converged{n})`. Iteration in this example proceeds by initializing the first iteration, `q{n=0}`, using the build rule. Next, termination of iteration is then checked by computing `converged`, if the test succeeds then the collapse rule terminates the iteration. Finally the iteration advances in time by the repeated application of the advance rule. Note that the completion of these rules may require invoking other rules specified in the rule database. In this case, rules that compute `converged{n}` and `dq{n}` will also need to be scheduled.

To support iteration, variables that exist in lower levels of the iteration hierarchy are automatically promoted up the iteration hierarchy. Thus a variable that is computed in iteration $\{n\}$ is communicated to iteration $\{n, it\}$ automatically. In addition, rules that are specified completely at the stationary level will be promoted to any level of the hierarchy. This allows for the specification of relations that are iteration independent (for example, $p = \rho\tilde{R}T$ implies $p^n = \rho^n\tilde{R}^nT^n$).

4 Scheduling

In Loci, the mesh, boundary conditions, initial conditions, and other modeling information is stored in a database of facts using stores, parameters, maps and constraints. The application provides a set of rules that may be used to solve the given problem. Given these databases, an application is formed by searching for a goal specified by the user. This basic strategy is illustrated in figure 3. The default goal is to compute a variable called solution, which plays a role similar to `main` in C++. The search for this goal may yield three possible results: 1) there is no way to obtain the request from the given database, 2) the request can be satisfied, but not without ambiguous results or unsatisfied constraints (assertion failures), or 3) the request can be satisfied and this is the resulting schedule that generates this goal. Generally, case 1 is caused by insufficient information in the given fact database, while case 2 is caused by internal inconsistencies that usually result from conflicting database specifications.

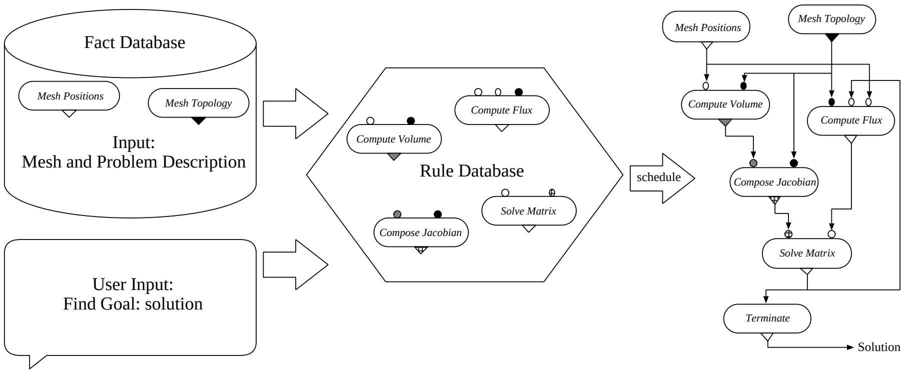

Fig. 3. The Loci methodology for application generation.

Scheduling occurs in three steps. The first step involves creating a dependency graph that connects the variables stored in the fact database to the goal using the rules in the rule database. This step involves iteratively exploring the space of known variables and determining which rules can apply, which may in turn generate new variables. Once this graph is produced it is pruned to only those rules that generate the requested goal, sorted into iteration hierarchies, and reduced to a directed acyclic graph (DAG) by clumping recursive dependency loops. The next step is an existential deduction phase which determines what attributes can be assigned to which entities. For example, the rule `p<-rho,R,T` specifies that entities that have attributes `rho`, `R`, and `T` also have the attribute `p`. The existential deduction begins with the given facts, and follows the DAG order computing the entities associated with each attribute until the goal is reached. During this existential deduction, recursive loops are iteratively evaluated until all possible attributes are generated. The result of the existential deduction phase is a concurrent schedule that obtains the requested goal. However, since it is possible that some attributes may exist for entities that don't contribute to the requested goal, a final optimization pass prunes this schedule. The pruning operation starts from the goal and works backwards through the DAG until the schedule only computes those values that are needed to provide the requested goal.

The scheduling process naturally produces a concurrent schedule. Only partitioning of entities to processors is needed to generate a schedule for parallel processors (on distributed memory architectures a communication schedule would also need to be deduced). Thus the numerical model does not have any references to parallel execution – this arises naturally from the specification.

5 Implementation

The fundamental design strategy in the Loci system has been the use of shallow inheritance hierarchies combined with templated containers and composers. The most basic data type for the Loci system is the `entitySet`, a value class that describes arbitrary sets of entities, and provides fast intersection, union, and complement operations. These entity sets are necessary for the existential deduction phase of scheduling and used for control and allocation functions. The data models described in section 2 are implemented as templated container classes. These containers provide features that facilitate their storage in the fact database and their automatic binding in rule invocations. The rules described in section 3 are implemented via a shallow inheritance hierarchy. Users create new rules by providing a constructor that creates the rule specification and a virtual member function called `compute` that performs the specified computation.

In Loci, a value class `sequence` is used to represent loop control structures. Sequences are automatically generated by the scheduling process and may describe both sequential and parallel looping structures. The template function `do_loop` provides a fast interface to the sequence control information. The combination of the two transfers the specific details of loop optimization to the Loci framework. Loci provides a `register_rule` template that adds rules to the global rule database when constructed. This allows rules in separately compiled modules to be automatically added to the global rule database.

6 Examples

The Loci system is currently being used to develop a finite-rate chemically reacting compressible flow solver for simulation of high speed and combustion flow problems. The code presently uses an implicit finite volume scheme for three dimensional unstructured mixed element grids using flux difference splitting schemes for stable solutions of flow fields containing discontinuities such as shocks. This application is currently being used by NASA Stennis space center to compute hydrogen oxygen combustion reactions to assess rocket motor measurement strategies. Two examples are provided from this flow solver implementation to illustrate some of Loci's features.

6.1 Area Computation

A generalized unstructured grid may consist of hexahedra, tetrahedra, prisms, and pyramids, which are in turn composed of triangular and quadrilateral faces. The finite volume integration method requires surface integrals of these elements which in turn requires an area evaluation of these faces. Two separate rules are required to generate areas for the quadrilateral and triangular faces. The quadrilateral faces are defined by the map variable `qnds`, while the triangular faces are defined by the map variable `tnds`. Nodal position vectors are stored in `pos`. Figure 4 illustrates the implementation of a Loci rule that computes the area of a quadrilateral face. In this rule, the area is determined by the cross product of the two diagonal vectors of the face. The constructor for this rule creates the specification "`area<-qnds->pos`", while method `compute` provides an interface for the actual computation of areas.

```
class quad_area : public pointwise_rule {
  const_store<vect3d> pos ;
  const_MapVec<4> qnds ;
  store<Area> area ;
public:
  quad_area() ;
  void calculate(Entity fc) ;
  virtual void
     compute(const sequence &seq) ;
} ;

quad_area::quad_area() {
  name_store("pos",pos) ;
  name_store("qnds",qnds) ;
  name_store("area",area) ;

  input("qnds->pos") ;
  output("area") ;
}
```

```
inline void quad_area::calculate(Entity fc) {
  vect3d dv1 = pos[qnds[fc][2]]-pos[qnds[fc][0]] ;
  vect3d dv2 = pos[qnds[fc][1]]-pos[qnds[fc][3]] ;

  area[fc].n = cross(dv1,dv2) ;
  real sada = sqrt(dot(area[fc].n,area[fc].n)) ;
  area[fc].n *= 1./(sada+EPSILON) ;
  area[fc].magnitude = 0.5*sada ;
}


// calculate areas for sequence of entities
void quad_area::compute(const sequence &seq) {
  do_loop(seq,this,&quad_area::calculate) ;
}

// register rule in global rule database
register_rule<quad_area> quad_area_registration ;
```

Fig. 4. Quadrilateral-face area computation rule in Loci

6.2 Gauss-Seidel Iteration

The linear system solution required by the implicit time integration in the solver utilizes a Gauss-Seidel iterative method. This method solves the equation $Ax = b$ by iteratively evaluating the equation $x^{k+1} = (L + D)^{-1}(Ux^k + b)$, where $A = L + D + U$, and L, U, and D are the lower, diagonal and upper matrices respectively. This method can be written as the point-wise recurrence relation

$$x_i^{k+1} = D_i^{-1}\left(b_i - \sum_{j=1}^{i-1} L_{ij}x_j^{k+1} + \sum_{j=i+1}^{n} U_{ij}x_j^k\right).$$

In the unstructured flow solver, the matrix, A, is formed from the connections between cells. These connections represent faces in the mesh. The data structure is constructed by first identifying a left and right cell to every internal face with maps `cl` and `cr`. The faces are constructed such that `cl` always points to the lowest color cell (in the matrix coloring). Mappings to the lower and upper parts of the matrix are formed by inverting the maps `cl` and `cr` such that `lower` = *inverse*(`cr`) and `upper` = *inverse*(`cl`).[3] In this data structure, the components of the lower matrix are stored at faces in variables called `L` and `U` while the diagonal is stored at cells in a variable called `D`. Using this data structure, the above recurrence relation form of the Gauss-Seidel algorithm becomes the recursive point-wise rule

```
x{k+1}<-D,b,lower->L,lower->cl->x{k+1},upper->U,upper->cr->x{k}.
```

Loci schedules recursive rules by repeatedly evaluating the rule until all possible entities are computed. What does Loci do for a red-black coloring? In this case all red cells have only left faces pointing to them, while all black cells have only right faces pointing to them as illustrated in figure 5. Thus red cells have a null mapping for `lower`, while black cells have a null mapping for `upper`. In the first step of the repeated evaluation, Loci will be able to evaluate this rule only for red cells since the lower map is null only for these cells and no entities have the attribute `x{k+1}`. For the second iteration of this scheduling process Loci will be able to apply this rule for all black cells since the red cells produced the necessary `x{k+1}` attribute. A following iteration will determine that no new attributes can be assigned and the schedule will be complete. A two step concurrent red-black algorithm will be the result. Interestingly, if instead a regular mesh is given with a canonical coloring, then Loci will derive the concurrent hyper-plane algorithm used in the NAS parallel benchmark LU[8].

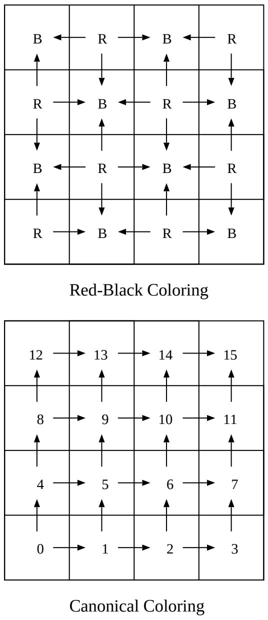

Fig. 5: Mesh Colorings

7 Performance

The performance of Loci is measured for the finite rate chemistry application using the hardware counters on the SGI 195mhz R10K processor. A medium sized grid for a converging diverging rocket nozzle consisting of 15,488 cells is used in the performance measurement. Two chemistry models are used in the simulation: 1) an ideal gas model, and 2) a 6 species 28 reaction hydrogen-oxygen reaction kinetics model. In addition, the performance of a structured ideal gas solver implemented in Fortran is also measured to provide a performance baseline. The performance is based on the ratio of floating point operations obtained from the R10K hardware counters to an overall execution time including Loci schedule generation. These results, listed in table 2, indicate that a deductive scheduling system combined with C++ templates and value classes can produce performance similar to Fortran based applications.

[3] An inverse of a mapping is simply its transpose.

Table 2. Loci Performance Results on Nozzle Simulation

Code	Model	Flops	Time (sec)	Rate (M-Flops)
Loci	Ideal Gas	41.41e9	1174	35.3
Loci	H2-O2 Kinetics	252.89e9	5803	43.6
Fortran	Ideal Gas	33.80e9	749	45.1

8 Summary

This paper outlined a programming model based on a data parallel logic programming strategy. The programming model has the advantage that it can coordinate various numerical models while enforcing many consistency guarantees. A side effect of this model is that parallel schedules can be generated from the specification. A preliminary implementation of this model in C++ demonstrates that the approach can be competitive with Fortran in performance while simultaneously providing consistency guarantees Fortran lacks.

References

1. J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
2. T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
3. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. *Modern Software Tools in Scientific Computing*, 1997.
4. S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array Design and Expression Evaluation in POOMA II. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
5. Scott R. Kohn. *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations.* PhD thesis, University of California, San Diego, 1995.
6. I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.
7. R. S. Bird and L. Meertens. Two Exercises Found in a Book on Algorithmics. In L. G. L. T. Meertens, editor, *Program Specification and Transformation*, pages pp 451–457. North-Holland, 1987.
8. E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga. Solution of regular, sparse triangular linear systems on vector and distributed-memory multicomputers. Technical report, NASA Ames Research Center, Report RNR-93-007.