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 system 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], Blitz++[2], and POOMA II [3], and 2) the development of application level frameworks or toolkits such as PetSc [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 many algorithms that are 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 transfers. 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 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.

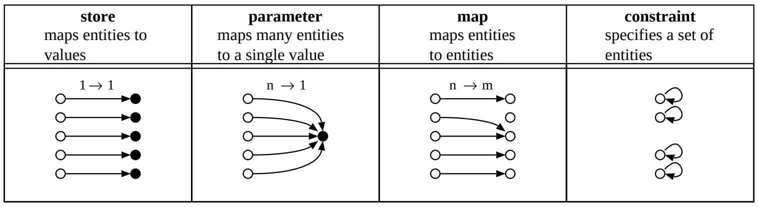

Fig. 1. Four Basic Database Constructs

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.

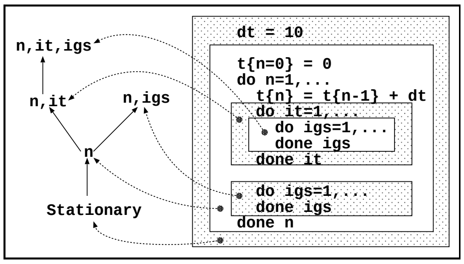

Fig. 2: Nested Iterations as a Hierarchy

The "->" operator is used to represent the application of a map in the access of values. Thus the indirection typically encountered in unstructured calculations can be encoded: accessing the value for the left side of a face identified by entity `f` may be coded using C arrays as `value[left[f]]`, whereas in Loci this access of `value` through the mapping `left` is written as `left->value`. Figure 3 illustrates the aggregate perspective of the indirection operation represented by the "->" operator.

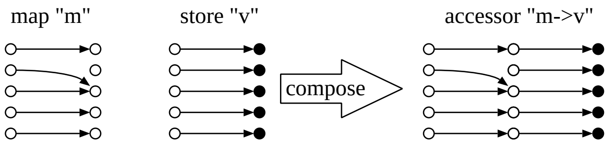

Fig. 3. An Illustration of the Mapping Operation '`m->v`'

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. For example, the rule signature `p<-rho, T, R` represents that a value for pressure (`p`) is provided when values for density (`rho`), temperature (`T`), and gas constant (`R`) are present.

A rule signature may also contain the mapping operator "->" to represent the use of indirection in a computation. For example, the rule signature of a computation that generates areas of faces utilizing positions of related nodes may be given as `area<-face2nodes->position`, where "`face2nodes`" is a mapping that connects faces to their defining nodes. Note that this rule may represent a calculation for a single face or for all of the faces in the mesh. It states that for all entities that satisfy all the properties given in the body, a computation can be performed to produce a value given in the head.

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. For example, if one wanted to specify the value of variable `u` at nodes where the Dirichlet boundary condition are applied, this might be specified by the rule signature: `u<-CONSTRAINT(Dirichlet)`. In addition to constraining rule applications, constraints also provide assertion semantics: a constraint implies that a rule must provide values for every entity in the constraint. In many cases this can be used to automatically detect inconsistencies caused by incomplete specifications. In addition, Loci identifies over-specification by requiring that each entity may have only one given value. Thus, if a boundary condition is applied to an interior node of the domain, then an error would result caused by a conflict between the boundary condition and stencil specifications.

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 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

[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.

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

$$r = f(v_1) \oplus f(v_2) \oplus \cdots \oplus f(v_i) \oplus \cdots \oplus f(v_N). \quad (1)$$

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

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

Note that the addition of the identity element, e, is used to indicate the requirement for the initialization of value on each processor: all reductions begin with the identity element. 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 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. Reduction Specification in Loci

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)

3.4 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 which computes values for `q` for the next iteration (`{n+1}`) given current iteration values at time level `{n}`. 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 4. 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 (under-specification), while case 2 is caused by internal inconsistencies that usually result from conflicting database specifications (over-specification).

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

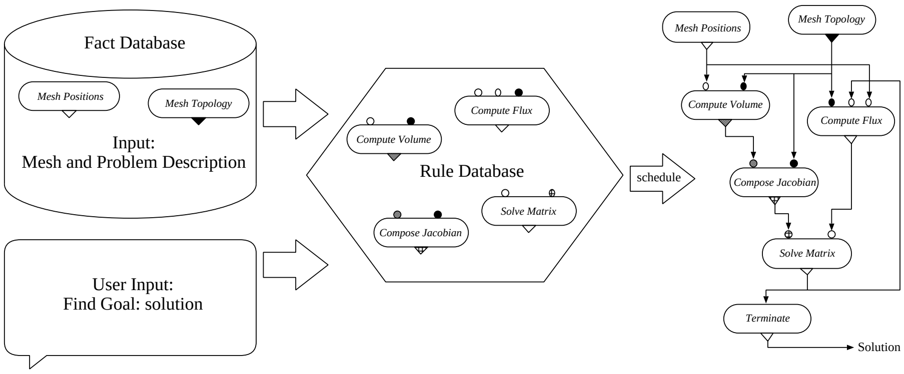

Fig. 4. The Loci Methodology for Automatic Application Generation.

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 do not 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.

A high-performance implementation strategy is based on moving all run-time polymorphic behavior (virtual functions and RTTI) to the level of collections of entities. This is facilitated by the scheduler which generates a sequence of rule applications over sets of entities. Each rule implementation provides a virtual member function `compute` that provides a collection level interface to rule evaluation. The scheduler generates a partial order of entity computations for each rule invocation represented by the `sequence` class. The developer of a rule implementation provides an inlined member function that performs the specific computations required by a single entity. A templated composer function, `do_loop`, provides the interface between the per-entity computation provided by the user and the collection-level request made by the execution schedule (in the form of an instance of `sequence`). This strategy provides a powerful division of labor with respect to optimization: while the scheduler can concern itself with global analysis such as the identification of concurrent or cache optimized evaluation orderings, the `do_loop` composer can provide the optimal looping interface (`e.g.` loop unrolling or parallelization directives). Most importantly, all of these optimizations can occur without change to the numerical algorithm as specified in the rule definition.

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 at the 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 5 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 {  | inline void quad_area::calculate(Entity fc) {
  const_store<vect3d> pos ;                |   vect3d dv1 = pos[qnds[fc][2]]-pos[qnds[fc][0]] ;
  const_MapVec<4> qnds ;                   |   vect3d dv2 = pos[qnds[fc][1]]-pos[qnds[fc][3]] ;
  store<Area> area ;                       |
public:                                    |   area[fc].n = cross(dv1,dv2) ;
  quad_area() ;                            |   real sada = sqrt(dot(area[fc].n,area[fc].n)) ;
  void calculate(Entity fc) ;              |   area[fc].n *= 1./(sada+EPSILON) ;
  virtual void                             |   area[fc].magnitude = 0.5*sada ;
     compute(const sequence &seq) ;        | }
} ;                                        |
                                           |
quad_area::quad_area() {                   | // calculate areas for sequence of entities
  name_store("pos",pos) ;                  | void quad_area::compute(const sequence &seq) {
  name_store("qnds",qnds) ;                |   do_loop(seq,this,&quad_area::calculate) ;
  name_store("area",area) ;                | }
                                           |
  input("qnds->pos") ;                     | // register rule in global rule database
  output("area") ;                         | register_rule<quad_area> quad_area_registration ;
}                                          |
```

Fig. 5. 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, D, and U 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`. A more detailed description of the matrix data-structure is provided in [8]. 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}.
```

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

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 6. 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[9].

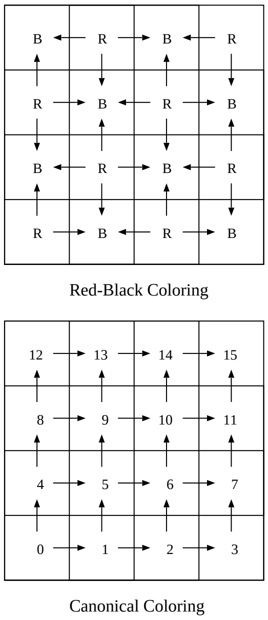

Fig. 6: Mesh Colorings

7 Performance

Evaluating the performance of the finite-rate chemistry application developed in Loci is complicated by the fact that there is no application which exactly matches its particular design goals. It is important, however, to obtain a performance baseline to determine if the approach has an unacceptable performance penalty. Unfortunately, simple linear algebra based benchmarks over-predict performance due to their unrealistically high percentage of multiply-add operations. In contrast, the finite-rate chemistry solver spends less than 20 percent of its execution time in matrix solution. To provide a more reasonable performance baseline, the performance of a Fortran ideal-gas solver[10] is measured. The basic algorithms used in both solvers are similar, although a more robust, but more expensive, method for Jacobian computations is applied in the finite-rate chemistry application.

The performance of a serial Loci implementation 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. The performance of the Fortran ideal-gas solver 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. Since the operations performed by scheduling is not included in the operation measurement, scheduling cost can only decrease measured performance. 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. The lower performance of the ideal-gas model is a result of a design that supports flexible chemistry models rather than particular overheads of the Loci system.

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 sequential implementation of this model in C++ demonstrates that the approach can be competitive with Fortran in performance while simultaneously providing consistency guarantees traditional design approaches lack.

The current implementation provides an application framework where high-performance value classes are used to describe entity-based computations, while the Loci scheduling system provides an automatic assembly of these fundamental descriptions into an application-level schedule. In this sense, the Loci system is both a library and a run-time system for the development of high performance graph-based applications. Although the present implementation focuses on the composition of entity-based computations, the same strategy could be applied to the case of aggregate computations (such as might be performed by a PetSc linear system solver, or an external commercial simulation application). However, these cases would need special treatment by the scheduler to ensure that the computations remained aggregate (for example, a matrix can not be inverted in segments), and may not be subject to the same parallelization opportunities.

There is a design argument that has been subtly referred to throughout this paper: in scientific computing, the needs of application-level structures are fundamentally different than that of the value-level. While recent advances in value class design are providing clear benefits, the problems associated with application-level structures have been less clearly addressed. A likely source of difficulties at this level is due to complex interactions between application components that do not fit neatly into a hierarchical, object-oriented structure, but rather appear to be more readily represented by axiomatic statements (*e.g.*

pressure exists whenever density, temperature, and gas constant exist). By this argument, a multi-paradigm approach offers the best of both worlds. In the case of Loci, C++ provides powerful mechanisms for creating high performance value classes, while Loci provides a powerful logic-based scheduling infrastructure for coordinating application-level program structures.

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. ISCOPE, Springer-Verlag, 1998.
3. 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. ISCOPE, Springer-Verlag, 1998.
4. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
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. A. Luke. *A Rule-Based Specification System for Computational Fluid Dynamics.* PhD thesis, Mississippi State University, December 1999.
9. 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, NASA Ames Research Center, Moffett Field CA 94035, April 1993.
10. J.M. Janus. *Advanced 3-D Algorithm for Turbomachinery.* PhD thesis, Mississippi State University, May 1989.