JFP **15** (3): 477–502, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005514 Printed in the United Kingdom

Loci: a rule-based framework for parallel multi-disciplinary simulation synthesis

EDWARD A. LUKE

Department of Computer Science and Engineering, Mississippi State University, MS 39762, USA
(*e-mail:* luke@cse.msstate.edu)

THOMAS GEORGE

Department of Computer Science, Texas A & M University, TX 77843-3112, USA
(*e-mail:* tgeorge@cs.tamu.edu)

Abstract

We present a rule-based framework for the development of scalable parallel high performance simulations for a broad class of scientific applications (with particular emphasis on continuum mechanics). We take a pragmatic approach to our programming abstractions by implementing structures that are used frequently and have common high performance implementations on distributed memory architectures. The resulting framework borrows heavily from rule-based systems for relational database models, however limiting the scope to those parts that have obvious high performance implementation. Using our approach, we demonstrate predictable performance behavior and efficient utilization of large scale distributed memory architectures on problems of significant complexity involving multiple disciplines.

1 Introduction

The current state-of-the-art in the development of scalable parallel numerical applications is primarily achieved using traditional sequential languages and library calls to the message passing libraries such as MPI (1997), PVM (Geist *et al*., 1994), or BSPLib (Hill *et al*., 1997). These applications are typically executed on distributed memory machines. Although this programming solution has demonstrated great success in numerous applications, this model remains tedious, and costly in practice. Often developers of such applications must pay a great deal of attention to parallel communication and resource management issues when time may be spent more productively on numerical methods. There have been many attempts to automate parallel application development with various degrees of success. The best known of these, High Performance Fortran (HPF, 1993), consists of a set of data-parallel extensions to Fortran. However, this approach failed to reliably deliver portable performance largely attributable to the lack of a consistent performance model. An alternative but less scalable approach that has been somewhat more successful in the context of shared memory machines is OpenMP (1997, 1998). OpenMP provides a set of loop parallelizing directives that allows users to incrementally parallelize sequential Fortran, C, and C++ codes and works reasonably well on tens of processors. Although not currently popular, we should mention the well known declarative approach to parallel numerical computing, SISAL (Cann, 1992). SISAL produced competitive performance results in the development of serial and shared memory applications in its day. Unfortunately, a scalable distributed memory implementation of SISAL was never fully demonstrated (Pande *et al.*, 1994).

What is a numerical software developer to use that would allow them to focus a little less on parallel processing and more on numerical model development? One strategy might be to use a coordination language such as PCN (Foster *et al.*, 1992). A coordination language holds the promise of allowing users to develop parallel or distributed application using the languages, libraries, or tools that are most appropriate for any particular computational task while allowing the coordination language to manage resources of a parallel or distributed implementation. These coordination languages usually have relatively large overheads making them impractical to apply at a fine-grain level (e.g. each computational entity). Unfortunately, the work required in the manual management of entity aggregation and associated inferred intra-aggregation data access is similar to the work required to parallelize applications. Thus, coordination languages are limited to contexts where the main objective is hooking together larger application level components where the relative overhead is suitably small. It would be helpful to have a coordination language that could manage aggregation as well. To address this issue, we have developed the *Loci*[1] coordination framework that is capable of coordinating and aggregating computations. The framework is unique in the sense that it uses a relational model of computations that naturally facilitates a fine-grain description that can be automatically aggregated using appropriate relational queries. A declarative logic-based strategy is used to define computations using an abstraction that is roughly a subset of the *Datalog* (Ullman, 1988) language for relational database queries. This facilitates an efficient and automatic data-parallel programming model that allows for coordination of computations that have fine-grained descriptions. Here we use fine-grained descriptions to indicate that computational abstractions are about the natural computational entities of the respective algorithms and not artificial aggregations created to achieve performance. Although the resulting programming model is not general, it is able to describe a wide range of numerical algorithms. We have implemented within the framework both finite-volume fluid mechanics solvers as well as finite-element solid mechanics models. In addition, we were able to seamlessly and easily couple these models to produce scalable parallel multi-disciplinary simulations.

2 Related work

Logical and functional declarative models have long been touted as having particular benefits as parallel programming languages. Since the order of computation is not explicitly specified, but rather emerges from data dependencies inherent in the

[1] *Loci* is not an acronym. The name *Loci* was given to represent its approach of discovering computation loop bounds at the locus of computational inputs.

algorithm description, it is argued that declarative languages provide a naturally parallel description of algorithms. In addition, referential transparency offered by these programming paradigms makes it easier to reason about parallel execution and to design optimizing transformations. In the early nineties there were several successful languages such as SISAL (Cann, 1992), Strand (Foster & Taylor, 1990), and NESL (Blelloch *et al.*, 1993) that had designs that were particularly suited to high performance computational architectures. As mentioned already, these efforts have not adapted well to more recent distributed memory architectures. More recent research on distributed memory architectures has included distributed implementations of declarative parallel languages such as Haskell (Trinder *et al.*, 1998), Eden (Breitinger *et al.*, 1997) and GdH (Pointon *et al.*, 2000). Many of these distributed memory approaches have included some form of explicit concurrency control in the form of embedded coordination languages (Trinder *et al.*, 2002). Unlike SISAL, the applicability of these efforts to scientific high performance computation is not certain.

An alternative approach to increasing automation in high performance computation is the development of domain specific computational frameworks such as Overture (Bassetti *et al.*, 1998) for computations on overset grids, or the Cactus Code (Gabrielle *et al.*, 2000) for Cartesian adaptive mesh refinement solvers. These frameworks provide facilities for quickly building applications and automatically managing parallel resources. However, these frameworks typically make specific assumptions about the approach used to solve a given problem. If the numerical strategy is sufficiently similar to the model used in the framework they can be quite helpful, but each appears to have rather narrow range of applicability. More recently, work on the Common Component Architecture (CCA) (Armstrong *et al.*, 1999; Govindaraju *et al.*, 2003) may provide a more generic approach for the creation of reusable components for high performance numerical simulations. To facilitate more efficient interactions among numerical components, a new Scientific Interface Description Language, SIDL (Cleary *et al.*, 1998), is used as it facilitates low overhead component interfaces, particularly when components happen to reside on the same architecture. We are beginning to see applications developed using these new standards for scalable linear system solvers, laser plasma simulations and adaptive mesh refinement applications (CASC, 2003).

A unique idea that is central to the approach described in this paper is that scientific computations are appropriately modeled using relational abstractions. Though this is not a common perspective, we have found a couple of examples where this observation has also been made. For example, the Janus (Gerlach *et al.*, 1998) C++ library for developing high performance simulations uses relations as an abstraction for the construction and manipulation of parallel data-structures. A much more bold use of the relational abstraction for high performance computations is employed in the development of a compiler for high performance sparse matrix solvers (Kotlyar *et al.*, 1997). Their compiler allows users to develop algorithms for sparse matrices by mapping much more terse dense matrix algorithms onto descriptions of sparse matrix data-structures. The compiler used relational representations of sparse data-structures facilitating the transformation of dense matrix loops into relational queries. These queries are then optimized in the compilation of parallel high performance sparse matrix solvers custom-tuned to arbitrary sparse storage formats. Although their focus is on the compilation of optimized sparse matrix algorithms from much more compact dense matrix algorithm descriptions, the notation and strategy used to describe their optimizations is strikingly similar to the strategies used in the *Loci* framework.

3 Key abstractions

A distinct feature of many numerical computations is the relative stasis of data-structures. In numerical computations, usually the same flow of information is repeatedly reused with different values as many iterations proceed. For example, in the conjugate gradient iterative matrix solver, the bulk of the algorithm is involved in sparse matrix-vector multiply operations on the same non-zero matrix structure. In fluid dynamic simulations, the same grid is reused over many time-steps, and within a time-step, many values are overlayed on the entities that form the discrete representation of space. Computations of this form end up having radically different run-time system requirements than more dynamic algorithms such as those found in typical sorting or computational geometry applications.

Because of this data-structure stasis, it is usually wise to spend significant computational effort on the organization of data for efficient resource utilization given that this cost shall be recovered during the value computation phase. Since the data-relationships change fairly infrequently, any expensive optimizations that are applied in organizing the representation of this data are easily amortized over many following iterations providing a much higher overall performance. Usually techniques applied to more dynamic algorithms ignore this distinction, and so, many systems that are tuned on dynamic algorithms where data relationships change frequently fail to perform well in numerical contexts.

We associate the relatively static data-structures as being primarily responsible for several unique attributes of most numerical software: usually this software is array-based, tends to use pointers infrequently, and has a distinct division between symbolic (data-structure assembly) and computational phases. Based on this observation, we identify two attributes of software abstractions that facilitate high performance numerical computation: (1) the abstraction should facilitate a staged execution model that alternates between symbolic and computational phases, and (2) the abstraction should support efficient mechanisms for overlaying new values on static computational data-structures. The first attribute implies that data-structures should exist in two distinct modes: one which is optimized for dynamic editing, and a second which is optimized for repeated accesses to the same static structure. A suitable "sorting" procedure is placed between these two stages to optimize for efficient access when in the static form. The second attributes implies that an efficient overlay mechanism is required to interface to the static form of these data-structures since most values have lifetimes that are relatively short compared with the static data-structure.

3.1 Two-phase model

The term two-phase was coined in the Janus library (Gerlach *et al.*, 1998) to describe containers that have two phases: 1) a write-only phase where the container

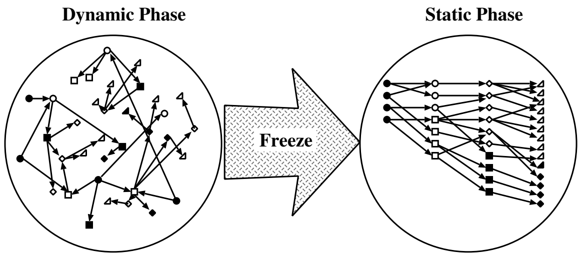


Fig. 1. An illustration of the two-phase data-structure concept.

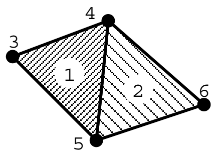
was created, and 2) a read-only phase where the container was used. The spirit of this idea is applied in *Loci* in a slightly generalized form. In *Loci*, data-structures exist in two phases: A dynamic phase where data and relationships are created in an arbitrary fashion and a static phase where data are accessed in an orderly fashion at lower cost. The dynamic phase supports fast insertion and editing with a relatively high access cost and the static phase supports dense memory management schemes using arrays for low access costs. The basic distinction is illustrated in Figure 1. In the dynamic-phase, the data-structure supports dynamic association of various components. The dynamic-phase data-structures use hash-tables to support dynamic and independent insertion of new entities and relationships. The static-phase data-structures use arrays that provide fast and efficient access to consecutively numbered entities. The process of converting from the dynamic to static phases is essentially a sorting process whereby entities of like type are grouped together to facilitate efficient random access.

3.2 An abstract model for containers

We find that a suitable abstraction for value containers in irregular computations is bags of tuples. The first entry of the tuple is the entity identifier. The second entry may either be a value or another entity identifier. Using this abstraction, it is possible to overlay values on existing data-structures by creating tuples that share the same entity identifiers as used in the tuples that define the irregular data-structure. When implemented using hash tables, the first entry of the tuple is hashed. Since hashed containers have no ordering associated cost, entity identifiers may be allocated as needed without a significant performance penalty. After freezing, when entity identifiers are numbered in contiguous allocations with respect to attribute assignments, arrays are used to store these values providing fast random access for computations. In addition, instead of using iterators to access these containers, entity sets can be used to describe aggregate accesses. These entity sets can be used to determine aggregated communication requirements of an aggregate access. Thus, this abstraction facilitates an efficient approach to manipulating distributed data-structures.

4 Logic Abstractions for Numeric Computations

We find that a logic abstraction is useful for describing numerical computations as it facilitates decoupling values from computations of values. For example, a



Id	n1	n2	n3
1	3	4	5
2	5	4	6

Id	pos
3	(0.00, 0.00)
4	(2.00, 0.75)
5	(1.75,-1.75)
6	(4.00,-1.00)

Fig. 2. A relational representation of an unstructured computational mesh.

description of the computation of momentum from velocity and mass should not contain assumptions about what values are required to compute velocity. Momentum is a product of mass and velocity regardless of context. A rule-based specification satisfies this requirement. When combined with the observation that a relational model is an appropriate model for representing the role of data-structures in these irregular numerical computations, it becomes interesting to consider a knowledge-base model of computing similar to *Datalog* (Ullman, 1988).

Consider how we may represent an unstructured computational mesh that is created from a triangular tessellation. A common way to represent such a mesh is in a tabular form. Spatial coordinates are associated with the vertices and triangular elements of the tessellation are represented by listing the three vertex identifiers that form any given triangle, for example see Figure 2. Thus a triangular tessellation of a two dimensional space may be represented using four relations: one relation that associates positions with vertex identifiers and three relations that associate each triangle with its defining three vertices.

Let us consider how one performs computations once given a data structure as above. Typically each relation described in figure 2 is represented as an array, where `n1`, `n2`, and `n3` are index arrays (arrays containing indices) and `pos` is an array containing positions. Suppose that we require the areas of the triangles for a computation, we could provide a subroutine for computing areas such as described in the following pseudo-code:

```
COMPUTE-AREAS($n1, n2, n3, pos, set$)
1  for $i \in set$
2     do $vec1 \leftarrow pos[n2[i]] - pos[n1[i]]$
3        $vec2 \leftarrow pos[n3[i]] - pos[n1[i]]$
4        $area[i] \leftarrow \frac{1}{2}(vec1 \times vec2)$
5  return $area$
```

In this subroutine line one represents a loop over the triangles for which we need to compute areas. Lines 2 and 3 define the vectors of two edges of the triangle, and line 4 computes the area of the triangle using the cross product of edge vectors. Note that to use this subroutine, we need to not only pass in the appropriate arrays, but we need to also pass in the appropriate *set* of indices. In the case of computing the area for the two triangles in Figure 2 we would need to pass in the set $\{1, 2\}$. In the development of complex applications, many similar subroutines are chained together. Coordinating these subroutine calls and the sets passed to them can become a rather complex task. For example, we may have both triangular and quadrilateral faces, each having different techniques for computing areas and requiring different sets of indices. For distributed memory applications, the complexity increases further as some subset of triangles may be computed on other processors and communicated, while others may be computed locally by the above subroutine. To simplify this process, we would like to annotate these subroutines so that the process of assembling them into an application is automatic. In addition, we would like the *set* passed to the subroutine to be consistent with the subroutines particular constraints (e.g. it would be incorrect to call the above subroutine with the set $\{3, 4, 5, 6\}$ when using the tables shown in Figure 2).

In *Loci*, we augment the above subroutine with a data access description. This data access description allows *Loci* to automatically schedule the execution of the subroutine when it is required by other computations and to automatically compute the set of indices that are passed to the loop. We derive these sets and a schedule of subroutine calls using a logic programming model similar to *Datalog*.

4.1 The applicability of the Datalog logic model

Since the logic model used in Loci is closely related to *Datalog* we find it useful to first describe how the above subroutine would be implemented in *Datalog* and what we might gain from such a description. In later sections, we will describe how Loci differs from *Datalog*. A detailed description of the *Datalog* logic model can be found in chapter 3 of Ullman's text (Ullman, 1988). We will only give a cursory description of this model here.

The *Datalog* model defines logic operations on relations. Logical rules similar to Prolog are used to describe how new relations can be derived from other relations. We typically start with an initial set of relations that are stored in a database. This initial set of relations is called the extensional database (EDB). In addition to the EDB relations, new relations may be created through the application of logical rules. These derived relations become part of the intensional database (IDB). Also note, as with *Loci*, the predicate symbols of *Datalog* rules are relations in either the intensional or extensional database.

At this point we can describe how the area computation mentioned earlier can be expressed as a *Datalog* rule. First we note that the data described in Figure 2 can be described by a set of four EDB relations given as `n1`, `n2`, `n3`, and `pos`. Given these relations, we can define the area of the triangles as an IDB relation which we give here as `area`. Given this, the triangle area computation can be described by the *Datalog* rule:

$$\begin{aligned} \mathtt{area(T,A)} \;:-\; & \mathtt{n1(T,I1)} \;\&\; \mathtt{pos(I1,P1)} \;\&\\ & \mathtt{n2(T,I2)} \;\&\; \mathtt{pos(I2,P2)} \;\&\\ & \mathtt{n3(T,I3)} \;\&\; \mathtt{pos(I3,P3)} \;\&\\ & \mathtt{A} = \frac{1}{2}(\mathtt{P2}-\mathtt{P1}) \times (\mathtt{P3}-\mathtt{P1}). \end{aligned}$$

In this rule we see that the derived relation, `area(T,A)`, is given as the head of the rule; and, the input to the computation is given by relations in the body. Note that we use a natural join for example between relations `n1(T,I1)` and `pos(I1,P1)` to achieve the same result as indirect access used in the pseudo-code for the area computation (as $pos[n1[i]]$ in *COMPUTE-AREAS*). This rule performs the exact same computation as the previous pseudo-code, however it has the advantage that it is context independent. For example, it can become part of knowledge-base of rules that can be applied to a certain class of data-structures. In such a form, values for areas emerge wherever sufficient information is present to compute them, while if there is no need for areas the rule is just another relationship in the body of descriptive rules of data properties. Unlike the subroutine version, there is no need to express the control for the loop as this spontaneously emerges from the logic model. In more complex problems, the advantage of the approach becomes more obvious as rather sophisticated selections of subsets of relations can be created. For example, the specification of a particular computation that is only valid at the interface between two different models (say fluid and solid materials) can be automatically applied to the interface simply by the logical implication of where data will be obtained. (For example, only at the interface would both fluid and solid properties be available.)

Note that we are not interested in re-implementing *Datalog* for scientific computations (although such a goal would likely prove to be interesting). Instead we are interested in capturing a higher level of abstraction for the methodologies that are presently used in numerical computations, particularly those that operate on irregular grids. Therefore, we have constructed a set of abstractions that is largely a subset of *Datalog*. It is important to identify this subset since it represents natural abstractions of approaches that have already been applied in the development of current high performance software. Thus, we expect that these abstractions will support similar performance to traditional message passing methodologies, only with greater levels of automation. By using this approach, we expect that the resulting abstractions will provide reasonable performance without requiring user explicit knowledge of particular high-performance idioms. If instead we attempted to maximize expressiveness, then the user would be burdened with remembering which peculiar subset of the programming model achieves predictable high performance. The remainder of this section will provide a quick overview of what specific abstractions are provided in *Loci*. Since the data model used in Loci is similar to the *Datalog* model, we will express these abstractions in comparison and contrast to *Datalog*.

4.2 Relations in Loci

In *Datalog* a relation may be an arbitrary k-arity tuple with no specific constraints on the types of the fields. In *Loci* however, we only support binary relations. In addition, the first entry in our tuple is always an index (or a key, or entity Id), while the second field may be either an index type or a value type. A value type is any data that does not contain any explicit index information (for example: the positions in

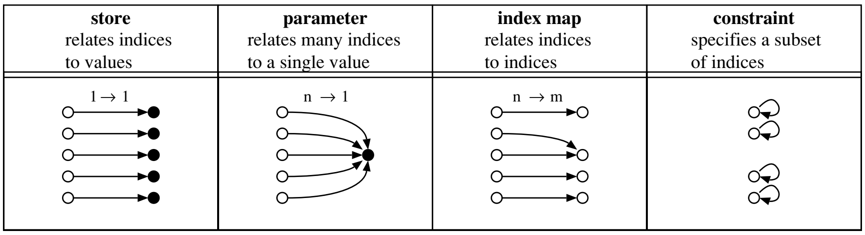

store
relates indices to values
parameter
relates many indices to a single value
index map
relates indices to indices
constraint
specifies a subset of indices
1 → 1
n → 1
n → m

Fig. 3. The four types of binary relations found in Loci.

the triangular mesh). Relations where the second field is a value type may come in two forms: a *store* or a *parameter*. The *store* is simply a relationship between indices and values, while the *parameter* describes a value that is shared among a group of indices. If the second field is an index, then the relation is called an *index map*. We use *index maps* to create relationships between entities in the data-structure, such as the `n1` relation used in the triangular mesh example. Finally, for the restricted case of the identity *index map*, we define a *constraint*. The *constraint* provides a mechanism to associate a valueless property with some subset of entities. These relations are illustrated in Figure 3 where we denote indices as open circles and values as filled circles.

In *Loci*, these four types of relations are provided by the user as EDB relations. The user (or an application program) also provides a set of transformation rules (which are simply documented subroutines). These rules can be used to create new IDB relations. However, since we are only concerning ourselves with programs that have static data-structures, we only consider rules that compute either *store* or *parameter* relations.

4.3 Loci rule signatures

In *Loci*, we add documentation to the *COMPUTE-AREAS* subroutine such that it can be viewed as an implication rule on relations. Once this specification is in place, the subroutine can be automatically scheduled in response to user queries. The Loci planner automatically computes sets of indices that are passed to the subroutines to be used for computation loop bounds. Since these sets are computed using the logic model of *Datalog*, they guaranteed to be consistent with the defined data-structures. We use the rule signature to document subroutine inputs and outputs. The grammar for the rule signature, given in Figure 4, is defined by a head and body. Both the head and body may contain a list of variables (names of relations in the intensional or extensional database). These relations may be composed with *index map* variables using the `->` operator. For example, the $pos[n1[i]]$ access in the *COMPUTE-AREAS* subroutine is denoted by `n1->pos` and has the same meaning as the `n1(T,I1) & pos(I1,P1)` in the given *Datalog* rule. Note that these *index map* composition operators can appear in the head or body of the rule. In addition, the rule signature may have a specified conditional execution argument. Finally, rules fall into five categories – `Pointwise`, `Singleton`, `Unit`, `Apply`, and

```
signature: head '<-' body

head: maplist

body: maplist optconditional ':' ruletype

maplist: mapseq
       | maplist ',' mapseq

mapseq: NAME
      | mapseq '->' NAME

optconditional:
              | ',' 'conditional(' NAME ')'

ruletype: 'Pointwise'
        | 'Singleton'
        | 'Unit'
        | 'Apply'
        | 'BlackBox'
```

Fig. 4. Rule signature grammar.

`BlackBox` – that determine the nature of the computations and how computed values are combined to form a resulting *store* or *parameter*.

4.4 Rule semantics

The rule signature is used to determine when a rule should be executed and what set of indices should be passed to the subroutine. The set of indices is called the context of the rule. The context of the rule can be determined from the rule signature by identifying the set of entities that satisfy all of the inputs. For example, a subroutine for the triangle area computation described earlier would have a signature of

`area<-n1->pos,n2->pos,n3->pos:Pointwise`,

where the explanation of the pointwise rule type will follow. The body of this rule documents the indirect accesses used to obtain the positions of the tree triangle nodes. Note that from this signature we are able to infer the set of indices that can be passed to the subroutine in order to compute areas. To do this, we first find the valid sets for each individual composition of *index map* and *store*, for example `n1->pos`, by forming a join between the two relations (*e.g.*, solving `n1(T,I1) & pos(I1,P1)`). In the case of `n1->pos`, the valid set is $\{1,2\}$ as only these indices satisfy the composition accessing a valid `n1` and `pos`. A valid context of the rule is formed through the intersection of the valid sets for all inputs.

4.4.1 Pointwise aggregation

The most common type of rule used in *Loci* applications is the pointwise rule. A pointwise rule documents a subroutine where the dependency of each iterate is explicitly through the (possibly indirect) accesses documented in the rule body. The computation that results from these accesses is referentially transparent and uniquely defines the resulting value that is placed in the output *store* described in the rule head. As a result, pointwise rules always generate *store* relations in the intensional database; however, they may read either *store* or *parameter* relations, possibly using *index maps* to access other relations indirectly. It is possible for multiple pointwise rules to generate the same relation provided that they assign values to independent indices. For example, it is possible to have many rules that compute areas, such as for both triangle and quadrilateral facets, provided that their outputs form a proper partition.

4.4.2 Singleton computations

Singleton rules are used to describe computations that derive one *parameter* relation from other *parameter* relations. Since *parameter* relations only have a single value that is associated with a group of indices, there is a single computation associated with these rules. Thus we identify such rules as singleton rules. By the nature of these computations, singleton rules are restricted to only have *parameter* relations as inputs or outputs.

4.4.3 Global and local reductions

In addition to pointwise and singleton rules, we provide a mechanism for performing reductions. We characterize reductions into two basic categories: Global and Local. In reductions, we introduce an associative (and commutative) operator to combine a set of values. In a global reduction, we combine all values to form a single result, whereas in a local reduction we combine subsets of values that are implicitly defined by *index maps*. In *Loci*, global reductions produce *parameters* as output, while local reductions produce *stores*. Figure 5 illustrates the difference between local and global reductions in *Loci*. In this figure, the open circles represent entity identifiers (indices), while the filled circles represent values. In the global reduction, we first use the input values to perform some computation, `f(x)`, and then we combine the results of these computations using a provided operator, $\oplus$. The result is stored in the value field of a *parameter* relation. In the local reduction, an *index map* is typically used in the head of the rule. We use this map to create a parallel computational reduction structure as shown in Figure 5. Here, we again perform some computation, `f(x)`, and then combine it using the structure of the *index map* provided in the head of the rule.

Reductions are specified in *Loci* using two different rule specifications. A `Unit` rule is used to initialize the values of the reduction to the identity of the reduction operator. In addition, a set of one or more `Apply` rules define the application of functions that will be accumulated to their outputs as illustrated in Figure 5. Once all possible apply rules execute, the value is made available for use as inputs to other rules.

4.4.4 BlackBox interface to external libraries

In some cases, we would like to relegate computation to external sources such as highly tuned numerical libraries. However, none of the previous rule types can

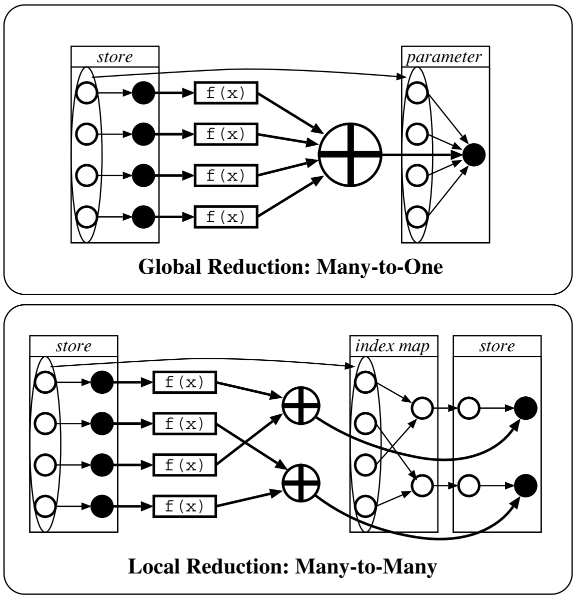


Fig. 5. Data flow in global and local reductions.

be used for this purpose since external libraries are likely to include internally generated relationships between values that constrain how the computation should be scheduled. Unlike pointwise rules, where the computations can be broken into cache sized chunks and computed in parts, the solution of external libraries (for example, consider a solver for a system of linear equations) must be performed as one monolithic operation. Thus, the semantics of an external library (particularly with respect to typical performance tuning transformations) is sufficiently different from other computations that it demands a unique interface. For documentation of calls to external libraries we reserve the `BlackBox` rule class. This rule class is used to inform the *Loci* planner that this computation is monolithic and contains hidden synchronization and communication that is not managed by *Loci.*

4.5 Similarities and differences with Datalog

As mentioned earlier, the logic model used in *Loci* is a restricted version of *Datalog*. The pointwise rule has semantics similar to a *Datalog* rule except that it is constrained in the following important ways: (1) in *Loci* all relations are binary with the first entry of the relation always representing an entity identifier (index), (2) joins between relations are restricted to either the shared entity identifier in the first field or simple *index map* compositions using the *Loci* `->` operator, and 3) results of pointwise rules are restricted to *store* type relations that only associate new values with already defined indices. Similar to *Datalog*, *Loci* support recursive specifications. However, we do not support stratified negation (although it could be added easily). Also, since our relations are implemented using arrays, we add a restriction that a value is uniquely associated with each entity of a relation. When this is violated, it is detected and provided as an error of model over-specification. *Datalog* does not define separate types of rules for singletons and reductions. However, the need for these rule types are primarily dictated by the restrictions that we have placed on the pointwise rule. In addition, the rule categories used in *Loci* represent standard idioms used in irregular numerical computations. For example, similar classifications exist in the directives of *OpenMP*.

5 Implementation

The *Loci* framework is implemented in C++. The framework provides a basic infrastructure for building numerical applications. The central component of the *Loci* framework is a planner that schedules subroutine calls, memory allocation, communication, and thread synchronization calls for execution on shared and/or distributed memory architectures. Here we only present the details of the distributed memory implementation.

In *Loci*, the user provides the *Loci* planner with a collection of rules (documented subroutine calls), the EDB relations that describe the problem, and a goal. The collection of rules are stored in dynamically shared objects and may be a combination of prepared rule libraries (for example libraries of standard time and space integration rules) and application specific rules that the user provides. In addition, *Loci* provides basic file reading functionality that builds many of the standard relations required to implement finite-volume or finite-element solvers.

5.1 Basic facilities

Several facilities are provided to support the planner. The most fundamental of these components are classes that describe sets of entity identifiers. Entity sets represent unordered sets of entity identifiers which are stored as a sorted list of disjoint intervals to support efficient set operations such as union and intersection. This compressed form makes them particularly efficient when manipulating large collections of entities that are consecutively numbered[2].

In addition to entity sets, *Loci* provides containers that implement the four types of relations described in Figure 3: the store, index map, parameter, and constraint. These containers have an interface that appears to be array-like in that the traditional array operator in C++ is overloaded to take an entity identifier. The array operator returns the value associated with the given entity identifier. Each container is provided in two forms, dynamic and static. The dynamic version implements association of indices with values using a radix trie. The dynamic versions support automatic allocation as new values are inserted and have facilities for editing relations as they are being generated. Static versions of these containers use a simple array and require consecutively numbered indices for memory efficiency (since they

[2] In this form, the entity set $\{[1, 1000], [2001, 3000]\}$ represents a set of 2000 entity identifiers using two intervals. The union of this set and the set of entities identified by $\{[1, 2000]\}$ produces the set represented by a single interval, $\{[1, 3000]\}$.

allocate an array from the minimum to maximum indices). Static containers require that their entity index space is allocated prior to insertion of values. Generally users build the EDB relations using dynamic containers. This facilitates using a global entity numbering strategy. During planning, *Loci* automatically converts the EDB relations into their static counterparts once an appropriate local numbering is determined. More details on this is described in the section on the *Loci* planner.

The fact database manages the extensional and intensional databases. It provides facilities for associating containers with symbolic names. As such, the fact database plays a key role in managing the flow of data between executing subroutines. The fact database also manages the distributed allocation of entity identifiers. Distributed allocation is provided by a collective call to the fact database where every processor requests a number of entities. The returned allocation is guaranteed to be a globally contiguous set of unique entity identifiers.

The rule database provides a mechanism for managing the rules that will be used by the planner. There are facilities for loading rules into the rule database from compiled shared object files. The rules in the shared object files are formed from instantiations of rule classes. These classes provide a constructor that documents the rule signature, an interface to obtain containers from any given fact database, and a compute method that is passed a set of entity identifiers that represents the context of the rule. The user can either manually loop over each interval in the set and perform the computation, or use a provided C++ template to automatically and efficiently iterate over the given context.

5.2 The Loci planner

The *Loci* planner assembles a schedule consisting of memory management, subroutine calls, and communication operations from a given fact database, rule database, and a user provided goal. Provided that only parameters or stores in the extensional database change, the plan can be repeatedly executed without regeneration. Plan generation is roughly divided into five steps: graph processing, partitioning/renumbering, existential deduction, pruning and communication scheduling, and finally plan generation.

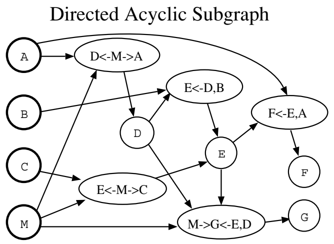
5.2.1 Dependency analysis and Proto-Plan

The graph processing step involves determining dependencies among rules as provided by the names of the relations found in the rule signatures as well as the names of EDB relations. The graph contains vertices for both relations and computations (rules). The graph is constructed from the goal, progressing toward known values. Once the dependency graph is generated, it is partitioned (based on functional groups such as iteration, or recursion) such that planning is managed over a collection of directed acyclic sub-graphs. These directed sub-graphs are used to create a proto-plan that is represented by a sequence of sets of rule signatures and relation names. This sequence can be considered as a set of data-parallel super-steps. In each step, the rules in the set can be executed concurrently, *Given EDB Relations*:

Store: `A,B,C`
Index Map: `M`

Given Rule Signatures:

```
D<-M->A
E<-M->C
E<-D,M->B
F<-E,A
M->G<-D,E
```



Proto-Plan

step	Rules	Relations
1		A,B,C,M

2	D<-M->A
	E<-M->C
		D

3	E<-D,B	E
4	F<-E,A M->G<-E,D	F,G

Fig. 6. Generation of a four step Proto-Plan.

while the relation names in the set are transitioning from being produced to being consumed. A communication step may be required with each super-step to achieve a consistent representation of the relations named in the step. In generating the proto-plan, we select an ordering that minimizes the number of steps by executing all rules that can execute in every step. Figure 6 illustrates a four step proto-plan generated from a given set of relations and rule signatures. In this plan, the first step contains no rules. Instead it includes all the given EDB relations. In the following step two rules are computed that contribute two IDB relations, `D` and `E`. However, only the relation `D` will be synchronized at the end of the step as other rules still will contribute other parts of the relation to `E`. The following step completes the computation of relation `E`. The last step of the schedule computes two additional relations, `F`, and `G`, that depend on `E`. The proto-plan is used to order both the computation and communication steps for the following planning stages. Since the proto-plan is based on the dependencies of relation names, and not on their contents, this plan is relatively cheap to generate. The proto-plan generation is duplicated on all processors. Once completed, every processor in the system has an identical proto-plan to base the communication of the remaining planner stages.

5.2.2 Partitioning and local numbering

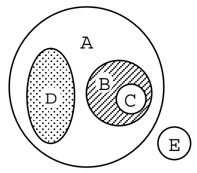
The extensional database is typically filled with dynamic containers that were created by the user. This facilitates economical construction of distributed data-structures using a global entity numbering strategy. While these containers are optimized for the creation of data structures, they are not efficient for computations. The next step in the process is to identify an efficient distribution of entities across processors and to create an ordering of entities such that entities of like type are consecutively numbered. Once this ordering is established, we can convert the dynamic containers into their static array-based counterparts for efficient computations (e.g. freezing the data structures). If the EDB relations are generated by one of Loci's provided file readers, then the distribution will already be optimized. Remaining non-optimized relations are distributed by using Parallel Metis (Karypis & Kumar, 1998) to partition the graph formed by the provided *index maps*.

Once we have an efficient distribution of entities, we need to identify entities that each processor will access. First, by definition, a processor will access all entities that have been allocated to it by the distribution step. However, due to the use of *index maps* in rules, each processor will also access some subset of entities that it does not own. These entities will need to be duplicated or ‘cloned’. We identify entities that need to be ‘cloned’ by finding the image of the entities each processor owns through all of the *index maps* documented in the rule signatures of the proto-plan. This step provides each processor with a set of owned entities and cloned entities. However, if we keep a global numbering at this stage, we will not be able to construct a compact contiguous numbering of the entities that each processor accesses.

We obtain a contiguous numbering by identifying a local numbering on each processor that establishes continuity not only for the entities each processor owns, but also the entities that each processor will access. The purpose of this local numbering is to organize relations such that they can be stored and accessed efficiently using static array-based containers instead of the more costly dynamic containers. The first element of the binary relation tuples is the index key. The domain of a container is the set of all index keys held in this first element. For the dynamic containers, there is no particular restriction on its domain. Static containers, however, need to have domains that are contiguously numbered for memory efficiency (as the gaps must be allocated as well). However, since the domains of several different containers may overlap, particular care must be taken to ensure that a numbering is able to give each container a contiguous numbering, or at least minimize the extent of gaps that are formed. To find an appropriate numbering for the entities in containers, we first identify the categories of entities formed by overlapping container domains. A category is defined by a set of relation names. An entity is in a category if the entity is in the domains of each relation in the category while also not in any other relations’ domain. Numbering entities to reduce the non-contiguous numbering of relation domains amounts to a problem of finding the appropriate order in which to number categories.

The general problem of ordering categories to minimize wasted gaps in allocation is somewhat similar to a knapsack problem. We use a heuristic approach that works well in practice. It will find the minimum for the case where the domains of relations are proper subsets of other relation domains. Our strategy for numbering categories is illustrated in Figure 7. In this example we have five relations that form five categories. The relationship between categories is illustrated in the Venn diagram. We start with a table of category sizes as listed in the leftmost table. Note, that if we numbered categories as found in the table, we would produce a gap in the numbering of relation B of size 1000. In our approach, we first compute the size of each relation. We then sort the relations in order of decreasing size. From this we are able to encode the category into a binary string where the bits of the string are one when that field’s relation is in the set of relations that define the category. We order this bit string such that the most significant bit is the largest relation. We then sort the categories according to this binary string as illustrated in Figure 7. We then number the entities in each category in their sorted order. Within each category, we number the entities that we own first, and then number the cloned entities. This ensures that computed entities that are passed to subroutine compute loops form contiguous numberings as well. Once this ordering is established, we

Venn Diagram



Category Sizes

category	size
A	100
A,B	75
A,D	1000
A,B,C	25
E	10

Relation Sizes

relation	size
A	1200
D	1000
B	100
C	25
E	10

Sorted Categories

key	category
00001	E
10000	A
10100	A,B
10110	A,B,C
11000	A,D

Fig. 7. An illustration of sorting categories to obtain contiguous numbering.

freeze our dynamic relations converting them into a local numbering that is suitable for their more efficient static forms.

5.2.3 Existential deduction

The next step in the process is to determine the existence of values (parameters and stores) that will become IDB relations. We perform this step by starting with the EDB relations, and then proceeding down the proto-plan. At each step we first evaluate the values that would be created in the IDB by the application of the rules given in the current step of the proto-plan. This evaluation starts by computing the set of entities that form the context of the rule. The context of the rule is simply the intersection of all of the inputs defined in the rule body. If an input contains the indirection operator (->) , then a pre-image of the *index map* must be performed. Since the entity sets that are intersected are stored in compressed form and entities are numbered in contiguous segments in the previous step, set intersections are economical. The most expensive procedure in the rule context determination is the *index map* pre-image operation. Since the same indirection operator is often used repeatedly, we cache these pre-images for later re-use. Once we have the rule context we are able to compute the creation of new parameters and stores for the IDB relations described in the head of the rule. If an indirection operator exists in the head of the rule, we find the image of the rule context through the given *index map* before computing the new existence entity set for the IDB relation. If any of the entities in the new IDB relation had previously been assigned a value, we identify this as a model consistency error and inform the user of the problem. Once we complete the IDB contributions of the rule in the current proto-plan step, we then perform a communication of the results for the synchronized relations listed for the current proto-plan step.

5.2.4 Pruning and communication discovery

Once the existential analysis is complete, we know the entities for which new IDB relations can be created. However, some of these values may not be required for a given goal. We follow the existential deduction process with a pruning phase to determine which computations are required. This step begins at the end of the proto-plan and works in reverse toward the beginning. In this case, we are evaluating requests for information and passing these requests to preceding rules in the proto-plan. First, we communicate any requests for relations listed in the current step of the proto-plan to the processors that own the requested entity. Since these communications will represent actual data communications in the final plan, we save these communication requests for plan generation. For the rules, we process the requests and compute the requested rule context. This value is saved for generating the computational loop bounds for the plan generation section. The pruned rule context is used to generate requests for the variables inputs. In turn, these requests will generate communication requests in the preceding relation synchronization steps in the proto-plan.

5.2.5 Plan generation

After the information on the communication requests and pruned rule contexts is computed from the pruning stage, we are ready to generate an execution plan. Note that the subroutines that the rules document are already compiled to machine executable form. Thus, the purpose of the plan is to coordinate these subroutine calls at run-time into an application that satisfies the user specified goal. The plan is essentially a tree of execute modules where execute modules are abstract base classes that can be specialized for specific functions such as executing a subroutine or communication operation. The plan is generated directly from the proto-plan. Each step in the proto-plan provides a set of subroutines (rules) to execute. The loop bounds for these subroutines as computed from the previous pruning stage are stored along with a function pointer to the subroutine in a subroutine execution module and appended to the tree. The list of relations at each step in the proto-plan may require communication if requests were transmitted during the pruning stage. At this point, three different types of communication may be required: (1) point-to-point message exchange for relations computed through pointwise rules, (2) a global reduction for parameters computed using unit/apply rules, and (3) a local reduction for stores computed using unit/apply rules. Each of these cases is dealt with separately.

For the point-to-point communication, the data is likely to be scattered among several relations at each step. In addition, for more complex objects, serialization may also be required. We use the simplest approach for dealing with this scattered data. We pack all of the data to be sent to each processor into a buffer for sending using calls to `MPI_Pack()`. This buffer is reused for each communication step to reduce memory allocation overhead. In addition, we are able to anticipate the size of the message buffer on the receiving side, and so are able to perform the communication of the buffer efficiently using a sequence of immediate receive calls followed by send calls.

For global reduction of parameters, all parameters that must be reduced are combined collectively with a single call to `MPI_AllReduce`. Before this communication operation all processors parameters contain their respective partial reduction of the values. When the operation is complete, all processor parameters contain the final combined value. For singleton rules, the parameter computations are duplicated on all processors, so no communication is necessary.

For local reductions, partial results are sent to the processor that owns the entities. The accumulated values are then sent to processors that request them using the point-to-point communication operation described earlier. This is generally efficient as the local reductions generally apply to low connectivity graphs (for example the triangles that are neighbors to an edge), and so there would be little savings from a more sophisticated approach.

5.3 Performance issues

For estimating parallel costs, the planning strategy fits nicely into a BSP (Valiant, 1990) like model of computation. Each step in the proto-plan is scheduled similar to a BSP super-step, with the exception that some of the communications operations at the end of the step include reduction operations. A loosely synchronous model is adopted whereby an actual barrier between steps is not required, but is instead implied by the two-sided message passing protocol. For the problem domains of continuum mechanics, we find that the number of clone entities can be kept relatively small (much smaller than the number of entities assigned to each processor). Similarly the number of processors that own clone entities appears to remain relatively fixed as we scale: the three dimensionality of the underlying graphs ensures high locality. The cost of global reductions grows as a logarithm of the number of processors, while the cost of local reductions is approximately twice that of the point-to-point communication as both share the same clone entities and communication footprint.

We adopt a philosophy of assuming that the network has high latency. Therefore, we attempt to combine communications as much as possible. For example, we schedule the proto-plan such that we try to minimize synchronization steps. Similarly, we combine all communications at each synchronization point as much as possible to reduce startup and other latency induced costs. This may have the effect of reducing constant factors associated with communication, but it is likely that the absolute costs are significantly more complex. For example, in some cases it may be more efficient to communicate less frequently with less volume of data by duplicating some computations. We do not make this optimization. For some configurations it may be more efficient to communicate more frequently directly from memory rather than resorting to the pack/unpack approach that incurs extra copying cost in exchange for a potential increased latency overhead. In addition we make no attempt to hide communication costs behind computations. However, these factors should have little effect on performance provided that our run-time remains bounded by computation. For the production problems we currently solve, runs appear to be computation dominated. It is likely that cache optimizations rather than communication optimizations will have a more dramatic effect in these cases.

From a serial performance perspective, we gain performance by aggregating computations over like type. While our specification is fine-grained, the computations are performed by tight, highly optimized loops. Without this aggregation facility, we would require dynamic dispatch within these tight loops if we wished to facilitate the same fine-grained specification. Since the plan generated by the Loci planner is calling pre-compiled subroutines and passing an aggregation of entities over which to perform the computations, the cost of dispatching the subroutine call is amortized over the aggregation. Since we compile loops over aggregations, the compiler is able to perform low level optimizations for instruction pipeline and register management that would be impossible to perform otherwise. In addition, converting the containers to their static array-based counterparts means that the actual computations are identical to the loops that would be used in a non-Loci program. You could, for example, pass the memory pointers and loop bounds directly to a Fortran subroutine if needed. In addition, we have found that using the Loci built in looping template to define rules produces serial codes that have practically the same execution time as similar C codes.

The generation of the plan is a unique cost of the Loci approach. Currently the planning phase is completed in a matter of seconds on problems involving hundreds of thousands of entities. Typically these problems run hours giving a negligible amortized planning cost.

6 Results

Performance benchmarking is always difficult to do well. Generally, performance can vary significantly with compilers and architecture as well as the amount of time that is spent by the implementer tuning an approach. Application performance usually includes a man-power cost that is often difficult to measure and include in the overall assessment. With regard to *Loci* performance, we will discuss some of our experiences that sets our baseline performance, and then use a application of substantial complexity to demonstrate that the approach can be scaled up to practical problems.

For simple applications, such as a two dimensional Laplace solver, we have observed that the serial performance of *Loci* programs are essentially equivalent to a similar C implementation(Luke, 1999). This is not surprising as the bulk of the computation time is spent in essentially identical computational loops. However, the *Loci* implementation was much more abstract as the implementation formed a knowledge-base of rules that could be applied in other contexts. From this experience, we note that *Loci* appears to provide a more abstract representation with little abstraction cost.

Another result comes from a *Loci* implementation of a finite-element thermal stress solver. For this case, we were able to compare the performance of a conjugate gradient iterative linear equation solver implemented in *Loci* to the PETSc (Balay *et al*., 2003) library implementation. We note that PETSc is a highly tuned parallel library that has had tens of man years of development effort invested in tuning its implementation for various architectures. In this case, we found the *Loci* implementation to perform approximately 50% slower than the PETSc optimized kernel. This was not surprising as this was comparing a naive *Loci* implementation to a highly tuned PETSc implementation. However, the comparison is somewhat moot, since instead of creating a less naive *Loci* implementation, we simply implemented an interface to the PETSc solver using the `BlackBox` interface to remove this performance penalty. This may seem like cheating, however, it is representative of what is standard practice for application programmers: use highly optimized kernels when possible.

We have also observed cases where *Loci* performance is better than naive implementations. For example, a student that implemented a parallel version of Conway's game of life using MPI as a class assignment recently implemented the same program using *Loci*. The resulting *Loci* program was faster, largely due to better memory management policies. As a result, the *Loci* implementation performed less data-copying. While the same savings could have been applied to the MPI program, it would have made the logic somewhat more complex, and the program somewhat less naive.

Based on these observations we find that the performance of *Loci* programs are generally better than naive implementations, but worse than highly tuned kernels. However, we suggest that when suitable highly tuned kernels are available, use the external library interface to call these kernels instead of re-implementing them in *Loci*. To evaluate the performance of *Loci* in a practical context we consider the performance of an application of significant complexity: the CHEM (Luke *et al.*, 2001) code. The CHEM code is a finite-rate non-equilibrium Navier-Stokes solver for generalized grids fully implemented using the *Loci* system. It implements a second-order space and time unstructured finite-volume solver that uses advanced high resolution approximate Riemann solvers, sophisticated multi-component transport properties, and advanced turbulence models. The code is suitable for solving both high and low speed combustion problems such as those encountered in space transportation system design. The CHEM code is implemented using approximately 700 *Loci* rules that are described using approximately 60K lines of C++ code. The CHEM code is in active and routine use by engineers at various NASA centers in the support of rocket system design and testing.

Here we are particularly interested in demonstrating the practical scalability of the approach. We wish to demonstrate that with reasonable per processor allocations of work we can continue to make parallel overheads a small fraction of the overall run-time. To demonstrate this, we select a representative problem and scale it such that the per processor assignment of work remains constant and measure the performance degradation.

Performance measurements were made on two platforms: (1) a 1038 processor Beowulf cluster, and (2) a 64 Processor SGI Origin 2000 195 Mhz R10K system. The 1038 processor Beowulf cluster is composed of dual 1.266 Ghz Intel Pentium III processor nodes with 1.25 Gb RAM. These nodes are interconnected within 32 node cabinets using 100 megabit Ethernet with a single large gigabit Ethernet switch connecting cabinets. No specialized high-speed low-latency communications hardware is used in the Beowulf cluster. A scaled speedup measurement was used to assess the scalability of the *Loci*/CHEM simulation. A simulation of hydrogen-oxygen combustion reactions in a rocket nozzle is used in this measurement where the problem size is increased for increasing number of processors by simply increasing resolution (grid size). The problem is scaled such that the number of cells per processor is held constant, and since the algorithm's serial cost is proportional to the number of cells, we are able to obtain the scaled speedup curve using the ratios

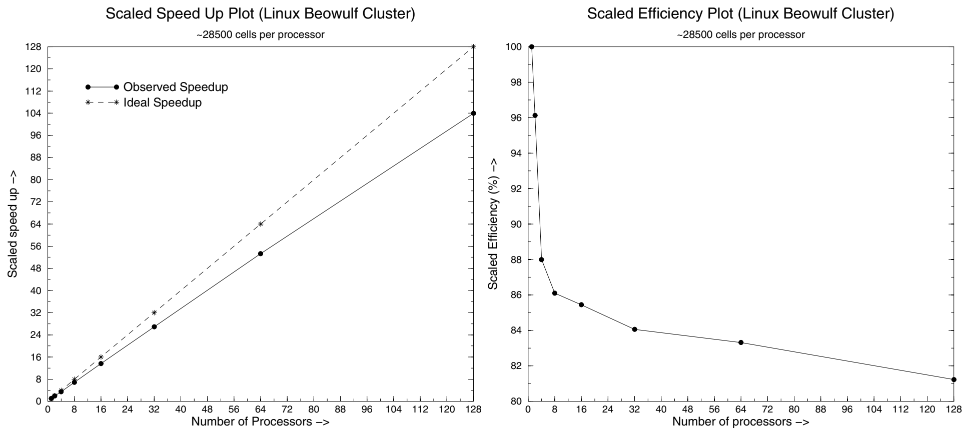


Fig. 8. Scaled speedup and efficiency curves for the Intel Beowulf cluster.

of serial to parallel per-cell execution times given by

$$S_{scaled} = \frac{N_p t_1}{N t_p}, \tag{1}$$

where t_1 is the time required to solve for N cells on one processor and t_p is the time required to solve for $N_p \approx p \times N$ cells on p processors. To ensure that the scalability of the system was measured as it would typically be used, the parallel execution time t_p was the measured wall-clock time for the entire simulation process, including file input and output, *Loci* parallel scheduling, and the simulation itself. Simulation times were approximately one hour where an average of four runs is used to estimate running time. With the exception of an occasional anomalous timing measurements caused by unusual system loads, timing measurements were largely consistent from one run to the next. The scaled speedup and efficiency plots for the Beowulf cluster for up to 128 processors are shown in figure 8, while the speedup results from the Origin 2000 for up to 32 processors are shown in Figure 9. As these systems are in active production use, we were not able to measure performance at full capacity (using all of the available processors). For the Beowulf cluster, most of the efficiency is lost going from one to four processors where efficiency drops to 88%. This is largely attributable to the transition from shared-memory inter-node communications to the 100 megabit Ethernet. Another drop in efficiency occurs from 8 to 32 processors, which can be attributed to an increased amount of inter-cabinet traffic (We noted that the batch queuing software tended to schedule jobs across cabinets for allocations larger than 8 processors, even though there were 64 processors per cabinet). Considering the relatively low-performance interconnect provided with this cluster, the scalability is quite good. In fact, no users reported efficiencies much higher than 80% for 128 processors on this cluster using other MPI based parallel simulation codes. Although the Origin 2000 speedup and efficiency results measure up to only 32 processors, efficiency is much higher with a 32 processor efficiency of 91% compared to just 84% on the Beowulf cluster. This is not surprising as the Origin has a better network to processor performance balance.

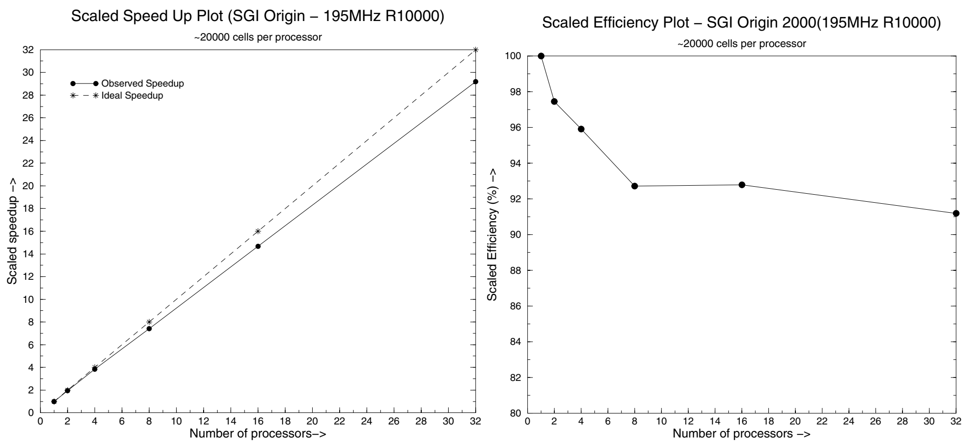


Fig. 9. Scaled speedup and efficiency curves for the SGI R10K origin.

In addition to these performance measurements, NASA engineers (West, 2003) reported an order of magnitude improvement in turn-around times in comparisons of the *Loci*/CHEM code to a specialized simulation code written using Fortran using the PVM message passing library. While some of this improvement is associated with different numerical algorithms, a significant fraction of the improvement is attributed to better parallel scalability of the *Loci*/CHEM software. However, this scalability is not related to specific improvements in communication performance or distinctions in other algorithm characteristics. Instead it is associated with the ad-hoc message-based parallel model used in the specialized simulation code. The ad-hoc implementation placed many constraints on how work could be partitioned to processors. Since communication of information at processor boundaries was incomplete, some simulated processes could not be accurately modeled across a processor boundary. In addition, new features were often not supported in the parallel implementation. However, since the *Loci* approach automated the parallel communication and execution scheduling, users could decouple modeling decisions from resource allocation decisions. This allowed users of the CHEM code to effectively use more processors for any given problem. This case is included to document the dangers of not including other costs in performance measurement. In the case of the specialized simulation code, the man-hour cost of developing a completely consistent MPI implementation was prohibitive and resulted in reduced effective scalability. This cost was not present in the *Loci* implementation as communication scheduling was automatic. This reliability of the parallel implementation contributed largely to the practical scalability of the *Loci*/CHEM code.

7 Conclusions

Developing scalable high performance numerical applications for scientific and engineering simulation is costly in part due to the tedious programming required using accepted parallel programming models. While many domain specific application frameworks can reduce the difficulty of developing this software significantly, we find that such frameworks usually commit the user to a specific discretization strategy and numerical approach. Many other parallel and distributed programming tools require coordinating coarse-grained computations to achieve efficiency; however, significant work is involved in developing software to reliably aggregate computations. As such, coarse-grained models are less helpful than they might first appear. To solve this problem, we have developed a framework that is based on a relational model of computation that facilitates automatic aggregation. Since such a model does not make assumptions about the particular discretization strategy, we provide a much more general abstraction. In addition, we find that it is useful to use a rule-based declarative strategy for describing computations since this approach allows developers to establish appropriate model invariants. In addition, a rule-based approach decouples the needs of value producers from value consumers (the consumer of values is not concerned with the particulars of how the value is created; only that it exists where it is needed).

Although the declarative programming techniques used in this framework are primitive by modern declarative language standards, the results of this work does present a new interesting idea for the development of a more general declarative model for high performance computing: the value of the relational model. We find that the abstraction is an attractive natural match for irregular computations. In addition, we demonstrate that a *Datalog* like abstraction can be used to develop scalable high performance applications. This abstraction allows us to think about computations at a fine-grain level, and allows the run-time system to manage aggregation and distribution appropriately.

Acknowledgements

We would like to thank Donna Reese and Yang Zhang for their suggestions and editing of the final draft of this paper. In addition we would like to thank the anonymous reviewers for their excellent suggestions. Finally, the financial support of the National Science Foundation (ACS-0085969), NASA GRC (NCC3-994), and NASA MSFC (NAG8-1930) is gratefully acknowledged.

References

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S. and Smolinski, B. (1999) Toward a common component architecture for high performance scientific computing. *Proceedings 8th IEEE International Symposium on High Performance Distributed Computation.* pp. 115–124.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M., McInnes, L. C., Smith, B. F. and Zhang, H. (2003) *PETSc 2.0 users manual.* Technical report ANL-95/11, Revision 2.1.6. Argonne National Laboratory.

Bassetti, F., Brown, D., Davis, K., Henshaw, W. and Quinlan, D. (1998) OVERTURE: An object-oriented framework for high performance scientific computing. *High Performance Networking and Computing (SC'98).* IEEE Computer Society, CDRom.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J. and Zagha, M. (1993) Implementation of a portable nested data-parallel language. *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 102–111.

Breitinger, S., Loogen, R., Ortega-Mallén, Y. and Pene, R. (1997) The Eden coordination model for distributed memory systems. *Workshop on High-level Parallel Programming Models*, pp. 120–124. IEEE Press.

Cann, D. (1992) Retire Fortran? A debate rekindled. *Comm. ACM*, **35**(8), 81–89.

CASC (2003) *High-performance component technology*. http://www.llnl.gov/CASC/sc2001_fliers/CompTech/CompTech01.html (Current Oct. 5, 2003).

Cleary, A., Kohn, S., Smith, S. G. and Smolinski, B. (1998) *Language Interoperability Mechanisms for High Performance Scientific Applications*. Technical report, LLNL. UCRL-JC-131823.

Foster, I. and Taylor, S. (1990) *Strand: New concepts in parallel programming*. Prentice-Hall.

Foster, I., Olson, R. and Tuecke, S. (1992) Productive parallel programming: The PCN approach. *J. Sci. Program.* **1**(1), 51–66.

Gabrielle, A., Benger, W., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E. and Shalf, J. (2000) The Cactus Code: A problem solving environment for the grid. *Proceedings Ninth IEEE International Symposium on High Performance Distributed Computing*. pp. 253–260.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) *PVM 3 Users Guide and Reference Manual*. Technical report, Oak Ridge National Labratory, Oak Ridge, Tennessee 37831.

Gerlach, J., Sato, M. and Ishikawa, Y. (1998) Janus: A C++ template library for parallel dynamic mesh applications. *Computing in Object-Oriented Parallel Environments, Proceedings of the Second International Symposium, ISCOPE 98: Lecture Notes in Computer Science 1505*, pp. 215–222. Springer-Verlag.

Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D. and Bramley, R. (2003) Merging the CCA component model with the OGSI framework. *Proceedings of CCGrid2003, 3rd International Symposium on Cluster Computing and the Grid*. pp. 182–189, IEEE Computer Society.

Hill, J. M. D., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. and Bisseling, R. (1997) *BSPlib: The BSP programming library*. Technical report PRG-TR-29-97, Oxford University Computing Laboratory.

HPF (1993) *High Performance Fortran Language Specification*. Technical report, Rice University, Texas, USA.

Karypis, G. and Kumar, V. (1998) Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel & Distributed Comput.* **48**(1), 96–129.

Kotlyar, V., Pingali, K. and Stodghill, P. (1997) Compiling parallel sparse code for user-defined data structures. *SIAM Conference on Parallel Processing for Scientific Computing*, vol. 8.

Luke, E. A. (1999) Loci: A deductive framework for graph-based algorithms. In: Matsuoka, S., Oldehoeft, R. and Tholburn, M., editors, *Third International Symposium on Computing in Object-oriented Parallel Environments: Lecture Notes in Computer Science 1732*, pp. 142–153. Springer-Verlag.

Luke, E. A., Tong, X. L., Wu, J., Tang, L. and Cinnella, P. (2001) A step towards "Shape-shifting" algorithms: reacting flow simulations using generalized grids. *Proceedings 39th AIAA Aerospace Sciences Meeting and Exhibit*. AIAA-2001-0897.

MPI (1997) *MPI-2: Extensions to the Message-Passing Interface*. Technical report, University of Tennessee, Knoxville, TN.

OpenMP (1997) *OpenMP Fortran Application Program Interface, Version 1.0*. Technical report, OpenMP Architecture Review Board.

OpenMP (1998) *OpenMP C and C++ Application Program Interface, Version 1.0*. Technical report, OpenMP Architecture Review Board.

Pande, S. S., Agrawal, D. P. and Mauney, J. (1994) A threshold scheduling strategy for SISAL programs on distributed-memory machines. *J. Parallel & Distributed Comput.* **21**(2), 223–236.

Pointon, R. F., Trinder, P. W. and Loidl, W.-H. (2000) The design and implementation of Glasgow Distributed Haskell. *International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Scienc 2011*, pp. 54–70. Springer-Verlag.

Trinder, P. W., Hammond, K., Loidl, H.-W. and Peyton Jones, S. L. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Program.* **8**(1), 23–60.

Trinder, P. W., Loidl, H.-W. and Pointon, R. F. (2002) Parallel and distributed Haskells. *J. Funct. Program.* **12**(4 & 5).

Ullman, J. (1988) *Principles of Database and Knowledgebase Systems*, pp. 53–66. Computer Science Press.

Valiant, L. G. (1990) A bridging model for parallel computation. *Comm. ACM*, **33**(8), 103–111.

West, J. (2003) NASA Marshall Space Flight Center, Personal Communication.