Squid: A Language for Discrete, Time-based Simulations Across an n-dimensional Lattice

Daniel Fasulo Jason Secosky

December 7, 1994

1 Introduction

Many languages support aggregate data structures, which may be informally defined as a grouping of data of varying types into a single structure. Similarly, many languages allow the user to define arrays of aggregate or atomic types, thereby defining a "collection" of data of uniform type. Collection-oriented languages take this one step further by providing not only mechanisms for defining such types, but also provide built-in operations which operate on entire collections. They may also provide ways of describing relationships between the elements in a collection. Thus, these languages can simplify programming tasks which involving manipulating large organized data sets, such as simulating heat conduction across a mesh.

In addition, collection-oriented languages extend well from sequential to parallel machines. They simplify the difficult task of parallel programming by allowing the user to focus on single representative items, rather than worry about controlling the interactions of several threads of control. Operations on collections also tend to contain large amounts of easily identifiable parallelism. For example, applying an operation to every element in an m-element collection can easily be thought of as doing m operations in parallel, so by applying a sequence of operations to a collection, the programmer can seem to be programming sequentially while actually generating a parallel algorithm.

Languages such as APL, Fortran 90, C** and ZPL all are considered to be collection-oriented, although each handles collections and operations on collections differently. In this paper, we briefly examine each of these languages' treatment of collections, and then specify a collection-based language for simulation, Squid, which incorporates the best features from each.

Squid is a dynamically-typed, collection-oriented imperitive language. In order to better support simulation, Squid incorporates a notion of time. We also envision a run-time support environment which allows the user to control the advance of time manually and to inspect values during the simulation.

We begin by examining some other collection-oriented languages, and then discuss the motivation for developing Squid. We then present an overview of a typical Squid program followed by some ideas on the language constructs necessary to concisely express such a program.

2 Prior Collection-oriented Languages

Each of the collection-oriented languages we consider are grid-ordered; that is, each language organizes its data into an n-dimensional array structure. This restriction is chosen since Squid operates on grid-oriented collections.

2.1 A Programming Language (APL)

Developed in the early 1960s, APL is the forerunner of collection based languages. It is noteworthy because it was the first language to introduce a compact notion for representing array operations with a fixed set of reductions. Amazingly, many of APL's constructs are used in languages such as Fortran 90 and ZPL.

Implicit function promotion is supported by APL. Function promotion is best illustrated by example. Suppose that a function like $\sin(x)$, which typically operates on one floating point operand, is applied to an array of floating point numbers. Promotion causes $\sin(x)$ to be applied to each value in the array. However, APL does not support nested collections, so an operation to print a collection of collections would not work since the print operation could not be propagated to the internal collections.

2.2 Fortran 90

Fortran 90 is similar to Fortran 77, with an added data parallel sub-language. Designed for vector and parallel machines, Fortran 90 treats collections as first class citizens. Arrays can be passed as arguments to, and returned as values from, user-defined functions. Unfortunately, the built-in data parallel operators are limited to functioning on numbers or strings, so they may not be applied to arrays of aggregate types.

2.3 C**

C** is a data-parallel, object-oriented language that includes many of the object-oriented features of C++ while adding an Aggregate class. This class is defined over n-dimensions and contains data values and functions which may operate on the entire Aggregate or just parts specified by for-loops over the dimensions. The state of an Aggregate is the collection of storage locations read or written during an invocation. In summary, Aggregates can be viewed as an array with a structure of data values at each position of the array and an associated group of functions which operate across the entire array.

2.4 ZPL

ZPL is a high-level, array-oriented language which supports data parallelism. ZPL derives many of its features such as implicit function promotion from APL, yet drops Fortran 90's strict loop requirements for array operations. There are two unique features of ZPL which separate it from the other languages being compared. First, ZPL provides the user with a way to specify sections of the array by named regions, and also to specify the locations of data items in the array with named vectors. This removes the need to use for-loops in operations on data, minimizing the possibility of making an error in array indexing. Second, ZPL contains special constructs for specifying boundary conditions on the array, thus reducing the amount of time and effort dealing with "special cases" and allowing the programmer to spend more time on the "general case" of interior array points.

On the negative side, ZPL does not support access to individual elements of an array so the programmer cannot write routines to operate on subportions of the array, as can be done in C** or Fortran 90. This makes it difficult to perform such tasks as permuting the data of an array.

3 Motivation for Squid

The three main goals for Squid are to support array-based simulations which are parameterized by some notion of time, to provide a supportive graphical environment, and to operate on a sequential machine.

We consider a language-defined notion of time to be essential to easily implementing time-dependent simulations, especially on sequential computers. Previous collection-oriented languages which lack support for time-

dependent simulations suffer from two flaws. First, it is possible to define relationships on the elements in the array which prevent their values from converging. For example, if we define a mesh-based simulation of heat conduction, then we know that the temperature at each point will stabilize eventually, thus signaling that the computation is done. However, a related simulation which models heat transfer in a fluid by convection will never converge to an unchanging state. Thus, in order to bound the computation, we must be able to provide a specific time in which we are interested. Second, implementing time-based simulations on a sequential machine is difficult, since it requires that the data being generated for time $t + 1$ be buffered until the *all* data for time $t + 1$ has been generated. By placing the notion of time into the language, we make this issue the responsibility of the language implementation rather than the programmer, allowing much more concise programs representing time-dependent simulations which will still run on sequential machines.

The runtime environment can easily be specialized to support the above goals. We envision an interactive environment which allows one to step through the simulation by given positive time values and examine its data. We would not allow the simulation to proceed backwards since we cannot guarantee that at any given time there is enough information to exactly deduce the state at a previous time value. The language should provide sufficient IO features to store all simulation data at any given time and to provide more sophisticated, bit-mapped data visualization routines, although we would only be able to view two dimensions of the array at once using bitmaps. Similarly, there should exist a tool for viewing the data dependencies of each node graphically over two dimensions by displaying a subsection of the array, and for any selected node, drawing lines from it to each other array point which contains data upon which it depends (the resulting squid-like image is the source of the language's name). These features would provide strong support for array-based, time-dependent simulation.

Although the languages discussed in Section 2 do not contain the features outlined above, they do contain many useful constructs for array-oriented languages, and we borrow from them heavily.

4 An Overview of Squid

4.1 Program Organization

A Squid program can be thought of as defining a large data structure similar to C**'s Aggregate which we call a Lattice. The Lattice contains the following:

1. A set of "global" variable declarations. The global integer `Time` is implicitly defined to be the current time step in the simulation;
2. An n-dimensional array of Nodes, for which we can specify the size of each dimension separately;
3. A declaration of the data stored at each Node in the Lattice;
4. A set of actions which the Lattice may request that the Node perform;
5. A set of initial values for each Node;
6. A set of specifications for each Node, to be described below;
7. A set of actions which may be performed on the Lattice; and
8. A function of time which may be used to invoke actions at specific times

There are several important design issues reflected in this organization. First, by making the Node's data a property of the Lattice, we implicitly force all of the Nodes to contain exactly the same information. However, as we shall see, the rules for updating the data belong to each Node object, so the uniformity of data restriction does not really limit the programmer. This organization is done to prevent the user from defining some action or rule which accidently accesses data that a particular node does not contain.

There are two distinct classes of data: *simulation variables*, which are "simultaneously" updated at the end of every time step, and *temporary variables* which are updated immediately upon assignment. The latter type is provided for the programmer's convenience. Finally, note that each Node has implicitly defined read-only variables called `x1`, `x2`, ..., `x`n, representing its coordinate in the Lattice.

We also provide a set of global, temporary variables that may be accessed from any Node. These are useful more for storing data about the Lattice

in general than for allowing communication between Nodes, for which there exists other constructs. Perhaps the most useful global variable is the `Time` variable which stores the number of discrete simulation steps have been performed on the Lattice.

Next in the structure comes the array declaration for the Nodes, and a section for initializing their data. The language provides general iteration constructs that allow us to assign certain data to sets of Nodes conveniently.

The set of actions are methods in the object-oriented sense of the term which may be invoked on the Lattice. For example, the Lattice may be able to output itself to a file, or generate a two-dimensional bitmap representing itself. Lattice actions may call the actions and examine data of any Node, but may not perform assignments to Node variables. The Lattice also has an implicitly-defined action called `step` which increments `Time`, updates the simulation variables in each Node according to the Node's rules, and, after performing the update, checks the assertions on each Node.

The actions for a Node are very similar to those of a Lattice; they allow the Node to perform some useful action such as return a value for output. They may also assign to global variables, thus allowing the programmer to collect statistics about the state of the Lattice, but may not assign to any simulation variables. We can define Node action "promotion" as well; a Node action may be invoked as a Lattice action, with the implicit result that the Node action is carried out on each Node in the Lattice in an unspecified order. This default behavior can be overridden by explicitly defining a Lattice action with the same name, however.

Finally, the Lattice has a function which takes `Time` as its input and may use its value to decide if it wishes to perform any of its actions. This function is called by the runtime environment each time that `Time` is incremented.

The program is also responsible for setting up each of its Nodes. As stated above, each Node has an identical set of variables, but other portions of the Nodes may vary significantly. Each Node may include the following information about itself (all of which is optional):

1. a set of rules which describe how to update the Node's variables;

2. a set of assertions about the Node's variables.

Rules are simply assignment operators that compute new values for some or all of the variables at a Node. In the case of temporary variables, the assignment takes effect as soon as the rule is executed. Such variables are only intended to be used for storing common subexpressions and temporary

results, and cannot be referenced outside of the Node. On the other hand, assignments to simulation variables can logically be thought of as taking place simultaneously at the end of a time step, thus making irrelevant the order in which the assignments are actually performed. Perhaps the most critical feature of the rules of a node is that they may reference the simulation variables at another Node by specifying that Node as an offset from the current position in the array. This allows the programmer to easily express in general terms the relationships between the Nodes in the array.

Finally, the assertions for a Node are boolean expressions which are checked after each time step, as well as after the Node's variables are initialized. If an assertion fails then the Lattice will stop stepping and an error message stating the Node's coordinates and the failed assertion will be printed by the runtime environment.

Why do we place the assertions and rules in the Nodes, and the data and actions in the Lattice? The answer is type compatibility. We must guarantee that whenever, for example, the neighbor of a Node is expected to have a certain variable in order to evaluate a rule, the neighbor will have that variable, no matter which Node it happens to be. Similarly, the Lattice now cannot "accidently" invoke an action on a Node and discover that the Node does not define that action. Also, the promotion scheme requires that all Nodes understand the same methods. If, for some reason, it is imperative that a specific Node not perform some action, then its coordinates can be checked in the action implementation, but we believe that the cases when this will happen are very rare.

In addition to the definition of the Lattice data structure, Squid programs may contain macro definitions, and "helper function" definitions which take a set of inputs and return a value.

Now that we have described the general structure of a Squid program, we detail those portions of the language which are critical to expressing the declarations described above.

4.2 Life: A Sample Program

The following example of the Game of Life will aid in presenting some of the features of Squid. The two-dimensional version of the Game of Life has the following rules. Each position of the lattice is a cell. We will start with an initial condition where some cells are alive and some dead. The following rules will be checked at each time step of the simulation:

1. If none of your neighbors are alive you die of loneliness

2. If two or three of your neighbors are alive, you continue living or are reborn.

3. If all four of your neighbors are alive you die of overcrowding.

To begin we need to set up some direction vectors which will use later in the program to access data at a specified offset from each data element.

```
#define north Node[0,-1]
#define south Node[0,1]
#define east  Node[1,0]
#define west  Node[-1,0]
```

The game will be executed on a 10 by 10 element array. The size of the array is specified at the beginning of the program.

```
Lattice Life (10,10)
```

The next stage declares the variables to be used in the lattice during the simulation. Globals are accessible to all routines defined. `Nodedata` specifies the number and type of data items each Node in the lattice will define. There are two type modifiers for nodedata types, `sim` and `temp`.

Simulation variables are updated at the each of every time step. Temporary variables are not held to this restriction. Temporary variables may be updated at any time of the lattice simulation. In this case, a global variable is used to keep track of how many cells are currently alive. Each point in the lattice has two values, one for determining if it is alive, and the other for adding up how many of its neighbors are alive. The variable `neighbors` is declared `temp` so that its value can be updated and used before the end of a time step.

```
globals
  int numAlive;

nodedata
  sim int alive;
  temp int neighbors;
```

The next phase of the program defines the functions to operate on each of the Nodes and initializes the alive information. The assignment of 1's to different alive variables in the lattice is controlled by applyall and applyto operators. Applyall applies the expression to every Node in the lattice, while applyto applies the expression to a subportion of the lattice.

```
nodeactions
  void incNumAlive(void) {
    if ( alive ) numAlive = numAlive + 1;
  }

initialize
  numAlive = 0;
  applyall { alive = 0 };
  applyto ( 3..6, 2..5 ) { alive = 1 };
```

The following assertion will be checked at after the lattice has been initialized and again at the end of each time step. If an assertion fails to hold for a Node, then an error will occur specifying which node produced the error.

```
assertions
  applyall ( ( alive == 0 ) || ( alive == 1 ) );
```

The next sections specify the what occurs when the simulation is running. The `rules` section defines a set of expressions that will be implicitly executed at each Node during each step of the simulation. In this example we are computing whether the cell is alive or dead. The `actions` section specifies functions which can be evaluated during the simulation. In this case we are outputting the value of each Node to the standard output.

```
rules
  applyto ( 1..8, 1..8 ) {
    neighbors = south.alive + north.alive +
                east.alive + west.alive;
    if ( ( neighbors == 0 ) || ( neighbors == 4 ) )
      alive = 0;
    else
      alive = 1;
  }

actions
  void print(void) {
    int x, y;

    for ( x = 0; x < 10; x++) {
```

```
      for ( y = 0; y < 10; y++)
        printf("%d", Node[x,y].alive)
      printf("\n");
    }
  }
```

Analogous to C's main program construct is Squid's `eachstep` phase. During each time step the rules for each Node are evaluated, then the expression within the `eachstep` clause is executed. In this case after calculating whether a cell is alive or dead, we print a representation of the lattice and modify the variable which is keeping track of the number of live cells.

```
eachstep {
  print();
  incNumAlive();
}
```

Notice that the function incNumAlive is a promoted function. Defined to operate on a single value, this function is automatically promoted to operation on an entire lattice. The function sums the number of live vertices and stores this information in an information file.

5 Conclusion

In this paper we have defined Squid, a collection-based language which incorporates features from ZPL, C**, and APL. Unlike these other languages, however, Squid is specialized for describing time-based iterative simulations, and is also capable of running on sequential or parallel machines. Squid's main task is to remove the burden of time management from the programmer while providing constructs to manipulate an array data structure. Squid's `applyto` and `applyall` mechanisms give programmers the capability to map a function across the entire lattice or portions of the lattice. A vector based access scheme also provides an intuitive feel for accessing data via relative coordinates.

Although Squid is only useful for a single class of problems, we believe that the fact that this class is quite large provides a demand for this type of language. For example, Squid can describe fluid flows, masks for image manipulation, sequential circuit simulations, and many types of dynamic systems. Given the frequency with which these problems occur, especially

in engineering, we believe that Squid would achieve widespread use if anyone ever bothered to implement a interpreter for it.

References

[1] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, Inc., 1962.

[2] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report 1126, University of Washington, November 1992.

[3] Calvin Lin and Lawrence Snyder. *ZPL: An Array Sublanguage*. Department of Computer Science and Engineering, University of Washington.

[4] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. Technical report, Carnegie Mellon University, March 1991.