Incorporating Dynamic Communication Patterns in a Static Dataflow Notation

Pritish Jetley*, Adarsh Keshan† and Laxmikant V. Kale*
*Department of Computer Science
University of Illinois at Urbana-Champaign
†Department of Computer Science
Indian Institute of Technology Kanpur

***Abstract*—**
Charisma is a higher-level notation for writing parallel *static* data flow programs. It provides a convenient publish-consume paradigm for the expression of static flows of data and control. Although the syntax of the language allows the elegant expression of an important class of programs, it is restrictive, in that not all types of static data flow program can be written in Charisma. In this paper, we discuss a new incarnation of the language. *Charisma 2.0* has been designed to expand the expressive scope of the language, while retaining the productivity benefits of the publish-consume paradigm and a simple semantics of execution. We also examine the syntactic and semantic constructs of Charisma 2.0, and how they enhance expressivity.

I. Introduction

Although it has always been challenging to program large, parallel machines, recent developments have increased the difficulty of this task manifold. As of this writing, the ten fastest supercomputers in the world all have multi-petaflop capability. The largest of these, the IBM Blue Gene/Q at LLNL, has an astonishing 1.6 *million* processor cores. Needless to say, it is a daunting task to write parallel code that efficiently utilizes the resources of such large-scale machines. This challenge is amplified by the increasing sophistication of numerical algorithms and parallel applications in use today. In particular, applications routinely incorporate multiple interoperating modules, each of which may present different computational characteristics. Moreover, these modules must be *efficiently composable* both programmatically, and at run-time. The former engenders *programmer productivity* and software *maintainability*, whereas the latter allows the idle time in one module to be dynamically overlapped with useful computation in another. Given these challenges of productivity, there is increasing interest in alternatives to the traditional, SPMD/BSP style of programming.

It is in this context that several researchers have sought to create programming systems that help programmers to write high performance codes in a productive manner. We take a *divide-and-conquer* approach that combines three key elements: *modularity*, *language specialization* and *interoperability*. Briefly, a programmer *divides* his application into cohesive and relatively self-contained modules. Each of these modules is written in a higher-level, specialized notation that is specifically designed to allow the elegant expression of a certain (sufficiently narrow) class of algorithms or programs. Such specialized notations are by design *incomplete*, in that we cannot hope to write *every possible* type of parallel program in a given notation. However, by allowing modules written in different notations to *interoperate*, we overcome (*conquer*) the expressive limitations of individual languages. This approach enhances **productivity**, since each constituent module is written in a notation that, by design, closely matches its semantics. Good **performance** is enabled by two aspects of the run time system (RTS). First, the RTS instruments the execution of a multi-module program and performs optimizations such as dynamic load balancing and message streaming/combining. Second, since each notation enjoys a limited semantic scope, RTS designers can create optimized components for managing the execution of modules written in different notations. Finally, **completeness of expression** is ensured by the interoperability of the constituent notations among themselves and with a *recourse* language such as CHARM++, which is complete in itself.

In this paper, we discuss one such specialized notation, namely *Charisma*. The expressive scope of Charisma is limited to *static* macro data flow, which is described in the following section. In particular, we focus on the enhancement of expressive scope of the Charisma language. We begin with a discussion of the language in § II. As the reader will see in § III, this notation has its advantages in terms of compactness, elegance and abstractness of specification. Although Charisma cannot hope to capture *data-dependent* data flows, we will see how it fails to capture several important *static* data flow patterns as well. This will inform the discussion in § IV, wherein we examine the dimensions along which Charisma has been redesigned. The result of our efforts is **Charisma 2.0**, a language that retains the elegance of its predecessor, but strikes new ground in terms of expressive scope. In § V we look at several parallel algorithms that can be expressed in this new language, but not in Charisma.

II. The Charisma language

Charisma [1] provides an object-oriented means of writing parallel macro data flow programs. Briefly, a Charisma program consists of a series of statements that specify the dependencies between parallel objects. Data dependencies are implied through a *publish-consume* mechanism, and are inferred by

the Charisma compiler. Control flow constructs modulate the global flow of control, such that the resultant program has a deterministic semantics of execution. The compiler generates efficient CHARM++ [2] code as described elsewhere [3]. Below, we briefly enumerate the salient features of the language.

(1) Indexed collections of parallel objects. The basic unit of parallelism in an Charisma program is an *object*. Objects encapsulate both data and work, thereby promoting a locality-aware, fine-grained decomposition of data and work. The statements of a Charisma program specify a sequence of methods to be invoked on these objects in order to realize some computational work. As we shall see below, these objects can communicate with each other to exchange information. Charisma provides indexed *collections* of objects in the form of *object arrays*. This ability to identify individual objects through indices provides a convenient way for the programmer to decompose parallel work. Actions can be specified collectively on all the members of an object array through the *foreach* construct of Charisma. For instance, the following statement invokes a method *foo()* on all the objects in a 2D array A:

```
foreach (x,y in A) A[x,y].foo();
```

Note that the *foreach* construct is intentionally distinct from the *forall* construct found in many other languages. The *foreach* construct is meant to connote an action taken on a collection of *objects*. This is different from the semantic import of the *forall* construct of other languages, which usually provides a way to specify a data-parallel operation over an *iteration space*.

(2) Parameters, values and data flow. In Charisma, communication is represented by read and write operations on so-called *parameters*. A parameter is an indexed collection of *typed values*. A value, in turn, represents data that is to be communicated between objects. A value has a user-specified type and may itself be an array of elements. Each value of a parameter is identified uniquely by the name of the parameter and a multidimensional subscript. For instance, the user may declare a parameter p to be a 2D collection of values of type `double`. Each value in parameter p is identified as $p[x, y]$, where the ranges of x and y are defined by the programmer. An object that writes a value is said to *publish* it, whereas an object that reads a value is said to *consume* it. Arrays of objects can collectively read and write parameter values, and such statements take the form $LHS \leftarrow RHS$, where the values read by an object (or an array of objects) are the inputs to a method given in the RHS. This method is invoked on the object in question to write (publish) the list of parameter values given in the LHS. This is shown in the example below:

```
foreach (i in A) (p[i]) <- A[i].foo();
foreach (i in B) B[i].bar(p[i-1]);
```

Here, each $A[i]$ publishes a single value in parameter p, namely $p[i]$, and each object $B[i]$ in array B consumes a single value $p[i-1]$. Therefore, there exists a data dependency between each object of A and an object of B. From the indices of the published and consumed parameter values, the compiler infers the point-to-point communication pattern $A[x] \rightarrow B[x+1]$ for all x. Note that indices wrap around by default. In general, in order to obtain the communication pattern implied by a particular pair of published and consumed parameters, the compiler must *invert* the expression involving the *foreach* indices at the consumer's end. This allows for a wide variety of patterns in addition to point-to-point communication, e.g. reductions, multicasts, scatters and gathers, and the combinations of these operations. We will present some of these communication patterns in the examples of § III. However, as we will discuss in § IV the inversion requirement limits the expressive power of Charisma.

(3) Control flow. Straightline code in Charisma consists of a series of *foreach* statements that publish and consume data. Note that for a data dependency to exist between two statements s_1 and s_2, where s_1 publishes values of parameter p and s_2 consumes them, s_1 must occur *before* s_2. This linear flow of control can be modulated by loop constructs, namely *for* and *while*, and the *if-then-else* conditional execution construct. In particular, the loop constructs allow the construction of backward or loop-carried dependencies. These constructs have the usual semantics that we associate with loop and conditional constructs in serial languages such as C++. However, as noted elsewhere [3], the message-driven CHARM++ code generated by the compiler overlaps loop iterations to the extent allowed by the dependencies in the program. Therefore, while it is useful to think of, say, a *for* loop as ending with an implicit barrier, this is not necessarily how the generated program behaves. Moreover, it is worth restating that these constructs express the *global* flow of control in a program.

(4) Separation of parallel and sequential code. In addition to giving a symmetric and intuitive mechanism for the specification of communication, the publish-consume paradigm enables a separation between the parallel and sequential parts of a program. Recall that a *foreach* statement invokes a method on a collection of objects. The description of the method is only complete up to the identities of the parameter values that it consumes and publishes. The actual definition of the method is specified separately, as plain C++ code. Moreover, the method invocation is guaranteed to complete in a non-preemptible manner, greatly simplifying the semantics of execution. Given this separation of concerns, Charisma provides a mechanism by which data from sequential method implementations can be associated with parameter values (and hence communicated), and *vice versa*. However, for the purpose of this paper, it suffices to only say that such a mechanism exists, and we do not discuss it further.

III. EXPRESSING PARALLEL ALGORITHMS IN CHARISMA

In the previous section, we provided a very brief overview of the features of Charisma. We now give the reader a sampling of Charisma programs, with the objective of discussing the expression of various communication patterns, and more generally to demonstrate the expressive scope of Charisma. This section will also give us some context in which to discuss extensions and modifications to the language, which

are proposed in § IV. We present several well-established parallel numerical algorithms and applications, and see how the publish-consume paradigm of Charisma allows for their succinct and elegant expression.

A. Jacobi Solver

We begin with the Jacobi method for iteratively solving a second order PDE over a structured grid of points. The method repeatedly applies a stencil relaxation operation on grid points until convergence. Here, we discuss the decomposition of a 2D grid over a 1D object array J. That is, for an $N \times N$ point grid each $J[i]$ receives a *slab* of grid points of dimensions $(N/n) \times N$, where n is the number of objects in J. In order to apply the stencil function along the boundaries of its slab, object $J[i]$ requires the values of grid points from its "neighbors" $J[i-1]$ and $J[i+1]$. Therefore, in every iteration, each $J[i]$ publishes its top ($t[i]$) and bottom boundaries ($b[i]$), and follows up by consuming the top boundary of its bottom neighbor ($t[i+1]$) and the bottom boundary of its top neighbor ($b[i-1]$). It uses these boundaries to perform a stencil computation on its tile and computes an error for the current iteration, which is contributed it to a global reduction. The next iteration is started if the global error is above a certain threshold. This is shown in Listing 1.

Listing 1: Jacobi relaxation

```
while (err > Threshold)
  foreach (i in J){
    (t[i],b[i]) <- J[i].boundaries();
    (+err) <- J[i].compute(t[i+1],b[i-1]);
  }
```

B. Matrix-Matrix Multiplication

Agarwal *et al.* [4] have designed a 3D algorithm for the dense matrix multiplication operation $C = A \cdot B$, where A (sized $M \times K$) and B (sized $K \times N$) are the input matrices and C (sized $M \times N$) is their product. This algorithm achieves a factor $P^{1/6}$ reduction in communication over 2D variants, where P is the number of processors. The matrix-matrix multiplication operation can be viewed as a 3D space of individual product operations. We decompose the work in this space over a 3D array of objects, W (size $p_1 \times p_2 \times p_3$). The dimensions of W are labeled d_1, d_2 and d_3 We define $m = M/p_1$, $n = N/p_2$ and $k = K/p_3$. Object $W[i,j,l]$ is charged with the computation of $D^{(l)}_{ij} = A_{il} \cdot B_{lj}$, with $0 \leq i < p_1, 0 \leq j < p_2$ and $0 \leq l < p_3$. Matrices A and B are initially equidistributed over elements of W. Equidistribution is ensured by decomposing submatrix A_{il} among objects along d_2. In particular, $W[i,j,l]$ holds a contiguous partition of columns of A_{il}, denoted $A^{(j)}_{il}$, where $0 \leq j < p_2$. Therefore, each such $A^{(j)}_{il}$ is of size $m \times k/p_2$. Similarly submatrix B_{lj} is distributed among objects along d_1 in such a way that $W[i,j,l]$ holds a contiguous partition of columns of B_{lj}, denoted $B^{(i)}_{lj}$, with $0 \leq i < p_1$. That is, $B^{(i)}_{lj}$ has dimensions $k \times n/p_1$.

In the first step of the algorithm each $W[i,j,l]$ gathers submatrices $A^{(\hat{\jmath})}_{il}$ along the d_2 dimension of W, and submatrices $B^{(\hat{\imath})}_{lj}$ along the d_1 dimension, where $0 \leq \hat{\jmath} < p_2$ and $0 \leq \hat{\imath} < p_1$. In the Charisma code, each $W[i,j,l]$ publishes its A- and B-submatrices ($A[i,j,l]$ and $B[i,j,l]$). Then, it gathers A-submatrices along the d_2 dimension, and B-submatrices along the d_1 dimension, by consuming $A[i,*,l]$ and $B[*,j,l]$.

Through this gather, $W[i,j,l]$ obtains A_{il} and B_{lj}, which it uses to compute $D^{(l)}_{ij}$. Now, the D-submatrices held by individual objects must be combined along the d_3 dimension, so as to obtain submatrices of the result C. Although a simple reduction operation along d_3 would accomplish this, in order to ensure equidistribution of C across *all* objects in W, Agarwal *et al.* perform an *all-to-all* along the d_3 dimension. In effect, object $W[i,j,l]$ scatters submatrices of $D^{(l)}_{ij}$ along the d_3 dimension. In particular, the r-th partition of contiguous columns of $D^{(l)}_{ij}$ is sent to $W[i,j,r]$, where $0 \leq r < p_3$. Therefore, each $W[i,j,r]$ receives p_3 D-submatrices, which it adds together to obtain $C^{(l)}_{ij}$. In Charisma the scatter is realized by having each $W[i,j,l]$ produce a *labeled set* of parameter values, $D[i,j,l,*]$. Object $W[i,j,l]$ then gathers all the parameter values labeled with its l index by consuming $D[i,j,*,l]$.

Listing 2: 3D matrix multiplication

```
foreach (i,j,l in W){
  (a[i,j,l],b[i,j,l]) <- W[i,j,l].slices();
  (c[i,j,l,*]) <- W[i,j,l].mul(a[i,*,l],b[*,j,l]);
                  W[i,j,l].add(c[i,j,*,l]);
}
```

C. Transpose-based 1D FFT

Several application domains require an efficient numerical algorithm for computing the Fourier transform and its inverse, and this is given by the Fast Fourier Transform (FFT) algorithm. In a distributed memory setting, it is common practice to phrase the 1D FFT as a combination of two phases separated by a transpose operation[5]. In this treatment of the algorithm, both phases are communication-free, and perform different parts of a so-called *butterfly* operation on local data [6]. To begin with, each object $F[i]$ in a 1D object array F holds k input data elements $x[ik : ik + k - 1]$. The first phase corresponds to the first $\lg N/k$ butterfly steps of the algorithm, which operate on "nearby" elements of data. This is followed by a transpose, in which each object sends a different piece of data to every other object. As in the case of § III-B, this is expressed in Charisma as the combination of a scatter and a gather. That is, each $F[i]$ produces a set of tagged values $p[i,*]$. The size of this set is implied by the number of objects in the consuming array, which is F itself. Therefore, each object produces N/k p-values, each of which is tagged with an index from $[0, N/k)$. Note that each p-value encapsulates k^2/N complex numbers. In order to begin the next phase (the "far" phase) of the butterfly, each $F[i]$ consumes those p-values whose tags match its own index i. Once it has received all N/k

p-values, $F[i]$ performs the far phase of the butterfly without any communication:

Listing 3: 1D Transpose-based FFT

```
1 foreach(i in F){
2   (p[i,*]) <- F[i].butterfly_near();
3   F[i].butterfly_far(p[*,i]);
4 }
```

IV. Redesigning Charisma to Increase Expressivity

In the previous section, we showed the elegant and compact manner in which static data flow algorithms can be expressed in Charisma. In this section, we investigate the shortcomings of this notation and present three dimensions along which we have redesigned Charismain order to address these issues. We follow up with several examples in § V that demonstrate the resultant expansion in the expressive scope of the language. Taken together, the modifications and extensions that we suggest here constitute the basis for the new incarnation of Charisma, namely Charisma 2.0.

A. *Consumer-Centric* vs. *Publisher-Centric Paradigm*

The Charisma compiler infers communication patterns from the program text's sequence of published and consumed parameter values, as well as their indices. That is, in Charisma, communication patterns are *implied* by the program text. In particular, Charisma requires that the index vector of the published parameter value be a prefix of the object index vector, or *vice versa*, with the consumed parameter value's index being some (simple) function of the consuming object's index vector. We call this the *consumer-centric* paradigm, since the actual communication pattern is implied by the index of the consumed value's index vector. The consumer-centric approach requires the compiler to *invert* the index expression of the consumed value. To be concrete, consider a *foreach* statement s_1 on object array A. Suppose that s_1 publishes a parameter p, and another *foreach* statement s_2, over array B, consumes its values. Let us denote the object index vectors specified by s_1 and s_2 as $\mathbf{x_1}$ and $\mathbf{x_2}$, respectively. Furthermore, let us denote the expression over the index vector in the consumed p-value as $(\beta_1(\mathbf{x_2}), \ldots, \beta_n(\mathbf{x_2}))$, where the β_i are functions over index vectors of B. Then, while generating code for the $s_1 \rightarrow s_2$ data dependency, the compiler is given $\mathbf{x_1}$ (the index vector of the publishing object in A), and must find the index vector $\mathbf{x_2}$ in B, such that $\beta_i(\mathbf{x_2}) = \mathbf{x_1}(i)$. It is possible to statically invert β_i for only very restricted classes of index vector expressions (e.g. *uniform* and *affine* expressions [7]). However, this is hard to do for expressions of general form. For a set of general β_i, $\mathbf{x_2}$ may have to be obtained by enumerating all possible $\mathbf{x_2}$ vectors, and checking whether $\beta_i(\mathbf{x_2}) = \mathbf{x_1}(i)$ for every i. This method of enumeration can be expensive for large, and higher-dimensional object arrays. Therefore, the implicit specification of communication patterns restricts the scope of expression of Charisma.

The alternative to the consumer-centric scheme is the *publisher-centric* pardigm. In the publisher-centric paradigm, the parameter value in the *LHS* of s_1 would have index vector $(\alpha_1(\mathbf{x_1}), \ldots, \alpha_n(\mathbf{x_1}))$, for some set of functions α_i. The consumed parameter value would simply be $\mathbf{x_2}$. Therefore, in order to get $\mathbf{x_2}(j)$, the compiler would only have to evaluate $\alpha_j(\mathbf{x_1})$ which is much easier to do than invert general functions. This lets us specify general functions α_j in the indices of the published parameter values without affecting the compilation procedure. As we shall see in § V-D and § V-E, this generality is essential in the expression of some important algorithms.

Finally, we note that our adoption of the publisher-centric paradigm requires the introduction of some extra syntax for the specification of multicasts. Recall that in (consumer-centric) Charisma, multicasts were implied by the production of a parameter value by a single producer, and its consumption by multiple consumers. However, in the publisher-centric paradigm, the multicast operation results from the production of a set of *identical*, but differently indexed, parameter values by a single producer, and the consumption of each value in the set by one consumer. Technically, this could be achieved through the wildcard operator, '$*$'. However, this would obscure the fact that the values thus produced are identical, thereby precluding optimizations for multicast communication. Therefore, we introduce the *identicality* operator, '='. The semantics of this operator are best explained through an example. In the code below,

```
foreach (x in A) (p[x,=]) <- A[x].foo();
foreach (x,y in B) B[x,y].bar(p[x,y]);
```

Each object $A[x]$ produces a set of parameter values $p[x, 0]$, $p[x, 1]$, etc. We state the identicality of these parameter values by placing an '=' in the second dimension of the produced values' subscript expression. We will encounter another example of this syntax in § V-C.

B. *Object Index Subspaces*

The Charisma *foreach* construct invokes a particular method on *all* the elements of a specified object array. However, certain classes of algorithms require method invocations on *subsets* of object arrays. This is the case for several dense linear algebra algorithms. For example, in § V-C an algorithm for the *LU* decomposition of dense arrays is expressed in terms of method invocations on *sections* (columns and rows) of a 2D object array. Moreover, some algorithms require a more general definition of subsets than contiguous subranges. As we shall see in § V-A, the tree patterned communication structure of Blelloch's parallel scan algorithm [8] requires the ability to address strided sections of object arrays.

More generally, let us say that an algorithm employs a communication pattern in which, objects from a 1D array A of size N participate in a point-to-point communication only if their indices are in the strided set $\{0{:}N{-}1{:}d\}$. For a given d, we could write Charisma code that would achieve this functionality. However, if d were provided as input to the program, or if it depended, say, on the value of a loop index variable, we would be unable to express it in Charisma.

To address this expressive gap in Charisma we introduce the notion of *object index subspaces*. An object index subspace allows the programmer to *filter* out a subset of pertinent indices drawn from a superset. The superset from which members of an index subspace are drawn is usually a Cartesian product of contiguous (or strided) index ranges. This dissociates the indices of the objects addressed in a *foreach* statement, from the array to which those objects belong. For instance, the programmer may write:

```
ispace sp = {0:N-1:2};
foreach (i,j in sp*sp : MyPred(i,j))
 (p[i,j]) <- A[i,j].f();
```

Here we assume that A is a 2D object array of size $N \times N$. The programmer first creates an index subspace sp, which consists of the even indices along a dimension of A. Next, sp is used in a *foreach* statement to address all $A[i,j]$ such that i and j are both even, and satisfy $MyPred(i,j)$. This is in contrast to the *foreach* construct of Charisma (*cf.* § II), wherein a specified action could only be invoked on *all* the objects of an array. Furthermore, object index spaces, as well as the pruning Boolean predicate may be functions of loop index variables. In this way, we can obtain *dynamic instances* of static communication patterns, a feature that is useful in some of the examples that we will see later § V.

Note that the index subspace concept is superficially similar to the concept of *array domains* in Chapel [9]. However, inasmuch as the latter defines an iteration space for a data-parallel operation via a *forall* statement, it is distinct from the index subspace concept (*cf.* § II).

C. Publishing Ranges of Indexed Values

Finally, we note that in Charisma, the *LHS* of a *foreach* statement can only provide a *constant* number of parameter values. Consequently, each object addressed by the statement can only publish a *constant* number of values. This number is specified in one of two ways. First, the *LHS* may specify a given number of published parameter values, as seen in the Jacobi example of § III-A. There, each $J[i]$ publishes four parameter values, $t[i]$ and $b[i]$. Second, the *LHS* may specify a tagged *list* of published parameter values, as was the case for the matrix multiplication algorithm and the FFT algorithm of § III. In this case each object publishes multiple parameter values. However, the number of published values is still constant – it is the size of the consumer object array along the dimension bearing a '$*$'.

However, for certain applications, the number of parameter values published by an object may depend on inputs given to the program, or the index vector of the object, or perhaps even on loop index variables. For instance, in the current notation we would have to write separate programs for Jacobi relaxations over input domains of different dimensions, even though the parallel structure of the algorithm is independent of the number of dimensions. Therefore, we introduce the ability to publish *ranges* of parameter values through the *foreach* statement. These ranges are specified in a similar manner to object index spaces, and as such can include expressions of loop index variables and parameter value index vectors. For instance, in Listing 9, $PairCalc[s_1, s_2, p]$ publishes a 2D range of parameter values, $d[s_1 g_2 : s_1 g_2 + g_2 - 1, s_2 g_2 : s_2 g_2 + g_2 - 1]$. We do not provide an example in which the expressions of published ranges involve loop index variables.

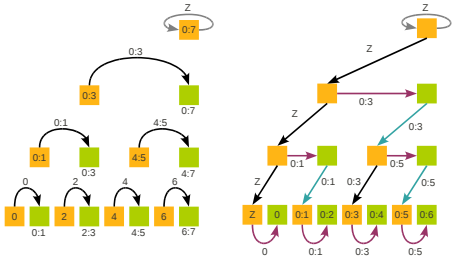
To complete the picture, an object must also have a way to *consume* a variable number of parameter values. For this purpose, we introduce the $*(k)$ notation. In particular, we write $B[\mathbf{y}].g(p[y_1, \ldots, y_{m-1}, *(k), \ldots, y_n])$ to mean that object $B[\mathbf{y}]$ consumes k parameter values along the m-th dimension of the space of published p-values. § V-E presents an example of this notation.

V. A Qualitative Study of the Expressive Power of Charisma 2.0

We now demonstrate that the language extensions proposed in the previous section, although conceptually simple, yield a significantly broader scope of expression than Charisma. In the following, we examine the expression of a number of important static data flow algorithms in Charisma 2.0. We remark at the outset that none of these algorithms can be expressed in Charisma, but all of the algorithms expressible in Charisma (including those in § III) can be expressed in Charisma 2.0.

A. Work-Efficient Parallel Scan

Blelloch [8] has provided a parallel, work-efficient (i.e. $O(n)$) algorithm for the scan operation. We describe its realization in Charisma 2.0, assuming that the input array x has been equidistributed over the elements of an object array A.



(a) Upward phase (b) Downward phase

Fig. 1: Blelloch's parallel scan algorithm.

The algorithm comprises two stages: the *upward* phase, in which a strided point-to-point pattern is employed to compute the intermediate prefix sums at different objects in A (*cf.* Figure 1(a)). The code for this phase is given by the first *for* loop in Listing 4. It involves two object index subspaces, created by the predicates $Sender(i,d) \equiv (i - d/2 + 1) \bmod d == 0$, and $Receiver(i,d) \equiv (i - d + 1) \bmod d == 0$ (lines 4, 7 of Listing 4). Here, loop variable d decides the distance across which partial sums are communicated.

The *downward* phase (given by the *for* loop on line 12 and depicted in Figure 1(b)) propagates sums from the left part

of each subarray to the right. This is done by maintaining a *temporary* sum and a *current* sum.

Listing 4: Blelloch parallel scan

```
1  ispace All = {0:N-1};
2
3  for(d = 2; d <= N; d *= 2){
4   foreach(i in All : Sender(i,d))
5    (p[i+d/2]) <- A[i].right();
6
7   foreach(i in All : Receiver(i,d)) A[i].up(p[i]);
8  }
9
10 A[N-1].startDownward();
11
12 for(d = N; d >= 2; d /= 2){
13   foreach(i in All : Active(i,d))
14     (p[i-d/2]) <- A[i].pubSum();
15
16   foreach(i in All : LastInLeft(i,d))
17     (r[i+d/2]) <- A[i].consAndPubSum(p[i]);
18
19   foreach(i in All : Active(i,d))
20      A[i].consSum(r[i]);
21 }
```

The downward phase is initiated by the $startDownward()$ method on line 10. The $startDownward()$ method sets the temporary sum to the current sum and the current sum to zero. The body of the loop consists of three *foreach* statements over different object index subspaces. The predicate associated with the first (line 13-14) is defined as: $Active(i,d) \equiv (i-d+1) \bmod d == 0$. All objects that satisfy this predicate, publish their current sums via the $pubSum()$ method. Thereafter, the last objects in the left subarrays ($LastInLeft(i,d) \equiv (i-d/2+1) \bmod d == 0$) of the publishers consume the published sums via the $consAndPubSum()$ method (lines 16-17; black arrows in the Figure 1(b)). Method $consAndPubSum()$ sets the temporary sum to the current sum and the current sum to the consumed left sum. It then publishes the *temporary* sum for its $d/2$-away right neighbor via the $consSum()$ method (lines 19-20; rightward, red arrows in Figure 1(b)), whose index necessarily satisfies $Active$. Like $consAndPubSum()$, this method swaps the temporary and the current sums. Note that in the next iteration of the *for* loop, the objects that satisfy $Active$ will be the ones that satisfied $Active$ or $LastInLeft$ in the previous iteration.

B. Gauss-Seidel Solver

The Gauss-Seidel method [10] for solving linear systems exhibits, in general, better convergence rates than the Jacobi method (§ III-A). In contrast to the Jacobi method, the Gauss-Seidel method uses updated point values from the *current* iteration, to the extent possible. For instance, if points in a 2D grid are updated in a left-to-right and top-to-bottom manner, each point uses values from the *previous* iteration for its right and bottom neighbors, and values from the *current* iteration for its left and top neighbors.

Figure 2 shows the *wavefront* of computation that results from this dependency structure. Each color denotes the $N/m \times N$ slab of the 2D grid assigned to one of m objects in G. In order to exploit the inherent parallelism, we set up a pipeline by partitioning each object's slab into tiles of size $N/m \times g$, where g is a grain-size control parameter. As each chare finishes the relaxation on a given tile, it communicates its bottom boundary to its bottom neighbor (solid, downward arrows) and proceeds to its next tile (dashed, rightward arrows).

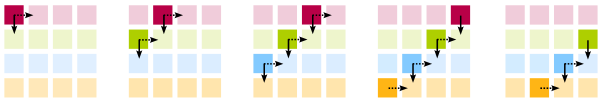


Fig. 2: Gauss-Seidel relaxation.

Listing 5: Gauss-Seidel solver

```
1  while(err >= Threshold){
2   for(I = 0; I < N/g; I++){
3    foreach(j in {0:min(I,m-2)}) (b[j+1]) <- G[j].r1();
4    foreach(j in {1:min(I+1,m-1)}) G[j].r2(b[j]);
5   }
6
7   for(I = 1; I < m-1; I++){
8    foreach(j in {I:m-2}) (b[j+1]) <- G[j].r1();
9    foreach(j in {I+1:m-1}) G[j].r2(b[j]);
10  }
11
12  foreach (j in {0:N-1}) (+err) <- G[j].residue();
13 }
```

The code in Listing 5 shows the complete code in two parts: the first *for* loop constitutes the *fill-up* and *steady-state* phases of the pipeline, whereas the second *for* loop represents the *empty-out* phase of the pipeline. Method *r1* performs the relaxation over grid points, and publishes the boundary for the lower neighbor. Method *r2* performs slightly different tasks depending on the actual index of the object it is invoked on. For $j < m-1$, it simply buffers the boundary received from the top neighbor for the next iteration of relaxation. However, since there is no lower neighbor for $j = m-1$, it performs the relaxation itself.

C. Dense LU Decomposition

The *LU* decomposition algorithm is vital to the efficient solution of dense linear systems. Here, we present the parallel, blocked version of the algorithm in Charisma 2.0. Note that in order to be brief, we do not address the issue of numerical stability and, in particular, *pivoting* in our discussion here. However, it is worth emphasizing that it is certainly possible to express pivoting operations in Charisma 2.0.

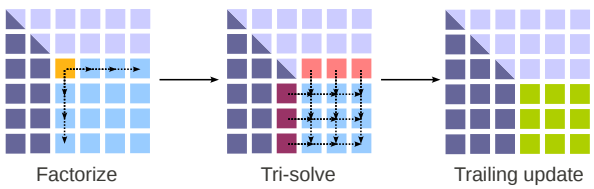


Fig. 3: LU decomposition.

We assume a blocked decomposition of matrix A among a set of 2D objects. In the K-th step of the algorithm the K-th

diagonal block is factorized (yellow block in Figure 3; line 4 of Listing 6), followed by the multicast of the factorized block along the K-th subdiagonal column and row. This leads to two triangular solves (pink and purple; lines 6 and 7) along the so-called *active panels*. Each active row block (pink) multicasts its results to objects in its column below it, and each active column block (purple) multicasts its results to objects in its row and to its right. When these row and column multicasts are received by the trailing submatrix blocks (green; line 11), they perform an in-place update (green).

Listing 6: Dense LU decomposition

```
1  for(K = 0; K < N/g; K++){
2    ispace Trailing = {K+1:N/g-1};
3
4    (d1[K,=], d2[=,K]) <- A[K,K].factorize();
5    foreach(j in Trailing){
6      (c[=,j]) <- A[K,j].utri(d1[K,j]);
7      (r[j,=]) <- A[j,K].ltri(d2[j,K]);
8    }
9
10   foreach(i,j in Trailing*Trailing)
11     A[i,j].update(r[i,j],c[i,j]);
12 }
```

D. Multigrid Solver

The Jacobi and Gauss-Seidel methods exhibit slow convergence rates, especially for low frequency error components. Here, we discuss the multi-grid algorithm, which remedies this problem by performing stencil computations on a hierarchy of structured, nested grids of varying resolutions [11].

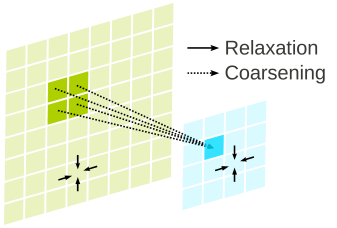


Fig. 4: Coarsening during MG restriction phase.

In each step, the algorithm performs several stencil-based relaxations on a grid of some resolution, and obtains an approximate solution v. Then, the residual $r = b - Av$ is calculated and *restricted*, along with the system matrix A, to the next coarser grid. We then iteratively solve $Ae = b$ to estimate the error e in v. Next, the error e is *prolonged* onto the finer grid, and used in the calculation of an updated approximate solution $u = v+e$. Note that the iterative solution of $Ae = b$ on the coarser grid could be carried out recursively on progressively coarser grids, yielding a better estimate of the actual error e. In the restriction phase, it may happen that the number of points per task falls below a certain threshold, so that the relaxation becomes expensive. This problem is overcome by *coarsening* the decomposition of the grid onto a smaller set of tasks, as depicted in Figure 4. The opposite applies to the prolongation phase, wherein the decomposition may have to be made finer. Due to space constraints, we present the Charisma 2.0 code for only the restriction phase of the multi-grid algorithm. The prolongation phase is similar.

Listing 7: Restriction phase of *V*-cycle MG algorithm

```
1  int n_a = ncharesPerDim;
2  ispace Active = {0:n_a-1}*{0:n_a-1};
3
4  for(level = 0; level < K; level++){
5   while(err >= T_relax){
6    foreach(i,j in Active){
7     (n[i+1,j],s[i-1,j],e[i,j-1], w[i,j+1])
8           <- M[i,j].bdry();
9     (+err) <- M[i,j].rlx(n[i,j],s[i,j],e[i,j],w[i,j]);
10   }
11  }
12
13  foreach(i,j in Active) M[i,j].restrict();
14
15  if(N/(n_a * 2^level) < C && n_a/R > 0){
16   foreach(i,j in Active)
17    (d[i/R,j/R,i%R,j%R]) <- M[i,j].send();
18
19   n_a /= R;
20   Active = {0:n_a-1}*{0:n_a-1};
21
22   foreach(i,j in Active)
23    M[i,j].gather(d[i,j,*(R),*(R)]);
24  }
25 }
```

The code performs a number of restriction steps, denoted by the outer *for* loop (line 6 of Listing 7). In each one of these, a number of Jacobi relaxations are performed (*cf.* § III-A), as shown by the *while* loop on line 8. Next, the problem is restricted to a coarser grid (line 17), which might cause the grain size of relaxation to fall below a programmer-controlled threshold (line 20). If this happens, grid points are communicated to a smaller section of the object array M, so as to coarsen the decomposition of the grid. Notice how the size of the section of active objects in M is altered on lines 24 and 25.

E. *Car-Parrinello* Ab Initio *Molecular Dynamics*

Ab initio approaches to molecular dynamics were designed to address the limitations of classical, forcefield-based approaches. Here we discuss a production-quality implementation of the Car-Parrinello *ab initio* MD technique called OpenAtom [12]. It comprises 10 phases involving 15 chare arrays, so a complete description of the application is beyond the scope of this paper. Moreover, most of the phases employ communication patterns that can be expressed in Charisma itself. Therefore, we focus on those phases that can be expressed in Charisma 2.0, but not in Charisma.

We discuss the interactions of three of the fifteen object arrays that OpenAtom comprises. The $GSpace$ object array holds components of electronic states (of which there are n_s) in Fourier space. Each state is a 3D array of data, whose planes are decomposed onto objects in the 2D $GSpace$ object array of size $n_s \times N_g$, where N_g controls computation granularity. $GSpace$ objects communicate data to objects of a 3D, so-called $PairCalc$ array. $PairCalc$ objects collectively perform

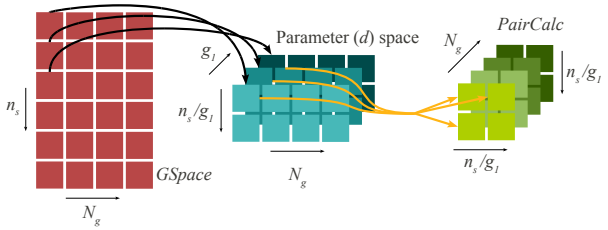


Fig. 5: OpenAtom: *GSpace* to *PairCalc* communication.

a finely decomposed matrix-matrix multiplication operation. Each $GSpace[s,p]$ object must send data to all $PairCalc$ objects that are in the same plane p as it, and which have s as either their first, or second index. This results in row and column multicasts from $GSpace[s,p]$ to $PairCalc[s,*,p]$ and $PairCalc[*,s,p]$. Moreover, in order to ensure good DGEMM performance, the $PairCalc$ objects must receive reasonably sized matrices. Therefore, we have only n_s/g_1 objects along the row and column dimensions of $PairCalc$, each the target of g_1 multicasts from the $GSpace$ objects. In Charisma 2.0 this is expressed as:

Listing 8: OpenAtom: *GSpace* to *PairCalc* phase

```
1 ispace G_all = {0:n_s-1}*{0:N_g-1};
2 ispace PC_all = {0:n_s-1}*G_all;
3
4 foreach(s,p in G_all)
5   (d[s/g1,p,s%g1) <- Gspace[s,p].phase7a();
6
7 foreach(s1,s2,p in PC_all)
8   PairCalc[s1,s2,p].phase7b(d[s1,p,*(g1)],
9                             d[s2,p,*(g1)]);
```

In effect, the values produced by $GSpace[s,p]$ objects in a particular plane p are *folded* onto an extra dimension of the space of published d-values (black arrows in Figure 5; Listing 8, line 4). However, to distinguish its value from those produced by other objects for which s/g_1 yields the same coordinate, each object places its value in the unique coordinate $s \bmod g_1$ of the folding dimension. Next, $PairCalc[s_1,s_2,p]$ consumes each d-value whose index in the first dimension is either s_1 or s_2. For instance, in the figure, yellow arrows show $PairCalc$ objects with $s_1 = 0$ or $s_2 = 0$ and in plane $p = 0$ consuming $d[0,0,*]$ from the 0-th plane of the d parameter space. Since each one of these d-values was published by a single $GSpace$ object, and is consumed by several $PairCalc$ objects, the compiler infers the row and column multicasts that we intend.

Thereafter, the partial results computed by the $PairCalc$ objects must be reduced along the p dimension to obtain the result of the matrix multiplication (depicted by the dashed arrows along the p dimension of the $PairCalc$ array in Figure 6). Moreover, the result must be distributed across a 2D object array, $Ortho$ which performs a so-called *orthonormalization* procedure to recover numerical accuracy. In order to ensure a fine-grained decomposition of work across $Ortho$ there are $n_s g_2/g_1$ objects along each of its dimensions, where $g_2 > 1$ is another constant that controls grain size. Therefore, each

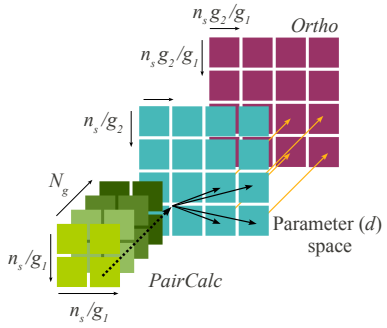


Fig. 6: OpenAtom: *PairCalc* to *Ortho* communication.

$PairCalc$ object scatters its portion of the reduced result to g_2^2 $Ortho$ objects, as shown in Listing 9.

Listing 9: OpenAtom: *PairCalc* to *Ortho* phase

```
1 foreach (s1,s2,p in PC_all)
2   (+d[s1*g2:s1*g2+g2-1, s2*g2:s2*g2+g2-1])
3     <- PairCalc[s1,s2,p].phase8a();
4
5 foreach (s1,s2 in {0:n_o-1}*{0:n_o-1})
6   Ortho[s1,s2].phase8b(d[s1,s2]);
```

VI. Conclusion

In this paper, we have examined Charisma, a language for the elegant and compact expression of static data flow algorithms. The language is part of our vision of *incomplete* but interoperable sets of higher-level notations, which engender a modular, and productive and performance-oriented approach to parallel programming. Due to its incomplete nature, one cannot express data-dependent data flows in Charisma. However, we saw that in its previous form, the language was unable to express some important and widely used static communication patterns, too. We rectified this gap in expressive scope by devising three simple but powerful extensions to Charisma. These extensions are embodied by Charisma 2.0, which has a significantly broader scope of expression than its predecessor. We demonstrated the expressive ability of Charisma 2.0, as well as the fact that it retains the compactness of Charisma, through a number of important algorithms written in the new language. Future work will center on the efficient translation of Charisma 2.0 programs to message-driven specifications, and a comparison of performance between Charisma 2.0-generated code and hand-written Charm++ code.

References

[1] C. Huang and L. V. Kale, "Charisma: Orchestrating migratable parallel objects," in *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[2] L. Kalé and S. Krishnan, "Charm++ : A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[3] P. Jetley and L. V. Kalé, "Static Macro Data Flow: Compiling Global Control into Local Control," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops 2010*, 2010.

[4] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[5] M. Frigo and S. Johnson, "Fftw: an adaptive software architecture for the fft," *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, May 1998.

[6] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.

[7] S. V. Rajopadhye and R. Fujimoto, "Synthesizing systolic arrays from recurrence equations," *Parallel Computing*, vol. 14, no. 2, pp. 163–189, 1990.

[8] G. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Transactions on Computers*, vol. 38, no. 11, November 1989.

[9] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[11] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A Survey of Parallelization Techniques for Multigrid Solvers," in *Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds. SIAM, 2006, ch. 10.

[12] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna, "Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 159–174, 2008.