Bulk-Synchronous Communication Mechanisms in Diderot

John Reppy, Lamont Samuels
University of Chicago
{jhr, lamonts}@cs.uchicago.edu

***Abstract*—Diderot is a parallel domain-specific language designed to provide biomedical researchers with a high-level mathematical programming model where they can use familiar tensor calculus notations directly in code without dealing with underlying low-level implementation details. These operations are executed as parallel independent computations. We use a bulk synchronous parallel model (BSP) to execute these independent computations as autonomous lightweight threads called strands. The current BSP model of Diderot limits strand creation to initialization time and does not provide any mechanisms for communicating between strands. For algorithms, such as particle systems, where strands are used to explore the image space, it is useful to be able to create new strands dynamically and share data between strands.**

In this paper, we present an updated BSP model with three new features: a spatial mechanism that retrieves nearby strands based on their geometric position in space, a global mechanism for global computations (i.e., parallel reductions) over sets of strands and a mechanism for dynamically allocating new strands. We also illustrate through examples how to express these features in the Diderot language. More, generally, by providing a communication system with these new mechanisms, we can effectively increase the class of applications that Diderot can support.

I. Introduction

Biomedical researchers use imagining technologies, such as *computed tomography* (CT) and *magnetic resonance* (MRI) to study the structure and function of a wide variety of biological and physical objects. The increasing sophistication of these new technologies provide researchers with the ability to quickly and efficiently analyze and visualize their complex data. But researchers using these technologies may not have the programming background to create efficient parallel programs to handle their data. We have created a language called Diderot that provides tools and a system to simplify image data processing.

Diderot is a parallel domain specific language (DSL) that allows biomedical researchers to efficiently and effectively implement image analysis and visualization algorithms. Diderot supports a high-level mathematical programming model that is based on continuous tensor fields. We use *tensors* to refer to scalars, vectors, and matrices, which contain the types of values produced by the medical imagining technologies stated above and values produced by taking spatial derivatives of images. Algorithms written in Diderot can be directly expressed in terms of tensors, tensor fields, and tensor operations, using the same mathematical notation that would be used in vector and tensor calculus. Diderot is targeted towards image analysis and visualization algorithms that use real image data, where the data is better processed as parallel computations.

We model these independent computations as autonomous lightweight threads called *strands*. Currently, Diderot only supports applications that allow strands to act independently of each other, such as direct volume rendering [1] or fiber tractography [2]. Many of these applications only require common tensor operations or types (*i.e.*, reconstruction and derivatives for direct volume rendering or tensor fields for fiber tractorgraphy), which Diderot already provides. However, Diderot is missing features needed for other algorithms of interest, such as particle systems,where strands are used to explore image space and may need to create new strands dynamically or share data between other strands.

In this paper, we present new communication mechanisms for our parallelism model that include support for inter-strand communication, global computations over sets of strands, and dynamic allocation of strands. Inter-strand communication is a spatial mechanism that retrieves the state information of nearby strands based on their geometric position in space. Global communication is an mechanism based on sharing information on a larger scale within the program using parallel reductions. Finally, new strands can be created during an execution step and will begin running in the next iteration.

The paper is organized as follows. In the next section, we discuss our parallelism model and applications that benefit from this new communication system. We then present the communication system's design details in Section III and describe important aspects of our implementation in Section IV. A discussion of related work is presented in Section V. We briefly summarize our system and describe future plans for it in Section VI.

II. Background

Diderot is based around two fundamental design aspects: a high-level mathematical programming model, and an efficient execution model. The mathematical model is based on linear algebra and properties of tensor calculus. More background details about our mathematical model and its design are covered in an earlier paper [3]. This paper focuses more on the execution model of Diderot. This deterministic model executes these independent strands in a bulk-synchronous parallel (BSP) fashion [4] [5]. This section provides a brief overview of our execution model and discusses potential new applications that motivates us to provide additional features to this model.

A. Program Structure

Before discussing our exeuction model, it would be beneficial to provide a simple example that describes the structure of a Diderot program. A program is organized into three sections: global definitions, which include program inputs; strand definitions, which define the computational core of the algorithm; and initialization, which defines the initial set of strands. We present a program that uses Heron's method for computing square roots (shown in Figure 1) to illustrate this structure.

```
1  int{} args = load("numbers.nrrd");
2  int nArgs = length(args);
3  input real eps = 0.00001;
4
5  strand SqRoot (real arg)
6  {
7    output real root = arg;
8    update {
9      root = (root + arg/root) / 2.0;
10     if (|root^2 - arg| / arg < eps)
11       stabilize;
12     }
13 }
14
15 initially { SqRoot(args{i}) |
16                        i in 0 .. nArgs-1 };
```

Fig. 1: A complete Diderot program that uses Heron's Method to compute the square root of integers loaded from a file.

Lines 1–3 of Figure 1 define the global variables of our program. Line 3 is marked as an **input** variable, which means it can be set outside the program (input variables may also have a default value, as in the case of `eps`). Line 1–2 loads the dynamic sequence of integers from the file `"numbers.nrrd"` and binds the dynamic sequence to the variable `args` and retrieves the number of elements in the sequence `args`.

Similar to a kernel function in CUDA [6] or OpenCL [7], a strand in Diderot encapsulates the computational core of the application. Each strand has parameter(s) (*e.g.*, `arg` on Line 5), a *state* (Line 7) and an **update** method (Lines 10–12). The strand state variables are initialized when the strand is created; some variables may be annotated as **output** variables (Line 7), which define the part of the strand state that is reported in the program's output. Heron's method begins with choosing an arbitrary initial value (the closer to the actual root of *arg*, the better). In this case, we assign the initial value of *root* to be our real number *arg*. Unlike globals, strand state variables are mutable. In addition, strand methods may define local variables (the scoping rules are essentially the same as C's).

The **update** method of the `SqRoot` strand performs the approximation step of Heron's method (Line 10). The idea is that if *root* is an overestimation to the square root of *arg* then $\frac{arg}{root}$ will be an underestimate; therefore, the average of these two numbers provides a better approximation of the square root. In Line 11, we check to see if we achieved our desired accuracy as defined by `eps`, in which case we *stabilize* the strand (Line 12), which means the strand ceases to be updated.

The last part of a Diderot program is the initialization section, which is where the programmer specifies the initial set of strands in the computation. Diderot uses a comprehension syntax, similar those of Haskell or Python, to define the initial set of strands. When the initial set of strands is specified as a collection, it implies that the program's output will be a one-dimension array of values; one for each stable strand. In this program, each square root strand will produce the approximate square root of an integer.

B. Execution Model

Our BSP execution model is shown in Figure 2. In this model, all active strands execute in parallel execution steps called *super-steps*. There are two main phases to a super-step: a strand update phase and a global computation phase.

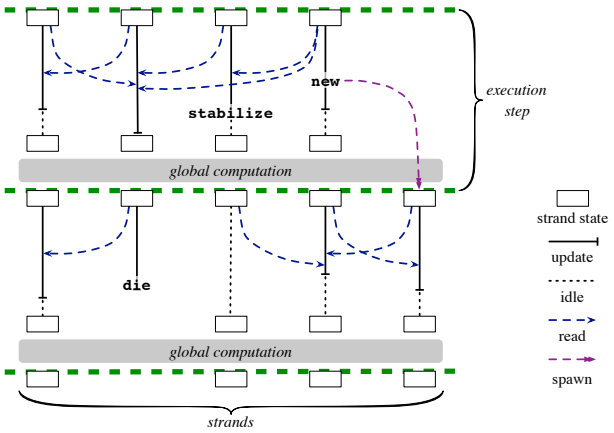

Fig. 2: Illustrates two iterations of our current bulk synchronous model.

The strand update phase executes a strand's update method, which changes its state. During this phase, strands may read the state of other strands but cannot modify it. Strands see the states as they were at the *beginning* of a super-step. This property means we must maintain two copies of the strand state. One for strand reading purposes and one for updating a strand state during the execution of an update method. Also, strands can create new strands that will begin executing in the next super-step. The idle periods represent the time from when the strand finishes executing its update method to the end of the strand update phase. Stable strands remain idle for the entirety of its update phase. Dead strands are similar to stable strands where they remain idle during their update phase but also do not produce any output. Before the next super-step, the global computation phase is executed. Inside this phase, global variables can be updated with new values. In particular, global variables can be updated using common reduction operations.

These updated variables can be used in the next super-step, but note they are immutable during the strand update phase. Finally, the program executes until all of the strands are either stable or dead.

C. Supporting Applications

Particle systems is a class of applications that greatly benefit from this updated BSP model. One example is an algorithm that distributes particles on implicit surfaces. Meyer uses a class of energy functions to help distribute particles on implicit surfaces within a locally adaptive framework [8]. The idea of the algorithm is to minimize the potential energy associated with particle interactions, which will distribute the particles on the implicit surface. Each particle creates a potential field, which is a function of the distances between the particle and its neighbors that lie within the potential field. The energy at each particle is defined to be the sum of the potentials of its interacting neighbors. The global energy of the system is then the sum of all the individual particle energies. The derivative of the global energy function produces a repulsive force that defines the necessary velocity direction. By moving each particle in the direction of the energy gradient, a global minimum is found when the particles are evenly spaced across the surface. The various steps within this algorithm require communication in two distinct ways: interactions between neighboring particles (*i.e.*, computing the energy at a particle is the sum of its neighbors' potentials) and interactions between all particles (*i.e.*, computing the global energy of the entire system). These distinctions motivate us to design a communication system that provides mechanisms for both local and global interactions. Strands need a way for only interacting with a subset of strands or with all the strands in the system.

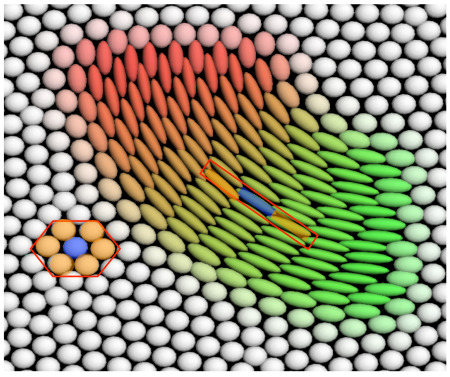

Fig. 3: Glyph packing on synthetic data [9]. The red grids are specialized queries for retrieving neighboring glyphs. The orange glyphs represent the neighbors of the blue glyph performing the query.

Another design goal for inter-strand communication is to provide different ways of collecting nearby neighbors. For example, Kindlmann and Westin [9] use a particle system to locate tensors at discrete points according to tensor field properties and visualizes these points using tensor glyphs. Particles are distributed throughout the field by a dense packing method. The packing is calculated using the particle system where particles interactions are determined via a potential energy function derived from the tensor field. Since the potential energy of a particle is affected by its surrounding particles, neighboring particles can be chosen based on the current distribution of the particles. Figure 3 illustrates an example of using glyph packing on a synthetic dataset. We use this figure to show two possible ways in which neighbors are collected: hexagonal encapsulation and rectangular encapsulation. Both mechanisms only retrieve the the particles defined within those geometric encapsulations and use queries that represent the current distribution in their area. In this example, using specialized queries can lead to a better representation on the underlying continuous features of the field. We need to provide various means of strand interaction, where researchers have the option of choosing the best query that suits their algorithm or their underlying data.

III. Communication Design

A major design goal of Diderot is to provide a programming notation that makes it ease for programmers to implement new algorithms. With the addition of our communication system, we want to continue that philosophy by designing these new features to be ease to use when developing programs. This section provides an overview of the design of the new communication system and examples of its syntax.

A. Spatial Communication

The idea behind spatial communication is shown in Figure 4. A Strand Q needs information about its neighboring strands. One way to retrieve Strand Q's neighbors is by encapsulating it within a spherical shape (the green circle) given a radius r. Any strand contained within this circle is returned to Strand Q as a collection of strands (*i.e.*, Strands A, B, and C). In Diderot, this process is done using predefined *query* functions. The queries are based on the strand's position in world space. Currently we only support this spherical or circle (in the 2D case) type of query, but plan to support various other type of queries, such as encapsulating the strand within a box. Once the collection is returned by the query, Strand Q can then retrieve state information from the collection of neighbors.

Query functions produce a sequence of strand states. Using the strand state, a strand can then gain access to its neighbor's state variables. As mentioned earlier, queries are based on a strand's position in world space; therefore, the strand state needs to contain a state variable called *pos*. The position variable needs to be defined as a `real pos`, `vec2 pos`, or `vec3 pos`.

Processing the queried sequence of strands is performed using a new Diderot mechanism called the `foreach` statement. The foreach statement is very similar to for statements used in many modern languages, such as Python and Java. An iteration variable will be assigned to each strand in the collection returned by the query function. During each iteration of the loop, a strand can use this iterator variable to gain access

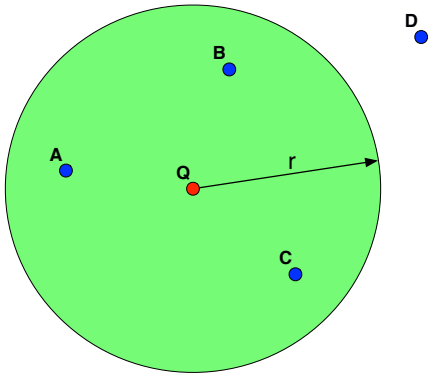

Fig. 4: An example of showing a spherical query (*i.e.*, a circle in 2D). Strand Q (red) produces an encapsulating circle (green) with a predefined radius. Any strands within the encapsulating circle is returned to Strand Q. In this case, the query would return Strands A, B, and C.

to its neighbor's state. Inside foreach block, strands can access the neighbor's state by using the selection operator (*i.e.*, the . symbol) followed by the name of the strand state variable and then the name of the field (similar to accessing a field in a struct variable in the C language).

```
1  real{} posns = load("positions.nrrd");
2  real{} energy = load("energies.nrrd");
3
4  strand Particle (real e, real x,real y) {
5    vec2 pos = [x,y];
6    real energy = e;
7    output real avgEnergy = 0.0;
8    update {
9      int neighborCount = 0;
10     real total = 0.0;
11     foreach(Particle p in sphere(10.0)){
12       neighborCount += 1;
13       total += p.energy;
14     }
15     avgEnergy = total/count;
16     stabilize;
17   }
18 }
```

Fig. 5: A snippet of code demonstrating the spatial communication mechanism

An example of using spatial communication is shown in Figure 5. This code snippet calculates the average energy of each particles neighbors. The code loads positions and energies from a file (Lines 1-2) and assigns (Lines 5-6) each particle (*i.e.*, a strand) a position and energy. It declares accumulator variables (Line 9-10) to hold the total energy and count of the neighbors. The foreach statement (Line 11) declares variable p with its typing being the strand name. The program uses a spherical query with a given radius (*e.g.*, the value of 10 in this example) to retrieve the collection of neighbors. Each neighbor will then be assigned to p for an iteration of the loop. The accumulator variable, neighborCount, is incremented for each neighbor within the collection (Line 12). We use the selection operator (Line 13) to retrieve the energy state variable and added it to the total energy. Finally, we use the accumulator variables (Line 15) to calculate the final output, *avgEnergy*.

B. Global Communication

As mentioned earlier, strands may want to interact with a subset of strands, where they may not be spatially close to one another. This mechanism can be seen as allowing strand state information to flow through a group of strands to produce a result that gives insightful information about the entire state of a group. Once the result is computed, a strand can then use it as way updating their state during the update phase or updating a global property (*i.e.*, a global variable) of the system. We call this flow of information global communication. This feature is performed by using common reduction operations, such as **`product`** or **`sum`**.

Figure 6 shows the syntax of a global reduction in Diderot. r represents the name of the reduction. Table I provides the names of the reductions and a detailed description of their semantics. The expression e has strand scope. Only strand state and global variables can be used in e along with other basic expression operations (*e.g.*, arithmetic or relational operators). The variable x is assigned to each strand state within the set t. This variable can then be used within the expression e to gain access to state variables. The set t can contain either all active strands, all stable strands, or both active and stable strands for an iteration. N represents the name of the strand definition in the program. Syntactically, if a **`strand`** P was defined as the strand definition name then a program can retrieve the sets as follows: *P.active* (all active Ps), *P.stable* (all stable Ps), or *P.all* (all active and stable Ps).

t	::=	N.**`all`**	strand sets
	|	N.**`active`**	
	|	N.**`stable`**	
e	::=	...	previously defined expressions
	|	$r\{e \mid x \; in \; t\}$	reduction expression

Fig. 6: Syntax of a global reduction

The reduction operations reside in a new definition block called **`global`**. Global reductions can only be assigned to local and global variables in this block. Reductions are not permitted to be used inside the strand definition. Before the addition of the global block, global variables were immutable but now they can be modified within this block.

TABLE I: The meanings of reduction operations

Reduction	Semantics	Identity
all	For each strand, *S*, in the set t compute the value e and return TRUE if all the e values in t evaluate to TRUE, otherwise FALSE.	true
max	For each strand, *S*, in the set t compute the value e and return the maximum e value from the set s.	$-\infty$
min	For each strand, *S*, in the set t compute the value e and return the minimum e value from the set t.	$+\infty$
exists	For each strand, *S*, in the set t compute the value e and return TRUE if any one of the e values in t evaluate to TRUE, otherwise FALSE.	false
product	For each strand, *S*, in the set t compute the value e and return the product of all the e values in t.	1
sum	For each strand, *S*, in the set t compute the value e and return the sum of all the e values in t.	0
mean	For each strand, *S*, in the set t compute the value e and return the mean of all the e values in t.	0
variance	For each strand, *S*, in the set t compute the value e and return the variance of the e values in t.	0

```
1  int{} energies = load("energies.nrrd");
2  int nEnergies= length(args);
3  real  maxEnergy = -∞;
4  int iter = 1;
5  strand Particle (real initEnergy) {
6    real energy = initEnergy;
7    update {
8      if(iter == 2 && maxEnergy == energy)
9        print(maxEnergy);
10     if(iter == 2)
11       stabilize;
12   }
13 }
14
15 global {
16   iter += 1;
17   maxEnergy = max{P.energy |
18                   P in Particle.all};
19 }
20 initially [ Particle(energies(vi)) |
21             vi in 0..(nEnergies-1)];
```

Fig. 7: Retrieves the maximum energy of all active and stable strands in the program.

A trivial program that finds the maximum energy of all active and stable strands is shown in Figure 7. Similar to the spatial code, The code loads energies from a file (Line 1) and assigns each strand an energy value (Line 6). The program also defines a global variable *maxEnergy* (Line 3) that holds the maximum energy of all the strands. Inside the global block (Lines 16-18), the **max** reduction is used to find the maximum of the energy state variable. The reduction scans through each *P.energy* value from all active and stable strands and assigns *maxEnergy* the highest value. Inside the update method (Lines 7-9), the strand with the maximum energy prints it out and all strands stabilize. An important aspect to note about this particular code is the need for the *iter* global variable. Remember, the first evaluation of the global computation phase is only after the first execution of the strand update phase. Thus, *maxEnergy* will not have the actual maximum energy until the second iteration. This caveat is why the strands stabilize and the maximum energy is printed when *iter* reaches two.

C. Dynamic Strand Allocation

Diderot does allow the number of strands to vary dynamically, via its die statement, but there is no way to increase the number of active strands in a program. New strands can be allocated during the strand update phase by using the **new** statement within the **update** method. Using the new statement:

```
new P (arguments);
```

where *P* is the name of the strand definition, a new strand is created and initialized from the given *arguments*. Figure 8 shows a snippet of code that uses the new statement. If a strand is wandering alone in the world then it may want to create more strands in its neighborhood. In this case, new strands are created by using the strand's position plus some desired distance from the strand.

```
1  real d =  5.0;
   // desired distance of a new particle.
2  ...
3  strand Particle (real e, real x,real y) {
4    vec2 pos = [x,y];
5    update {
6      int count = 0;
7      foreach(Particle p in sphere(10.0)){
8        count = count + 1;
9          ...
10     }
11     if(count < 1) {
12       new Particle(pos + d);
13     }
14     ...
15   }
16 }
```

Fig. 8: A snippet of code showing how new strands are created when there are no surrounding neighbors.

IV. Implementation

The addition of strand communication to the Diderot compiler produced minimal changes for its front end and intermediate representations. But we added a significant amount of code to our runtime system. In particular, implementing the spatial scheme for spatial communication and determining an efficient way of executing global reductions was required. In this section, we give a brief overview of the Diderot compiler and runtime system and discuss the implementations details of the communication system.

A. Compiler Overview

The Diderot compiler is a multi-pass compiler that handles parsing, type checking, multiple optimization passes and code generation [3]. A large portion of the compiler deals with translating from the high-level mathematical surface language to efficient target code. This process occurs over a series of three intermediate representations (IRs), ranging from a high-level IR that supports the abstractions of fields and derivatives to a low-level language that decomposes these abstractions to vectorized C code. Also at each IR level, we perform a series of traditional optimizations, such unused-variable elimination and redundant-computation elimination using value numbering [10]. The code the compiler produces is dependent the target specified. We have separate backends for various targets: sequential C code with vector extensions [11], parallel C code, OpenCL [7]. Because these targets are all block-structured languages, the code generation phase converts the lowest IR into a block-structured AST. The target-specific backends translate this representation into the appropriate representation and augment the code with type definitions and runtime support. The generated code is then passed to the host system's compiler.

B. Runtime Targets

The runtime system implements the Diderot execution model on the specified target and provides additional supporting functions. For a given program, the runtime system combines target-specific library code and specialized code produced by the compiler. We current support three versions of the runtime system:

1) Sequential C: the runtime implements this target as a loop nest, with the outer loop iterating once per super-step and the inner loop iterating once per strand. This execution is done on a single-processor and vectorized operations.

2) Parallel C: The parallel version of the runtime is implemented using the Pthreads API. The system creates a collection of worker threads (the default is one per hardware core/processor) and manages a work-list of strands. To keep synchronization overhead low, the strands in the work-list are organized into blocks of strands (currently 4096 strands per block). During a a super-step, each worker grabs and updates strands until the work-list is empty. Barrier synchronization is used to coordinate the threads at the end of a super step; although along with the sequential c target, the scheduler can be specialized to eliminate this barrier, which can significantly improve performance.

3) GPUs: Lastly, the GPU runtime is implemented using the OpenCL API. In OpenCL, work items (*i.e.*, threads) are separated into workgroups and execution is done by *warps* (*i.e.*, 32 or 64 threads running a single instruction). Similar to the parallel C runtime, strands are organized into blocks. Each strand block contains a warp's worth of strands for execution. Instead of the runtime using the GPU scheduler, the system implements the idea of *persistent thread* to manage workers [12]. We use multiple workers per compute unit to hide memory latency. Currently, the communication system is implemented only for the sequential and parallel C targets. We plan to add communication support for GPUs in the future.

C. Spatial Execution

When choosing a spatial scheme, it is important to consider how it affects a program's performance. For instance, if a Strand Q queried for its neighbors using a spherical query then a naive implementation would sequentially perform pairwise tests with the rest of the strand population to determine if a strand lies within the sphere's radius. For n strands, this requires: $O(n^2)$ pairwise tests [13]. A Diderot program can contain thousands of active strands at any given super step. This scheme can become too expensive even with a moderate number of strands due to the quadratic complexity. We use a tree-based spatial partitioning scheme, specifically, k-d trees [14][15]. A k-d tree allows one to divide a space along one dimension at a time. We use the traditional approach of splitting along x, y, and z in a cyclic fashion. With this spatial scheme, nearest neighbor searches can be done more efficiently and quickly because we are searching smaller regions versus the entire system. This process thereby reduces the number of comparisons needed.

As stated earlier, the position state variable of a strand is used for constructing the spatial tree. We use a parallel version of the medians of medians algorithm [16] to select the splitting value. As the the tree is built, we cycle through the axes and use the median value as the splitting value for a particular axis. A strand's position can potentially change during the execution of the update method, or new stands can be created during an update. Thus, the tree is rebuilt before the beginning of the update method to take into account these changes.

D. Global Reductions

As stated previously, the global reductions reside in the new global block. This block of code allows for the modification of global variables and is executed at the end of the super-step. The parallel C target uses one of its threads to execute the block in a sequential fashion with the exception of reductions. The process of executing reductions is described below:

1) Reduction Phases: The reductions inside the global block are executed in parallel phases. Each reduction is assigned and grouped into execution phases with other reductions. After parsing and type checking, a typed AST is produced and converted into a simplified representation, where temporaries are introduced for immediate values and operators are applied only to variables. It is during this stage of the compiler reductions are assigned their phase group. The phase in which a reduction is assigned is dependent on whether a another reduction is used to calculate the final value for a global variable. Figure 9 shows an example on how reductions are grouped. Initially, each global variable assigned to a reduction will automatically begin in phase 0 and added to a hash-map that contains all global reduction variables. If the right hand side (rhs) expression contains other global reduction variables the phase assigned to the variable will be: $1 + \theta$, where θ is the highest phase among the rhs global reduction variables. Spitting reductions into phases is important for increasing

performance. Calculating a reduction in parallel occurs a large amount of overhead for running and synchronizing threads. Thus, reductions that can be executed in parallel with other reductions reduces this overhead.

Variable	Phase
a	0
b	1
c	2
d	0
e	0

```
a = mean(...);
b = sum(...) + a;
c = sum(...) * 10;
d = mean(...)/ b;
e = product(...) + 3.0;
```

Fig. 9: Reduction variables are assigned into phase groups for execution. A phase group for a variable is incremented depending on whether other reduction variables reside in the same expression. This process greatly reduces the amount of overhead of performing a reduction versus individual execution.

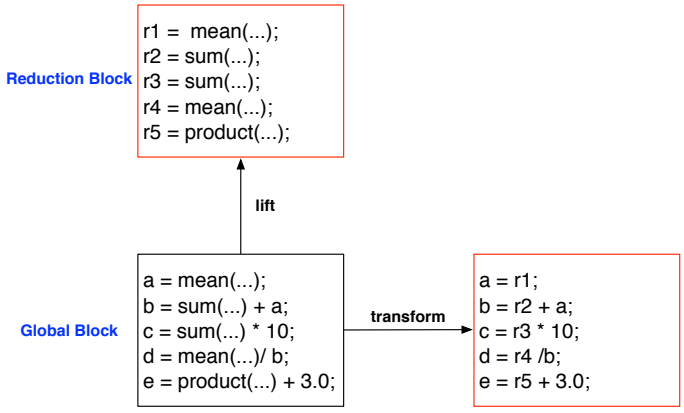

Fig. 10: Reduction lifting makes it easier to group reductions when generating target code. Each reduction is replaced with a new variable in the expression and is lifted into a special code block for reductions.

2) Reduction Lifting: Also during phase identification, we perform an operation called reduction lifting. This process is shown in Figure 9. Each reduction expression is replaced with a temporary variable and lifted into a new block called the *reduction block*. Lifting reductions simplifies the grouping of reductions into their correct phase groups during code generation.

3) Phase Execution: The code generation phase breaks the reductions into their assigned phases. This phase also determines at what time to execute a particular phase. The assignments inside the global block are scanned for reduction variable usage. If an reduction variable is used on the rhs then we look up its phase group and insert a phase execution call before the assignment as shown in Figure 9. The phase execution call is only needed before the first occurrence of any reduction variable in that particular phase.

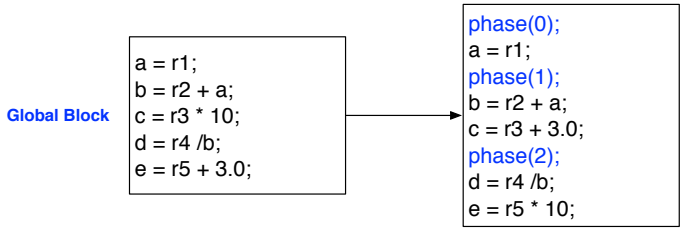

Fig. 11: Phase execution calls are inserted before the execution of the assignment that uses the reduction.

E. Allocating Strands

Diderot maintains a contiguous chuck of memory that represents the strand states for a program. During world initialization (i.e., before the first update method call), We allocate this chunk of memory with enough space to allocate the initial set of strands along with an additional amount of strands that will remain uninitialized. Remember, worker threads process blocks of strands during a super-step. If a worker thread needs to allocate a new strand then it can request an uninitialized strand from that block memory, which it then can initialize once received.

V. Related Work

The work presented in this paper is novel to the area of visualization and image analysis languages. Currently, we are unaware of any other languages that provide the spatial query mechanism that is directly built into the language itself. Although, the concepts of spatial and global communication are studied in various other research fields.

The ideas behind spatial communication in Diderot was influenced by previous works that use agent-based models [17]. These models use simulations based on local interactions of agents in an environment. These agents can represent variety of different objects such as: plants, animals, or autonomous characters in games. With regards to spatial communication, we explored models that are spatially explicit in their nature (*i.e.*, agents are associated with a location in geometric space).

Craig Reynolds's boids simulation is an example of an spatially explicit environment [18]. This algorithm simulates the flocking behavior of various species. Flocking comes from the practice of birds flying or foraging together in a group. A flock is similar to groups of other animals, such as the swarming of insects. Reynolds developed a program called Boids that simulates local agents (*i.e.*, boids) that move according to three simple rules: separation, alignment, and cohesion. The boids simulation influenced our spatial communication design

in regards to how it retrieved its neighbors. When boids are searching for neighboring boids, they are essentially performing a query similar to our queries in Diderot. In particular, they are performing a circle query that encapsulates the querying boid and any nearby boids bounded within a circle. However, our query performs much faster because we use tree-based scheme to retrieve neighbors, while Reynold's query runs in $O(n^2)$ because it requires each boid to be pairwise tested with all other boids.

Agent interactions have also been modeled using interprocess communication systems [19] and computer networks [20]. In these models, processes or nodes can send and receive messages (*i.e.*, data or complex data structures that represent tasks) to other processes. Once agents are mapped to processes, they then can use the messaging protocol defined by the system to query about nearby agents or exchange state information with each other. However, this process requires an application to explicitly adapt or layer its spatial communication model to work within these systems. In Diderot, there is only an implicit notion that strands are defined within a geometric space and one uses query functions to retrieve state information of neighboring strands, which differs from the layering requirement needed for these other communication systems.

The execution of global reductions in Diderot is similar to the MapReduce model developed by Dean and Ghemawat [21]. MapReduce is a programming model used to process parallelize problems that use large data sets, where data mapped into smaller sub-problems and is collected together to reduce to a final output. Many data parallel languages have supported parallel map-reduce, such as the NESL language [22], which has also influenced our design of global reductions. The global block allows reductions to be mapped to global variables After each super-step, the global phase (*i.e.*, our "reduce" step) executes the global block, which performs the actual computation for each reduction. The programmers do not need to worry about lifting the reductions into their own phase, this work is all done by the Diderot compiler.

VI. Discussion and Future Work

Diderot limited itself by only providing features for autonomous strand computations, which excluded other algorithms of interest, such as particle systems. These algorithms require additional communication mechanism to be implemented within the language. With the addition of spatial communication and global reductions we have given programmers the ability to implement and explore more applications with the Diderot language. Although, there are few areas in we plan to improve and work on in the future to provide a better communication system.

Currently we only support a limited number of query functions. We plan to provide additional query functions such as ellipsoidal and hexagonal, to bring more diversity to the options researchers can use within their algorithms. We also are exploring the ability to support abstraction spatial relationships. For example, defining a query to retrieve the 26-neighbors in a 3D grid, or support mesh based methods, where a strand corresponds to a triangle in a finite-element mesh. Query functions are the basis behind spatial communication in Diderot, so allowing for various query options gives a larger range in the types of algorithms we can support.

The new BSP communication mechanism poses a difficult implementation challenge for our GPU target. GPUs are not as flexible in terms of allocating memory dynamically, which can only be done on the host side device. This restriction means that we have to produce a scheme for managing memory efficiently. This scheme needs to determine the appropriate times to dynamically allocate more memory, which can incur a large overhead cost if done naively. With having various components of a Diderot program being allocated on the GPU (*i.e.*, strand state information, spatial tree information, GPU scheduler information, and potential image data), we can potentially run out of memory on the device. If this happens then we may need to come up with a scheme that offloads certain components to the CPU and load only the data that is need for a given iteration. These complications need to be considered when implementing strand communication on the GPU.

VII. Acknowledgments

Portions of this research were supported by the National Science Foundation under award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[1] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '88. New York, NY, USA: ACM, 1988, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/54852.378484

[2] A. Filler, "The history, development and impact of computed imaging in neurological diagnosis and neurosurgery: CT, MRI, and DTI," *Nature Precedings*, 2009. [Online]. Available: http://dx.doi.org/10.1038/npre.2009.3267.5

[3] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A parallel DSL for image analysis and visualization," in *Proceedings of the 2012 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. New York, NY: ACM, Jun. 2012, pp. 111–120.

[4] D. Skillicorn, J. M. Hill, and W. McColl, "Questions and answers about BSP," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[5] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[6] *NVIDIA CUDA C Programming Guide (Version 4.0)*, NVIDIA, May 2011, available from http://developer.nvidia.com/category/zone/cuda-zone.

[7] *The OpenCL Specification (Version 1.1)*, Khronos OpenCL Working Group, 2010, available from http://www.khronos.org/opencl.

[8] M. D. Meyer, "Robust particle systems for curvature dependent sampling of implicit surfaces," in *In SMI 05: Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI 05*. IEEE Computer Society, 2005, pp. 124–133.

[9] G. Kindlmann and C.-F. Westin, "Diffusion tensor visualization with glyph packing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1329–1336, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2006.134

[10] P. Briggs, K. D. Cooper, , and L. T. Simpson, "Value numbering," *Software – Practice and Experience*, vol. 27, no. 6, pp. 701–724, Jun. 1997.

[11] *Using vector instructions through built-in functions*, Free Software Foundation. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

[12] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Inovative Parallel Computing (InPar '12)*, May 2012.

[13] C. Ericson, *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[15] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sep. 1977. [Online]. Available: http://doi.acm.org/10.1145/355744.355745

[16] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, Aug. 1973. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(73)80033-9

[17] N. R. Jennings, "Agent-based computing: Promise and perils," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, ser. IJCAI '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 1429–1436. [Online]. Available: http://dl.acm.org/citation.cfm?id=646307.687432

[18] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 25–34, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37406

[19] "The message passing interface (MPI) standard," http://www.mcs.anl.gov/research/projects/mpi/, accessed: 20/03/2013.

[20] D. C. Walden, "A system for interprocess communication in a resource sharing computer network," *Commun. ACM*, vol. 15, no. 4, pp. 221–230, Apr. 1972. [Online]. Available: http://doi.acm.org/10.1145/361284.361288

[21] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[22] G. E. Blelloch, "NESL: A nested data-parallel language," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1992.