Strand Communication in Diderot

Lamont Kenneth Samuels

April 26th, 2013

Abstract

Diderot is a parallel domain-specific language designed for biomedical image-analysis and visualization algorithms that provides a high-level mathematical programming model. Diderot allows domain experts to implement familiar image analysis and visualization algorithms directly in terms of tensors, tensor fields, and tensor operations, using the same mathematical notation that would be used in vector and tensor calculus. These operations are executed as parallel independent computations. We model these independent computations as autonomous lightweight threads called strands. The current design 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.

We present a strand communication system with two mechanisms: a spatial mechanism that retrieves nearby strands based on their geometric position in space, and a global mechanism for global computations (i.e., reductions) over sets of strands. We also provide a new mechanism for dynamically allocating new strands. We show through experiments that our strand communication system can provide good performance for our sequential runtime.

Contents

List of Figures

List of Tables

Chapter 1

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 [3] that provides tools and a system to simplify image data processing.

Diderot is a 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 imaging technologies mentioned 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. These computations are executed by a collection of autonomous lightweight threads called *strands*. Currently, Diderot only supports applications that allow strands to act independently of each other, such as direct volume rendering [5] or fiber tractography [7]. 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. These features include support for: inter-strand communication, global computations over the strands, and dynamic allocation of strands. By providing a communication system with these new mechanisms, we can effectively increases the class of applications that Diderot can support.

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 [12]. 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.

Another design goal for strand communication is to provide different ways of collecting nearby neighbors. For example, Kindlmann and Westin [11] 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 1-1 illustrates an example of using glyph packing on a synthetic dataset. The neighbors are retrieved using an ellipsoidal encapsulation; however, we use this figure to show two possible ways in which neighbors are collected: hexagonal encapsulation and rectangular encapsulation, even though their distribution is a by-product of the underlying data. 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.

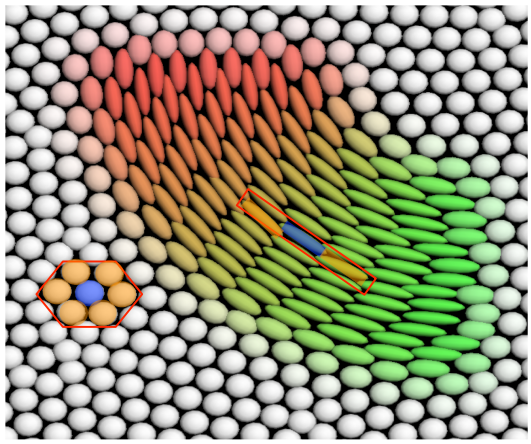


Figure 1-1: Glyph packing on synthetic data [11]. The red grids are specialized queries for retrieving neighboring glyphs. The orange glyphs represent the neighbors of the blue glyph performing the query.

We present a communication system implemented for Diderot that provides two distinct mechanisms of communication. One mechanism is based on interactions with nearby strands and the other mechanism is based on global computations over the strands in the system. In the coming chapters, we describe the additions to the language that allow for strand communication and how they are developed within our compiler. In the next chapter, we provide an overview of the Diderot language and system. We then present the language design of strand communication in Chapter 3 and dynamic strand creation in Chapter 4. We then discuss in Chapter 5 the implementation aspects of strand communication. Chapter 6 presents performance results, followed by a discussion of related work in Chapter 7. Future work is discussed in Chapter 8 and we then summarize our results in Chapter 9.

Chapter 2

Diderot Overview

As mentioned earlier, Diderot targets applications that consist of many independent subcomputations (*i.e.*, strands). We use C-like syntax for our computational notation, but have added common concepts and the direct-style notations from tensor calculus. In this chapter, we provide an overview of Diderot's: types, program structure, execution model, and semantics.

2.1 Types and Values

Diderot has six types of basic values: booleans, strings, integers, tensors, and fixed and dynamic sequences (*i.e.*, sequences that have no predefined size and can also vary in size) of values. The type **`tensor`**`[`σ`]` is a tensor with *shape* $\sigma \in \{i \mid 1 < i\}^*$. The number of dimensions in σ (*i.e.*, the length of σ) is the *order* of the tensor. For example, the types of a scalar and a 3D vector are **`tensor`**`[]` and **`tensor`**`[3]`, respectively. We consider these types to be a 0-order tensor and a 1-order tensor. We also define synonyms for common tensor types, such as **`real`** and **`vec3`**.

Diderot also provides three forms of abstract types: images, fields, and kernels. Images are multidimensional arrays of tensor data. The type **`image`**`(`d`) [`σ`]` is the abstract type of image data, where $d \geq 1$ specifies the dimension of the data and σ specifies the shape. The discrete data within the image is reconstructed using convolution kernels. The type **`kernel#`**k is the type of a kernel that is of class C^k (*i.e.*, k is the number of continuous derivatives it has). Diderot provides a number of helpful built-in kernels, such as the C^1 interpolating Catmull-Rom cubic spline `ctmr`, and the C^0 `tent` for linear interpolation. Lastly, **`field#`**k`(`d`) [`σ`]` is the type of continuous tensor fields, where k is the number of continuous derivatives, d is the dimension of the field's domain, and σ is the shape of its range. A field is a function from d-dimensional space to tensors with the shape σ.

2.2 Program Structure

A Diderot 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. To illustrate this structure, we present a program that uses Heron's method for computing square roots (shown in Figure 2-1) as a running example.

```
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
9      update {
10         root = (root + arg/root) / 2.0;
11         if (|root^2 - arg| / arg < eps)
12             stabilize;
13     }
14 }
15
16 initially { SqRoot(args{i}) | i in 0 .. nArgs-1 };
```

Figure 2-1: Heron's method code

2.2.1 Global Definitions

Lines 1–3 of Figure 2-1 define the global variables of our example. Global variables in Diderot are immutable. 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`). The Diderot compiler synthesizes glue code that allows input variables to be set on the command-line or by a library call. Line 1 loads the dynamic sequence of integers from the file `"numbers.nrrd"` and binds the dynamic sequence to the variable `args`. The `load` function may only be used in global part of the program. Line 2 uses the `length` function to retrieve the number of elements in the sequence `args`.

2.2.2 Strands

Similar to a kernel function in CUDA [14] or OpenCL [10], 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). A stabilized strand ceases to be updated. A strand may also have a **stabilize** method that is invoked when the strand stabilizes to perform any final computations.

2.2.3 Initialization

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. For example, we specify a collection of integers, where each integer is retrieved from the dynamic sequence variable (*args*) in Line 16.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. For example, each square root strand will produce the approximate square root of an integer.

Diderot also allows one to specify an initial *grid* of strands by using "`[ ]`" as the brackets around the comprehension (instead of " { }'). In this case, The grid structure is then preserved in the output.

2.3 Bulk Synchronous Model

Diderot uses a bulk-synchronous parallelism model [18, 19] as shown in Figure 2-2. In this model, all active strands execute in parallel steps, which we call *super steps*. During a super step, each strand's update method is evaluated once. The idle periods represent the time from when the strand finishes executing its update method to the end of the update phase. A stable strands remain idle

for the entirety of its update phase. The program executes until all of the strands are either stable or dead.

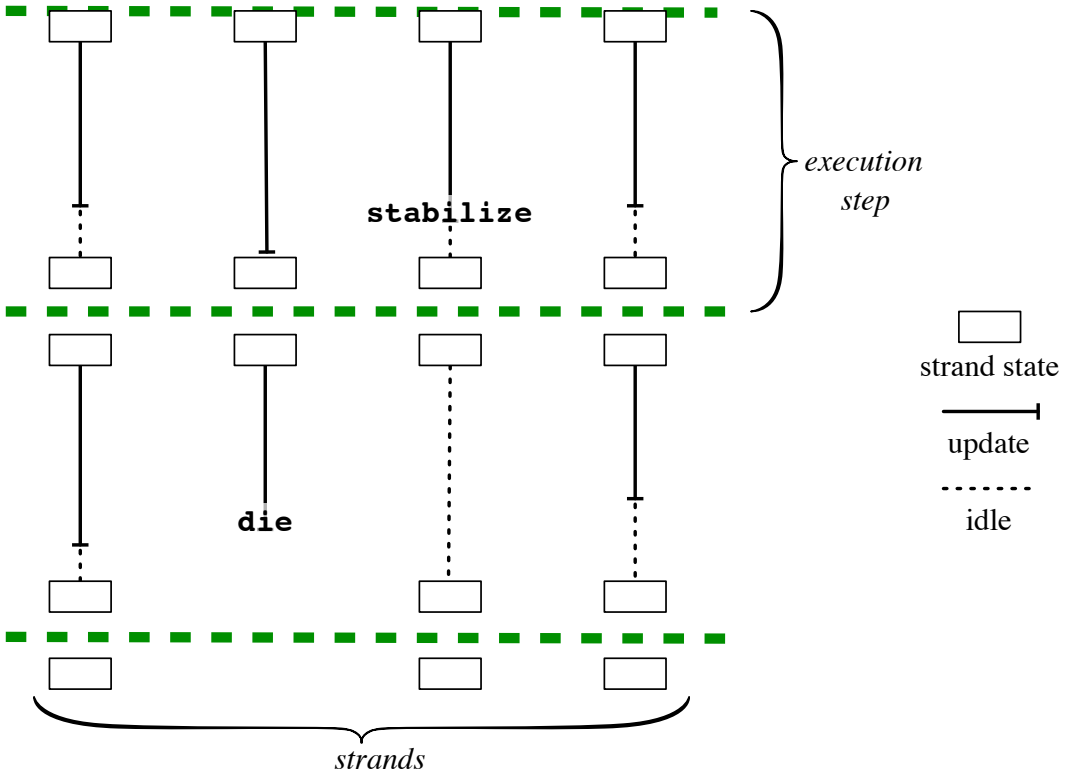


Figure 2-2: Illustrates two iterations of our current bulk synchronous model.

2.4 Semantics

Before discussing our semantics, we define the notation that we use throughout its description. If A and B are two sets, we use the notation A $\xrightarrow{\text{fin}}$ B to denote the set of finite maps from A to B and A $\cup$ B as there union. We write $\{a_1 \mapsto b_1, ..., a_n \mapsto b_n\}$ for the finite map that maps a_1 to b_1, etc. We write the notation $(B \text{ of } C)$ to mean we are selecting the value B from the tuple C. For a finite map, f, we write dom(f) for its domain and rng(f) for its range. if f and g are finite maps, we write $f \pm g$ for the finite map

$$\{x \mapsto g(x) \mid x \in dom(g)\} \cup \{x \mapsto f(x) \mid x \in dom(f) \setminus dom(g)\}$$

and we write $f[x \mapsto v]$ to represent a binding of a variable to a value that is defined as:

$$f[x \mapsto v](y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{if } x \neq y. \end{cases}$$

2.4.1 Syntax

We use the following classes of identifiers in the syntax of Diderot.

τ	$\in$	TYPES	types
x	$\in$	VAR = {y,z,...}	local, global, and state variables
f	$\in$	PRIMFUNS	primitive operations and functions
c	$\in$	CONST	constants (e.g.: booleans,integers,etc..)
v	$\in$	VALUE	values
e	$\in$	EXPR	expressions
D	$\in$	STATEMENTDECLARATION	statement declarations
s	$\in$	STATEMENT	statements
S	$\in$	STRANDDEFINITION	strand definition
N	$\in$	STRANDNAME	strand name
i	$\in$	INITIALLYDEFINITION	initially definition
m	$\in$	STRANDMETH	strand method
p	$\in$	PROGRAM	Diderot program

The syntax of Diderot is given in Figure 2-3. A Diderot program consists of a sequence of statement declarations, which represent the global variables, a strand definition, and an initially block. A strand definition contains: an expression that can be used to initialize its state, a sequence of statement declarations, which represent its state, and methods, such as the **`update`** method. A strand method include statements, such as assignments and conditionals. Common tensor calculus operations and functions are defined as expressions. The Diderot syntax uses unicode characters to represent mathematical constants (π) and a rich set of operations on tensors, such as: dot product (`u•v`), cross product (`u×v`), tensor product (`u⊗v`), and vector norm (`|u|`). Continuous tensor fields are created using high-order operations, such as addition and subtraction and convolving image data with kernels (`img⊛bspln3`). We also support differentiation of fields using the notation ∇ (for scalar fields) and $\nabla\otimes$ (for higher-order tensor fields). Finally, the **`initially`** block is used to allocate the initial collection of strands.

$$
\begin{array}{rcl}
p & ::= & D; S; i \\
\\
D & ::= & \tau\ x\ =\ e \\
 & | & D_1; D_2 \\
\\
S & ::= & \texttt{strand}\ (\tau\ x)\{D; m\} \\
\\
m & ::= & \texttt{update}(s) \\
\\
i & ::= & \texttt{initially}\{e_1 \mid x\ \texttt{in}\ e_2\ \texttt{..}\ e_3\} \\
\\
\tau & ::= & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \\
 & | & \tau\{n\} \\
 & | & \tau\{\} \\
 & | & \texttt{tensor}[\sigma] \\
 & | & \tau \mapsto \tau' \\
 & | & \texttt{image}\,(d)\ [\sigma] \\
 & | & \texttt{kernel\#}k \\
 & | & \texttt{field\#}k\,(d)\ [\sigma] \\
 & | & \texttt{strand} \\
\\
c & ::= & 1, 2, 3, ... \\
 & | & true, false \\
 & | & ... \\
v & ::= & c \\
 & | & vs \\
\\
vs & ::= & [] \mid v :: vs
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & v \\
 & | & e_1\{e_2\} \\
 & | & f(e_1, ..., e_n) \\
 & | & e_1\ \texttt{if}\ e_2\ \texttt{else}\ e_3 \\
\\
s & ::= & \{s\} \\
 & | & D \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & | & x\ =\ e \\
 & | & s_1; s_2 \\
 & | & \texttt{stabilize} \\
 & | & \texttt{die}
\end{array}
$$

$$
\begin{array}{rcl}
d & \in & \{n \mid n > 1\} \\
\sigma & \in & \{d_1\ , ...,\ d_n\} \\
k & \in & \{n \mid n \geq 0\}
\end{array}
$$

Figure 2-3: The Diderot syntax

2.4.2 Static Semantics

The Diderot typing judgments are written with respect to a static environment Γ, which consists of a variable environment (ψ), and a strand variable environment (γ), which contains the types of the strand state variables. These environments are defined in Figure 2-4. The judgement forms and their meanings are defined in Table 2.1.

$$
\begin{array}{rcll}
\psi & \in & \textsc{VarEnv} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type} & \text{variable typing environment} \\
\gamma & \in & \textsc{StrandTyEnv} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type} & \text{strand state typing environment} \\
\Gamma & \in & \textsc{Env} = \textsc{VarEnv} \times \textsc{StrandTyEnv} & \text{typing environment}
\end{array}
$$

Figure 2-4: Definitions of the typing enironments

Table 2.1: Judgement forms and their meanings

Judgement Form	Meaning
$\Gamma \vdash s : \mathbf{ok}$	Statement s is *well-formed* w.r.t. Γ.
$\Gamma \vdash e : \tau$	Expression e has type τ.
$\Gamma \vdash S : \gamma$	Strand definition S *defines* environment $(\gamma \ of \ \Gamma)$.
$\Gamma \vdash i : \mathbf{ok}$	Initially definition i is *well-formed* w.r.t. Γ.
$\Gamma \vdash D : \Gamma'$	Declaration D *defines* environment Γ'.
$\Gamma \vdash m : \mathbf{ok}$	Method m is *well-formed* w.r.t. Γ.
$\Gamma \vdash p : \Gamma'$	Program p *defines* environment Γ'.

The rules for declarations, strand definitions and Diderot programs are shown in Figure 2-5. The first two rules show how the variable typing environment is augmented when typing new declarations. The next rules illustrate how the strand the strand typing environment is created from the declarations defined in the strand definition. Lastly, the typing rule for a program produces the initial typing environment, where $(\psi \ of \ \Gamma)$ is built from typing the global declarations, and $(\gamma \ of \ \Gamma)$ is created from typing the strand definition.

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma' = \langle \{x \ \mapsto \tau\}, \emptyset \rangle}{\Gamma \vdash \tau \ x \ = \ e : \Gamma'}$$

$$\frac{\Gamma \vdash D_1 : \Gamma' \qquad \Gamma + \Gamma' \vdash D_2 : \Gamma''}{\Gamma \vdash D_1 \ ; D_2 : \Gamma' \pm \Gamma''}$$

$$\frac{(\psi[x \mapsto \tau] \ of \ \Gamma) \vdash D : \Gamma' \qquad \langle (\psi[x \mapsto \tau] \ of \ \Gamma) \pm (\psi \ of \ \Gamma'), (\psi \ of \ \Gamma') \rangle \vdash m : \mathbf{ok}}{\Gamma \vdash \textbf{strand} \ (\tau \ x)\{D; m\} : (\psi \ of \ \Gamma')}$$

$$\frac{\langle \emptyset, \emptyset \rangle \vdash D : \Gamma' \qquad \Gamma' \vdash S : \gamma \qquad \langle (\psi \ of \ \Gamma'), \gamma \rangle \vdash i : \mathbf{ok}}{D; S; i : \langle (\psi \ of \ \Gamma'), \gamma \rangle}$$

Figure 2-5: Static semantics for a declaration, strand definition, and a program

2.4.3 Dynamic Semantics

We augment the Diderot syntax to include the runtime form of a strand definition and its components as shown in Figure 2-6.

T	$\in$	STRANDSTATE	strand state
I	$\in$	STRANDIDENTIFIER	strand identifier
k	$\in$	STRANDSTATUS	strand status
σ	$\in$	VARSTATE $=$ VAR $\rightarrow$ VALUE	local variable state
I	$::=$	c	
k	$::=$	**active** | **stable** | **dead**	
T	$::=$	$\langle I;\ k;\ \sigma \rangle$	
v	$::=$	...	previously defined values
	|	T	strand state

Figure 2-6: The runtime form of a strand definition

The form $\langle I;\ k;\ \sigma \rangle$ denotes a strand state. The *I* portion represents a strand identifier, which is an unique value assigned to each strand. The *k* portion represents a strand status, which determines if a strand is **active**, **stable**, or **dead**. The σ portion denotes a local variable state, which is a function that binds state variables to values.

To define the evaluation relation, we need some additional definitions as show in Figure 2-7.

Δ	$=$	STRANDDEFINITION	strand definition of the program
η	$\in$	STRANDIDENV $=$ STRANDID $\xrightarrow{\text{fin}}$ STRANDSTATE	strand identifier environment
Σ	$\in$	GLOBALSTATE $=$ (STRANDDEFINITION $\times$ STRANDIDENV $\times$VARSTATE $\times$ STRANDTYENV)	global state

Figure 2-7: Evaluation environments and functions

Δ stores the strand definition of a Diderot program. Currently, there can only be one strand definition defined within a Diderot program. We use Δ to produce a *fresh* copy of a strand definition when a new strand is allocated. The strand identifier environment is a finite function from strand

identifiers to strand states. The global state is a tuple that stores the strand definition, the strand identifier environment, a local variable state, which binds global variables to values, and the strand typing environment. This environment is needed during the initialization of a new strand. The variables stored in the strand's local state are defined by the domain of the strand typing environment.

The dynamic semantics is split into two phases. The first phase produces the global state by evaluating a program's global variables, strand definitions, and initially block. The second phase repeatedly evaluates the update function of a strand state, while its status remains **`active`**. Figure 2-8 specifies the semantics of evaluating an expression and statement within a strand method. Expressions and statements are evaluated with the global state context and a local state context (σ), which represents the method's local variable bindings. The first rule states that an expression evaluates to a value (v). The next rule shows how the evaluation of a statement leads to a new local state. For example, if an assignment statement is executed then the variable is changed to a new value within the local state; therefore, this will lead to a new local state with that variable updated.

$$(1) \quad \Sigma, \sigma \vdash \ e \Downarrow \ v$$

$$(2) \quad \Sigma, \sigma \vdash \ s \Downarrow \ \sigma'$$

Figure 2-8: Expression and statement evaluation rules

Chapter 3

Strand Communication Design

We provide two mechanisms for strand communication. The first is spatial communication, which allows strands to interact with each other based on their world coordinates. Each strand uses special query functions to identify neighboring strands. The second involves communication between a subset of the strand population. For example, if strands have a state variable called *energy* then they may want to know the average energy of all active strands. We refer to this type of communication as global communication. We provide common reduction operations, such as mean, sum, product, etc., to work across a set of strands. Therefore, strands can retrieve the average energy of all active strands by using the mean reduction. The remainder of this chapter provides a more detailed description of the semantics of these mechanisms and how they can be used within a Diderot program.

3.1 Spatial Communication

The idea behind spatial communication is shown in Figure 3-1. A Strand Q needs information about its neighboring strands. One way to retrieve Strand Q's neighbors is by encapsulating it within a circle (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 rather than a sphere or circle. Once the collection is returned by the query, Strand Q can then retrieve state information from the collection of neighbors.

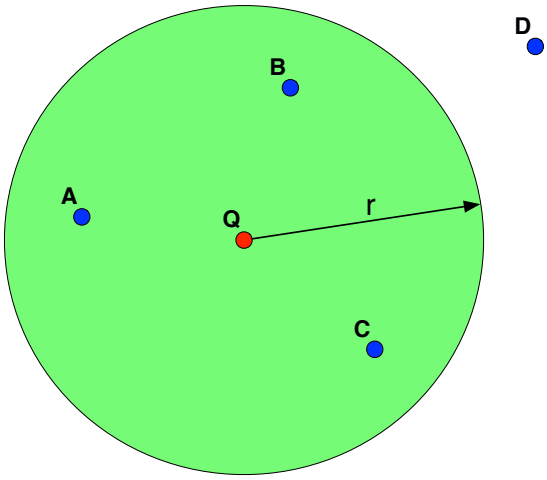


Figure 3-1: An example of showing a circle query. Strand Q (red) produces an encapsulating circle (green) with a predefined radius. Any strands within the encapsulating circle is returned to Strand Q. In this case, the query would return Strands A, B, and C.

3.1.1 Query Functions

The static semantics of a query function is depicted in Figure 3-2. 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`**.

$$
\begin{array}{lcll}
q & \in & \text{QUERYFUN} = \{\texttt{sphere},...\} & \text{query functions} \\
e & ::= & ... & \text{previously defined expressions} \\
 & | & q(e) &
\end{array}
$$

$$
\frac{\Gamma \vdash e : \tau \qquad pos \in dom(\gamma\ of\ \Gamma) \qquad q : \tau \mapsto \texttt{strand}\{\}}{\Gamma \vdash q(e) : \texttt{strand}\{\}}
$$

Figure 3-2: Typing rule for query functions

The dynamic semantics of a query function is described in Figure 3-3. The dynamic semantics includes a special function called *query*, which takes in the global state, the querying strand's position, the type of query (*e.g.*, a spherical query) and a value (*i.e.*, the argument(s) of the query function). The query function produces a dynamic sequence of strand states based on the position of the querying strand and the type of query requested.

$$\frac{\Sigma, \sigma \vdash e \Downarrow v \qquad vs = query(\Sigma, \sigma(pos), q, v)}{\Sigma, \sigma \vdash q(e) \Downarrow vs}$$

Figure 3-3: Dynamic semantics for query functions

3.1.2 Strand Iteration

Processing the queried sequence of strands is performed using a new Diderot mechanism called the *foreach* statement. Figure 3-4 shows the typing rule for the foreach statement. The expression e_1 produces a sequence of type τ. The foreach iterates over the sequence and assigns its iterator variable *x* to have type τ for the judgement of the statement s_0.

$$\begin{array}{rcll} s & ::= & \ldots & \text{previously defined statements} \\ & | & \texttt{foreach}(x \texttt{ in } e)s & \text{foreach statement} \end{array}$$

$$\frac{\Gamma \vdash e_1 : \tau\{\} \qquad (\psi \text{ of } \Gamma)[x \mapsto \tau] \vdash s_0 \textbf{ ok}}{\Gamma \vdash \texttt{foreach}(x \texttt{ in } e_1)\ s_0 \textbf{ ok}}$$

Figure 3-4: Typing rule for the foreach statement

Figure 3-5 shows the dynamic semantics of the foreach statement. The evaluation of the expression e_1 will produce a value as shown in the first rule. The second rule binds the first value of the sequence to the iterator variable *x* in the local state σ, which is used to evaluate the statement block, s_0. Any values updated after the processing of s_0 are now defined within the new state, σ'. We use this new state to evaluate the rest of the sequence *vs*, which produces the final updated state, σ''. The third rule illustrates the termination of the foreach. When the sequence is empty then we continue executing the rest of the program.

$$\frac{\Sigma, \sigma \vdash e_1 \Downarrow vs \qquad \Sigma, \sigma \vdash \texttt{foreach } (x \texttt{ in } vs)\ s_0 \Downarrow \sigma'}{\Sigma, \sigma \vdash \texttt{foreach}(x \texttt{ in } e_1)s_0 \Downarrow \sigma'}$$

$$\frac{\Sigma, \sigma[x \mapsto v] \vdash s_0 \Downarrow \sigma' \qquad \Sigma, \sigma' \vdash \texttt{foreach } (x \texttt{ in } vs)\ s_0 \Downarrow \sigma''}{\Sigma, \sigma \vdash \texttt{foreach}(x \texttt{ in } v :: vs)\ s_0 \Downarrow \sigma''}$$

$$\frac{}{\Sigma, \sigma \vdash \texttt{foreach}(x \texttt{ in } [])\ s_0 \Downarrow \sigma}$$

Figure 3-5: Dynamic semantics for foreach statement

3.1.3 Neighbor Count Example

An example of using spatial communication is shown in Figure 3-6. This example simply counts the number of neighbors returned by the query function. The program loads positions from a file (Line 1) and assigns (Line 8) each particle (*i.e.*, a strand) a position. It declares an accumulator variable (Line 11) to keep track of the number of neighbors. The foreach statement (Line 12) declares variable *p* of type `name`. The program uses a spherical query (Line 12) to retrieve its neighbors, where each neighbor will be assigned to the foreach iterator variable, *p*. The accumulator variable is incremented for each neighbor within the collection (Line 13). Finally, the neighbor count is assigned to the strand's output variable (Line 15).

```
1  real{} posns = load("positions.nrrd");
2  int numParticles = 64;
3  int dimSize = 2;
4  int strandSize = 2;
5  real queryRadius = 10;
6
7  strand Particle (real x,real y) {
8      vec2 pos = [x,y];
9      output int neighborCount = 0;
10     update {
11         int count = 0;
12         foreach(Particle p in sphere(query_radius)){
13            count = count + 1;
14         }
15         neighborCount = count;
16         stabilize;
17     }
18 }
19 initially {Particle(posns{i*dimSize},posns{i*dimSize+1}) | i in 0 .. numParticles-1 };
```

Figure 3-6: Diderot code that counts the number of nearby neighbors for a Particle.

3.1.4 Accessing Strand State Information

A variable assigned to a strand state can access its state variables. For example, the variable *p* in the **`foreach`** statement in Figure 3-6 can be used to retrieve the values of the *pos* and *neighborCount* state variables. The selection operator can be used to retrieve state variables of a strand state. The typing judgement for the selection operator is depicted in Figure 3-7. The rule states that the type of the expression e is a strand type. This is required since only variables of type strand can access state variables. The rule shows that the type of the state variable (x) in γ is the resulting type of the selection operator.

$$
\begin{array}{rcll}
e & ::= & \ldots & \text{previously defined expressions} \\
 & | & e\,\texttt{.}\,x & \text{selection operator}
\end{array}
$$

$$
\frac{\Gamma \vdash e : \textbf{strand} \qquad (\gamma \text{ of } \Gamma)(x) : \tau \quad x \in dom(\gamma \; of \; \Gamma)}{\Gamma \vdash e\,\texttt{.}\,x : \tau}
$$

Figure 3-7: Typing judgement of the selection operator

Figure 3-8 illustrates the dynamic semantics for accessing a state variable using the selection operator. The evaluation of e is a strand state. The local variable state defined within the strand state is used to find the value of state variable (x).

$$
\frac{\Sigma, \sigma \vdash e \Downarrow \langle I;\ k;\ \sigma;\ m \rangle \qquad \sigma(x) = v}{\Sigma, \sigma \vdash e\ \texttt{.}\,x \Downarrow v}
$$

Figure 3-8: Dynamic semantics for the dot expression

3.1.5 Average Energy Example

Figure 3-9 is an example of how strands retrieve state information from neighboring strands. In particular, strands are accessing each neighbor's energy state variable and then averaging the engeries. We use the selection operator (Line 11) to retrieve each neighbor's energy variable using the syntax *variable*`.`*state_variable* (*e.g.*, p.energy).

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

Figure 3-9: A snippet of code showing how to access strand state information

3.2 Global Communication

As mentioned earlier, strands may want to interact with a subset of strands. The idea of this communication mechanism is based on sharing information on a larger scale within the system. Strands can communicate with other strands regardless of where they are positioned in world space. Global communication is performed by using common reduction operations, such such as **`product`** or **`sum`**. Once a reduction is computed, all strands have access to that value.

3.2.1 Reductions

A general typing judgement for reductions is listed in Figure 3-10. Similar to query functions, the set expression (e_2) produces a dynamic sequence. The evaluation of e_1 should have the same type as the reduction's return type. We currently provide eight different reduction operations. All logical reductions (*i.e.*, all, exists) should produce a final type of **`bool`**. The reductions: minr, maxr, product, mean, variance and sum produce a resulting type of **`real`**. Since the judgement form of a reduction has a type τ', the evaluation of e_1 and $(R\ of\ \Gamma)(r)$ have the same type (τ').

$$
\begin{array}{rcll}
r & \in & \textsc{Reductions} = \{\texttt{all},\ \texttt{max},\ \texttt{min}, \text{ etc..}\} & \text{global reductions} \\
R & \in & \textsc{ReductionsEnv} = \textsc{Reductions} \mapsto \textsc{Type} & \text{reduction environment} \\
\Gamma & = & (\ldots \times \textsc{ReductionsEnv}) & \text{typing environment}
\end{array}
$$

$$
\frac{\Gamma \vdash e_2 : \tau\{\} \qquad (\psi \text{ of } \Gamma)[x \mapsto \tau] \vdash e_1 : \tau' \qquad (R\ of\ \Gamma)(r) = \tau'}{\Gamma \vdash r\{e_1 \mid x\ in\ e_2\} : \tau'}
$$

Figure 3-10: Typing judgment for reduction operations

Figure 3-11 illustrates the syntax of a global operation in Diderot. The expression e has strand scope. Only strand state variables can be used in e. The variable *x* is assigned to each strand state within the set *t*. It can be used within the expression e to gain access to state variables. The set *t* can contain all active strands, all stable strands, or both active and stable strands for an iteration. Syntactically, if a **`strand`** P was defined then a program can retrieve the sets as follows: *P.active* (all active Ps), *P.stable* (all stable Ps), or *P.all* (for both active and stable Ps).

$$
\begin{array}{rcll}
t & ::= & N.\texttt{all} & \text{strand sets} \\
 & | & N.\texttt{active} & \\
 & | & N.\texttt{stable} & \\
e & ::= & \ldots & \text{previously defined expressions} \\
 & | & r\{e \mid x \; in \; t\} & \text{reduction statement}
\end{array}
$$

Figure 3-11: Syntax of a global reduction

3.2.2 Operational Meanings

Table 3.1 provides a detailed description of each reduction in Diderot. The table gives both the syntax and semantical meanings for each reduction.

Table 3.1: Semantical meanings for each reduction operation

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
maxr	For each strand, S, in the set t compute the value e and return the maximum e value from the set s.	$-\infty$
minr	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

3.2.3 The Global Block

The reduction operations reside in a new definition block called **global**. The global reduction variables are assigned to their reduction operation inside this block. Before the addition of global communication, global variables were immutable but now they can be modified within this block. Figure 3-12 shows the addition of the global block to a program's definition.

$$g \in \text{GLOBALBLOCK} \quad \text{global block definition}$$

$$g ::= \{s\}$$

$$p ::= D; S; g; i$$

Figure 3-12: Syntax of the global block

3.2.4 Finding the Maximum Energy Example

The code in Figure 3-13 shows an example of using global communication. The program finds the maximum energy from all active and stable strands. The code loads energies from a file (line 1) and assigns each strand an energy value (line 4). Inside the global block (Lines 15-17), the maxr reduction is used to find the maximum energy in the set. Inside the update method, each strand checks to see if their energy is the maximum energy and the strand with the maximum energy prints it out.

```
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(maxEnergy == energy)
9              print(maxEnergy);
10          if(iter > 2)
11             stabilize;
12     }
13 }
14
15 global {
16      maxEnergy = maxr{P.energy | P in  Particle.all};
17      iter +=1;
18 }
19 initially [ Particle(energies(vi)) | vi in 0..(nEnergies-1)];
```

Figure 3-13: Retrieves the maximum of the program

3.3 Strand State Values

When accessing a state variable, the value returned is the value assigned from the previous iteration. Recall that Diderot uses a bulk synchronous model of execution. In this model, strands are executed in *parallel* steps; therefore, strands can potentially be updating their state at different times, which may lead to nondeterministic results if accessed by another strand. For example, if a Strand A is

accessing a Strand B's state while Strand B is updating it then it is unknown if the state Strand A receives is the old or updated state. Thus, strands will access the values computed from the previous super-step. Figure 3-14 diagrams an example of using the previous iteration's strand values. The top green boxes represent the strand state variables for a Strand A and Strand B after Iteration i. During an iteration, the strand state variables *energy* and *pos* are updated before executing the foreach statement in the update function. Assuming there are only two strands running, the strands retrieve each other's energy values (*i.e.*, nEnergy) within the foreach block. But the values assigned are from iteration i and not iteration i + 1. This result is shown in the bottom green boxes. Furthermore, querying functions also use the *pos* values from the previous iteration. Thus, the querying function, *sphere*, will use the values $\{2.0, 3.0\}$ for Strand A and $\{3.0, 4.0\}$ for Strand B as their positions.

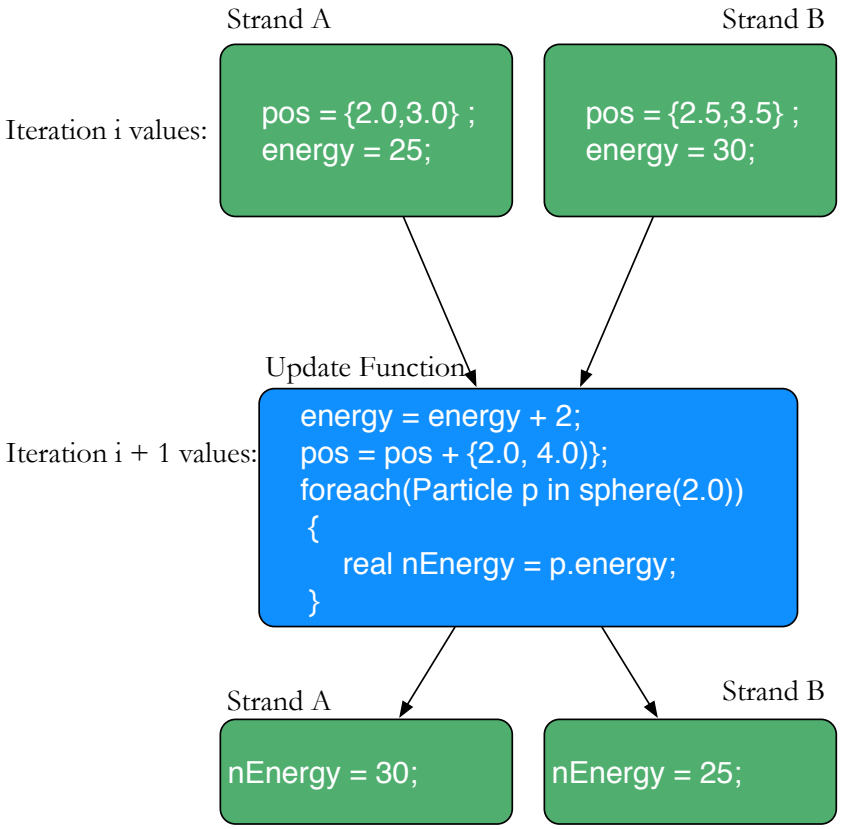


Figure 3-14: Illustrates the use of a strands state variables from the previous iteration and not the current one.

Chapter 4

Dynamic Strand Creation

The current design of Diderot limits strand creation to the initialization phase. 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. Algorithms, such as particles systems, require the ability to vary the strand population. In particular, particles tend to explore the space around them; therefore, a new feature is needed that will dynamically creates strands instantly. For example, one of the motivating factors for strand communication is the work being done with using particles to serves as a representation of implicit surfaces [12]. Since the energy of a particle comes from the interactions of its neighbors, a particle with low energy has too few neighbors, whereas a particle with high energy has too many neighbors. The fluctuation of the system energy owed to these interactions can potentially cause an effect on whether the system has enough energy to evenly distribute the particles on the surface. One way to balance the system energy is to insert or delete particles, which will increase or decrease the total system energy. This approach requires the ability to dynamically create strands when needed.

4.1 Design

New strands can be allocated during a super-step by using a new mechanism called the **new** statement. Figure 4-1 shows the semantics for the new statement. We retrieve the strand definition from (Δ of Σ) in order to retrieve a fresh copy for initialization. The declarations (*i.e.*, the state variables) is evaluated and produces a new local state. The domain of this state is the variables defined within the static state environment of the global state, which holds the state variables defined within a strand definition. We update the global state by assigning the new strand's identifier to its strand

state within the strand identifier environment η.

$$\begin{array}{rcll} e & ::= & \ldots & \text{previously defined types} \\ & | & \textbf{new}\ e & \end{array}$$

$$\frac{\Sigma, \sigma \vdash e \Downarrow v \qquad (\Delta \text{ of } \Sigma) = \textbf{strand}\ (\tau\ x)\{D; m\} \qquad \Sigma, \sigma[x \mapsto v] \vdash D \Downarrow \sigma' \quad \text{I is fresh}}{\Sigma, \sigma \vdash \textbf{new}\ e \Downarrow (\eta \text{ of } \Sigma)[I \mapsto \langle I;\ \textbf{active};\ \sigma' \mid dom(\gamma\ of\ \Sigma);\ m\rangle]}$$

Figure 4-1: Dynamic semantics for new statement

Figure 4-2 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 }
```

Figure 4-2: A snippet of code showing how new strands can be created.

4.2 Implementation

Strands are represented as a memory pool of their state information. Whenever a new statement is performed we retrieve an uninitialized strand state from the memory pool. We initialize the strand state by calling the strand's initialization function. The new strand will begin executing in the next iteration, which is in line with our bulk synchronous execution model. If the memory pool becomes full then we allocate another chuck of memory for additional strand states. We use a memory pool because allocating memory for a single strand state in the pool every time a new strand is added can potentially decrease performance due to the allocation overhead cost.

Chapter 5

Implementation

The addition of strand communication to the Diderot compiler produced minimal changes for the front end and intermediate representations. But we added a significant amount of code for strand communication to the code generation phase. In particular, implementing the spatial scheme for spatial communication and adding a new phase to the bulk synchronous execution model for global communication was required. In this chapter, we discuss these implementations details and choosing a efficient spatial and global scheme.

5.1 Spatial Communication

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. 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. Ericson et al. [6] suggest using schemes that divide world space into two phases. A *broad phase* separates strands that may be near each other into smaller regions called *cells* and excludes those far way. The *narrow phase* then performs the pairwise tests on the strands within each cell. This approach is called *spatial partitioning* and it can significantly increase performance. Now a spherical query can limit itself to cells that only lie within its bounds, thereby reducing the number of pairwise tests. Thus, choosing an effective spatial scheme is important for producing much faster results.

5.1.1 Uniform Grids

We use a uniform grid as our spatial subdivision scheme [8]. The grid effectively overlays world space by dividing it into grid cells of equal size. Each strand is associated with the cell that covers it. A strand's cell can be calculated in constant time because of the uniformity of the grid. The application's domain window, grid dimensions and cell sizes are determined by the programmer. These parameters are set by declaring the global variables `qWinDim`, `qGridDim` and `qCellSize`, which are required when using query functions. The types of these variables depend on the type of the `pos` variable declared in the strand definition. For example, if the position variable is declared to have type **`vec2`** then the `qGridDim` and `qCellSize` will both have type **`vec2`**. The window size represents the x-axis's minimum and maximum values and the y-axis's minimum and maximum values (and z-axis's minimum and maximums for the 3D case). Figure 5-1 shows an example of performing a spherical query on a two-dimensional grid. If Strand H queries for its nearest neighbors with a radius of four units then the query first determines the boundary of its search. Since the cell dimension is four units by fours units and the search radius is four units then the boundary of the search is defined within the dotted red area. The query covers at least the length of the radius. If the radius was eight units then the boundary would be extended by one in each direction. Only cells defined within the bounds of the query are pairwise tested with the querying strand. In this example, the Strands I, J, G and D are pairwise tested with H's position to determine if they qualify as neighbors. If a strand is a neighbor it is added to this sequence of neighbors, which is returned to the querying strand.

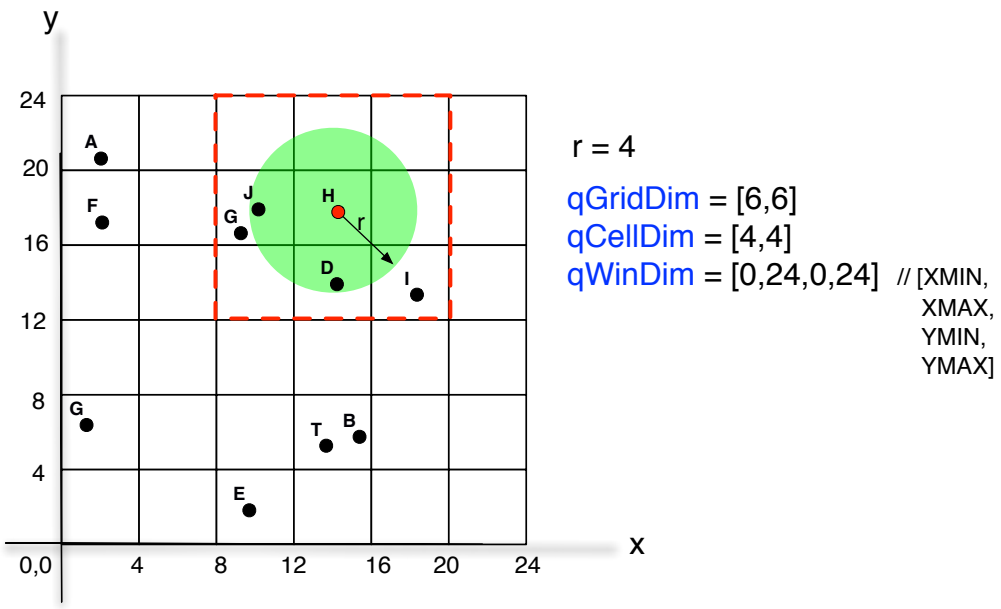


Figure 5-1: An example of performing a spherical query on a 6x6 uniform grid. The width and height of a cell has a length of four. If Strand H performs a spherical query with radius four units then query is defined with the dotted red area. Pairwise tests will only be performed on Strands J, G,D, and I to check if they are valid neighbors.

5.1.2 Clamping Strands to the Grid

Diderot strands are defined within an unbounded world space. In order to implement an unbounded grid we would need an infinite memory space, which is impossible. Our solution to this problem involves clamping wondering strands that are outside the grid dimensions to the boundary cells of the grid. Once we determined our grid dimensions, we extend the dimensions by one in each direction. This creates a border around the main grid for strands that lie outside the actual grid dimensions. Figure 5-2 shows four greyed out strands that lie outside the grid space. We look at their world position and simply clamp them to their corresponding boundary cell within the grid. We chose the clamping mechanism because of its simple implementation and because the clamping is performed in constant time.

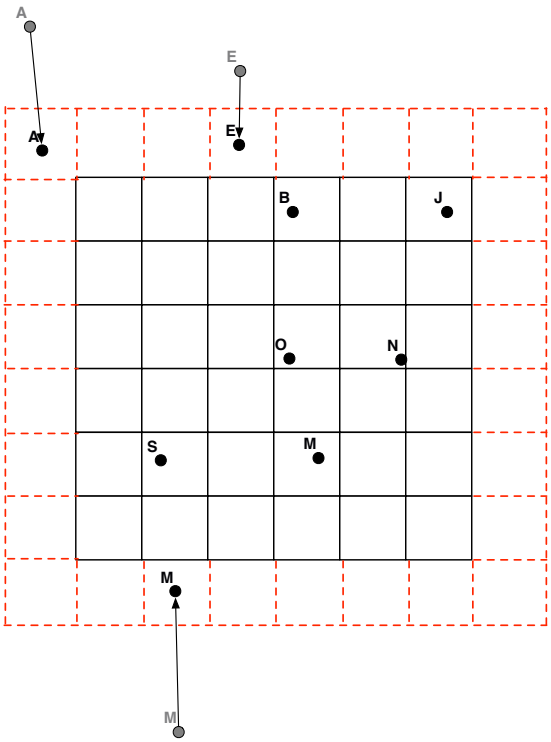


Figure 5-2: Clamping strands to the edge cells of the grid

5.2 Global Communication

Global computations occur after each iteration in a new phase called the *global phase*. During this phase, the actual reduction gets computed. Figure 5-3 depicts the addition of the global phase to our bulk-synchronous model. In this example, each strand has a state variable called energy. After a super-step, the global phase computes the average of energy (*i.e.*, the mean reduction) across all active and stable strands. Only global variables can be assigned to reductions inside the global definition block. The global variable assigned to the reduction can then be accessed within the update function of a strand definition.

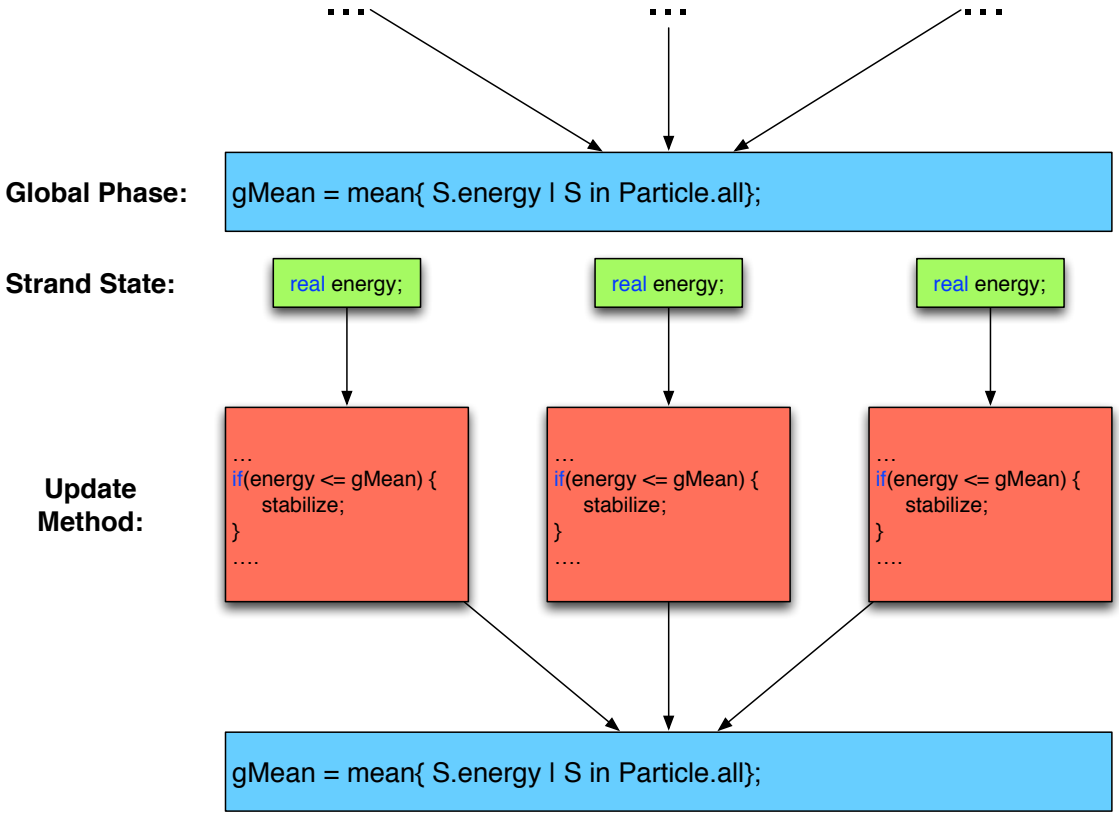


Figure 5-3: Shows how the global phase works within the bulk-synchronous model.

Chapter 6

Performance Evaluation

With the addition of these new communication mechanisms to our programming model, we are interested in how they affect an application's performance. In this chapter, we present results from two benchmarks that use our strand communication system. Our test machine is a Macbook Pro with a 2.67 GHz quad-core Intel Core i7 processor and 8Gb of memory running Mac OS X 10.7.3. For each benchmark, we report the average of 30 runs on a lightly-loaded machine. In this chapter, a description of each benchmark is presented, followed by a summary of the results.

6.1 Benchmark: Boids

Craig Reynolds designed an algorithm to simulate the flocking behavior of various species [17]. 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 shoaling of fish, or 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. Diderot code that implements this algorithm is shown in Figure 6-1. Each boid is given an initial position and velocity (Lines 14-15) and then queries for its neighbors (Line 24). We then compute the vectors that represent the three rules of the algorithm. The separation rule requires boids to avoid colliding with neighboring boids by applying a short range repulsion to maintain an adequate distance (Lines 30-33, Lines 42-44). The alignment rule states that each boid steers towards the average heading of its surrounding boids (Lines 28, 39). Finally, the cohesion rule implies that a boid move towards the average position of neighboring boids (Lines 27, 38). These rules are used to update a boid's position and velocity (Lines 48-52) and illustrate a realistic way a flock would move in a real-world environment.

```
input int width;
input int height;
real queryRadius = 50;
int DESIRED_SEPARTION = 30;
...

real cellSize=queryRadius;
vec4 qWinDim = [0,width,height,0];
vec2 qGridDim = [ floor(width/cellSize) ,floor(height/cellSize)];
vec2 qCellDim = [cellSize,cellSize];

strand Boid(real vx, real vy, real px, real py) {

    vec2 pos = [px,py];
    vec2 vel = [vx,vy];

    update {
        vec2 cohereVec = [0.0,0.0];
        vec2 alignVec = [0.0,0.0];
        vec2 seperateVec = [0.0,0.0];
        int  neighorCount = 0;
        int  seperateCount = 0;

        foreach(Boid neighbor in sphere(queryRadius)){
            real d  =  |pos - neighbor.pos|;

            cohereVec += neighbor.pos;
            alignVec += neighbor.vel;

            if(d < DESIRED_SEPARTION) {
                seperateVec = seperateVec + (normalize(pos-neighbor.pos)/d);
                seperateCount+=1;
            }
            neighorCount+=1;
        }

        if(neighorCount > 0) {
            cohereVec /= real(neighorCount);
            alignVec /= real(neighorCount);
        }

        if(seperateCount > 0) {
            seperateVec /= real(seperateCount);
        }

        ...

        vec2 acceleration = cohereVec + alignVec + seperateVec;

        vel += acceleration;

        pos += vel;
        ...
    }
}
...
```

Figure 6-1: A Diderot program for the Boids algorithm.

6.2 Benchmark: Unit-Circle

The unit-circle benchmark distributes particles evenly around the unit circle. A Diderot implementation is shown in Figure 6-2. This algorithm was inspired by Meyer's work that uses a particle system to distribute the particles on an implicit surface [12]. The idea of the algorithm is to minimize the potential energy associated with particle interactions. Each particle is given an initial random position on the unit circle and has an energy associated with it (Lines 17-18). The energy at position $\mathbf{p}_i$ is due to the proximity of a nearby neighbor, $\mathbf{p}_j$:

$$E_i = \sum_j \phi(|\mathbf{r}_{ij}|) \text{ (Line 32)}$$

where $\mathbf{r}_{ij} = \mathbf{p}_i - \mathbf{p}_j$ (Line 29) and ϕ is an energy function that states the potential around a single particle. The potential of a particle is due to distance between $\mathbf{p}_i$ and $\mathbf{p}_j$, which is $|\mathbf{r}_{ij}|$. The force F_i felt at $\mathbf{p}_i$ is the negative gradient of energy:

$$F_i = -\sum_j \phi'(|\mathbf{r}_{ij}|)\mathbf{d}_{ij} \text{ (Line 33)}$$

where the derivative is with respect to $\mathbf{p}_i$ and $\mathbf{d}_{ij}$ is the unit-length direction from particle j to i (Line 30). A particle's position is then updated based on this repulsive force F_i:

$$\mathbf{p}_i += h * F_i \text{ (Line 38)}$$

where h is the numerical integration step size, which controls how quickly a super-step changes a particle's position. If there are no neighbors around a particle then new particles are created at a desired distance (Line 40). Particles positions are continuously updated until the system reaches convergence that is, when the particles are evenly distributed. Convergence is determined by computing the standard deviation of the particle energies (Lines 44-47). If the standard deviation is below a certain threshold (determined by the programmer) then all particles stabilize (Lines 21-23).

```
input real rr = 0.2;       // actual particle radius
real d = rr * 2;           // desired distance for a new particle.
real nRadius = rr+0.0;     // neighbor query radius
real stdev = ∞;            // used to test for convergence
real threshold = 0.2;      // covnergence threshold
...

vec2 xDom = [-1,1];
vec2 yDom = [-1,1];
real xSamples = floor((xDom[1] - xDom[0])/RR);
real ySamples = floor((yDom[1] - yDom[0])/RR);
vec4 qWinDim = [xDom[0],xDom[1],yDom[0],yDom[1]];
vec2 qGridDim = [xSamples,ySamples];
vec2 qCellDim = [RR,RR];

strand Particle (real posx, real posy) {
    output vec2 pos = normalize([posx,posy]);
    real energy = 0;
    update {

        if(stdev <= threshold)  {
            stabilize;
        }

        energy = 0;
        vec2 force = [0,0];

        foreach (Particle p_j in sphere(RR)) {
            vec2 r_ij = (pos - p_j.pos)/rr;
            vec2 d_ij = normalize(r_ij);
            if (|r_ij| < 1) {
                energy += (1 - |r_ij|)^4;
                force += - (-4*(1 - |r_ij|)^3) * d_ij;
            }
        }
        if (energy > 0.0) {  // we have neighbors
            ...
            pos = normalize(pos +  hh*force);
        }else {
            new Particle(pos[0] + d, pos[1] + d);
        }
    }
}
global{
  real var = variance{P.energy | P in Particle.all};
  stdev = sqrt(var);
}
...
```

Figure 6-2: A Diderot program for evenly distributing particles around a unit circle.

6.3 Benchmark Results

Figure 6-3 and Figure 6-5 show the runtime of the boids and unit-circle benchmarks with various numbers of strands. The factor that significantly affects the performance of both benchmarks is the grid's cell size, as one would expect when using a uniform grid as a spatial scheme. As we increase the cell size above the query radius, the execution time also increases. This result is plausible because as the grid cell size increases there are fewer grid cells and each cell has a larger number of strands. Thus, as the cell size increases the querying execution time approaches the worst-case scenario of the all-pairs test. Similarly, when the cell size decreases below the query radius, the execution time increases, because the query checks a larger number of cells. To support this analysis, Figure 6-4 and Figure 6-6 show how many strands are tested on average per iteration. With smaller cell sizes, fewer strands are tested and performance is affected because of the overhead of searching through a larger number of cells, rather than pairwise testing a large number of strands. When cell sizes are larger, more strands are pairwise tested, resulting in longer execution times. Choosing a grid cell size to be at or around the query radius achieves the best performance in these benchmarks because the query's boundary contains a smaller number of cells to check. Ultimately, the performance of a uniform grid scheme depends on choosing the optimal grid cell. If the optimal grid cell size is unknown then choosing it to be near the query radius achieves the best performance, as indicated from our results.

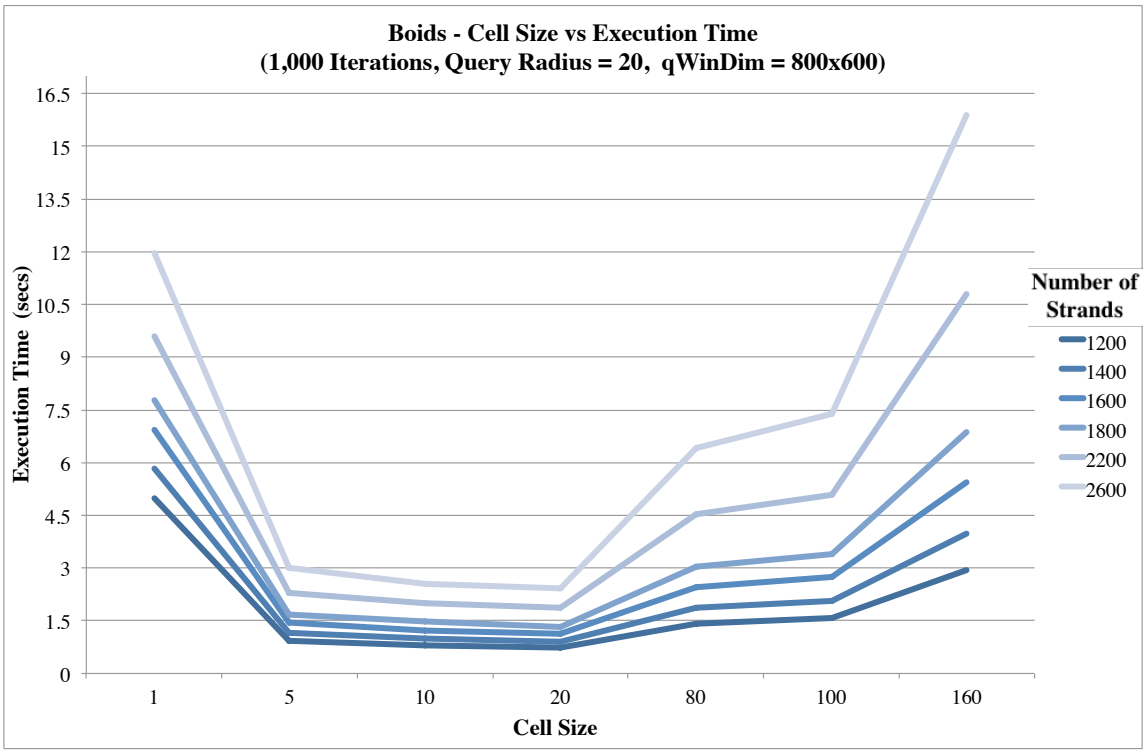


Figure 6-3: Boids benchmark with the cell size variation

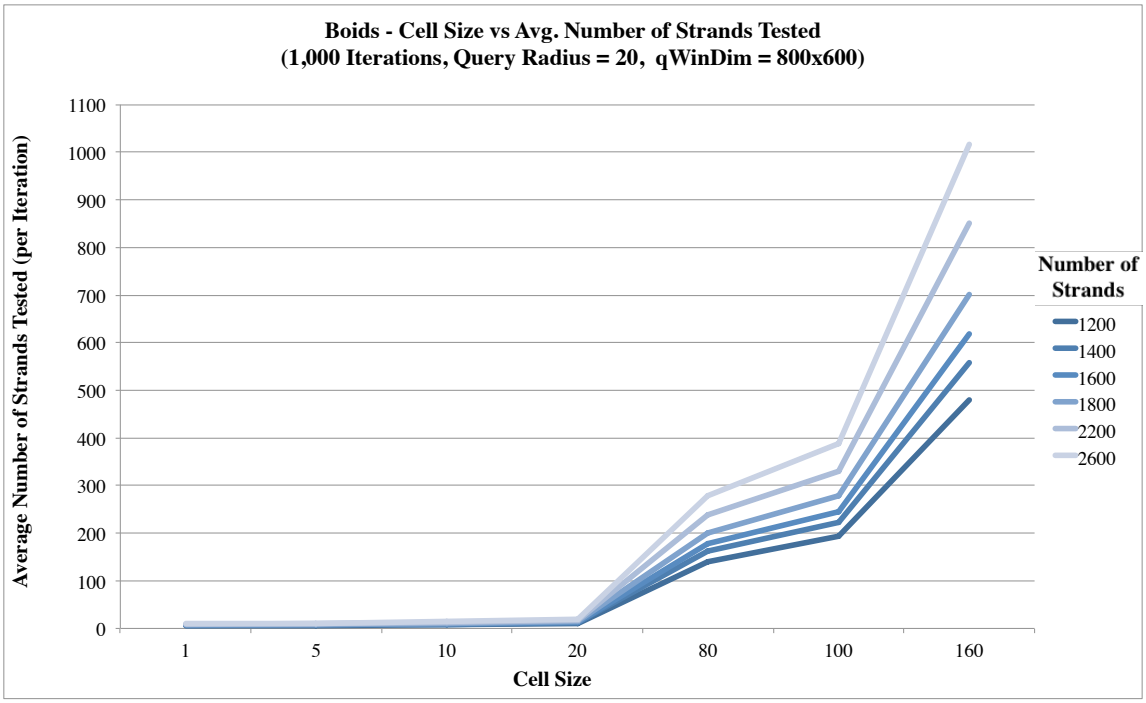


Figure 6-4: Boids benchmark showing the average number of strands tested per iteration

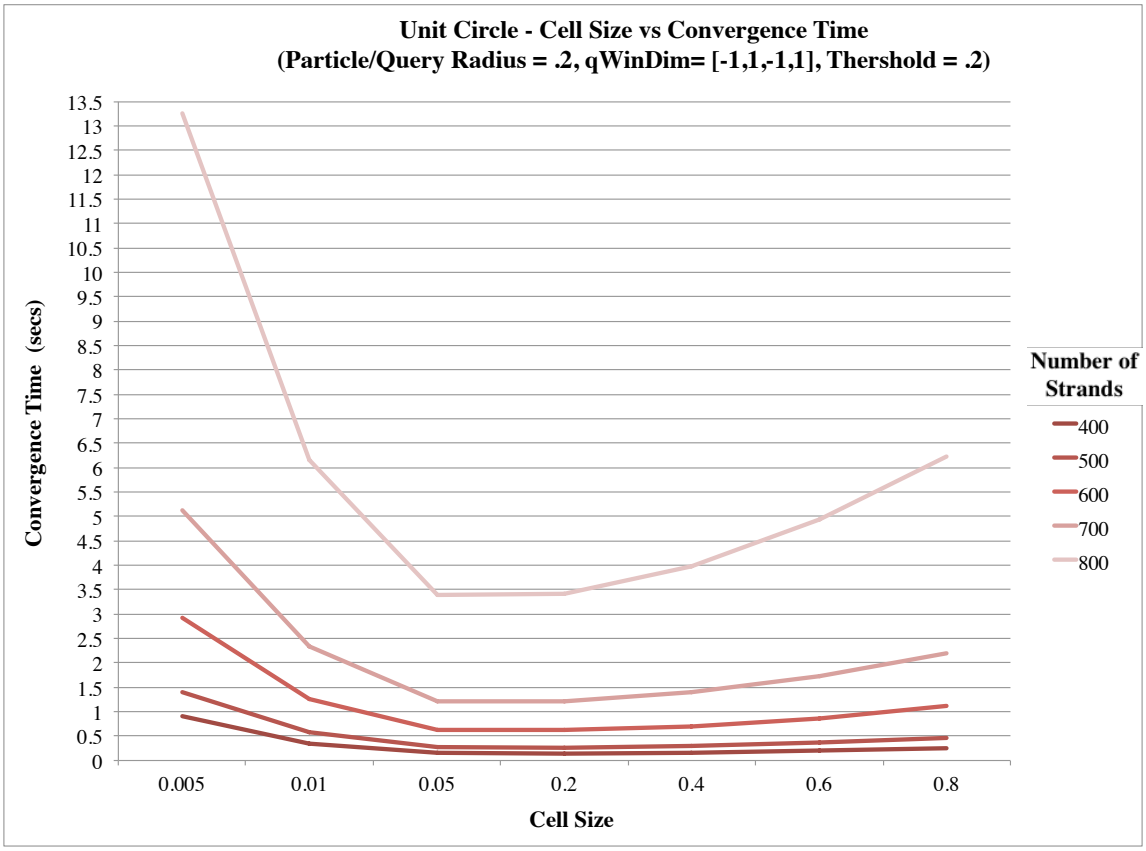


Figure 6-5: Unit Circle benchmark with the cell size variation

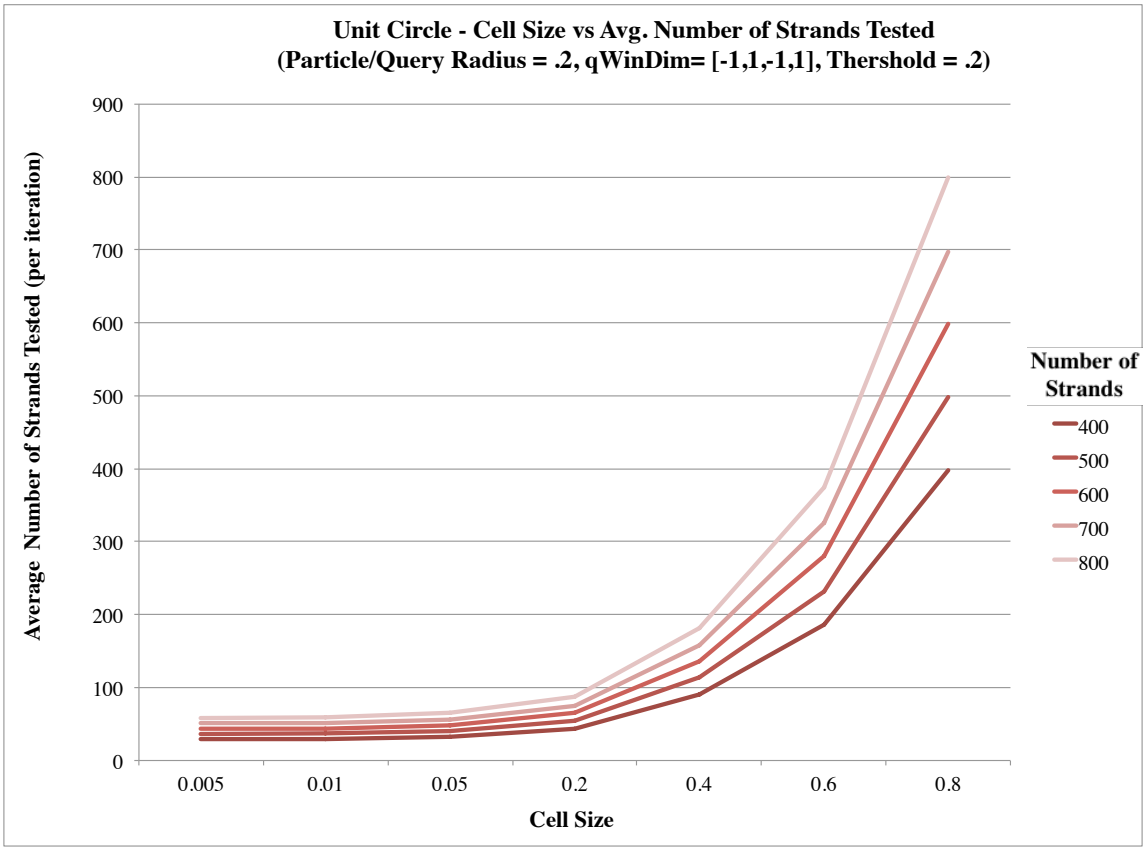
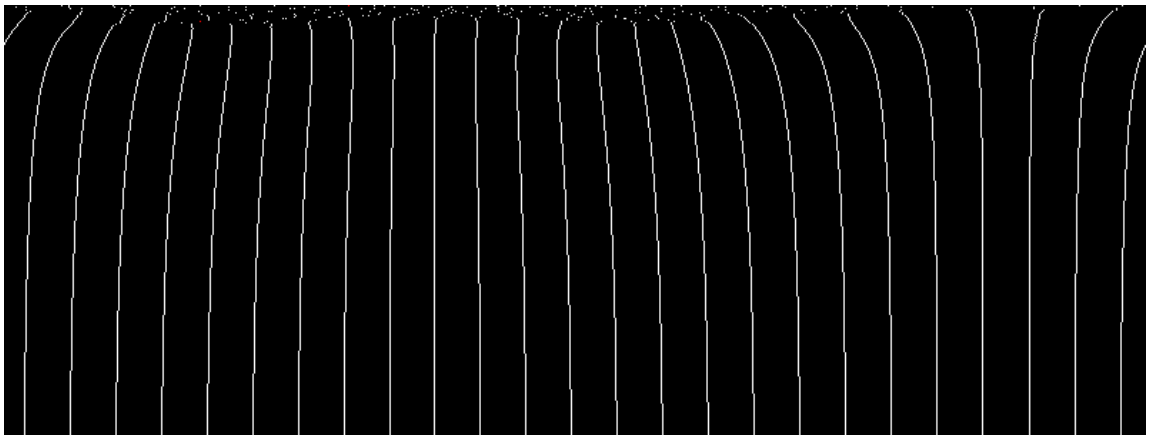


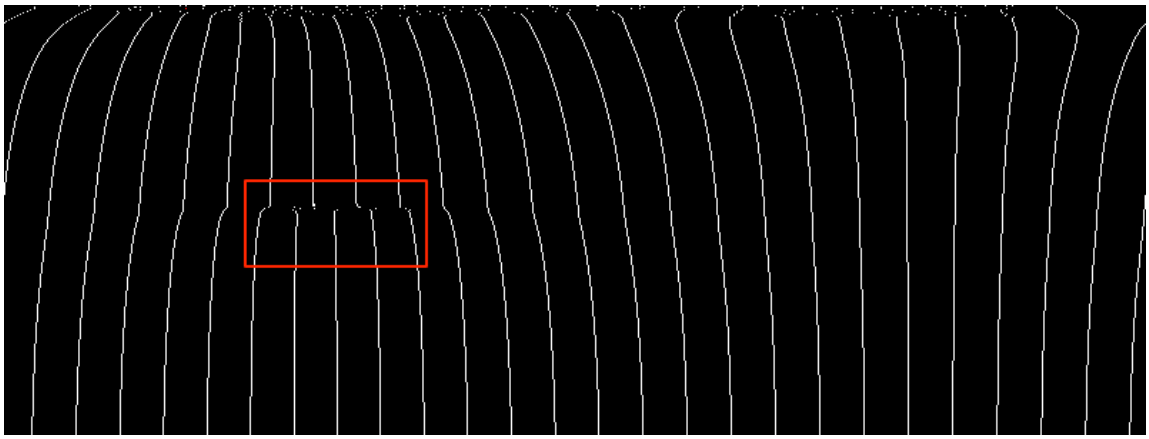
Figure 6-6: Unit Circle benchmark showing the average number of strands tested per iteration

6.4 Strand Allocation

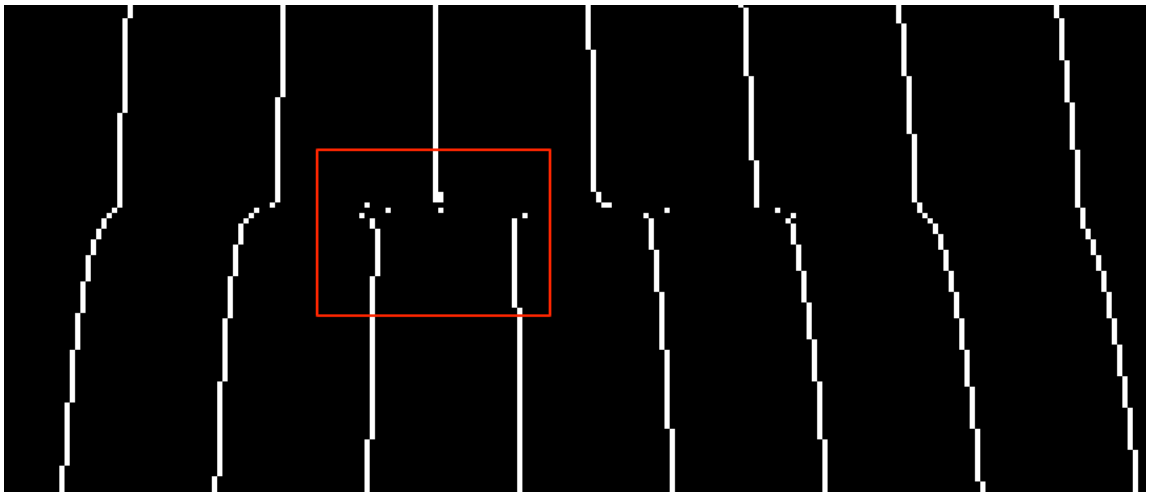
The unit-circle benchmark also demonstrates our strand allocation mechanism. As mentioned earlier, a new strand can be allocated at some desired distance. Figure 6-7 shows images that represent the history of strand angles as they move around the unit circle per iteration. Initially, the strands are significantly repelling each other, which is producing the ”scattering” effect seen in the image. After a number of iterations, the strands start to evenly distribute themselves on the unit-circle, which decreases the amount their angles increase by per iteration. This effect is seen as a ”jittering” line in the image because there are fewer interactions. In this example, a new strand is allocated at iteration 140, which is shown in the last two images of Figure 6-7.



(a) The history of strand angles without creating a new strand.



(b) An example of creating a new strand at iteration 140.



(c) A zoomed-in image of a new neighbor being allocated.

Figure 6-7: The images show the history of strand angles as they move around the unit-circle per iteration.

Chapter 7

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 these mechanisms of communication. 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 [9]. 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).

As described in the previous chapter, the boids simulation is an example of an spatially explicit environment. 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 spatial binning 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-based models are used in various scientific fields, especially in an ecological simulation. For example, Benes, Cordoba, and Soto [1] developed a plant ecological simulation that uses agents that interact with the plants and other agents based on their geometric locations. The ecosystem is represented as a homogenous continuous area where plants are competing between the same species and other species. Plants species die or experience grow due to conditions in the system or

plant resources (e.g. food). The agents are responsible for eliminating weeds, removing incorrectly placed plants, or old plants. Each agent maintains a small memory queue that stores the various tasks an agent needs to accomplish. If an agent experiences an overflow of tasks in its queue, it communicates with the closest agents to synchronize their tasks (*i.e.*, exchange or relieve tasks to each other).

A significant contribution of this work is the idea of allowing agents to share their data with other agents. The exchanging of agent information (*i.e.*, tasks) in the plant simulation is a similar to how we exchange strand state information between strands. Agents within the plant simulation can share tasks amongst each other, which is how strands can retrieve state variables from their neighbors. In the plant simulation, the agents are bounded by a certain number of tasks it can hold within their local memory, whereas in Diderot, strands can maintain large amounts of strand state information.

Agent interactions have also been modeled using interprocess communication systems [13] 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 [4]. MapReduce is a programming model used to process parallelizable problems that use large data sets. The model is typically implemented on clusters in a distributed computing environment. The framework is split into two major steps, which are operated by a master node (*i.e.*, a computer). The first step is called Map. In this step, the master node takes in the input data and divides it into smaller sub-problems. It then distributes the sub-problems to worker nodes. A worker node processes a sub-problem and passes the result back to the master node. The last step is called Reduce. In the Reduce step, the master node retrieves all the results from the worker nodes and combines them in a particular way (defined by the programmer) to produce an output. Many data parallel languages have supported parallel map-reduce, such as the NESL language [2], which has also influenced our design of global reductions.

In Diderot, we also spilt a reduction calculation into two parts similar to MapReduce. Our global block allows reductions to be mapped to global variables. In essence, this global variable represents the key for the reduction calculation. The actual reduction calculation (*i.e.*, the value) is calculated in the global phase. 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 MapReduce model has inspired new languages that provide simple mechanisms to use and build its underlying framework. Researchers at Yahoo developed a platform called Pig [15], which produces sequences of MapReduce programs using predefined implementations from the Hadoop project [21]. The language of the Pig infrastructure is called Pig latin. The language abstracts away the Java MapReduce idiom and provides a notation that is easier to use while working with a large data sets. Another example is the language Sawzall developed by researchers at Google Inc. [16]. Sawzall is a DSL used to process large numbers of individual log records. Writing a MapReduce program that analyzes these records can be a time-consuming task but with the simple notation of Sawzall the program becomes much easier to implement. Sawzall programs are only executed in the Map phase of the MapReduce model. This phase process one record at a time to produce intermediate values (*i.e.*integers, strings, tuples, etc.) that are combined with values from other records. The reduce takes in these intermediate results and are further processed by aggregators, which collect and reduce the intermediate values to create the final results. The reduce phase is handled automatically without the need for the programmer's input. The programmer only needs to worry about the script being processed by the Map phase.

Both these languages allow researchers to focus solely on their algorithms rather than the underlying implementation details of how their data will be processed. Pig and Sawzall produce the MapReduce programs based on the simple scripts their users provide. This concept is also one of the major goals of Diderot. We want to allow researchers to focus on implementing their algorithms quickly and efficiently without worrying about the implementation details. Similar to the straight forward notation Pig and Sawzall provide to their users, we also support a simple set notation that many researchers are already accustomed to using. They do not need to worry about lifting the reductions into a different phase (*i.e.*, the global phase), this work is all done by the Diderot compiler.

Chapter 8

Future Work

8.1 Hierarchical data structures

When using query functions, the programmer is required to set the cell size for the uniform grid. One of the most significant problems with uniform grids is their performance depends on choosing an appropriate cell size. If the cells are too small then the query will have to search many cells, which will degrade performance. If the cells are too large, then there will be many objects in each cell, which can deteriorate to the all-pairs test. Thus, we can improve on our implementation by providing additional spatial schemes that deal with various cell sizes. In particular, Ericson [6] describes using hierarchal grids or tree based representations, such as octrees and quadtrees, as one way to handle this problem because the cell sizes only change as you divide up the environment space.

8.2 Additional Query Functions

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.

8.3 Strand Communication for our Parallel Targets

Our parallel targets (Pthreads and GPUs) need strand communication implementations. Our Pthread target benefits from using the C language and libraries, which provide common synchronization and allocation mechanisms. However, our GPU implementation of strand communication poses a more difficult challenge. GPUs are not as flexible in terms of allocating memory dynamically, which can only be done on the host side device. This restriction means if we want to create strands dynamically we have to transfer to the host side, allocate a memory pool on the GPU device, and then transfer back to the GPU to perform the initialization of the strand state. Performing this process can incur a large overhead cost; therefore, deciding the appropriate time and how much to allocate is important to consider. Furthermore, the GPU has a limited amount of memory available for a program in comparison to the virtual memory environment on the host side device. With having various components of a Diderot program being allocated on the GPU (*i.e.*, strand state information, spatial grid meta 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.

Chapter 9

Conclusion

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. Figure 9-1 shows our updated parallelism model with our new communication mechanisms. We provided a spatial mechanism that allows strands to read state information from nearby strands based on predefined query functions. A global mechanism was added to allow strands to share state information with large set of strands. This process is done by using common reduction operations that are defined within the global block of a program. After a super-step the global block is computed by the global phase within our parallelism model. Finally, we provide a mechanism for dynamically generating new strands, which will begin executing in the next super-step. The evaluation of our communication system showed that application can execute faster because of our builtin uniform grid scheme, which can look at a smaller area for retrieving local neighbors than the worst case scenario of the all-pairs test.

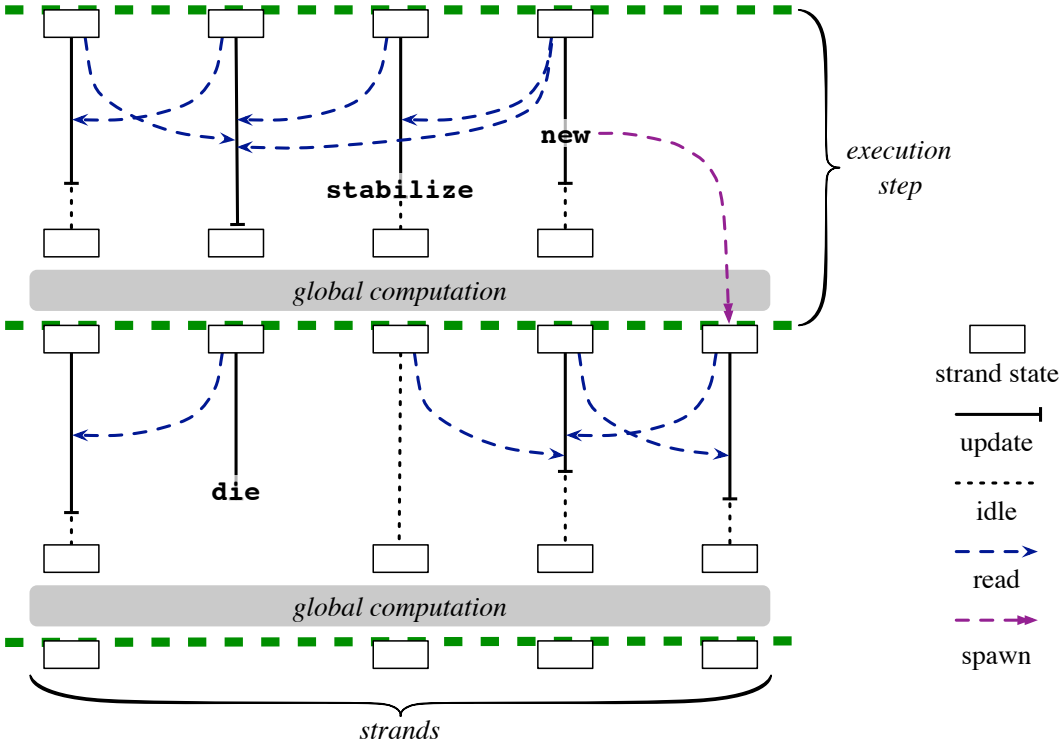


Figure 9-1: Updated bulk synchronous model with new communication mechanisms.

Bibliography

[1] Bedrich Benes, Javier Abdul Cordoba, and Juan Miguel Soto. Interacting agents with memory in virtual ecosystems. In *WSCG*, 2003.

[2] Guy E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[3] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: a parallel DSL for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[5] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 65–74, New York, NY, USA, 1988. ACM.

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

[7] Aaron Filler. The history, development and impact of computed imaging in neurological diagnosis and neurosurgery: CT, MRI, and DTI. *Nature Precedings*, 2009.

[8] Marc Gissler, Markus Ihmsen, and Matthias Teschner. Efficient uniform grids for collision handling in medical simulators. In *GRAPP'11*, pages 79–84, 2011.

[9] Nicholas R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 1429–1436, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[11] Gordon Kindlmann and Carl-Fredrik Westin. Diffusion tensor visualization with glyph packing. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1329–1336, September 2006.

[12] Miriah 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*, pages 124–133. IEEE Computer Society, 2005.

[13] The message passing interface (MPI) standard. `http://www.mcs.anl.gov/research/projects/mpi/`. Accessed: 20/03/2013.

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

[15] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[16] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, October 2005.

[17] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[18] D.B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[19] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[20] David C. Walden. A system for interprocess communication in a resource sharing computer network. *Commun. ACM*, 15(4):221–230, April 1972.

[21] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.