# Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy

by
**Michael J. Chen**

June 6, 1994

ERL Technical Report UCB/ERL M94/16
Electronics Research Laboratory
University of California
Berkeley, CA 94720 USA

# Contents:


1.0 Introduction....................................................3

2.0 Dataflow in Ptolemy, SDF and MDSDF....................................................3

2.1 SDF and Ptolemy Terminology.................................................... 3

2.2 MDSDF Graphical Notation .................................................... 5

3.0 Features and Examples of MDSDF ....................................................8

3.1 Schedule Expressiveness.................................................... 8

3.2 Nested Resetable Loops and Delays .................................................... 10

3.3 Data Parallelism and Multiprocessor Scheduling .................................................... 12

3.4 Natural Syntax for 2-D System Specifications.................................................... 15

4.0 Scheduling and Related Problems ....................................................15

4.1 Calculating Repetitions .................................................... 15

4.1.1 Sample Rate Inconsistency and Deadlock .................................................... 17

4.2 Generating a Schedule.................................................... 18

4.3 Delays.................................................... 21

4.3.1 Alternative Definitions of Two-Dimensional Delays .................................................... 22

4.3.2 The MDSDF Definition of Two-Dimensional Delays .................................................... 23

4.4 Extended Scheduling Example.................................................... 27

5.0 Ptolemy Implementation Details....................................................28

5.1 Two-dimensional data structures - matrices and submatrices .................................................... 28

5.2 Buffering and Flow of Data.................................................... 29

5.3 Scheduling and Schedule Representation .................................................... 31

5.4 Delays and Past/Future Data .................................................... 32

5.5 ANYSIZE Inputs and Outputs .................................................... 33

5.6 Writing MDSDF Stars.................................................... 34

5.7 Efficient forking of multidimensional data .................................................... 37

6.0 Conclusion ....................................................38

7.0 References....................................................38

# 1.0 Introduction

Multidimensional dataflow is the term used by Lee [1] to describe an extension to the standard graphical dataflow model implemented in Ptolemy [2]. The concept involves working with multidimensional streams of data instead of a single stream. Unlike other interpretations of multidimensional dataflow [9,10] which focus more on data dependency and linear indexing issues in textual and functional languages, our focus is primarily on the graphical representation of algorithms, such as those used in multidimensional signal processing and image processing, and exposing data parallelism for multiprocessor scheduling.

This report discusses some of the issues that arose during the development of a multidimensional synchronous dataflow (MDSDF) domain in Ptolemy. The initial goal was to implement support for a two-dimensional extension of the synchronous dataflow (SDF) domain that could simulate MDSDF systems on a single processor system. Therefore, throughout this paper, the terms MDSDF will most often refer to only a two-dimensional implementation, although we hope that many of the ideas can be generalized to higher dimensions. In implementing a simulation environment running on a single processor machine, we made a number of simplifying assumptions, which we will explain in this paper. We will also discuss some of the difficulties we foresee in implementing a full multiprocessor version.

Due to the fact that MDSDF is closely related to single dimension SDF, we will contrast their differences throughout this report. Chapter 2 will explain the graphical representation used for SDF in Ptolemy and the terms we use to describe the components of an SDF system. We will also introduce the graphical notation of MDSDF and explain how the two differ. Chapter 3 will present the features of MDSDF with a series of example systems. Chapter 4 will discuss in more detail the attributes of an MDSDF system and the problems in implementing a simulation domain. Chapter 5 will discuss the low-level implementation issues involved in the creation of the MDSDF simulation domain in Ptolemy, covering design issues such as data representation, buffering, schedule representation, and writing stars for the MDSDF domain. Chapter 6 will conclude with a summary of what has been accomplished and the areas that still need to be worked on.

# 2.0 Dataflow in Ptolemy, SDF and MDSDF

## 2.1 SDF and Ptolemy Terminology

Since Ptolemy [2] is the environment for our implementation, we will introduce its terminology in this chapter. In many ways, MDSDF is simply an extension of the capabilities of SDF [3] so we begin with a discussion of the representation of one-dimensional SDF systems in Ptolemy. Note that the presentation of SDF in this chapter is intended as a summary and not as an in-depth discussion. Much work has been applied to formalize the concepts of SDF, so we strongly suggest that the reader refer to the papers on SDF and Ptolemy in the reference section, especially paper [3], for better understanding.

In SDF and other graphical models of one-dimensional dataflow, the data transferred between functional blocks (or *actors*) is of simple form, i.e. a single value that can be a floating-point number, an integer, a fixed-point number, or a complex number. In Ptolemy, these values are held in a container structure called a *particle,* and these particles are transmitted between Ptolemy actors. Ptolemy also supports structural hierarchy, so that a collection of actors can be grouped and represented as a single actor. At the finest level, actors in Ptolemy are called *stars*, and these are usually implemented by a C function or C++ class. A collection of stars can be grouped together to form a *galaxy.* The overall system, formed by a collection of interconnected stars and galaxies, is called an *universe*. Ptolemy also supplies the ability to transfer more complex data structures, such as vectors and matrices, using a special container structure called a MessageParticle. A simple SDF universe in Ptolemy is pictured below:

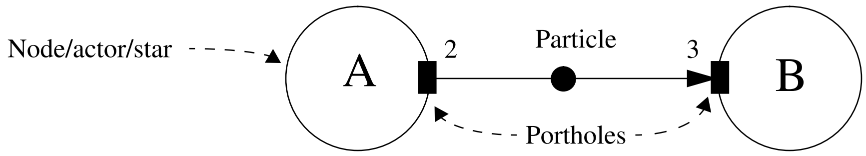



**FIGURE 1. A simple SDF universe**

Actors are connected together by arcs that represent FIFO queues. The arcs are attached to an actor at a location called a *porthole*. An actor can have more than one input or output porthole. The numbers along the arc connecting the two actors specify the number of particles generated or consumed by each star every time it executes (also called a star *firing* in Ptolemy). In the above example, actor A generates two particles at each firing and actor B consumes three particles.

The fact that the number of inputs and outputs for every actor in a SDF system is known at compile time gives the scheduler of the SDF domain (note that SDF is just one model of computation supported by Ptolemy, each of which is called a *domain*) the ability to generate a compile-time schedule for simulation and code generation purposes. This schedule is called a *periodic admissible sequential schedule* (PASS). A PASS is a sequence of actor firings that executes each actor at least once, does not deadlock, and produces no net change in the number of particles on each arc. Thus, a PASS can be repeated any number of times with a finite buffer size, and moreover, the maximum size of the buffer for each arc is a constant that is determined by the exact sequence of actor firings in the schedule. We call each of these repetitions of the PASS an *iteration*.

SDF systems also support the concept of feedback and delays. A delay is depicted by a diamond on an arc, as shown in Figure 2. The delay is specified by an integer whose value is interpreted as a sample offset between the input and the output. It is implemented simply as an ini-

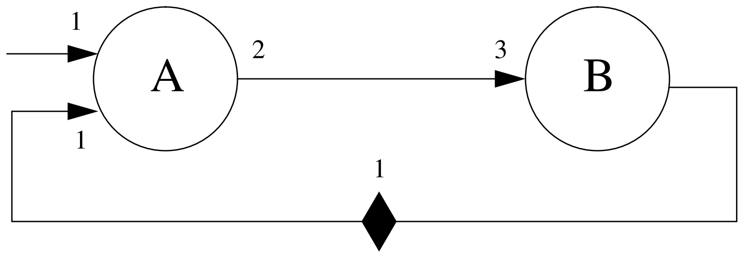



**FIGURE 2. A SDF system with a delay.**

tial particle on the arc between the two actors, so that the first particle consumed by actor B when it fires is the value of the delay (most often this value is zero, but Ptolemy allows the user to give delays initial values). The delay allows the system with feedback to work by giving the source actor A an initial particle to consume on its lower input arc. Note that in [5], delays and the associated problem of accessing past samples in SDF are shown to be problematic in that they often disallow the use of static buffering.

## 2.2 MDSDF Graphical Notation

Although the graphical notation of MDSDF is closely related to SDF and in many ways just a simple extension, the added freedom of the multidimensional system introduces numerous choices of how system specifications can be interpreted. Such flexibility can lead to confusion by both the user of the system and the person implementing it if they do not agree on what the syntax means. Examples of such possible areas of confusion are how to interpret two-dimensional delays and how to define an actor that needs access to data in the "past" or in the "future". This section presents the definitions of MDSDF syntax, but some alternative interpretations will be discussed in Chapter 4.

In MDSDF, the graphical notation is extended by adding an extra dimension to the input/output specifications of each porthole of a star. A MDSDF star in our current two-dimensional implementation has input and output portholes that have two numbers to specify the dimensions of the data they consume or generate, respectively. These specifications are given as a (*row*, *column*) pair, and we use parenthesis to denote this pair. For example, Figure 3 shows a MDSDF star that has one output that generates data with dimensions of two rows by one column.

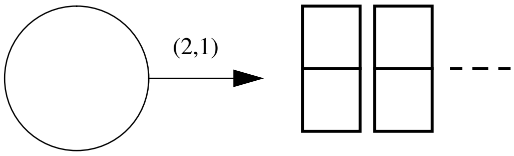



**FIGURE 3. A simple MDSDF star.**

Unlike the SDF case, which can support two-dimensional data objects using the `Matrix` class, the data generated by a MDSDF star is not a self-contained monolithic structure but is considered part of a underlying two-dimensional indexed data space. SDF is able to transmit two-dimensional data objects, such as matrices, using the MatrixParticle construct. However, these data objects are of fixed size, and all actors working on the data stream must be aware of the size of the object (usually by specifying some parameters to the star) and can only manipulate each particle of the stream individually. On the other hand, the input/output specifications of a MDSDF star simply gives us directions on how to arrange the data consumed/produced by the star. For the case of an output data block, once the data has been generated, it no longer has a fixed sized structure, and the system is free to rearrange or combine data generated from multiple firings of the source star into a differently sized data block.

Another way at looking at the specifications of the dimension of the data generated or consumed by a MDSDF star is to consider the specifications as the size of a window into an underlying data space. The origin of the window is determined by the firing index of the star itself. This is best illustrated by an example.

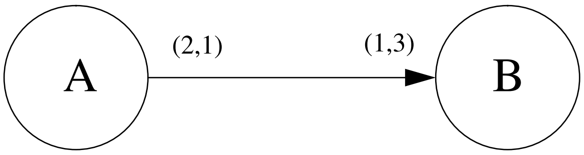



**FIGURE 4. A MDSDF extension of the universe in Figure 1.**

Figure 4 shows a possible MDSDF extension to the SDF system of Figure 1. Actor A still produces two data values, but they are now considered to be arranged as a block that has dimensions of two rows and one column. Similarly, actor B still consumes at each firing three data values, but these three values are required to be structured as a block with dimensions of one row and three columns. The underlying data space for this system would look like:

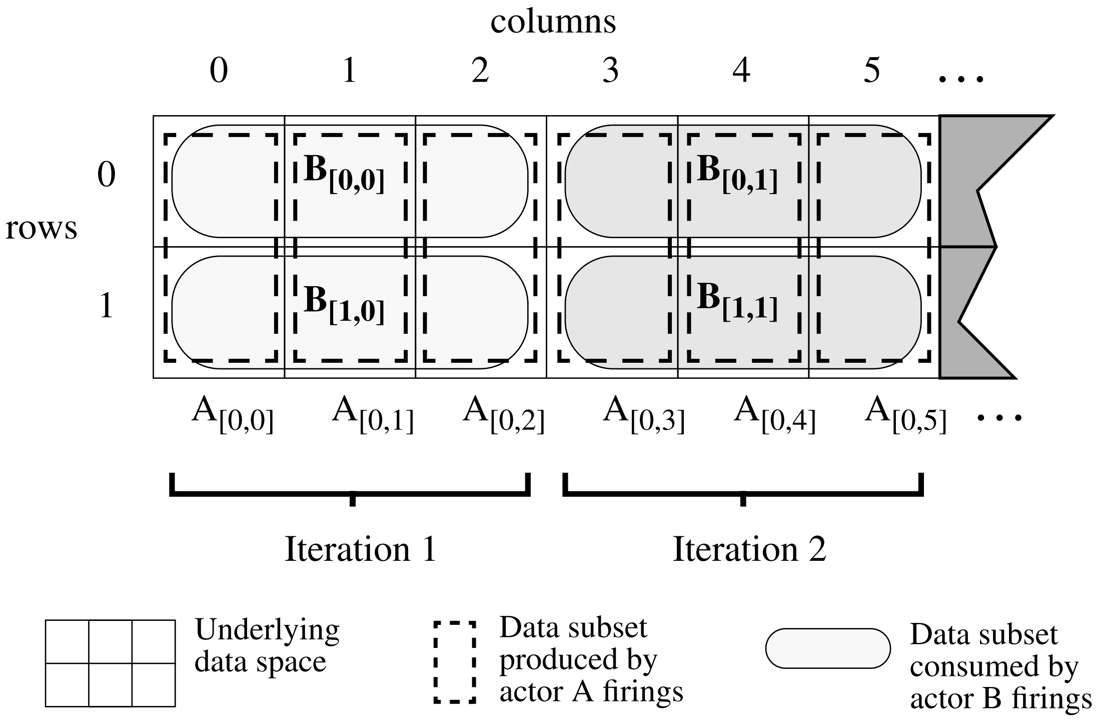



**FIGURE 5. The data space for the system of Figure 4**

Here, the figure shows how the underlying data space has two rows and many columns. First look at the section marked as Iteration 1. This section of the data space is of size two rows by three columns, which is the lowest common multiple of the row and column dimensions of the two actors in Figure 4. The first firing of actor A, which we denote with a firing index using square brackets, is $A_{[0,0]}$ (note the starting index in each dimension is zero), and is mapped to the data space as a two row by one column block at location **d[0,0]** and **d[1,0]**, where **d** represents the underlying data space. We notice that since actor B needs data blocks that have three columns, the only way actor A can fulfill such a demand is by firing two more times along the column dimension. These two firings are denoted $A_{[0,1]}$ and $A_{[0,2]}$, and their associated data space are the two columns next to that of firing $A_{[0,0]}$. Once the three firings of A have produced the data, now considered as a two-row by three-column block, star B can fire twice, with the second firing proceeding along the row dimension. Thus, firings $B_{[0,0]}$ and $B_{[1,0]}$ will consume all the data that the three firings of A produced, and their respective subsets of the data space are portrayed in the diagram as the shaded regions. These five actor firings can be listed as $A_{[0,0]}A_{[0,1]}A_{[0,2]}B_{[0,0]}B_{[1,0]}$, which constitutes an infinitely repeatable schedule for the MDSDF system.

Note that the firing index of an actor is directly associated with a fixed location in the data space, but they are not exactly equivalent. We need to know the size of the blocks produced or consumed by the actor to determine the exact mapping between the firing instance of the actor and its corresponding data space.

Additionally, an important feature about the above firing sequence is the fact that the two sets of firings for actor A and actor B could have clearly been scheduled for parallel execution. In other words, we can see from the data space diagram that the three firings of actor A are independent and can be executed in parallel. Similarly, once all three firings of A are complete and the data they produce are available, the two firings of actor B are also data independent and can be scheduled for parallel execution. We will give more examples of this important aspect of MDSDF in the next chapter.

For a second iteration of the schedule, we can see in Figure 5 that the data space of the second iteration is laid alongside the data space of the first, incremented along the column dimension. This was a design decision, to increment along the column dimension rather than the row dimension. We even considered defining a two-dimensional iteration count, so that we could iterate in both dimensions. We do not know if this latter definition is needed, and all the systems we have implemented thus far have been definable using just the column incrementation definition of a schedule iteration. One issue that is clear is the fact that if there are no delays in the system and there are no actors in the system that require access to “past data” (delays and accessing past data will be described next), then each iteration is self-contained, in the sense that all data produced is consumed in the same iteration. The next iteration of the schedule can reuse the same buffer space as the previous iteration, so the buffer can be of constant size. So although the index of the data increases as the firing indices increase for each iteration, we do not need an ever increasing buffer to represent the data space. This is essentially a two-dimensional extension of static SDF buffering (see [5] for a discussion of static one-dimensional SDF buffering). The index space increases in the column dimension for each iteration, but the actual buffer is from the same memory locations.

The last two basic features of MDSDF that we must explain deal with dependency of an actor on data that is “before” or “after” in the two-dimensional data space. In SDF, the model of interpreting the arcs as FIFO queues implies an ordering of where particles are in time. Therefore, we could discuss how stars could access data in the “past.” In MDSDF, since one of our main goals is to take advantage of multiprocessor scheduling, we do not impose a time ordering along the two dimensions of the data buffer for one iteration (note that there is an ordering between the data of successive iterations). Therefore, for lack of a better term, we use “before” or “past” and “after” or “future” in each dimension to refer to data locations with lower or higher index, respectively, in each dimension. So data location **d[0,0]** is before **d[0,1]** in the column dimension but not the row dimension.

A related concept is the idea of a delay in two dimensions, which can have a number of interpretations. We have chosen to interpret a two-dimensional delay as if they were boundary conditions on the data space. For example, Figure 6 shows a MDSDF system with a two-dimen-

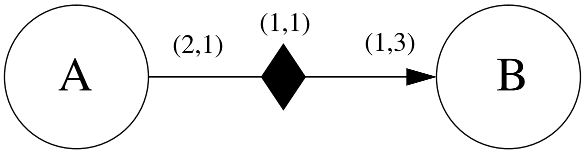



**FIGURE 6. A MDSDF system with a two-dimensional delay.**

sional delay. The delay, just like the portholes of a MDSDF actor, has a (*row*, *column*) specification. The specifications for a two-dimensional delay tell us how many initial rows and columns the input data is offset from the origin **d[0,0]**. We see that in Figure 7, firing $A_{[0,0]}$ is now mapped

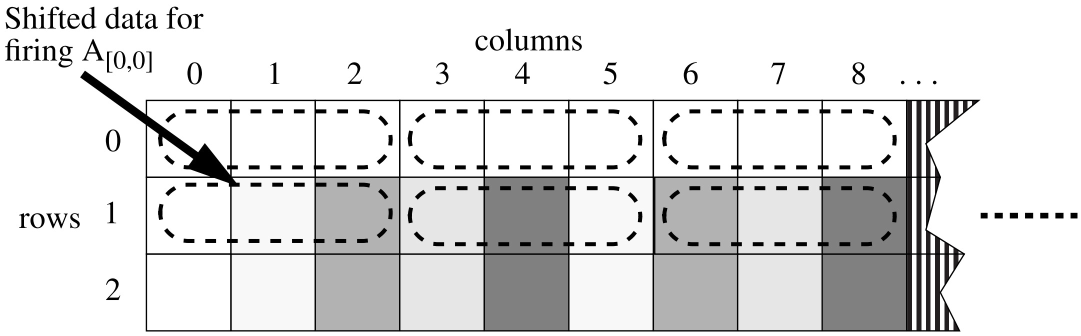



**FIGURE 7. A MDSDF system with a two-dimensional delay.**

to buffer locations **d[1,1]**, **d[1,2]**, **d[2,1]**, **d[2,2]**. We will discuss the effects of two-dimensional delays on scheduling and other complexities that it introduces in Section 4.0. We note that another possible interpretation of the specifications of a two-dimensional delay is simply as one fixed sized data block with the given dimensions, instead of an infinite stream along each dimension. We feel that our interpretation is the proper extension of SDF delays and has some useful advantages over other interpretations, as we shall show in the next chapter.

# 3.0 Features and Examples of MDSDF

Now that we have presented all the building blocks and definitions of a MDSDF system, this chapter will present the various features and possibilities that the increased capabilities provide us. Note that these features and examples are just the ones we have been able to identify in the short time we have worked with the model. We hope that with increased experience, we will discover many additional uses for this model of dataflow.

## 3.1 Schedule Expressiveness

The seemingly simple augmentation of the input/output specifications of MDSDF portholes by just one additional parameter has made these system very much different from their SDF cousins. One advantage that MDSDF has over SDF is the ability to express a greater variety of dataflow schedules in a more graphically compact way. For example, Figure 8 shows a simple


**FIGURE 8. A SDF system for scheduling example.**

multirate SDF system. In terms of scheduling, it can easily be seen that it actor A needs to fire three times for every two firings of actor B in order for the production and consumption rates to balance.

We can formalize this more clearly by looking at the precedence graph and the distribution of data for the above system. These are shown in Figure 9. Since the arc connecting the two actors

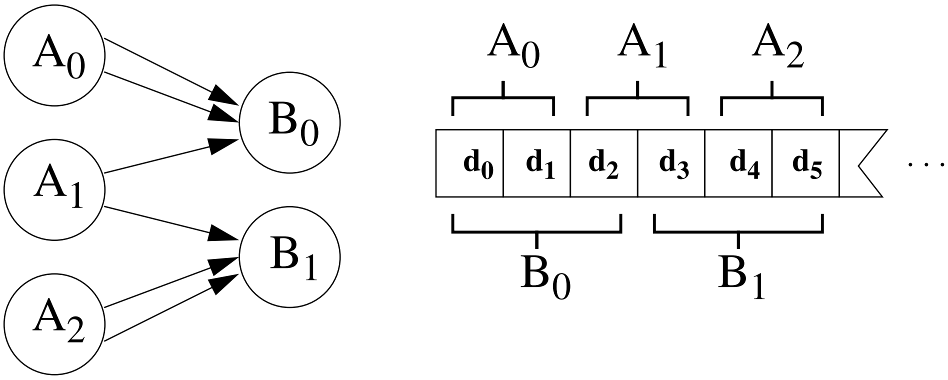



**FIGURE 9. Precedence graph and data distribution for system of Figure 8.**

is considered to be a FIFO queue, the order of the data produced by the various firings of actor A are consumed in order by actor B, as shown in both the precedence graph and the data distribution diagram. The data distribution diagram is similar to the two-dimensional data space buffer diagrams we have shown for MDSDF systems before, but it is only a single dimensional stream. The left most entry, labeled $\mathbf{d_0}$, is the first particle in the stream. Therefore, $\mathbf{d_0}$ and $\mathbf{d_1}$ are the first two particles generated by the first firing of actor A.

Figure 10 shows a possible MDSDF extension of the previous system. Again, actor A produces two data values each time it fires and actor B consumes three, but the extra information inherent in the dimensions specified for their portholes results in a much different distribution of data between the two actors.

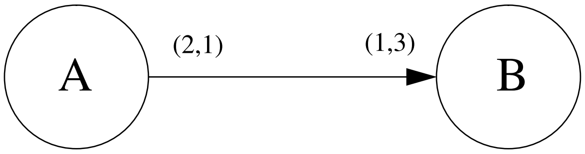



**FIGURE 10. A MDSDF system with a two-dimensional delay.**

Again this is more clearly understood if we take a look at the precedence graph and a diagram of the data space involved, which we show in Figure 11. Here we see that because the data

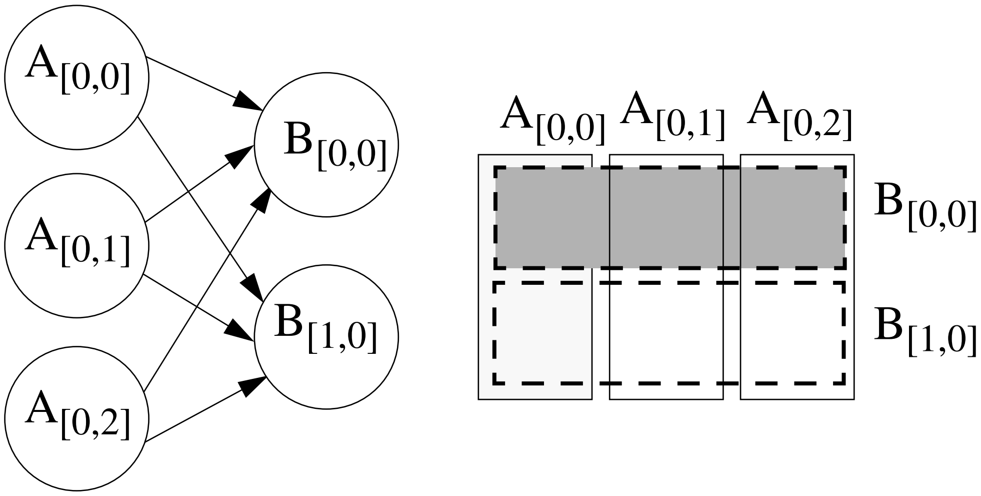



**FIGURE 11. Precedence graph and data distribution for system of Figure 10.**

produced by actor A is arranged as a column of the data space, the two output values of each firing of actor A is distributed to each firing of actor B. So even though the actors in the SDF and MDSDF systems both produce and consume the same number of data values, and the schedules for the two systems are similar in that actor A fires three times and actor B fires twice in both schedules, the data distribution of the two systems is quite different. Note that the MDSDF model is more general since it can express the dataflow of the SDF system by varying one of the dimensions and keeping the other dimension fixed at one. We can also express the precedence graph of Figure 11 in SDF, but we would have to lay out the system exactly as shown, using five nodes and connecting them up exactly as we showed in Figure 11, which makes it clear that MDSDF is a more expressive model of dataflow and can express a larger set of systems more compactly than SDF.

## 3.2 Nested Resetable Loops and Delays

Besides having greater expressive power than SDF, MDSDF can also support some functionality that SDF cannot. One such functionality is the ability to represent nested resetable loops using reinitializable delays. This type of functionality is needed when you try to implement a system like a vector inner product. In SDF, an attempt at expressing such a system might look like the graph in Figure 12. Actors A and B generate four particles per firing, which we can consider

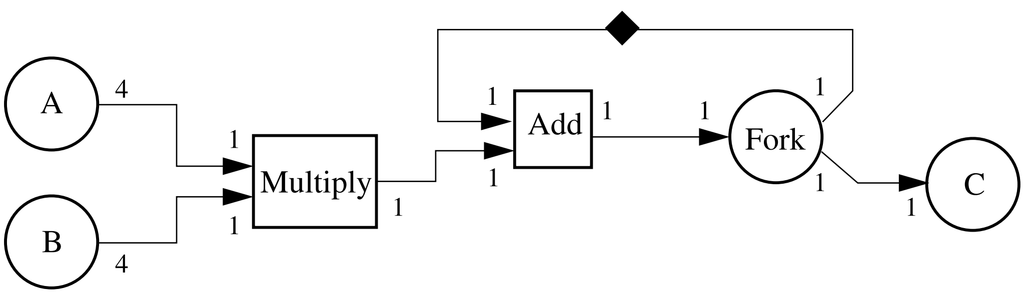



**FIGURE 12. A SDF system to do vector inner product.**

to be vectors with four entries. Each respective entry of the two vectors is multiplied together and the sum is accumulated using a recursive structure of an Add star with delayed feedback. C++ code equivalent to the system above is shown in Figure 13. A problem arises when one would like

```
C = 0;
for (counter = 0; counter < iterationCount; counter++) {
        for (i = 0; i < 4; i++) {
                C += A[i] * B[i];
        }
}
```

**FIGURE 13. C++ code for vector inner product SDF system.**

to make this into a module such that each time the system is run, one would like to have it do the inner product of two four-entry vectors. The problem is that because of the stream orientation of the system, there is no way to reset the accumulator output C. A second iteration of the system would have C to accumulate the sum of the inner product of the first pair of vectors with the inner product of the second pair of vectors.

One possible way to make the system do what we desire is if we could somehow reset the delay at every iteration. A delay is usually considered to be an initial particle on the arc and we set its value to be zero. This is how the first iteration computes the inner product correctly because it essentially sets the initial value of C to be zero. If we could have the delay insert another initial particle at every iteration, this would achieve the functionality we desire. To do this in SDF, we often had to resort to various tricks to hardwire a reset to actors or delays in order to implement this controlled reset of nested loops.

MDSDF can implement such functionality by using the fact that successive iterations are along a new column in the data space. By using our definition of a delay as an entire row or column of initial values in the data space, we can implement the inner product function as shown in Figure 14. Here, all the input/output specifications of the actors in the SDF version have been

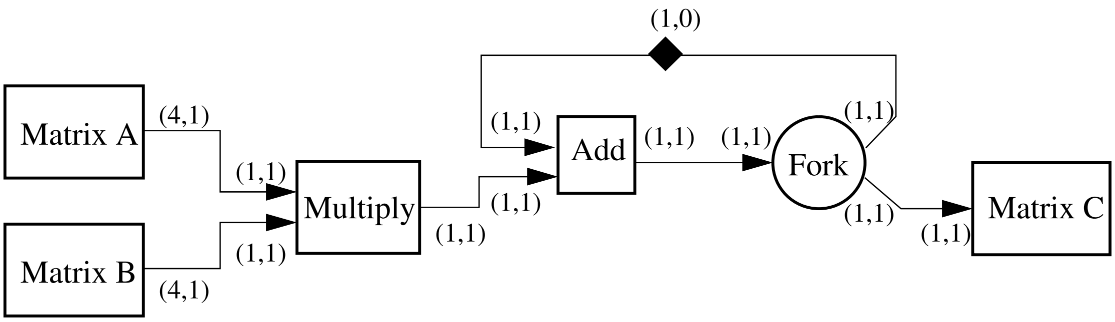



**FIGURE 14. A MDSDF system to do vector inner product.**

augmented to a second dimension. The specification of the second dimension in most of these extensions have been set to one, which implies a trivial use of the second dimension. It is primarily the specification of the two-dimensional delay, and the use of the implicit use of a new column for each successive iteration that makes this system different. The effect of the two-dimensional delay is best illustrated by a diagram of the data space buffer for the arc containing the delay. We show this is Figure 15. The two-dimensional delay in the system was declared to

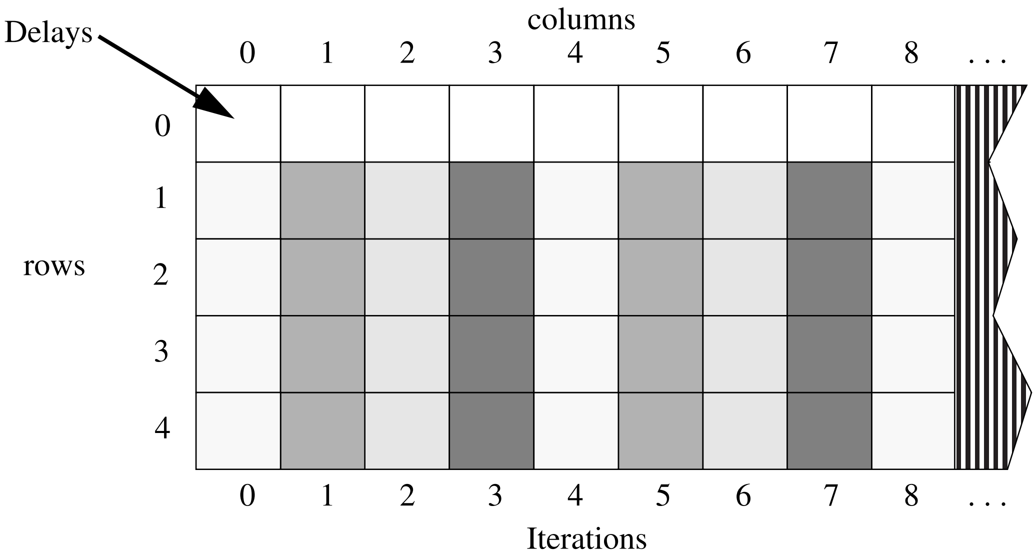



**FIGURE 15. A MDSDF system with a two-dimensional delay.**

have one row and no columns. This implies that the entire first row of the data space is set to the initial value of zero. Thus, at every iteration, the Add actor will have its upper input reset, which is equivalent to resetting the output result C at the beginning of each iteration. This example shows one of the features of our interpretation of two-dimensional delay specifications as infinite along a row or column.

## 3.3 Data Parallelism and Multiprocessor Scheduling

One of the original motivations for the development of MDSDF was the possibilities we saw inherent in the model for revealing data parallelism in algorithms. Although the implementation of MDSDF in Ptolemy has only progressed to the stage of supporting simulations under a single processor, we hope to soon add support for multiprocessor scheduling using the extra information provided by the MDSDF model.

In the last chapter, we introduced how MDSDF can reveal data parallelism in a system. We now present a couple of more interesting examples from field of two-dimensional signal processing. The first, shown in Figure 16, is a simple system that computes the two-dimensional Fast

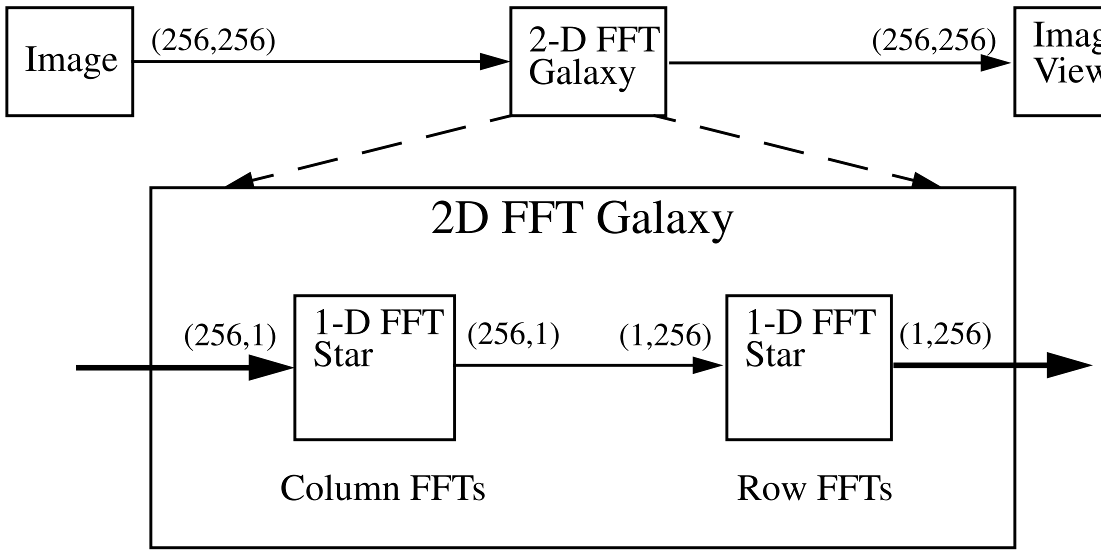



**FIGURE 16. A two-dimensional FFT system using row-column decomposition.**

Fourier Transform (FFT) of an image. One easy way to compute a two-dimensional FFT is by row-column decomposition, where we apply a 1-D FFT to all the columns of the image and then to all the rows [7][8]. This simple concept is straightforwardly expressed in MDSDF as we see in the figure. The diagram shows how we can use the graphical hierarchy of Ptolemy to implement the 2-D FFT as a module made of the two 1-D FFT components. The 1-D FFT stars of the 2-D FFT galaxy are identical, except that we have specified the inputs and outputs to work along the columns and rows of the image, respectively.

We could describe something similar in SDF, but we would be limited to either working with the entire image (as in Figure 17) or adding a series of matrix-vector conversions and trans-

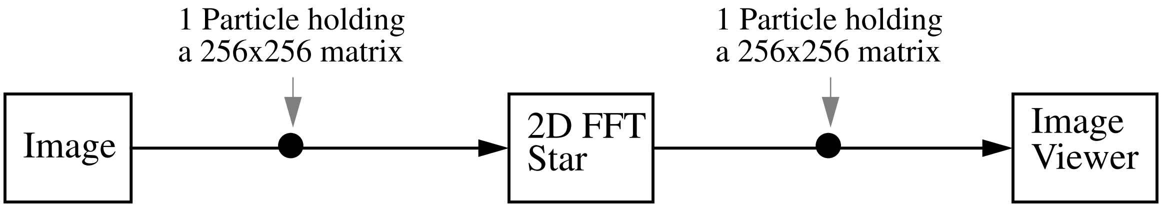



**FIGURE 17. A SDF implementation of 2D FFT as one star.**

positions to manipulate the 1-D vectors to the correct orientation (as shown in Figure 18). The first alternative is not very attractive because we would not be able to take advantage of the data parallelism in the algorithm for multiprocessor scheduling, especially the data parallelism that the MDSDF system reveals. The second alternative is also unattractive because it is quite cumbersome and awkward to have all the data manipulation stars that do not really contribute to understanding the algorithm. The two-dimensional image, considered in SDF as a single monolithic matrix, needs to be converted to a series of vectors so that we can apply the 1-D FFT star on the rows. Then, the vectors must be collected again into a large data block and then transposed and converted to vectors so that we can apply the 1-D FFT star on the columns. Finally, the vectors must be collected again, and then transposed again to undo the previous transposition. The MDSDF representation is much clearer and reveals both the data parallelism and automatically handles the computations along either dimension.

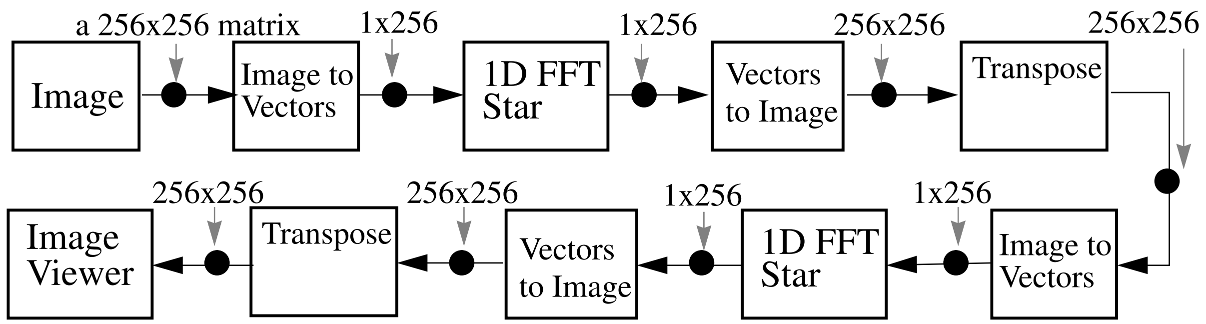



**FIGURE 18. A SDF implementation of 2D FFT revealing the data parallelism awkwardly.**

Once a multiprocessor scheduler is developed to take advantage of the data parallelism revealed by the MDSDF representation, we see that there is also the potential to prototype the system targeted to different numbers of multiprocessors. This is essentially the ability to scale the amount of parallelism that the system designer wishes to exploit in the final implementation. The MDSDF simulation should be able to give the designer information about when the communications costs outweigh the benefits of increasing the number of processors in the system.

For example, Figure 19 shows a MDSDF system that implements a two-dimensional FIR

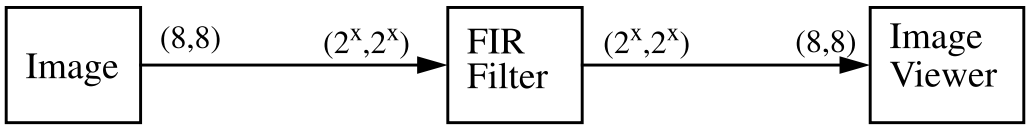



**FIGURE 19. A two-dimensional FIR system.**

filtering system [7][8]. We use a very small image size so that we can show the data space diagram more easily in Figure 20. Here, we show that the designer can have the ability to choose dif-

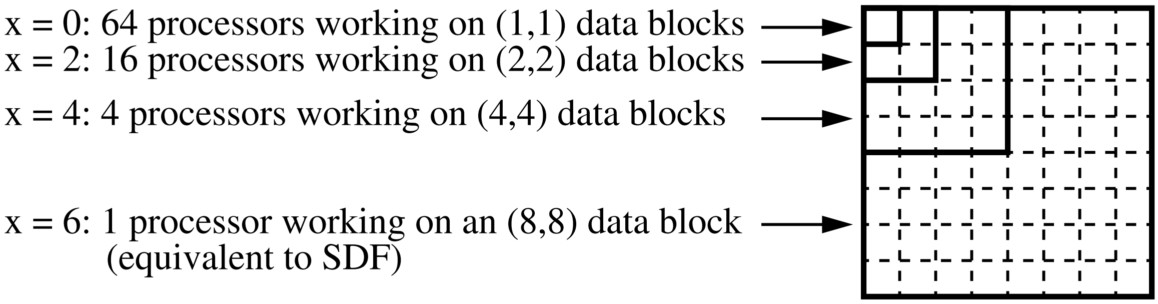



**FIGURE 20. Different subsets of the buffer for a two-dimensional FIR system.**

ferent levels of granularity for the parallelism he wishes to exploit in the system. Although we can specify systems that have actors that access past and future data along the two dimensions, the current implementation is quite limited and such flexible scaling as shown above is not yet possible. One limitation is that a star that desires to access past or future blocks of data can only access blocks that have the same dimension as the current block. In the case of having four processors working on (4,4) blocks of data for the FIR system, those four actors only need one column in the past or future (assuming an FIR filter that is specified by taps that only access one index back or forward in either dimension), but our current specification would only allow those actors to access (4,4) blocks in the past or future. Nevertheless, it should be clear that once we have the ability to do multiprocessor scheduling, MDSDF will allow the user some degree of flexibility to control the amount of parallelism in the system by allowing him/her to tune the ratios of the dimensions of the inputs and outputs of the actors in the system.

### 3.4 Natural Syntax for 2-D System Specifications

The examples we saw in the previous section on two-dimensional FIR filtering and two-dimensional FFT implementation show that the syntax used in MDSDF is a natural one for describing two-dimensional systems. We feel that even without the multiprocessor scheduling attribute, the MDSDF model will be useful for developing two-dimensional systems, such as image processing systems, in Ptolemy.

## 4.0 Scheduling and Related Problems

This section discusses in greater detail some of the theoretical problems we have encountered in defining a workable MDSDF system. We have solutions for many of these problems when dealing with a single processor simulation system for MDSDF, but many of the problems for a true multiprocessor system are still unresolved. We will present the problems we encountered, some potential solutions (when we have identified more than one) and our solution for those problems, and a discussion of the problems remaining to be solved.

Many of the problems in developing a workable MDSDF specification are concerned with the task of scheduling a MDSDF system. Part of the complexity of implementing MDSDF is the fact that so many of the issues are interrelated, and a design decision in one area will have major impact in many others.

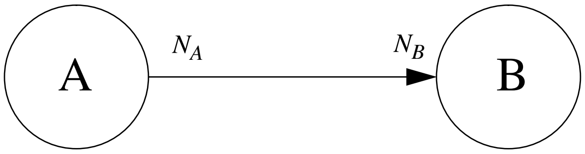
We will present the discussion by scheduling topic, first summarizing how the problem is defined and solved in SDF, and then presenting the MDSDF definition and solution. This discussion will be more formal than what we presented in Section 2.0. The reader is referred to [3],[4],[5] for a more complete presentation of SDF topics.

### 4.1 Calculating Repetitions

The first step in computing a schedule in SDF is to calculate the number of times each actor needs to be repeated during one iteration period. This is accomplished by solving the *bal-*

*ance equations*. The balance equations for a SDF system are a set of equations relating the number of samples consumed and produced by each pair of stars associated with an arc.

In Figure 21, the system has only one arc, so there is only the single balance equation.

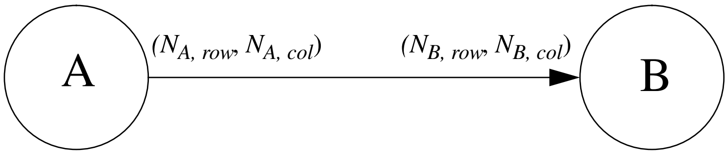



$$r_A N_A = r_B N_B$$

**FIGURE 21. A simple SDF system and its balance equation.**

The unknowns $r_A$ and $r_B$ are the minimum *repetitions* of each actor that are required to maintain balance on each arc. $N_A$ and $N_B$ are the number of output and input particles produced and consumed by actors A and B respectively. The scheduler first calculates the smallest non-zero integer solutions for the unknowns, which we saw to be $r_A = 3$ and $r_B = 2$ for the universe of Figure 8.

The MDSDF extended universe differs because we no longer consider the arcs connecting the actors to be a FIFO queue but rather a two-dimensional data space. We adopt a similar definition of an iteration for the MDSDF case such that at the end of one iteration, the consumption of data should be balanced with the production so that all buffers are returned to the same state as at the beginning of the iteration. In terms of repetitions, this definition involves a simple extension so that there are now two sets of balance equations, one for each dimension:




$$r_{A,\,row} N_{A,\,row} = r_{B,\,row} N_{B,\,row}$$
$$r_{A,\,col} N_{A,\,col} = r_{B,\,col} N_{B,\,col}$$

**FIGURE 22. A simple MDSDF system and its balance equations.**

Each equation can be solved independently to find the row repetitions and column repetitions for each actor. We consider this two-dimensional repetition specification to represent the number of *row firings* and the number of *column firings* for that actor in one iteration. We use the curly brace notation {*row firings, column firings*} to denote the repetitions of a MDSDF actor. The product $rowfirings \times columnfirings$ gives us the total number of repetitions of that actor in one iteration period.

### 4.1.1 Sample Rate Inconsistency and Deadlock

In SDF, it is possible to specify a system such that its balance equations have no integer repetition solutions. This situation is called *sample rate inconsistency* [4]. An example of such a system is shown in Figure 23. Since actor A has a one-to-one production/consumption ratio with

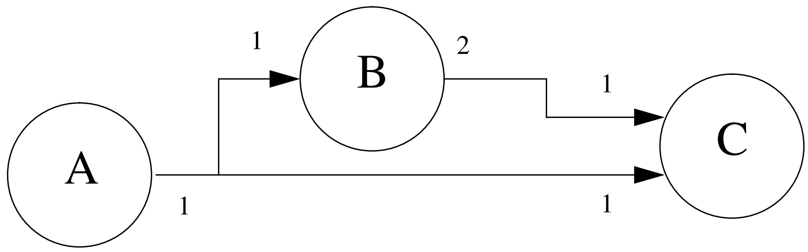



**FIGURE 23. A SDF system with sample rate inconsistency.**

actors B and C, they should have the same number of repetitions in one iteration period. Unfortunately, actor B produces twice as many particles per firing as actor C consumes, which implies that actor C should fire twice as often as actor B in one iteration. Thus, there is an inconsistency in the number of repetitions for each actor in one iteration.

It is also possible to specify MDSDF systems with sample rate inconsistencies. The user needs to be even more careful when specifying MDSDF systems because it is possible for same rate inconsistencies to occur on both dimensions. An example of an MDSDF system with sample rate inconsistencies is shown in Figure 24.

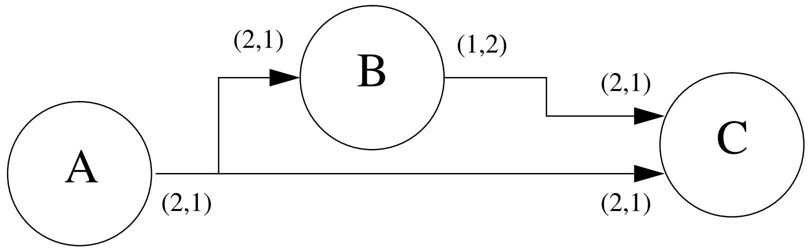



**FIGURE 24. A MDSDF system with sample rate inconsistency.**

A related problem is when a user defines a non-executable system due to insufficient data on an input for the first iteration. This situation, which we term a *deadlock* condition, can occur in systems with feedback, as shown in the SDF system of Figure 25. For the first firing of actor A, it

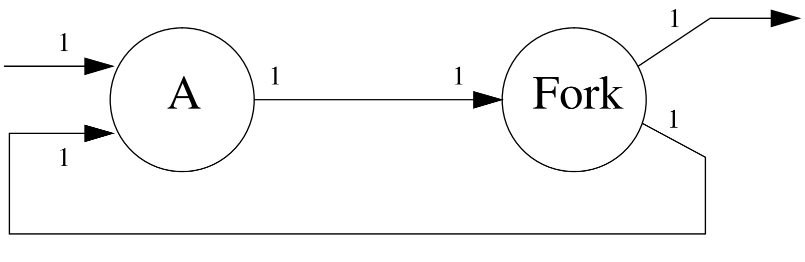



**FIGURE 25. A SDF system with a deadlock condition.**

cannot fire because there is a dependency on its lower arc for data from a non-existent previous firing of A. The solution to this problem would be to add a delay on the lower arc, which would supply an initial particle for the first firing. MDSDF systems can also be specified to have feedback, so they are vulnerable to the same deadlock conditions and MDSDF delays applied similarly to remove these deadlock situations.

## 4.2 Generating a Schedule

The above discussion only gives us the number of times each actor of the universe needs to fire in one iteration. There is still the full scheduling problem of determining when each actor should fire, i.e. we need to generate an actual schedule. For the SDF system in Figure 8, all we know from the repetitions calculation is that actor A fires three times and actor B twice per iteration. There is actually more than one possible schedule for the iteration. One such schedule would be to have actor A fire three times consecutively, and then have actor B fire twice. Another schedule would have actor A fire twice first, producing four data values for the FIFO queue. Actor B would then fire once to consume three of those data values, leaving one value left in the queue. Then actor A could fire its third time to update the queue storage to three values, and actor B could then fire its last time to empty the queue. In a short hand notation, the first schedule can be written as AAABB and the second schedule can be written as AABAB.

The difference between the two SDF schedules has to do with the fact that the second schedule defers the last firing of actor A when it realizes that actor B was *runnable* after the first two firings of actor A. This “smarter” schedule has the advantage of being able to use a smaller buffer between the two actors. For the example above, the first schedule requires a buffer of size six, while the second schedule requires a buffer of size four. There is a cost in using the second schedule that has to do with the fact that the first schedule can be written so that is uses less memory for the code than the second schedule. This is because the first schedule can be expressed as a *loop* schedule 3A2B, which means that the code for actor A is simply placed inside a loop that executes three times and the code for actor B is placed inside a loop that executes twice. If we try to loop the second schedule, the best we can do is A2(AB), which requires us to repeat the code for actor A an extra time (note that in real DSP systems, code for modules are often repeated rather than called as functions since function calls are slower and take stack memory as well). Considerable work has been done on how to schedule SDF graphs to minimize the two often opposing criteria of code size and buffer size [5,6].

In an attempt to make a simple scheduler for MDSDF, we have chosen to implement an extension to the first type of schedule, in which we schedule all the firings of an actor that are runnable as soon as possible, rather than deferring any for future scheduling.

The critical problem to solve in generating any schedule is knowing when the destination actor has enough data to fire. This is not too difficult a problem to solve in the SDF case where all buffers are modeled as FIFO queues. A simple scheduler for SDF graphs simply keeps track of the number of particles at the input to an actor. If an actor has no inputs, then it is always runnable and can be added to the schedule. So, source actors are always runnable. Otherwise, the only condition for an SDF actor with inputs to be runnable is that there are enough particles on each of its input buffers to satisfy the number required. Thus, an SDF scheduler can determine when an actor is runnable simply by keeping track of the number of particles on the buffer.

The MDSDF case is much more complex if we allow the most general multiprocessor scheduling. First, let us look at some simplifications that we can make when we are limited to a single processor scheduler. On a single processor machine, since only one firing of an actor can run at any time, we felt it best to have the scheduler follow a deterministic ordering when scheduling an actor that can run multiple times in one iteration. That is, if an actor can be fired more than once in one iteration period, the scheduler will follow a fixed rule of what order to schedule the various row and column firings. We have adopted a row-by-row approach in scheduling, so that we schedule all firings from the first row of a star before proceeding to the second row of firings. Each row is scheduled in increasing order from lowest to highest. The second rule we use is that we schedule a runnable actor as many times as it needs to be repeated in the iteration immediately and do not attempt to defer any to be scheduled later.

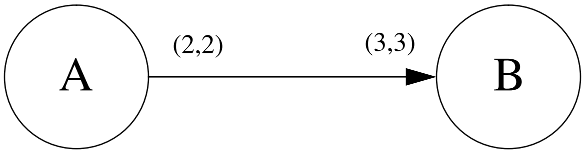



**FIGURE 26. A MDSDF universe for scheduling.**

For example, consider the universe of Figure 26. Using the techniques from the previous section on calculating the row and column repetitions, it is easy to determine that actor A needs to be fired {3,3} times and actor B {2,2} times for one complete iteration. Since actor A can fire a total of nine times, we will schedule it to do so immediately, before the four firings of actor B. Using the row-by-row scheduling rule we mentioned above, we schedule the first three row firings of actor A, starting from firing $A_{[0,0]}$ and incrementing in the column dimension, and then proceed to the next two rows. At completion of scheduling, the schedule that our simple single processor MDSDF scheduler generates is

$$A_{[0,0]}A_{[0,1]}A_{[0,2]}A_{[1,0]}A_{[1,1]}A_{[1,2]}A_{[2,0]}A_{[2,1]}A_{[2,2]}B_{[0,0]}B_{[0,1]}B_{[1,0]}B_{[1,1]}.$$

From the experience of using our MDSDF scheduler on systems with large two-dimensional *rate changes*, it became clear that a shorthand notation for such a schedule is needed because there are often many firings of each actor per iteration (especially for systems like image processing). For the single processor case, when we know that there is a specific order of firings, we can use the shorthand notation $A_{[0,0]\text{-}[2,2]}B_{[0,0]\text{-}[1,1]}$ to represent the above schedule.We still have the problem of determining when the destination actor can fire. In the one-dimensional SDF case, the solution was to simply count the number of particles on the buffer between the actors. In the previous example, actor B was runnable when the buffer had enough particles, and when it fired, it would remove the first $N_B$ particles from the buffer. The seemingly simple extension to working on a two-dimensional data stream actually results in a quite complex problem. We cannot simply talk about “when is star B runnable?” We need to talk about a specific instance of the firing of star B, like “when is the instance of $B_{[0,0]}$ runnable?” This is because of the fact that the buffers between MDSDF actors can no longer be represented as simple FIFO queues and each firing of a MDSDF star has a fixed block of data that it needs to produce or consume, depending on its firing index.

To illustrate this point, let's return to the example of Figure 26. Figure 27 shows a representation of the two-dimensional data buffer between actors A and B for that system. We can see

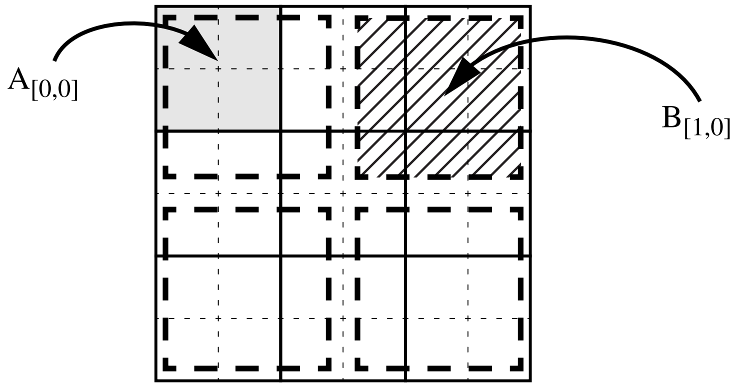



**FIGURE 27. Two-dimensional data buffer for system in Figure 26**

that firing $A_{[0,0]}$ produces data that correspond to buffer locations **d**[0,0], **d**[0,1], **d**[1,0], **d**[1,1], where **d** represents the two-dimensional buffer. Similarly, firing $B_{[1,0]}$ requires that buffer locations **d**[0,3], **d**[0,4], **d**[0,5], **d**[1,3], **d**[1,4], **d**[1,5], **d**[2,3], **d**[2,4], **d**[2,5] all have valid data before it can fire. We can also tell that firing $B_{[1,0]}$ requires firings $A_{[0,1]}$, $A_{[0,2]}$, $A_{[1,1]}$, and $A_{[1,2]}$ to precede it. The problem is how to determine such dependencies quickly, without resorting to a two-dimensional state-space search to verify that the required data buffer entries are available. In a single processor scheduler, given the simplifications we mentioned before based on the fixed row-by-row execution order of firings, the problem is solved by simply keeping a pointer to the location of the last "valid" row and column in the buffer. Any rows above the *last valid row* (lvr) is assumed to have data filled by the source star already, and any column to the left of the *last valid column* (lvc) is similarly assumed to be valid.

For example, after firing $A_{[2,1]}$, lvr = 5 and lvc = 3 (see Figure 28). To check whether firing B[0,0] is runnable, we simply check the location of lvr and lvc. We know that actor B expects (3,3) blocks of data, and since this is the [0,0]th firing, we need lvr >= 2 and lvc >= 2. Similarly, firing B[1,1] would not be runnable in this example since we need lvr >= 5 and lvc >= 5.

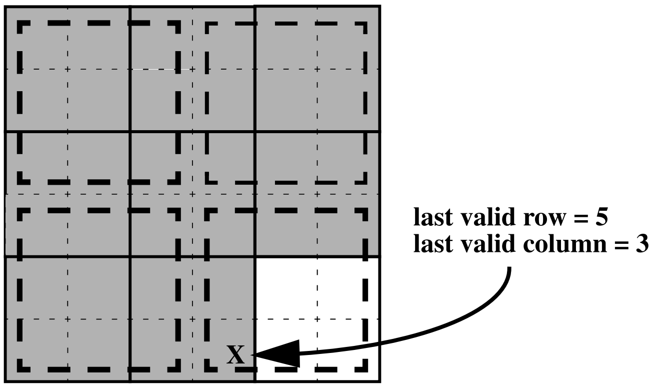



**FIGURE 28. Valid buffer locations after firing $A_{[2,1]}$.**

This method of using a pointer to the last valid row and column is suitable only for the single processor case, but is not flexible enough for multiprocessor scheduling since it is based on the strict firing order assumption. In a multiprocessor system, the various firings of actor A might be executed in parallel, and so firing A[2,2] might complete before firing A[0,0]. We have not yet implemented a multiprocessor scheduler, so we are uncertain whether there is an easier solution to this problem than a full two-dimensional search for all the valid input data values needed for a destination star to be runnable. We hope that there exists a simpler systematic solution because a two-dimensional search can be quite costly and would make extensions to higher dimensions unattractive and possibly unfeasible.

## 4.3 Delays

Delays are a common feature in one-dimensional signal processing system, but their extension to multiple dimensions is not trivial and can cause many problems for both scheduling and buffer management. In one-dimensional SDF, delays on an arc are usually implemented as initial particles in the buffer associated with that arc. The initial particles act as offsets in the data stream between the source and destination actor, as show in Figure 29. Effectively, the output of actor A has been offset by the number of particles set by the delay.

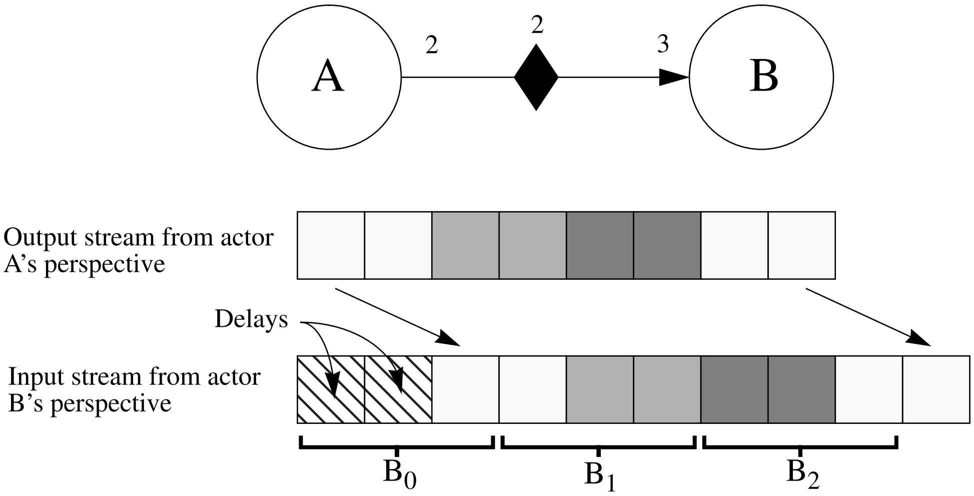



**FIGURE 29. Delays in SDF.**

Unfortunately, the extension to more than one dimension is not so simple. In our attempts at implementing multidimensional delays, we were at first uncertain how to even define them. We see at least two ways to interpret the meaning of a delay on a multidimensional arc, and we have adopted the definition that seems more logical and attractive to us, but we still had to limit its functionality to aid us in implementation. It is not yet clear to us whether our definition is the "correct" one, but more experience in using MDSDF to model real problems should settle the matter. For now, we will present the various alternative definitions and go into more detail about the definition we have adopted. We will explain some of the problems we found in implementing our definition and the restrictions we had to place on it to simplify our implementation.

### 4.3.1 Alternative Definitions of Two-Dimensional Delays

The notation we use for specifying a two-dimensional delay is similar to how we specify the portholes of a MDSDF actor. This is seen in Figure 30, in which we have specified the delay

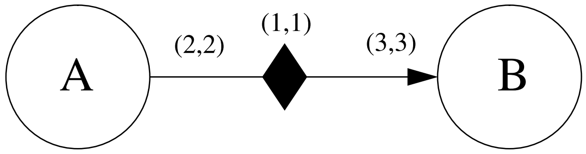



**FIGURE 30. A MDSDF system with a two-dimensional delay.**

to have dimension (1,1). Since MDSDF actors work on an underlying data space, one possible interpretation of the delay is as a finite block with the dimensions given by the delay arguments. This is depicted in Figure 31. The delay block is the first (1,1) block in the space. Notice how it

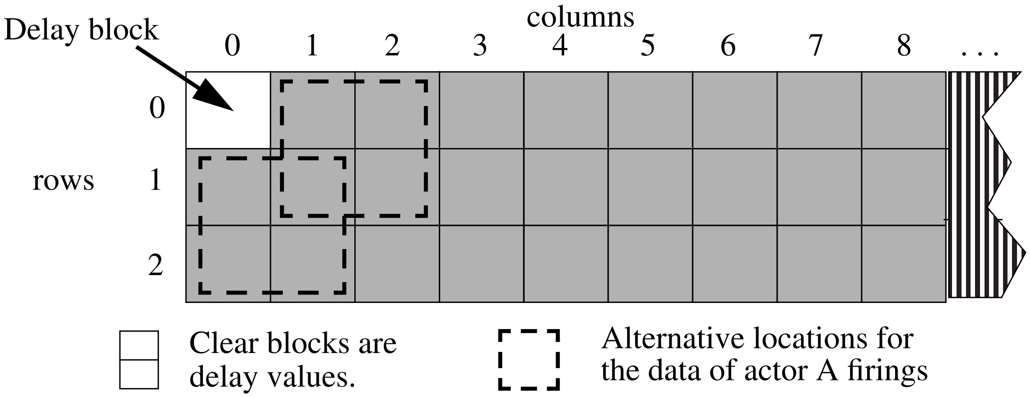



**FIGURE 31. A finite block interpretation of a two-dimensional delay.**

distorts the data space so that it is even unclear how the data from subsequent firings of actor A should be placed in the data space. Although a limited definition (where we limit the dimensions of the delay to be some multiple of the input dimensions) of such finite block delays might be useful in some cases, we do not think this is the "correct" definition of multidimensional delays.

Another possible way to define 2-D delays is to be multiples of the input dimensions. In SDF, delays were a count of how many initial particles, so if we consider MDSDF actors to produce arrays, we might consider delays to be a count of the number of initial arrays. This definition would be similar to the previous one when we limit the delay dimensions to be multiples of the input dimension. For the previous system, the data space would look like the diagram in

Figure 32. Notice how there is one less firing of actor A needed for the first iteration, so this delay

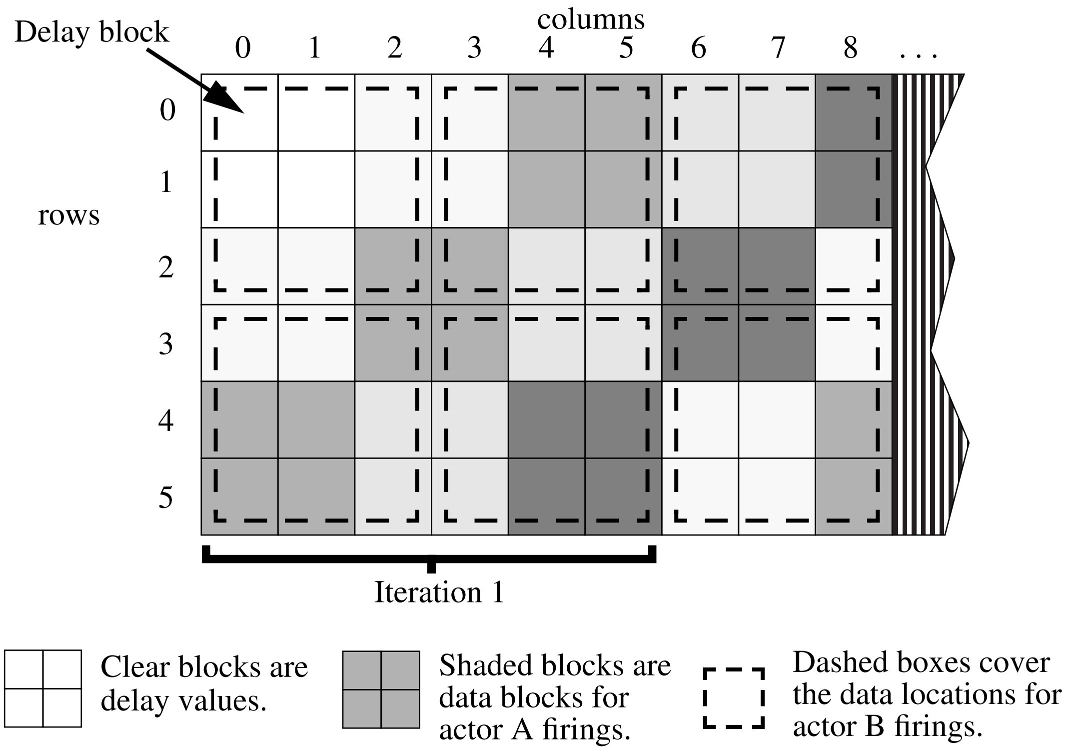



**FIGURE 32. An interpretation of delays as multiples of input blocks.**

interpretation actually changes the schedule generated for the system. Again, this definition may be useful in some cases, but we felt that it was not the "correct" extension of SDF delays since SDF delays do not change the number of times an actor is repeated in each iteration period (although delays might cause some data generated by an actor to be unused and left on the queue).

### 4.3.2 The MDSDF Definition of Two-Dimensional Delays

The last definition we present is the one presented in [1] and is the one we have adopted in our implementation. This interpretation of two-dimensional delays is one in which the delay dimensions cause a two-dimensional offset of the data generated by the source actor relative to the data that is consumed by the destination actor. This is similar to considering the two-dimensional delay specifications as boundary conditions on the data space. The two-dimensional specification of the delay, ($N_{row\ delays}$, $N_{column\ delays}$), is interpreted such that $N_{row\ delays}$ is the number of rows of initial delay values and $N_{column\ delays}$ is the number of columns of initial delays values. Although it is possible in SDF to specify non-zero initial values for delays, in the current implementation of MDSDF, delays are fixed to have zero initial values. We illustrate the data space diagram for this interpretation of the system in Figure 30 below.

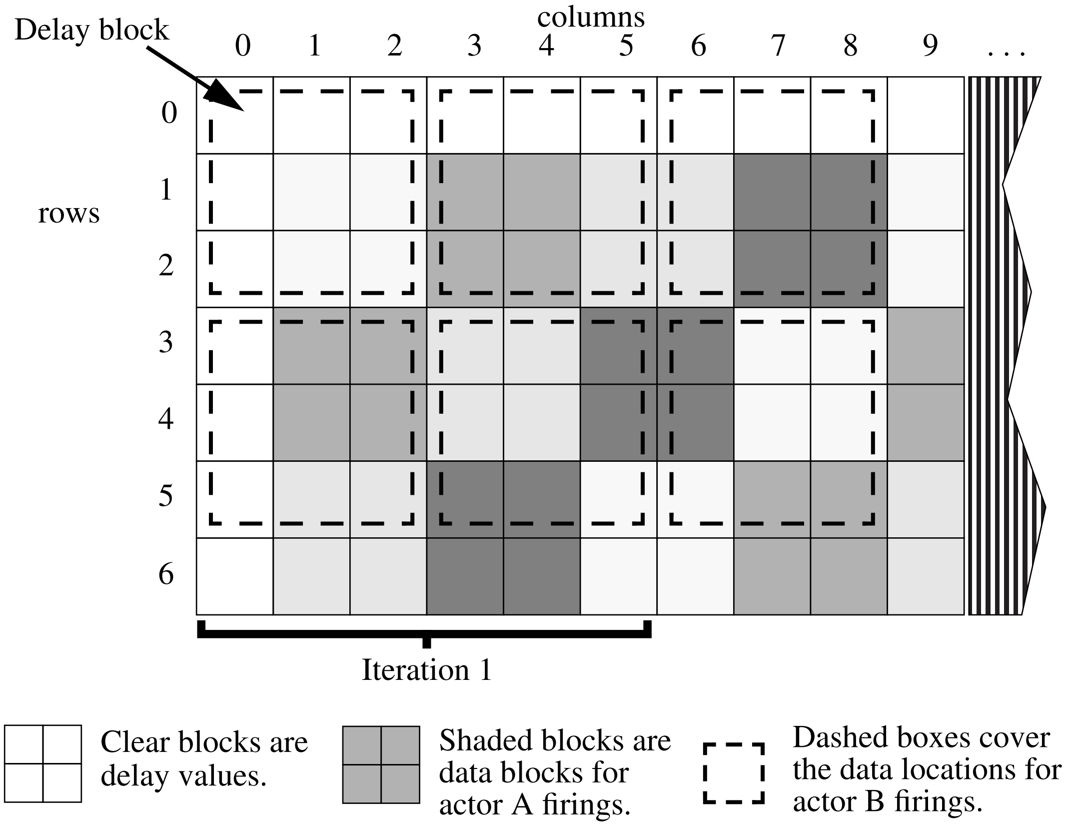



**FIGURE 33. An interpretation of delays as multiples of input blocks.**

We notice that similar to what happens with delays in SDF, there is left-over data on the buffer that will never be consumed, and the buffer size must be large enough to accommodate this extra data. In the row dimension, the delay has caused the last row of data produced by the source actor to be never consumed. Currently, we simply enlarge the buffer by the number of row delays, to give the producer a place to put the data generated. We could discard the data after this, or it might even be possible to discard it immediately when it is created so we do not have to buffer the data, but this would require the submatrix of the producer to be smart enough to know that the data being generated should be discarded. We feel the cost of this modification is not worth the savings at this time. The extra column data that is left unconsumed in the first iteration by column delays cannot be so discarded because subsequent iterations would consume it.

As we just showed, the column delays also increase the number of columns needed in the buffer, but this increase in column size results in much more complex problems than the increase in row size caused by the row delays. The problems have to do with determining how much to increase the column size of the buffer. If we simply increase the number of columns of the buffer by an amount equal to the number of column delays (the method used for the row delays), we encounter a problem that has to do with the implementation of the submatrices used to access subsets of the buffer. For example, if we used a buffer size of seven rows by seven columns for the system of Figure 30, we get the following:

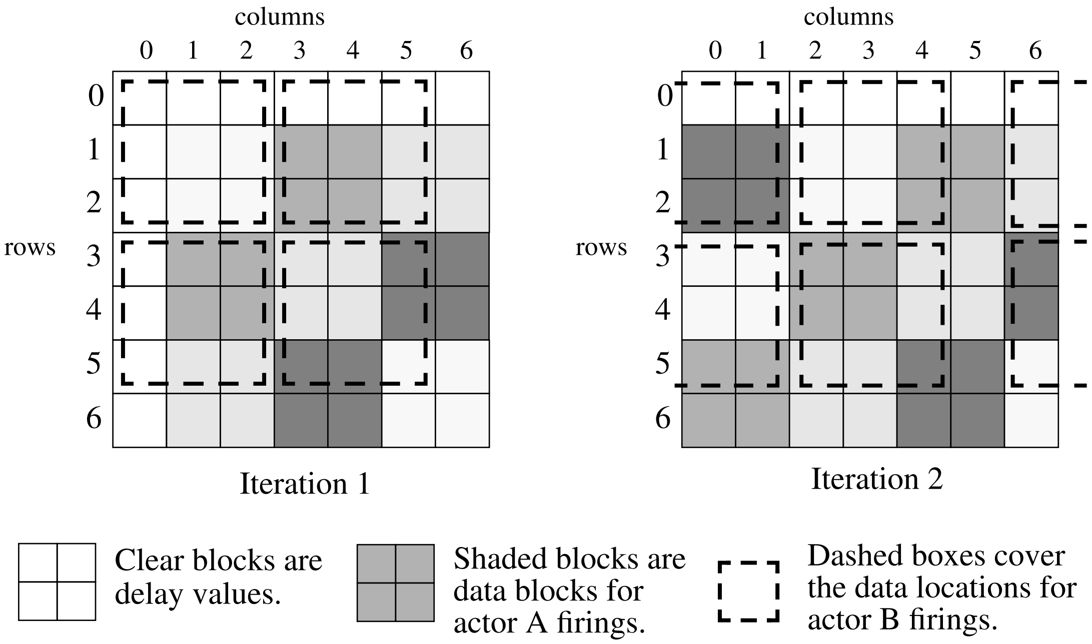



**FIGURE 34. Buffer usage in two iterations of a MDSDF system with delays.**

Notice how in the second iteration, the submatrices for firings $B_{[0,2]}$ and $B_{[1,2]}$ are no longer proper subsets of the buffer space. Similarly, firing $A_{[0,6]}$ will produce data into a submatrix that wraps around the boundary of the buffer space. In order to support such modulo addressing in the submatrices, their design would need to be much more complex, and the methods to access each entry of the submatrices would be much slower. These problems also exist in the first finite block definition we gave previously, but not in the second definition given above where the delay block size was a multiple of the input block size.

In an attempt to simplify the system and especially to keep the implementation of the submatrices as fast and efficient as possible, we chose not to support modulo addressing. We wanted submatrices to always access proper subsets of the buffer space. In order to do this, we had to adopt a constraint such that the number of column delays specified must always be a multiple of the column dimension of the input to the arc with the delay. This causes the column delays to behave like initial firings of the source actor onto the buffer space, and results in the submatrices used by the source actor to always fit as proper subsets of the buffer space. Unfortunately, this constraint is not sufficient to guarantee that the destination actor will use a submatrix that is a proper subset of the buffer space.

An additional constraint was needed, such that the number of columns in the buffer with delays is always a multiple of the number of columns of the original buffer with no delays. This is because there are instances where the source or destination actor works on the entire original buffer space, thus increasing the number of columns in the buffer only by the number of column delays still results in a submatrix being an improper subset of the buffer space. This can be seen in the example system and buffer diagram of Figure 35.

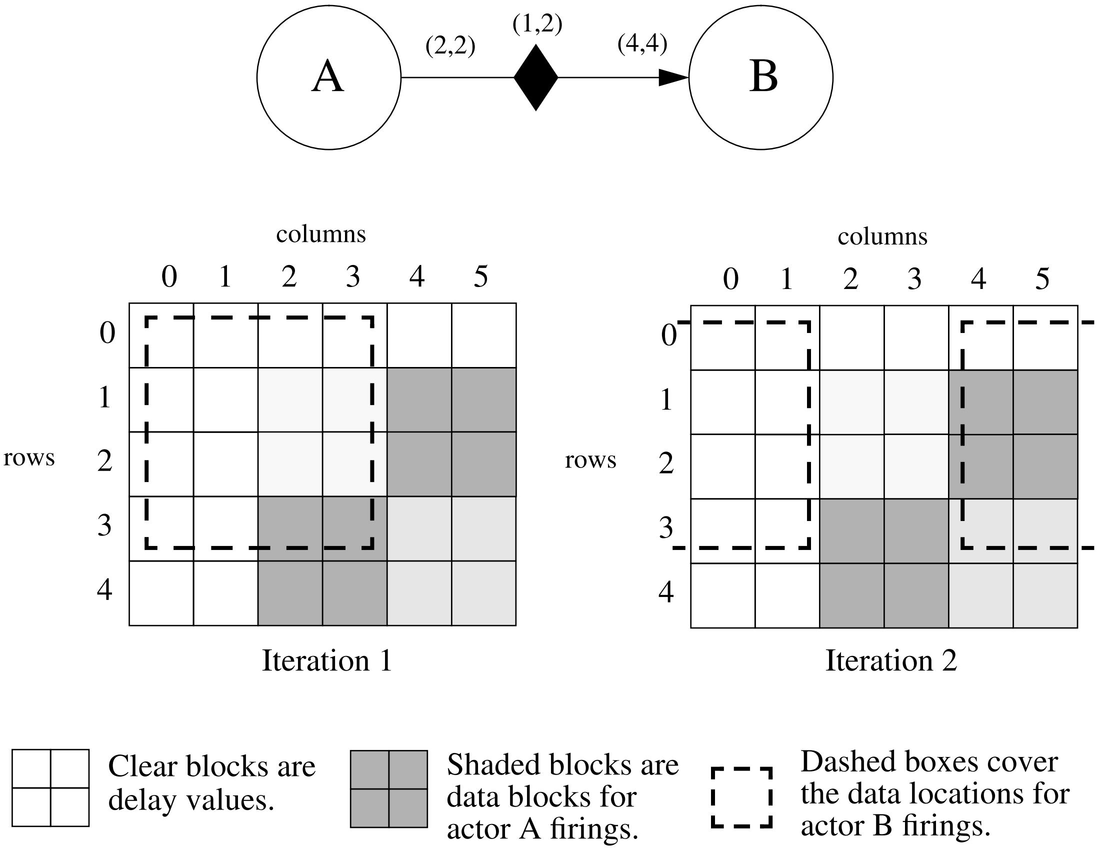



**FIGURE 35. Buffer usage in two iterations of a MDSDF system with constrained delays.**

We can see that the source actor produces submatrices that are always subsets of the buffer space. If the column size of the buffer is increased by a multiple of the original column size of the buffer without delays, then the submatrices of both the source and destination actors will always be proper subset of the buffer space, as shown in Figure 36.

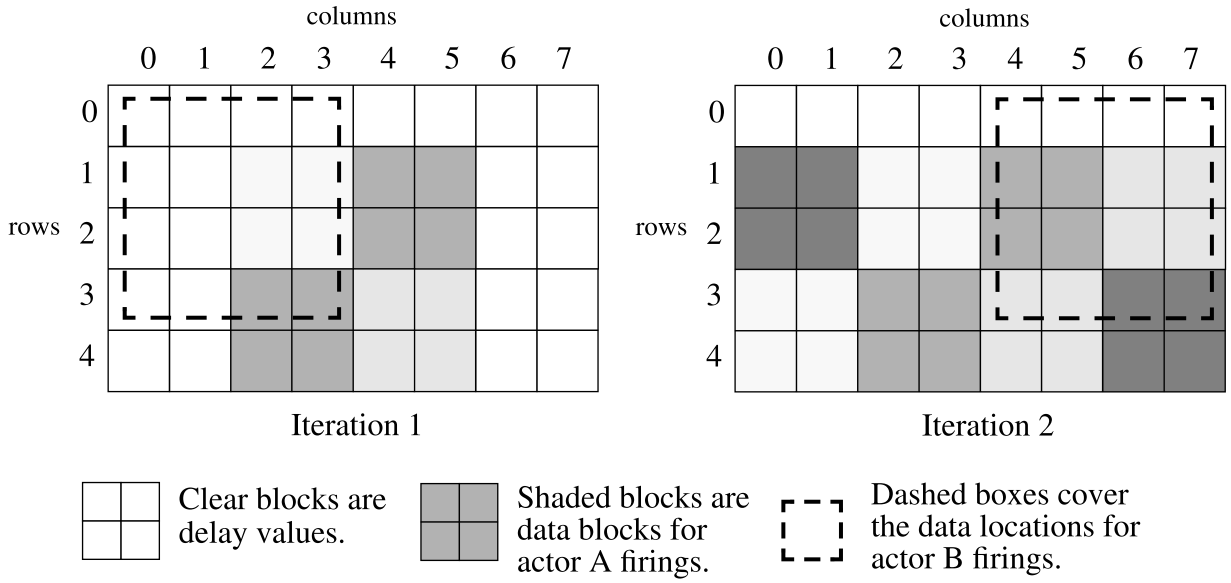



**FIGURE 36. Buffer usage for two iterations of a MDSDF system with constrained delays and where the column size of the buffer is a multiple of the column size of the buffer if there were no delays.**

## 4.4 Extended Scheduling Example

Let us go through an example of using the above rules and definitions to generate a single processor schedule for a larger MDSDF system. We will revisit the problem of generating the schedule for the vector inner product system, which we reproduce below:

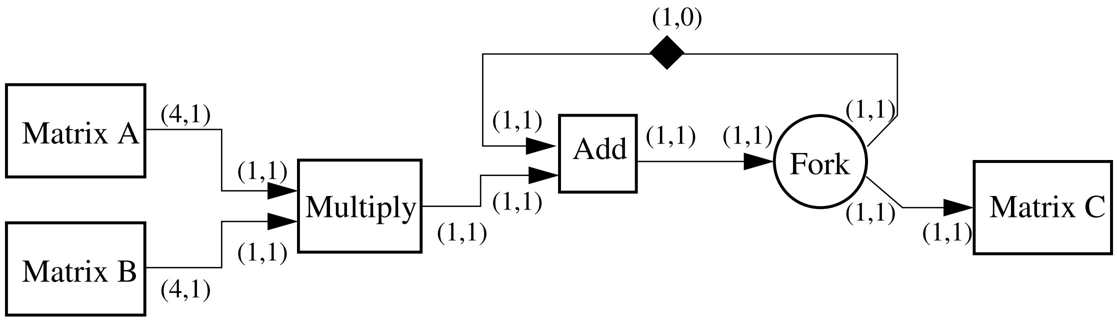



**FIGURE 37. A MDSDF system to do vector inner product.**

First, the balance equations for the system are:

$$4 * r_{A,row} = 1 * r_{mult\_input1,row}$$

$$1 * r_{A,col} = 1 * r_{mult\_input1,col}$$

$$4 * r_{B,row} = 1 * r_{mult\_input2,row}$$

$$1 * r_{B,col} = 1 * r_{mult\_input2,col}$$

$$1 * r_{mult\_output,row} = 1 * r_{add\_input2,row}$$

$$1 * r_{\text{mult\_output,col}} = 1 * r_{\text{add\_input2,col}}$$

$$1 * r_{\text{add\_output,row}} = 1 * r_{\text{fork\_input,row}}$$

$$1 * r_{\text{add\_output,col}} = 1 * r_{\text{fork\_input,col}}$$

$$1 * r_{\text{fork\_output1,row}} = 1 * r_{\text{add\_input1,row}}$$

$$1 * r_{\text{fork\_output1,col}} = 1 * r_{\text{add\_input1,col}}$$

$$1 * r_{\text{fork\_output2,row}} = 1 * r_{\text{C,row}}$$

$$1 * r_{\text{fork\_output2,col}} = 1 * r_{\text{C,col}}$$

We can solve these equations to generate the repetitions count for each actor, which are $A_{\{1,1\}}$, $B_{\{1,1\}}$, $\text{Mult}_{\{4,1\}}$, $\text{Add}_{\{4,1\}}$, $\text{Fork}_{\{4,1\}}$, $C_{\{4,1\}}$. Thus, for one iteration period, actors A and B fire one time each and the other actors all fire four times. The actors that fire four times each consume data down the rows of one column.

Using the scheduling rules we presented previously, the schedule for the vector inner product system is $A_{[0,0]}B_{[0,0]}\text{Mult}_{[0,0]\text{-}[3,0]}(\text{AddFork})_{[0,0]\text{-}[3,0]}C_{[0,0]\text{-}[3,0]}$. The schedule uses a short-hand notation to group the pair of sequential firings of the Add actor followed by the Fork actor. That sequence is executed four times, from index [0,0] to [3,0]. The Add actor can fire the first time because it has a initial data block provided by the delay on its upper input. After its first firing, it needs the output of the Fork actor to continue. Thus, the pair Add and Fork must fire together in series. After one iteration, the Add gets reset because its first input comes from a new column, which again has an initial delay value. The final result is that for each iteration, the system computes the inner product of the two vectors provided by actors A and B. We could make the system into a galaxy, and provide a different pair of input vectors for each call of this galaxy.

# 5.0 Ptolemy Implementation Details

This chapter discusses the details of the implementation of MDSDF in Ptolemy. The ideas do not necessarily require the reader to be a Ptolemy "hacker," but a good understanding of C++ and how the Ptolemy kernel operates would be beneficial.

## 5.1 Two-dimensional data structures - matrices and submatrices

Since MDSDF uses a model in which actors produce data that are part of a two-dimensional data space, the data structure used to represent both the buffers and the subsets of the buffer that the stars can actually work with is very important. Currently, the primary data structure used for the buffer is the `PMatrix` (the 'P' is silent) class from Ptolemy's kernel (please refer to the Ptolemy 0.5 Programmer's Manual for a complete description of the `PMatrix` class and its derivatives). A subclass of the `PMatrix` class was developed to act as the primary structure used by stars to access data from the buffer. There are four `SubMatrix` classes: `ComplexSubMatrix`, `FixSubMatrix`, `FloatSubMatrix`, and `IntSubMatrix`, to match the four corresponding types of `PMatrix` classes.

The primary function of the SubMatrix class is to provide an interface to a subset of the memory allocated by the PMatrix class. Every submatrix has a pointer to a "parent" or "mother" matrix, and many submatrices can have the same parent. Only the parent matrix has allocated memory to act as the buffer. A submatrix simply accesses a subset of this buffer, using the information it knows about its own dimensions and that of its parent's. The interface of the SubMatrix class, in terms of accessing individual entries, is quite similar to that of the PMatrix class. We have overloaded the [] operator and the entry() method so that they return the entry from the correct location in the memory of the parent matrix. The SubMatrix interface is thus exactly the same as that of the PMatrix interface, which is very natural for dealing with two-dimensional data. The various arithmetic operators, such as addition and multiplication, and matrix manipulation operators, such as transpose and inverse, are inherited from the PMatrix class and are still functional on the submatrices. An example of using the SubMatrix class interface is shown in Figure 38.

```
// If A, B, and C are all of type FloatSubMatrix

A = 1.0;          // make all entries of A equal to 1
B[0][1] = 1.1;    // assign individual elements in matrix B
B[0][2] = 2.5;
B[0][3] = 1.5;
C = A * B;      // multiply matrix A and matrix B, put result in C
```

**FIGURE 38. Use of the SubMatrix class interface.**

In the example, the three variables A, B, and C have been previously declared to be of type FloatSubMatrix. The assignment operator has been overloaded to allow us to assign all entries of a matrix to be the same value, as shown in the first code statement. We can also use the [] operator to access an entry of the matrix at a specific row and column, as shown in the next three code statements. The last code statement shows how we can use the * operator, which we have defined to implement matrix multiplication, on two source matrices A and B, and the result of that operation is then assigned to the destination matrix C. The ability to define operators for the Matrix and SubMatrix classes gives us the ability to treat matrices simply by their variable names and operate on them as if they were a new data type in the system.

## 5.2 Buffering and Flow of Data

In the current implementation of the SDF simulation domain, a lot of overhead is involved in moving data particles from one actor to another. A diagram of what this system looks like is shown in Figure 39. Each actor (or functional block) of the system is connected to another by a series of structures. First, there is the porthole, which acts as a buffer to hold the data particles, either before they are sent by output portholes or when they are received by input portholes. The

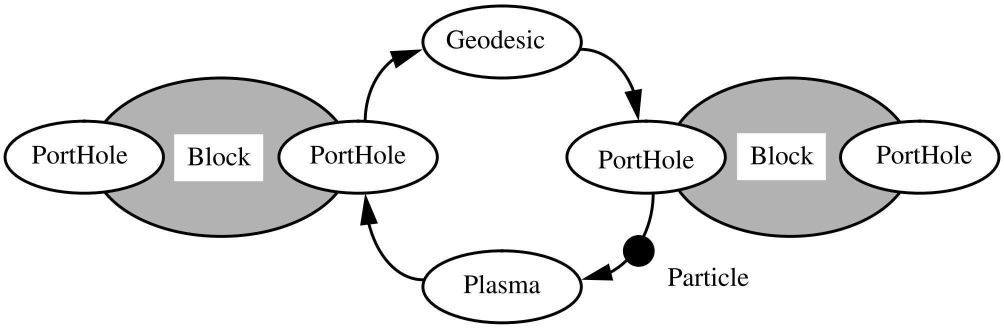



**FIGURE 39. Close-up of connections for data transfer between actors in the SDF simulation domain.**

arc connecting portholes is implemented using the geodesic structure, which also has a buffer that acts as a FIFO queue. The particles go into the geodesic buffer when the source actor has finished firing to produce the data. The particles move from the geodesic buffer to the buffer of the destination porthole when the destination actor is ready to fire. After the destination actor has fired, the "empty" particles are returned to the plasma, which acts as a repository of empty particles that can be reused by the source porthole.

We felt that this system of having three buffers (one in each porthole and one in the geodesic) per arc would be too inefficient for MDSDF. Many of the systems described in MDSDF have large rate changes, which results in a large number of particles flowing through the system if we use the old style of implementation. An example of such a system would be an image processing graph, where we wished to work at the pixel level. A typically sized image would generate thousands of particles of data if treated at such a level. This inefficiency is not inherent to SDF. On the contrary, SDF systems in general have very desirable qualities, such as the ability to make static schedules and perform static buffer allocation for them. These qualities have been implemented for SDF code generation domains, but not for the SDF simulation domain. MDSDF has similar qualities, so we have designed the MDSDF simulation domain to take advantage of these qualities to reduce the amount of buffering overhead in the system.

We mentioned in the previous section that stars in MDSDF access the data space of the buffer using submatrix structures instead of through particles like SDF stars. These submatrices are not buffered at all, but are created and deleted as needed when the star requests one for input or output purposes (it might be even more efficient to allocate a submatrix plasma to store "empty" submatrices so that we can reuse allocated memory for the structures). For example, a star that generates data would first request from the output porthole a submatrix to access the output buffer using the `getOutput()` method of that porthole. That star could then write to the entries of that submatrix using the standard matrix operations. Similarly, a star that receives input from another star could get access to the data using the `getInput()` method of its input porthole. This is in contrast to the standard SDF style of using the `%` operator of the portholes to access the current particle or any previously received particles in its buffer. We will illustrate how stars access these submatrices in a future section. Here, we want to emphasize that there are no buffers of particles or submatrices for data transfer purposes at all in the MDSDF simulation domain implementation. The storage for the data that passes on an arc is allocated by the geodesic as one large mother matrix. The stars at either end of the arc will access subsets of the memory allocated for the mother matrix using submatrices.

Although particles are not used to transfer data in MDSDF, they still are used to facilitate type resolution. Each porthole of an MDSDF system can be of type `COMPLEX_MATRIX`, `FIX_MATRIX`, `FLOAT_MATRIX`, or `INT_MATRIX`. Type resolution is carried out in a way similar to the SDF domain. Specifically, a plasma of the appropriate particle type (a `MatrixParticle` class, with four type specific subclasses named `ComplexMatrixParticle`, `FixMatrixParticle`, `FloatMatrixParticle`, and `IntMatrixParticle` has been created for this purpose) is created for each porthole once the type resolution function of the Ptolemy kernel has determined the appropriate type to be used by each connection. The geodesic code, when allocating a mother matrix to act as the buffer storage, simply gets a matrix particle from one of the portholes connected to it, and that matrix particle will know its own type and how to create appropriately typed matrices and submatrices. Thus, one matrix particle will always exist in the `MDSDFGeodesic`, which puts it in the `ParticleStack` buffer that was inherited from the kernel `Geodesic` class. This particle stack is also used to hold additional matrix particles that might be created if the destination porthole of the arc tries to access a subset of data space with negative indices or with indices beyond the bounds of the mother matrix's dimensions. In that case, the `MDSDFGeodesic` will create dummy matrix particles with no data to act as zero-filled matrices. The particlestack holds all these matrix particles so that their memory allocation can be properly recovered when the galaxy is deleted.

The buffers of the portholes are also still around in MDSDF, but they are not used to store data that is being transferred on the arc. Instead, to maintain backward compatibility with some stars that we copied from SDF that had to use the `%` operator and required a particle input, we have created a `%` operator for MDSDF portholes that will create a temporary matrix particle and copy the data from the submatrix that would normally have been accessed. This temporary matrix particle is stored in the porthole's buffer and is deleted when the porthole is deleted. Currently, the only case of this being used is to support the `MDSDFTkText` stars, which expect inputs of the `Particle` class. The star does not care what the dimensions of the data are, or even that it is a matrix. The reason for our modification is that the `TclTk` stars utilize Ptolemy kernel code that we did not want to duplicate or modify just for the MDSDF case.

In summary, although submatrices are similar to particles in that they should be able to be reused instead of being created and deleted repeatedly for every iteration, the primary difference in the way we treat submatrices is that we never buffer them in the portholes or geodesics. Although it might be possible to buffer the submatrices used by a star in the portholes for that star, which would give us the advantage of maintaining pointers to all the submatrices used so that the system could recover the memory used by the submatrices instead of forcing the star to do so, this would involve an additional complexity of maintaining a two dimensional buffer. In our first attempt at implementing a MDSDF simulation domain, we did not think this extra complexity would provide enough benefit to be justified.

## 5.3 Scheduling and Schedule Representation

The MDSDF scheduler is very similar to the SDF scheduler except for one crucial step. Because MDSDF systems only create buffers in the geodesics, the geodesics must be initialized after the scheduler has calculated the number of repetitions required for all the actors in the galaxy. Thus, the `setup()` method of each star must be called prior to the calculation of repetitions, but before the initialization of the portholes and geodesics of the system. In SDF, the `setup()`

method for each actor, along with the initialization of each of its portholes and their associated geodesics, is initiated by the `prepareGalaxy()` method, which is executed before the repetitions are computed. This would not perform correctly in the MDSDF case because the geodesics need knowledge of repetitions and buffer sizes before they can be properly initialized. Therefore, we had to change the `prepareGalaxy()` method so that only the `setup()` method of each star is called. The geodesics and portholes are initialized after the repetitions have been calculated by a call to a new method named `postSchedulingInit()`. The methods used to calculate the repetitions of each star are simply extensions of the SDF methods, except we now calculate repetitions for both rows and columns instead of just for the star as a whole.

We have created a slightly more complex schedule class for the MDSDF domain. The `SDFSchedule` class was essentially a sequential list of pointers to stars. An `MDSDFSchedule` needs to know more than just the order of the stars. The schedule entries must also have the firing index of the star since the firing index is the only way to determine how a particular firing of a star is mapped into the data space. This index is produced when the schedule is created and then stored along with the star pointer in a cell called the `MDSDFScheduleEntry`. The index stored is not just one (*row*,*column*) pair but actually a (*row*,*column*) star index and a (*row*,*column*) end index range. This allows us to express larger schedules more efficiently. We are essentially storing the syntax used to express single processor schedules like $A_{[0,0]-[4,4]}B_{[0,0]-[2,2]}$ instead of storing each firing of the star as one entry. For a multiprocessor scheduler, we will need to develop new structures to represent such schedules.

## 5.4 Delays and Past/Future Data

When a delay exists in the system, the buffer necessarily has to be larger to hold the data passed between actors. When the row dimension of the delay is greater than zero, we simply extend the row dimension of the buffer by that extra amount. The data values that are pushed down the data space by the initial rows are never used because we increment along the column dimension for subsequent iterations.

When the column dimensions of the delay is greater than zero, we will increment the column dimensions of the buffer by multiples of the original column size of the buffer. We cannot simply increment by the size of the column delay due to the issue that we discussed before about wanting submatrices to access proper subsets of the buffer storage. For example, for the system shown in Figure 40, we use a buffer with twice the column size as would be needed if there were no delay. The row size of the buffer is one greater than the row size that would be needed if there were no delay. The column size of the buffer when there is a delay in the system must be a multiple of the original column size because we want both the input and output submatrices to access proper subsets of the buffer. Since it is possible that either submatrix might access the entire original buffer as its block size, we need the column dimensions of the modified buffer to always be a multiple of the original buffer size. Similarly, if the system has actors that access data in the "past" along the column dimension, we must use a buffer size that has a column dimension that is some multiple of the original column size in order to guarantee that we have room to retain enough samples.

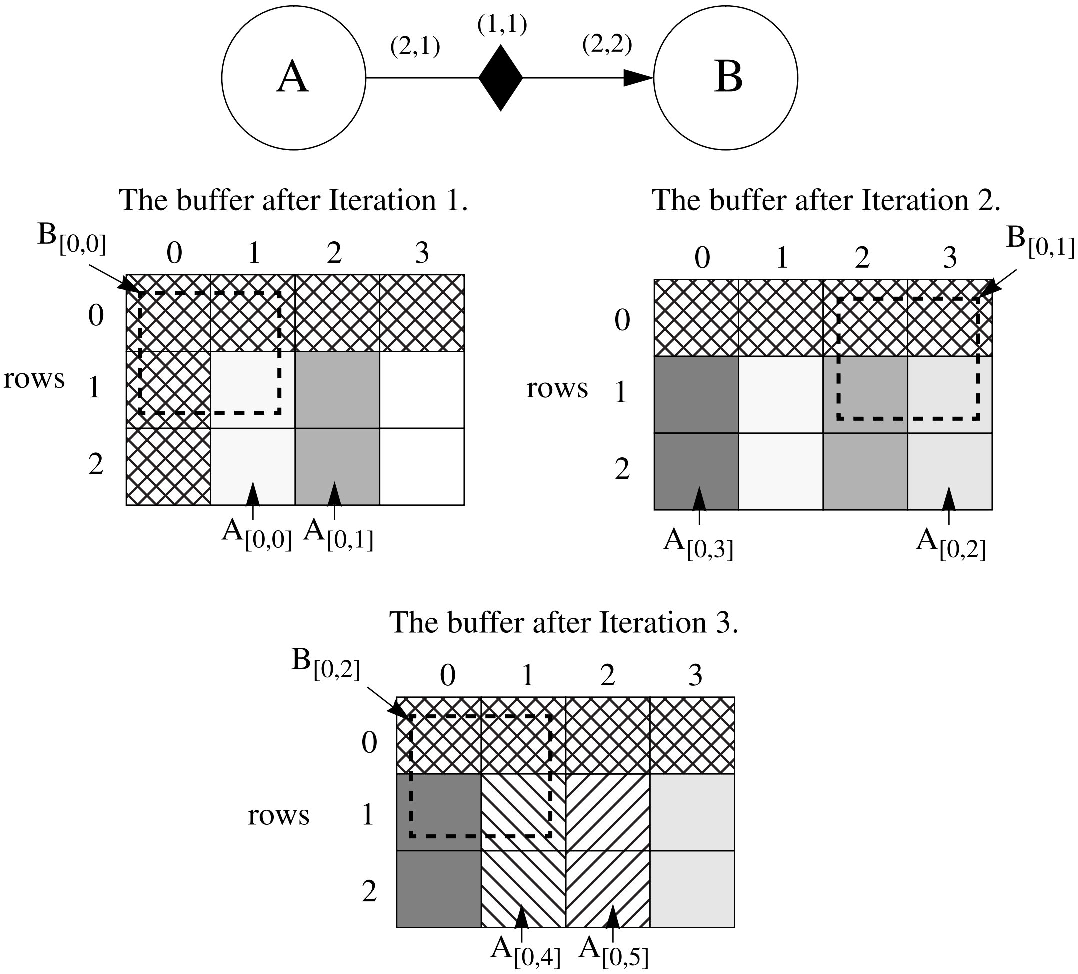



**FIGURE 40. Buffer evolution of a MDSDF system with delay.**

## 5.5 ANYSIZE Inputs and Outputs

There are situations where we would like an actor to be able to receive inputs that are of any dimensions. That actor could be a sink star, such as a star which displays the input and does not care about the type or size of the input, or the actor could be a fork star which simply gives copies of the input to multiple outputs.

We have implemented the ability to support stars which have portholes with specifications that are (*ANYSIZE, ANYSIZE*). The rules for resolving the size that the porthole uses is as follows:

1) No star can have more than one input porthole with ANYSIZE rows or columns.

2) A star with ANYSIZE rows or columns on an output porthole must have an input porthole that also has ANYSIZE rows or columns.

3) All portholes of a star that have ANYSIZE rows or columns will use the same resolved values for the dimensions.

4) ANYSIZE rows or columns are resolved by following the input porthole with ANYSIZE rows or columns and assigning the ANYSIZE row or ANYSIZE column dimension to the corresponding row or column dimension of the output porthole connected to it. If that output porthole itself has ANYSIZE rows or columns (as in the case of cascaded fork stars), then that star is resolved first, following the rules given here, until we find an output porthole which has determinate row and column dimensions.

## 5.6 Writing MDSDF Stars

MDSDF stars are written much differently than the standard dataflow stars in Ptolemy. First, every star should have in its `setup()` method a call to `setMDSDFParams()` for every porthole to declare its dimensions to the MDSDF scheduler. Secondly, since MDSDF stars access their data using submatrices instead of particles, these submatrices are acquired from the input and output portholes using the `getInput()` and `getOutput()` methods, respectively, instead of the `%` operator used by the other Ptolemy dataflow stars to access particles. The reason we adopted new methods for accessing the submatrices instead of overloading the `%` operator was because the `%` operator is limited to a single argument and in the cases where we wish to access past or future submatrix blocks in two dimensions, we need methods that can take two arguments. An example demonstrating these two points is shown below:

```
defstar {
      name { MatrixAdd }
      domain { MDSDF }
      desc {
            Matrix addition of two input matrices A and B to produce matrix C.
            All matrices must have the same dimensions.
      }
      version { %W% %G% }
      author { Mike J. Chen }
      copyright { 1994 The Regents of the University of California }
      location  { MDSDF library }
      input {
            name { Ainput }
            type { FLOAT_MATRIX }
      }
      input {
            name { Binput }
            type { FLOAT_MATRIX }
      }
      output {
            name { output }
            type { FLOAT_MATRIX }
      }
      defstate {
            name { numRows }
            type { int }
            default { 8 }
            desc { The number of rows in the input/output matricies. }
```

```
    }
    defstate {
        name { numCols }
        type { int }
        default { 8 }
        desc { The number of columns in the input/output matricies. }
    }
    ccinclude { "SubMatrix.h" }
    setup {
        Ainput.setMDSDFParams(int(numRows), int(numCols));
        Binput.setMDSDFParams(int(numRows), int(numCols));
        output.setMDSDFParams(int(numRows), int(numCols));
    }
    go {
        // get a SubMatrix from the buffer
        FloatSubMatrix& input1 =
                    *(FloatSubMatrix*)(Ainput.getInput());
        FloatSubMatrix& input2 =
                    *(FloatSubMatrix*)(Binput.getInput());
        FloatSubMatrix& result =
                    *(FloatSubMatrix*)(output.getOutput());

        // compute product, putting result into output
        result = input1 + input2;

        delete &input1;
        delete &input2;
        delete &result;
    }
}
```

Notice how we have declared the types of each porthole. The MDSDF stars use the types `COMPLEX_MATRIX`, `FIX_MATRIX`, `FLOAT_MATRIX`, and `INT_MATRIX`, in contrast to the SDF stars that act on the `PMatrix` class objects, which have portholes declared to be of type `COMPLEX_MATRIX_ENV`, `FIX_MATRIX_ENV`, `FLOAT_MATRIX_ENV`, and `INT_MATRIX_ENV`. The SDF matrix types have the `ENV` extension because the matrix particles in SDF use the `Envelope` structures to hold the matrices being transferred. The MDSDF star uses states that allow the user to change the dimensions of the inputs and outputs for the star as needed. The dimensions are declared in the `setup()` method, as we mentioned before. It is important to note how the calls to `getInput()` and `getOutput()` have been cast to the appropriate return type needed. Type checking is performed by the system during scheduling, so these casts should match the ones declared for the porthole types or else unexpected results will occur. The last thing to note is how we delete the submatrices used to access the data buffers at the end of the `go()` method. This is because the submatrices are currently allocated by the `getInput()` and `getOutput()` methods whenever they are called and no pointers to those submatrices are ever stored (unlike particles). Thus, to prevent memory leaks, the submatrices must be deleted by the stars that created them. The memory for the data actually referenced by the submatrices is not changed since the submatrices are simply access structures and do not allocated any memory of their own for storage purposes.

Often in image processing systems, the stars written will need to access data at the single pixel level. A pixel or any scalar can be accessed by declaring the portholes to provide or require (1,1) matrices, but the submatrix method of accessing these scalar information is inefficient. Therefore, we have provided two simpler functions `getFloatInput()` and `getFloatOutput()` to improve the performance when accessing single entry locations of the mother matrix in the geodesic. These functions return a double and a reference to a double, respectively, so no submatrices are created or need to be deleted. We currently only provide these methods for the `Float` data type, but support may be extended to the other data types supported by Ptolemy in the future. The use of these functions is illustrated in the following code fragment from the `go()` method of the `MDSDFFIR` star:

```
setup {
      input.setMDSDFParams(1,1);
      output.setMDSDFParams(1,1);
}

go {
      // get a scalar entry from the buffer
      double& out = output.getFloatOutput();

      out = 0;
      int tap = 0;

      for(int row=int(firstRowIndex); row <= int(lastRowIndex); row++) {
            for(int col=int(firstColIndex); col <= int(lastColIndex); col++) {
                  out += input.getFloatInput(row,col) * taps[tap++];
            }
      }
}
```

Currently, MDSDF supports a limited method of accessing data with indices to the past and future of the “current” data block. As we mentioned before, ever star firing is mapped to a specific block in the data space. If the star also desires to access data that is outside that block, it can do so, with some limitations. The limitations are that the star can only access data blocks within the current buffer. Data outside the current buffer is considered zero. We do not support dependency along the iterations such that a star that was firing at the last column of the current iteration buffer size would not force a subsequent iteration firing to produce the data for the forward reference. Similarly, a star that is the first firing of an iteration cannot access data from the buffer of the previous iteration. The syntax for making such references is shown in the code fragment for the `MDSDFFIR` star below:

```
defstate {
      name { firstRowIndex }
      type { int }
      default { "-1" }
      desc { The index of the first row of tap values }
}
defstate {
      name { lastRowIndex }
      type { int }
      default { 1 }
```

```
        desc { The index of the last row of tap values }
}
defstate {
        name { firstColIndex }
        type { int }
        default { "-1" }
        desc { The index of the first column of tap values }
}
defstate {
        name { lastColIndex }
        type { int }
        default { 1 }
        desc { The index of the last column of tap values }
}
defstate {
        name { taps }
        type { floatarray }
        default { ".1 .1 .1 .1 .2 .1 .1 .1 .1" }
        desc { The taps of the 2-D FIR filter. }
}
go {
        // get a SubMatrix from the buffer
        double& out = output.getFloatOutput();

        out = 0;
        int tap = 0;

        for(int row = int(firstRowIndex); row <= int(lastRowIndex); row++) {
                for(int col=int(firstColIndex); col <= int(lastColIndex); col++) {
                        out += input.getFloatInput(row,col) * taps[tap++];
                }
        }
}
```

The syntax is very similar to the normal ones used to access the block directly assigned to the firing except we can use negative and positive arguments to `getFloatInput()` and `getInput()` to access data backwards or forwards in the data space, respectively.

## 5.7 Efficient forking of multidimensional data

For a pure dataflow interpretation of one-dimensional SDF, forking amounts to copying of the input particle into two output particles. In our code generation implementation of SDF, we can optimize the fork case because the data does not really need to be copied. In dataflow, the destination stars are not allowed to modify their inputs. So, two destinations of a fork star could simply have a reference to the same input.

This concept is equally valid in the multidimensional case. Although currently not implemented this way, we should be able to have destination portholes of a fork share one geodesic, so that we do not have to have multiple copies of the data in separate geodesics for each output arc of the fork.

## 6.0 Conclusion

This paper has discussed various issues that arose while attempting to implement a MDSDF domain in Ptolemy. There are alternative models for data representation and numerous challenges in efficiently managing the large amounts of data that a typical MDSDF system would generate. We have presented the formal specifications of a workable MDSDF model, and presented some examples of its features. We have also presented a discussion of the complexities involved in implementing a simulation environment for MDSDF and the design decisions we chose to simplify the problems we encountered. Currently, a MDSDF single-processor simulation domain has been implemented in Ptolemy. It has been tested on small simple systems. Future work include implementing a multiprocessor scheduling target and examining possible extensions of the system to greater than two dimensions.

The author would like to acknowledge various people at U.C. Berkeley for numerous ideas and thought provoking discussion. I would like to thank my research advisor Professor Edward A. Lee, without whom I would undoubtedly never have embarked on this project. I would also like to thank the members of Professor Lee's research group, especially Sun-Inn Shih and Tom Parks, for their input on MDSDF and their assistance in implementing the domain in Ptolemy. Lastly, I thank my parents for their encouragement, love, support, and nagging throughout my years in school.

## 7.0 References

[1] E.A. Lee, "Multidimensional Streams Rooted in Dataflow," *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Jan 20-22, 1993, IFIP Transactions A (Computer Science and Technology), 1993, vol.A-23:295-306.*

[2] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, special issue on "Simulation Software Development," January, 1994.

[3] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Vol. 75, No. 9, pp. 1235-1245, September, 1987.

[4] E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing" *IEEE Transactions on Computers,* Vol. C-36, No. 1, pp. 24-35, January, 1987.

[5] Shuvra S. Bhattacharyya and E.A. Lee, "Memory Management for Synchronous Dataflow Programs," Memorandum No. UCB/ERL M02/128, Electronics Research Laboratory, U.C. Berkeley, November 18, 1992.

[6] Shuvra S. Bhattacharyya and E.A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, Dec. 1993, vol.6 (no3):271-88.

[7] Dan E. Dudgeon and Russell M. Mersereau, *Multidimensional Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[8] Jae S. Lim, *Two-Dimensional Signal and Image Proceeding*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

[9] Ingrid M. Verbauwhede, Chris J. Scheers, and Jan M. Rabaey, "Specification and Support for Multidimensional DSP in the Silage Language," *ICAASP '94*.

[10] Frank H. M. Franssen, Florin Balasa, Michael F. X. B. van Swaaij, Francky V. M. Catthoor, and Hugo J. De Man, "Modeling Multidimensional Data and Control Flow," *IEEE Transactions on VLSI Systems*, Vol. 1., No. 3, pp. 319-27, September 1993.