Parallel Programming With Coordination Structures

Steven Lucco* Oliver Sharp†

Computer Science Division, 571 Evans Hall
UC Berkeley, Berkeley CA, 94720

Abstract

Parallel programs display two fundamentally different kinds of execution behavior: synchronous and asynchronous. Some methodologies, such as distributed data structures, are best suited to the construction of asynchronous programs. In this paper, we propose a methodology for synchronous parallel programming based on the notion of a coordination structure, a direct representation of the multidimensional dataflow patterns common to synchronous programs. We introduce Delirium, a language in which one can concisely express many useful coordination structures.

1 Introduction

We are proposing a new methodology for writing parallel programs based on the idea of *coordination structures*. Coordination structures are a direct representation of the multi-dimensional data flow patterns common to a large class of parallel programs. We will begin by defining that class of programs. After introducing coordination structures and showing how to use them as a basis for parallel programming, we will describe Delirium, a coordination language which supports concise, declarative expression of coordination structures.

*Supported in part by an IBM Fellowship. Email address: lucco@fire.Berkeley.EDU

†Supported in part by a Hertz Fellowship and by the Lawrence Livermore National Laboratory. Email address: oliver@karakorum.Berkeley.EDU

All parallel programs have a *communication pattern* that characterizes the exchange of information and synchronization among the sub-computations of the program. A communication pattern has some *connectivity* and may possess the property of *uniqueness*. The connectivity determines which sub-computations can communicate. If a communication pattern has uniqueness, then any given pair of sub-computations can communicate only once (and only in one direction).

Parallel programs can exhibit two fundamentally different kinds of communication pattern: *synchronous* and *asynchronous*. We define synchronous programs to be those whose communication patterns have both uniqueness and deterministic connectivity. Another way to understand synchronous programs is in terms of an *execution graph*. An execution graph is simply a dataflow graph [1] where each node is a stateless sub-computation. Each unique pair of sub-computations in the communication pattern is an arc in the program's execution graph (see figure 1 for an example).

If all execution instances[1] of a program can be summarized by an execution graph, that program is synchronous. Summarization of asynchronous programs requires a more general model based on a *message graph* (see figure 1). Like execution graphs, a message graph shows which sub-computations exchange information during the computation. The differences between these two types of graphs are:

1. Exactly one data item travels along each arc of an execution graph; multiple items may travel along arcs in a message graph.

[1] possible communication patterns of the program for a given input

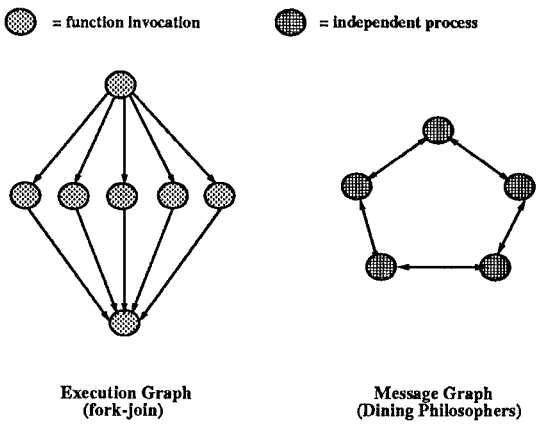

Figure 1: Execution vs. Message Graphs

2. A node in an execution graph corresponds to a state-less function; a node in a message graph corresponds to a process with local state.

A simple example of a problem with only asynchronous solutions is the Dining Philosophers. A more interesting example is the Delirium runtime system[22]. Fast fourier transform[27], and the ray tracer discussed below, are good examples of problems with useful synchronous solutions.

Having identified a program as synchronous or asynchronous, the programmer must choose a notation for expressing it. One purpose of the Delirium project is to investigate whether a single notation can support concise and efficient solutions for both types of problems.

A variety of languages, including SR [4], Sloop [18], and Linda [13], can express asynchronous solutions to problems which also have good synchronous solutions. Using such languages, one can construct protocols [2, 7] or distributed data structures[2] [10] which restrict the interactions between cooperating processes. These methodologies simplify the construction of correct asynchronous programs. However, the do not eliminate non-deterministic communication patterns, which are hard to debug, and hard to implement efficiently [21, 20, 17].

For problems with good synchronous solutions, an alternative is to use a functional language. Such languages support deterministic communication patterns but are unable to directly express asynchronous programs[3]. One might argue that a language which can not express asynchronous programs is too restrictive.

[2]Distributed data structures are a class of protocols.

[3]Given a stream of random numbers, a programmer could write a simulator that supported a truly asynchronous model. However, we are only interested in concise, efficient solutions.

The thesis of this paper, however, is that the best methodologies for writing synchronous programs are irreconcilably different from methodologies for writing asynchronous programs. We propose a methodology, *coordination structures*, which leads to concise, efficient synchronous programs. We show how to implement this methodology using a declarative, deterministic functional language. By declarative, we mean that the connectivity of the program's communication pattern is clear from the program text. In contrast, languages based on distributed data structures elaborate their communication patterns *procedurally*, and thus the pattern that emerges is dependent on the language's underlying evaluation model.

The remainder of this paper has the following organization. Section 2 describes coordination structures, showing their usefulness as a methodology for writing synchronous programs. Section 3 introduces the idea of a coordination language and argues for the linguistic decoupling of coordination from computation. Section 4 discusses Delirium, a coordination language that supports the construction of concise, efficient synchronous programs. Section 5 presents the Delirium implementation of a medium-sized application. Section 6 discusses related work.

2 Coordination Structures

We are proposing a methodology for implementing synchronous parallel programs that is based on *coordination structures*. A coordination structure is a (structured) collection of *coordination items*. Coordination items can be understood as individual ordering dependencies within a program. Imagine the dataflow graph for the following expression:

```
let x = f(<expr1>)
in g(x)
```

In a normal strict functional language, `x` is a name that corresponds to the result of evaluating the application of `f` to `<expr1>`. A different way to understand `x`, however, is that it expresses an ordering dependency. To evaluate the application of `g`, one must first have evaluated the application of `f`. Think of `x` as a pipe that connects an application of `f` to an application of `g`, through which data can flow. Each such pipe is a coordination item.

A parallel computation can be expressed as a multi-stage pipeline of coordination structures. At each stage in the pipeline, a function is applied to the data flowing through each member in the collection of data pipes. Following a function application, the order of items (data pipes) within the coordination structure may be

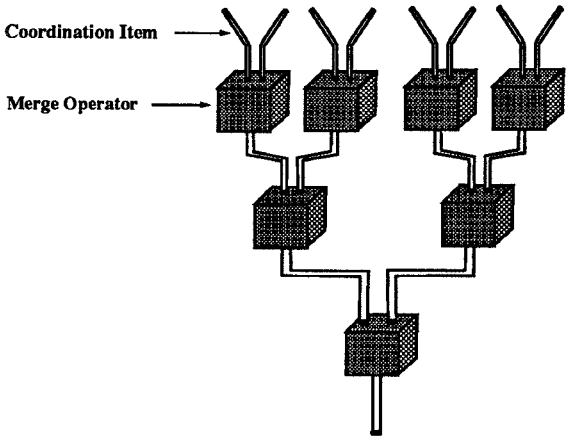

Figure 2: Coordination Structure for Mergesort

permuted to create the appropriate data organization for the next stage of the pipeline.

For example, a useful primitive for many parallel algorithms is binary reduction. A binary reduction takes N data items and applies some associative binary operation to successive pairs, yielding a group of $N/2$ results. The same operation is performed repeatedly until there is only one value left. Many algorithms are based on binary reduction, including, for example, merge sort. The pipeline for merge sort is shown in figure 2 and consists of $log(N)$ applications of the merge operator. If the original N values flow into the pipeline as a vector of pipes, the first step is to divide the pipes into $N/2$ pairs. Next, the program applies an instance of the merge operator to each of these pairs in parallel. One pipe flows out of each merge operator, so the cross section of the pipeline has been reduced to $N/2$ coordination items. The grouping and function application are done repeatedly, until at the end only a single pipe flows out of the pipeline and it contains the sorted list.

It is important to note that the coordination structure of the application as a whole looks like a tree, even though the data that moves through the pipes is probably organized into an entirely different data structure (like a list). We believe that an algorithm is much clearer if the coordination structure is linguistically decoupled from the underlying data structure. A binary reduction always looks like a tree, regardless of whether the structure being operated on is a set, a list, a tree, or an array.

Section 5 describes an application with a complex coordination structure and shows a Delirium realization of this structure. Many classes of algorithms have good synchronous solutions which can be expressed as coordination structures. A few examples are: wavefront algorithms (including many types of dynamic programming—see appendix), algorithms based on communication over trees (including the Delirium compiler [30]), and algorithms based on convolutions over grids (such as Laplace's equation and successive over-relaxation [27]). A significant proportion of numerical scientific programs fall into one of these categories [3].

Because coordination structures directly support data parallel operations, they encourage a programming style that incorporates techniques found in SIMD programs. However, SIMD architectures impose synchronization requirements which narrow their application domain. As illustrated by the application case study in [21], the class of data parallel problems with useful synchronous solutions is significantly larger than the class of problems with SIMD solutions [14].

3 A Case for Coordination Languages

Many parallel language proposals assume that people write parallel programs from scratch. Practical experience [22] and common sense both contradict this assumption.

When parallelizing an application, a programmer often begins with a working version in a sequential language. Typically, the programmer's environment includes useful debugging, profiling, and optimization (or vectorization) tools for this sequential language. In this situation, people don't rewrite their code, they restructure it. They decompose the program into sub-computations that could run in parallel, and then they write new code to coordinate the sub-computations. Usually the coordination code relies on low level synchronization primitives, such as locks or message passing, because no higher level of abstraction is available.

We believe that coordination among sequential sub-computations should be the basic function performed by a parallel programming environment. A *coordination language* performs only this function, giving the programmer the opportunity to express computation in the most convenient notation available.

4 Delirium

Existing coordination languages, such as Linda and Sloop, are *embedded*; they consist of a set of asynchronous coordination primitives which are accessed through statements scattered throughout a host language program.

Delirium is the first example of an *embedding* coordination language [22]. We call it an embedding language because a Delirium program specifies a frame-

work for accomplishing a task in parallel; sequential sub-computations called *operators* are embedded within that framework. To guarantee that a Delirium program will execute deterministically, one need only ensure that operators do not maintain state across invocations.

We believe that embedding coordination languages such as Delirium offer significant advantages for the expression of parallelism. One can express all the glue necessary to coordinate a mid-sized application on a single page of Delirium. This organizing principle makes parallelization easier. Instead of scattering coordination throughout a program, creating a set of ill-defined sub-computations, a Delirium programmer precisely defines sequential operators and embeds these operators within a coordination framework.

Each of these operators is callable directly from Delirium with the same syntax as a function invocation. Operators can be written in any language, including traditional imperative languages like C or Fortran. This allows the programmer to take advantage of existing tools, libraries, and coding strategies.

The Delirium environment currently runs on the Sequent Symmetry, Cray-2, Cray Y-MP, and the BBN Butterfly TC2000. We have developed an efficient run time system for executing Delirium which generally adds less than three percent overhead to the running time of an application. On the Butterfly, the runtime system uses the Tarmac distributed shared memory toolkit to manage caching of data passed between operators [19].

The environment includes an optimizing compiler (written in Delirium) [30], various tools for analyzing and improving execution speed, and a visualization tool for coordination structures. These tools support the incremental parallelization of an existing program. The program can be decomposed into a set of operators; when an operator is too expensive to execute sequentially, it is further decomposed into component pieces which can be computed simultaneously.

4.1 Basic Delirium

At heart, Delirium is a straight-forward functional language which supports first class functions, recursion, iteration, let-bindings, and conditional expressions. All functions are evaluated strictly. There are no computation primitives in the language; all real work is accomplished within operators that are defined in a different language.

Here is a sample Delirium program which solves the eight queens problem, expressing backtracking directly:

```
main()
  let board = empty_board()
  in show_solutions(do_it(board,1))

do_it(board,column)
  let h1 = try(board,column,1)
      h2 = try(board,column,2)
      h3 = try(board,column,3)
      h4 = try(board,column,4)
      h5 = try(board,column,5)
      h6 = try(board,column,6)
      h7 = try(board,column,7)
      h8 = try(board,column,8)
  in merge(h1,h2,h3,h4,h5,h6,h7,h8)

try(board,column,row)
  let new_board = add_queen(board,column,row)
  in if is_valid(new_board)
      then if is_equal(column,8)
            then new_board
            else do_it(new_board,incr(column))
      else 0
```

The code uses the operators `is_equal`, `is_valid`, `add_queen`, `show_solutions`, and `empty_board`. Because each of these does not involve much computation, the overhead for expressing all the backtracking in parallel is significant[4]. A simple solution to reducing overhead is to express only two levels of the recursion in Delirium, calling a recursive C operator to descend the rest of the search tree. The modified version redefines `try` to be:

```
try(board,column,row)
  let new_board = add_queen(board,column,row)
  in if is_valid_op(new_board) then
      if is_equal(column,2) then
        do_it_op(new_board,incr(column))
      else do_it(new_board,incr(column))
     else 0
```

This version, run on eight Sequent processors, is seven times faster than a sequential C program.

4.2 Support for Coordination Structures

This section describes how one can build coordination structures using Delirium *transforms*. Each application of a transform creates a structure which connects two successive stages in a pipelined computation.

Names in Delirium refer to either *integers*, *functions*, *transforms* or n-dimensional *arrays* of untyped values.

[4] On twelve processors, this version took three seconds versus five seconds for the sequential version.

A transform is a rule which replaces a set of *input arrays* by a *result array*; the result array represents some permutation (with possible copying) of the values in the input arrays. Transforms have two parts, a *reshape statement* and a set of *wiring rules*. The wiring rules describe how the transform will permute its input arrays. The reshape statement determines the shape of the transform's result. It also requires that the shapes of the transform's input arrays conform to an *input pattern*. Application of a transform to an array that does not match the transform's input pattern results in a runtime error.

The following grammar describes the syntax of transforms:

transform ::= *heading reshape-stmt*
[*wiring-rules*] [*fill-constant*]

heading ::= *identifier* '(' *identifier-list* ')'
identifier-list ::= *identifier* ',' *identifier-list*
| *identifier*

reshape-stmt ::= *result-shape* '<-' *input-pattern*
input-pattern ::= *subscript-expr*
result-shape ::= *subscript-expr*

wiring-rules ::= 'with' *wiring-rule-list*
wiring-rule-list ::= *wiring-rule wiring-rule-list*
| *wiring-rule*
wiring-rule ::= *lsubscript-expr* '=' *rsubscript-expr*

fill-constant ::= 'fill' *number*

lsubscript-expr ::= *identifier* '[' *simple-expr-list* ']'
simple-expr-list ::= *simple-expr* ',' *simple-expr-list*
| *simple-expr*
simple-expr ::= *number* | *identifier*

rsubscript-expr ::= *subscript-expr* | *number*
subscript-expr ::= *identifier* '[' *index-expr-list* ']'
index-expr-list ::= *index-expr* ',' *index-expr-list*
| *index-expr*
index-expr ::= *expr*
expr ::= any Delirium expression (should yield an integer at runtime)

The operator '=' in a wiring rule is read "depends on". Each wiring rule specifies how a section of the transform's result depends on its input[5]. For example, the transform shown below groups into pairs adjacent elements of a one-dimensional input array:

```
adjacent(P)
  C[n,2]<-P[n]
  with
    C[i,j]=P[i+j]
```

[5] For wiring rules in which the left and right subscript expressions refer to different arrays, '=' is semantically equivalent to an assignment. At present, this is the only kind of rule permitted in a transform.

The reshape statement, `C[n,2]<-P[n]`, contains the input pattern `P[n]`. This pattern matches any one-dimensional array, binding the name `P` to that array. The reshape statement declares that the result of the transform will be a two-dimensional array, `C`.

The single wiring rule of this transform maps successive (overlapping) pairs of adjacent elements from P into the corresponding columns of C. In general, the left-hand side (lhs) of the wiring rule specifies some elements of the transform's result. Array subscripts appearing on the lhs must be either *index variables* or constants. An index variable is an identifier which represents the entire range (with zero origin) of index values along a particular dimension of the result array. Index variables must be unique within a wiring rule and *consistent* within a transform's set of wiring rules. To be consistent, a given index variable must always refer to the same dimension of the result.

The right-hand side (rhs) of a wiring rule must be either a constant or a selection of some elements from one of the transform's input arrays. Array subscripts appearing on the rhs can contain arbitrary arithmetic expressions. Index variables bound on the lhs of a wiring rule can appear in its rhs. If an rhs subscript expression does not specify a valid index of the input array, the value of the entire rhs expression is determined by the *fill constant* of the transform. When defining transforms containing such rhs expressions, the programmer must explicitly specify a fill constant.

The semantics of a set of wiring rules are captured by the following algorithm:

for each wiring rule
  for each value in the range of each index variable
    compute the lhs expression
    if the indicated element of the result has already been specified
      then report error
      else assign the value of the rhs expression to this element

To understand how transforms work, imagine the dataflow graph of a computation to be a set of ribbon cables. These cables have a male and a female plug, both n-dimensional, and wires which connect each input of the female plug to one or more outputs of the male plug. The male plugs correspond to Delirium arrays. Female plugs correspond to the input patterns of transforms; they determine what shapes of input arrays can "plug into" the transform. The wiring rules of the transform constitute a schematic for creating a new male plug.

A computation is just a series of transformations, applied to the program's input "wires". For completeness, we should generalize the ribbon cable analogy so that, at any point, one can bifurcate a cable such that

it has two male ends (since a program may copy an array).

The only thing that remains is to specify how actual work gets done. To do this, we introduce the primitive `map`, which applies a function to groups of elements in Delirium arrays. Depending on how it is applied, `map` groups array elements in different ways; this flexibility is necessary to support functions with multiple array arguments and multiple return values. The simplest way to use `map` is to apply a single argument, single return value function to one array. The result is an array of the same shape as the input array, where each element in the new array is computed by applying the function to the corresponding element of the original one.

If `map` applies a function that returns r results, the dimensionality of the output array is correspondingly increased. An $n \times m$ array, for example, would be transformed into an $n \times m \times r$ array. Element $(8, 10, 2)$ is the third return value of the function when it is applied to element $(8, 10)$ of the input array.

There are two ways to use `map` with multiple argument functions. The first is to apply an n argument function to n arguments. All the arguments must *conform*, meaning that they must either be arrays with the same shape, or scalars, which are automatically replicated (and coerced into the array type). For example, if a four argument single return value function is applied to two $m \times q$ arrays and two scalars, the result would be an $m \times q$ array where each element is computed by applying the function to the two scalars and the two corresponding array elements.

The other way to handle multiple inputs is to get them all from a single dependency array. In this case, the programmer applies an n-element function to an array whose lowest-numbered dimension is n. If the function has r return values, the `map` operation will change the size of the array's lowest-numbered dimension from n to r. For example, if a three argument, two return value function is applied to a 20×3 dependency array, the result array will be of shape 20×2. The two elements in row ten of the result array are the two values returned by the function when it is applied to the three elements in row ten of the original array.

To summarize, `map` can apply a function to either one or n dependency arrays. In the first case, the input arity of the function must match the lowest-numbered dimension of the input array. In the second case, the input arity must be n. The lowest-numbered dimension of the output array will be the output arity of the function.

Note that the reshaping effect of `map` can be described as a transformation. The full semantics of `map` cannot be expressed in a transform, however, because transforms can only build coordination structures. They are not able to apply functions to data flowing through a coordination structure. With this restriction, Delirium linguistically enforces a decoupling between coordination and computation. Normal transforms create patterns of dataflow. The `map` transform applies a function across a pattern.

To demonstrate how `map` is used in practice, here is Delirium code that applies a function of two arguments to each pair of adjacent elements in a vector. The first step is to transform the vector into a two-dimensional array where each two-element column contains adjacent elements from the vector. Once the intermediate array is constructed, a simple `map` operation performs the computation:

```
adjacent(group,P)
  C[n,group]<-P[n]
  with
    C[i,j]=P[i+j]

f(a,b)
  op(a,b)

f-on-adjacent(vector)
  map(f,adjacent(2,vector))
```

This example generalizes the definition of adjacent to group by an arbitrary number (rather than just by pairs). In the example, `vector` is a one-dimensional array, `f` and `f-on-adjacent` are functions, and `adjacent` is a transform. The identifier `op` refers to a functional operator defined in a computational language (see the above description of basic Delirium). Note that `adjacent` makes use of an integer argument `group`. If a program passes an array in this argument position, it will generate a runtime type error.

4.3 Current Research

4.3.1 Expression of Coordination Structures

While the current transform mechanism is sufficient for the specification of any coordination structure, it is oriented toward building these structures one pipeline stage at a time. Coordination structures built this way are easy to understand and manipulate because they can be represented as relationships among n-dimensional arrays. To represent a general coordination structure, we need a more flexible data structure, such as a directed graph. Some coordination structures, such as wavefront computations, can be embedded in arrays. For such structures, it is useful in practice to support both views. Transforms expecting arrays should be able to take array-embedded coordination structures as arguments. In other contexts,

programmers need to be able to express a direct modification to the array-embedded graph.

We have developed *multi-stage transforms* which provide this flexibility. Multi-stage transforms also include a notion of recurrence. Recurrence relations are resolved into coordination structures that can be manipulated explicitly or implicitly (as array-embedded graphs) by transforms. A future report will describe both the semantics and compilation of multi-stage transforms.

Most current Delirium applications use a different strategy for expressing recurrences. Prior to the development of multi-stage transforms, we provided a recurrence notation similar to Crystal [11] which programmers could use as an alternative to writing recurrences as iterations over transforms. Crystal is a system that converts computations expressed as recurrence equations into a systolic architecture. At an intermediate step in this conversion, the Crystal compiler derives a set of linear combinations that express the communication inherent in the given recurrence. There is a straightforward mapping from such sets to Delirium transforms.

4.3.2 Discovery of Coordination Structures

Delirium is a notation for the expression of coordination structures. However, it does not directly support the *discovery* of coordination structures inherent in existing code. We are developing a workbench which supports this activity through analysis of sequential C and Fortran programs. The analysis tool can also assist in decomposing a program into sets of Delirium operators.

5 A Delirium Example

This section presents a case study of an application which can be parallelized using a synchronous communication pattern. The study demonstrates that Delirium programs can be both concise and efficient.

The UNIGRAFIX ray tracer is a 10,000 line C program developed at UC Berkeley [23]. A parallel version of this ray tracer has been implemented in C on the Sequent Symmetry [9], using the notion of replicated worker processes [32]. The pattern of data dependencies in this program is complex for two reasons. First, the program breaks scan lines into subsections with a fixed number of pixels. Second, the program uses an antialiasing strategy involving averaging across multiple scan lines. These two properties combine to create a computation with a three-dimensional data dependency graph.

The C language solution to this problem is based on distributed data structures. Specifically, the C program creates a set of queues, one for each averaging operation. Each worker process polls one of the work queues, waiting for a complete set of data to accumulate on the front of the queue. This solution requires 500 lines of C (with references to locking primitives). It completely obfuscates the underlying communication pattern of the algorithm. Even in a language like Linda, one has no choice but to express the communication pattern of this algorithm indirectly through a distributed data structure.[6]

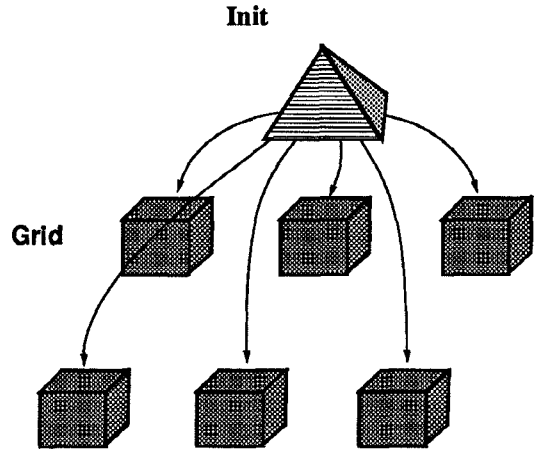

Figure 3: First Stage of Ray Tracing Computation

The coordination structure of the ray tracing task as a whole is a five stage pipeline, where the stages are `init`, `grid`, `super-sample`, `filter`, and `output`. The Delirium strategy for constructing this pipeline is to create a transform to represent each stage.

The first stage is `init`, which generates a database describing the image being rendered. A reference to the database is handed to each of $n \times m$ instances of the `grid` function. The first rerouting stage of the pipeline takes the single pipe emerging from the `init` function and branches it into an $n \times m$ array of pipes all containing the database reference (see figure 3). The `super-sampling` stage computes a value for each two by two block of `grid` values. Figure 4 is a two-dimensional representation of its three-dimensional coordination structure, using cubes to represent the `grid` functions and pyramids for `super-sample`. The `filter` stage computes a value for each three by three block of `super-sample` values. All the resulting values are passed to the `output` stage where they are processed linearly.

Here is a Delirium realization of the ray tracer's coordination structure:

[6] The Linda coordination code required 165 lines.

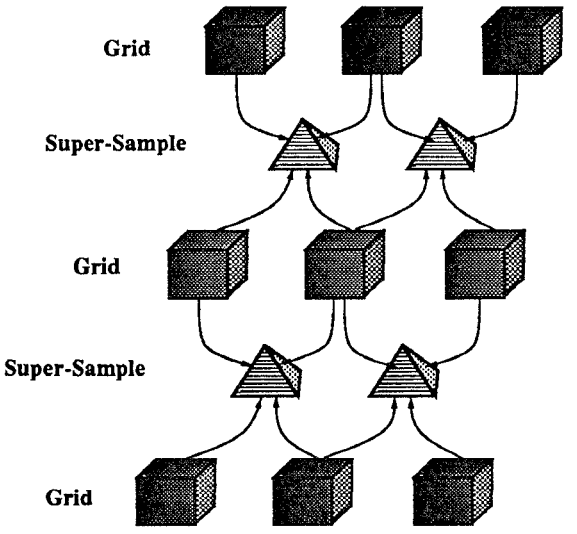

Figure 4: Second Stage of Ray Tracing Computation

```
replicate(n,m,value)
  C[n,m]<-value
  with
    C[n,m]=value

main()
  let
    database=init("scene-database-file")
    grid-points=
      map(grid,replicate(NSCANLINES,
                         NPIXELS,database))
    samples=
      map(super-sample,adjacent(2,grid-points))
    image=map(filter,adjacent(3,samples))
  in output(image)
```

This example uses the transform **adjacent** described above. It also introduces a new transform, **replicate**, whose purpose is to change a scalar value into a two-dimensional array. The function **init** returns a *reference* to a complex scene database; however, **replicate** treats the reference to this data structure as a scalar, since interpretation of the database takes place only within the computational code.

The example takes advantage of a special property of one-dimensional transforms. If a one-dimensional transform is applied to a multi-dimensional array, the transform is generalized to apply along all dimensions of the array. To compute the result array in this case, the transform is applied to successive dimensions of its k-dimensional input array, yielding k result arrays. The result arrays are then concatenated along their lowest-numbered dimension, yielding the single array that is the final result of the transform.

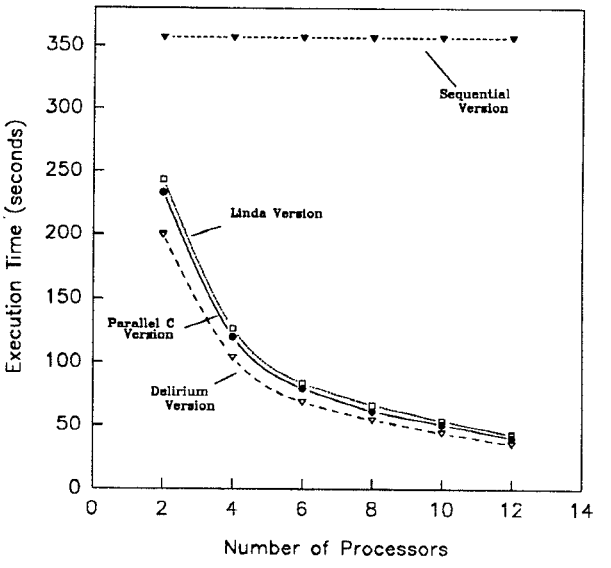

Figure 5: Performance of Ray Tracer Programs

In the absence of this property, one could define the two-dimensional **adjacent** transform as follows:

```
flatten(P)
  C[n*m]<-P[n,m]
  with
    C[i]=P[i/n,i mod n]

adjacent4D(group,P)
  C[n,m,group,group]<-P[n,m]
  with
    C[i,j,k,l]=P[i+k,j+l]

adjacent(group,P)
  flatten(adjacent4D(group,P))
```

Figure 5 compares the performance of this Delirium program with programs written in C and Linda for the same application. The Delirium program for the ray tracer runs 14% faster than the version in which the coordination is expressed in C and 22% faster than an optimized Linda version. One strategy that allows the Delirium run time system to implement transforms efficiently is called *summarization*. Due to their regular structure, transforms can often be subdivided into sections of optimal grain size. Figure 6 shows the effect of different subdivisions on the ray tracer's execution time.

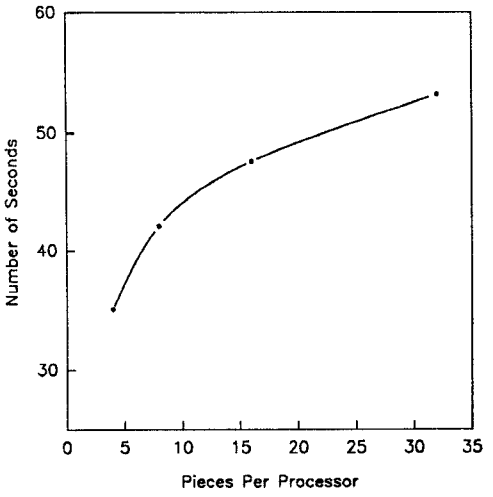

Figure 6: Effect of Partition Size

6 Related Work

6.1 Functional Languages

One can achieve the organizing effects of coordination structures using higher-order functions. For example, one could write a function that groups pairs of adjacent elements in a vector:

λP . λf .
construct array A
with range $i = 0$ to $length(P) - 1$
such that $A[i] = f(P[i], P[i+1])$

To realize the same degree of parallelism as the equivalent Delirium transform, this higher-order function must be implemented in a functional language which has *lazy aggregate construction* and strict function applications. By lazy aggregate construction, we mean that an array can be used before all its elements are computed. Combining this property with strict function application, a functional language could realize the multi-dimensional, pipelined communication pattern created by Delirium transforms.

6.1.1 Strict Functional Languages

Most dataflow languages, including VAL [24] and SISAL [25], are evaluated strictly. These languages have strict aggregate construction, and so they can not implement Delirium transforms. The language ParALFL [15], though not a dataflow language, also has strict arrays; however, one could implement coordination structures inefficiently in ParALFL using its lazy lists.

6.1.2 Lazy Functional Languages

Some dataflow languages, such as Id [5, 26], have lazy aggregate construction as well as lazy function application. To realize the parallelism of Delirium transforms, a coordination structure built in Id would require strict evaluation of function applications within lazy arrays. This unintuitive evaluation strategy could be arranged by a compiler that was designed to recognize coordination structures as a stylized idiom. In contrast, Delirium adds the transform mechanism to an otherwise strict language; using this mechanism, programmers directly express the desired evaluation behavior.

6.1.3 Array Comprehensions

Anderson and Hudak discuss the construction of "lazy arrays within a strict context" using a functional syntax called *array comprehensions* [3]. One can use such array comprehensions to implement coordination structures. However, array comprehensions can conveniently express only those coordination structures which can be embedded in arrays.

Like multi-stage transforms, array comprehensions can express recurrences. We will provide a detailed comparison between array comprehensions and multi-stage transforms in a future report. We have found that the complexity of compiling the Delirium recurrence notation into iterations over transforms is similar to the complexity of compiling Haskell [16] array comprehensions for sequential machines.

6.2 Asynchronous Coordination

As was mentioned above, protocols [2, 7] and distributed data structures [10] are useful in asynchronous languages because they place restrictions on the generality of communication patterns. Another approach suggested by Guy Steele [31] limits the types of operations that cooperating processes can perform on shared data.

For synchronous programs, Delirium's coordination structures are a better methodology than these asynchronous programming techniques. Coordination structures have two main advantages: they are deterministic and they express the program's communication pattern declaratively, making it easier to deduce from examination of the program text. We have completed a study [21] which demonstrates that for synchronous applications, programs written in terms of coordination structures are often more concise and more efficient than programs written in terms of distributed data structures.

6.3 Aggregate Primitives

6.3.1 Early Aggregate Languages

Delirium transformations have been heavily influenced by APL [12], which introduced the idea of a pipeline of functional transformations that modify an aggregate structure. This idea was significantly elaborated by FP [6], which provides a rich set of functional operators for creating new transformations. APL does not have first class functions, and so can't express coordination structures. FP's functional operators are similar to Delirium transforms, but operate on a data object that must model memory as well as coordination, and are thus difficult to implement efficiently. Water's series expressions [33] also create pipelined computations; however, like FP and APL, they provide a fixed set of operators for modifying dataflow through the pipeline. In contrast, Delirium transforms are a general mechanism for creating dataflow modification operators. The appendix lists some of the FP, series, and APL functional operators, implemented as Delirium transforms.

6.3.2 SIMD Languages

For many applications, data parallel operations are the most likely source of massive parallelism. Concise and efficient data parallel programs have been written for SIMD multiprocessors. The class of data parallel problems with useful synchronous solutions is significantly larger than the class of problems with SIMD solutions [14]. This is because SIMD architectures impose synchronization requirements which limit their application domain. Languages intended for SIMD programming, such as C* [29], incorporate these architectural limitations into their semantics.

C* programs compiled for MIMD machines experience performance degradation for two reasons [28]. First, the compiler must insert barriers to synchronize after each conditional branch within a data parallel operation (and to synchronize after the whole operation). Second, if data parallel operations on an array refer to other values within that array, the compiler must pre-copy the array to ensure data consistency.

C* includes as primitives useful data parallel operations such as *scan* [8]. In Delirium, one can use transforms to specify the dataflow pattern for this and many other such operations. For example, the Delirium transform for general reduction is:

```
# combine sets up for function
# call on grouped elements

combine(group,P)
  C[n/group,group]<-P[n]
  with
    C[i,j]=P[i*group+j]

# this function performs a reduction
# the repeated assignment does not
# imply serialization of the loops

reduce(f,group,P)
  while (shapeof(P)[0]>1)
    P=map(f,combine(group,P))
```

7 Conclusion

We have presented a framework which classifies the communication patterns of parallel programs into two types: synchronous and asynchronous. Asynchronous programs have inherently non-deterministic behavior. The challenge with these programs is to discipline the communication among a set of independent processes so that this non-determinism is manageable. With synchronous programs, communication is deterministic, but often complex and multi-dimensional. The two kinds of parallel programs map naturally onto different implementation methodologies. Distributed data structures are well-suited to the expression of asynchronous programs. Synchronous programs can be concisely and efficiently expressed in terms of coordination structures. We have proposed a language mechanism which supports the creation and manipulation of such structures. We have used this mechanism to construct concise and efficient implementations for several applications.

8 Acknowledgments

We would like to thank Guy Steele for his extensive comments on earlier drafts of this paper, Marti Hearst for her help in improving the presentation, and David Culler for his insights into functional languages.

References

[1] W. Ackerman. "Data Flow Languages". *Computer*, 15(2), February 1982.

[2] D. Anderson. "Automated Protocol Implementation with RTAG". *IEEE Transactions on Software Engineering*, 14(3):291–300, March 1988.

[3] S. Anderson and P. Hudak. "Compilation of Haskell Array Comprehensions for Scientific Computing". In *Proceedings of the ACM SIGPLAN*

'90 Conference on Programming Language Design and Implementation, pages 137–149, White Plains, NY, June 1990.

[4] G. Andrews. "The Distributed Programming Lanaguage SR—Mechanisms, Design and Implementation". *Software Practice and Experience*, 12(8):719–753, 1982.

[5] Arvind and K. P. Gostelow. "An Asynchronous Programming Language and Computing Machine". Technical Report TR114a, Dept. of Information and Computer Science, University of California, Irvine, December 1978.

[6] J. Backus. "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM*, 21(8), August 1978.

[7] A. Birrell and B. Nelson. "Implementing Remote Procedure Calls". *Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] G. Blelloch. "Scans as Primitive Parallel Operations". *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[9] B. Boothe. "Multiprocessor Strategies for Ray Tracing". Master's thesis, University of California/Berkeley, 1989. Tech Report 534.

[10] N. Carriero, D. Gelernter, and J. Leichter. "Distributed Data Structures in Linda". In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1986.

[11] M. Chen. "A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI". In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, pages 131–139, 1986.

[12] A. Falkoff and K. Iverson. "The Design of APL". In E. Horowitz, editor, *Programming Languages: A Grand Tour*, pages 240–250. Computer Science Press, Inc., 1987.

[13] D. Gelernter. "Parallel Programming in Linda". In *Proceedings of the International Conference on Parallel Processing*, pages 255–263, August 1985.

[14] D. Hillis and G. L. Steele Jr. "Data Parallel Algorithms". *Communications of the ACM*, 29(12):1170–1183, 1986.

[15] P. Hudak and L. Smith. "Para-functional Programming: A Paradigm for Programming Multiprocessor Systems". In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, pages 243–254, 1986.

[16] P. Hudak and P. Wadler. "Report on the Programming Language Haskell". Technical Report YALU/DCS/RR666, Yale University, Department of Computer Science, November 1988.

[17] S. Lucco. "A Heuristic Linda Kernel for Hypercube Multiprocessors". In *Proceedings of the Second Conference on Hypercube Multiprocessors*, September 1986.

[18] S. Lucco. "Parallel Programming in a Virtual Object Space". In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[19] S. Lucco and D. Anderson. "Tarmac: A Language System Substrate Based on Mobile Memory". In *International Conference on Distributed Computing Systems*, 1990.

[20] S. Lucco and K. Nichols. "A Performance Analysis of Three Parallel Programming Methodologies in the Context of MOS Timing Simulation". In *Digest of Papers: IEEE Compcon*, pages 205–210, 1987.

[21] S. Lucco and O. Sharp. "A Comparison of Two Strategies for Data Parallel Programming on MIMD Architectures (*submitted for publication*)".

[22] S. Lucco and O. Sharp. "Delirium: An Embedding Coordination Language". In *Proceedings of Supercomputing '90*, November 1990.

[23] D. Marsh. "UgRay—An Efficient Ray-Tracing Renderer for UniGrafix". Master's thesis, University of California/Berkeley, May 1987.

[24] J. R. McGraw. "The VAL Language: Description and Analysis". *ACM Transactions on on Programming Languages and Systems*, 4(1):44–82, January 1982.

[25] J. R. McGraw. "SISAL: Streams and Iteration in a Single Assignment Language". Technical report, Lawrence-Livermore National Laboratory, March 1985.

[26] R. S. Nikhil. "ID Reference Manual, version 88.0". Technical Report 284, MIT Laboratory for Computer Science, 1988.

[27] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computation.* Cambridge University Press, 1988.

[28] M. Quinn, P. Hatcher, and K. Jourdenais. "Compiling C* Programs for a Hypercube Multicomputer". In *Proceedings of the ACM/SIGPLAN Parallel Programming Experience Symposium*, July 1988.

[29] J. Rose and G. L. Steele Jr. "C*: An Extended C Language for Data Parallel Programming". In *Proceedings of the Second International Conference on Supercomputing*, May 1987.

[30] O. Sharp. "Pythia: An Parallel Compiler for Delirium". Master's thesis, University of California/Berkeley, 1990.

[31] G. L. Steele Jr. "Making Asynchronous Parallelism Safe for the World". In *Proceedings of the Seventeenth ACM Symposium on the Principles of Programming Languages*, pages 218–231, January 1990.

[32] M. Sullivan and D. Anderson. "Marionette: a System for Parallel Distributed Programming using a Master/Slave Model". In *International Conference on Distributed Computing Systems*, 1989.

[33] R. Waters. "Appendix A: Series". In G. L. Steele Jr., editor, *Common LISP: The Language, 2nd edition.* Digital Press, 1990.

A Appendix

Some APL, series, and FP functional operators implemented as Delirium transforms:

```
# concatenates two arrays
# along lowest-numbered dimension

catenate(P,Q)
  C[m+n]<-P[m],Q[n]

# removes first s1 x s2 elements from P

drop(P,s1,s2)
  C[n-s1,m-s2]<-P[n,m]
  with
    C[i,j]=P[i+s1,j+s2]

# APL outer product (function
# supplied separately by map)

outer-product(P,Q)
  P[n],Q[m]->C[n,m,2]
  with
    C[i,j,0]=P[i]
    C[i,j,1]=Q[j]

# reshape for inner product

set-up-ip(P,Q)
  P[m,n],Q[n,q]->C[m,q,n,2]
  with
    C[i,j,k,0]=P[i,k]
    C[i,j,k,1]=Q[k,j]

# perform inner product (see
# definition of reduce in text)

inner-product(plus-fn,times-fn,a,b)
  reduce(plus-fn,2,map(times-fn,set-up-ip(a,b)))
```

Recipe for dynamic programming:

```
# first N stages of 2D dynamic programming

dptop(P)
  P[n]->C[n,2]
  with
     C[i,j]=P[i-j+1]

# second N stages

dpbottom(P)
  adjacent(2,P)
```