Reports in Informatics no. 45, 1990 (revised 1997), pp. 26, ISSN 0333-3590.
Department of Informatics, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway

Dependencies and Space Time Algebras in Parallel Programming

Magne Haveraaen
University of Bergen, Norway

Abstract

One aim for the development of the functional programming language Sapphire is to give explicit control over the distribution of computations on the processors of massively parallel machines, yet Sapphire itself does not contain any references to processes nor communications. Sapphire is based on the view that an algorithm has a data dependency aspect and a computational aspect. A program is formed by combining the computational aspect, defined in a constructive recursive function, with an appropriate data dependency. The data dependency defines the dependency pattern of the algorithm, and is coded as a normal program module. In order to execute a program on a parallel computer, the program's dependency pattern has to be embedded in the computer's communication structure. In Sapphire this embedding is also expressed as a reusable program module. The approach is illustrated by defining a butterfly data dependency, and coding the broadcast-sum and the fast Fourier transform algorithms over it. These programs are targeted for two different parallel architectures, a linear array and a hypercube, by providing two alternative definitions of the butterfly.

1 Introduction

In the analysis of programs for the purpose of generating parallel code, see e.g. Chen & al. [5] and Sarkar [17], it has been known for a long time that the data dependencies inherent in a program is of utmost importance. Most of these approaches rely on extracting and analyzing the data dependencies implicit in the code. In this paper we show that it is possible to separate the definition of an algorithm's data dependency from its computational aspect, and that both aspects may be defined by simple program modules. This turns out to be very useful in porting programs to, and between, parallel machines. The computational aspect remains unchanged, while the data dependency modules are defined to distribute the program on the parallel machine. The modules are reusable components, and the same data dependency may be used by many algorithms. In this paper we will show two different algorithms that have the same data dependency, illustrating the reusability of these components.

The data dependency modules define a class of mathematical structures we call data dependency algebras, or *dda*'s for short. Every dda is isomorphic to a graph, so a dda is just another way of expressing the data dependency graph. Every node in the dda corresponds to a computation in the algorithm, while an arc represents dependencies between computations. We have chosen to develop a functional programming language to work with this concept. The recursive functional style gives the programming a more holistic view, helping us avoid much of the technical detail an imperative style would force us to work with.

The space-time communication structure of a parallel machine may also be viewed as a dda. We call this the space-time algebra (*sta* for short) for that machine. Since we are using discrete time, the sta's basically model the class of SIMD (single instruction multiple data) computers. In the sta a node corresponds to a processing element at a specific time-step, and an arc corresponds to a communication channel. In the paper Miranker & Winkler [16] it was proposed that a method for programming parallel machines would be to embed the program's data dependency graph in the space-time graph of the target parallel machine. Since we are able to express these graphs as program code, the embedding of an algorithm's dda can also be expressed using normal program code. We do this by expressing the dda in terms of the transitive closure, the *tca*, of an sta. The dda's nodes are given space-time coordinates by this, and dependency arcs become communication paths in the sta. This defines precisely the distribution and communication structure of the algorithm on a parallel computer. With this information we may compile a functional program to parallel imperative code with the recursion removed. The approach is best suited for massively parallel machines, since, in principle, we want to have one data item per processor.

We have developed the functional programming language Sapphire for exploring this approach. Its theoretical bases is studied in the paper Čyras and Haveraaen [8].

This paper is organized as follows. In the next section we introduce some basic concepts of Sapphire, some of its built-in types, and define the butterfly dda using these tools. Then we express two algorithms, the fast Fourier transform and the broadcast-sum, in terms of the butterfly. In section four we define sta's for a hypercube and a linear array, and show how a butterfly may be embedded in these space-time structures. We then compare our approach with some other approaches before the concluding section.

2 A Functional Programming Language

Sapphire programs are based on the theory of many-sorted algebras (see, e.g., Wirsing [21]). Important here is the separation between the syntactic declaration, the *signature*, of types and functions and their *implementation.* This is a feature shared with languages such as Ada [1], and Sapphire signatures are parameterized much the same way that Ada's generic packages are parameterized, i.e., with types and functions as parameters. In Sapphire we will often have several implementations (package bodies) for a signature, e.g., different embeddings of a butterfly dda for different machine architectures.

```
signature $\mathcal{D}$
  parameters
    sort  dda, dir
    operations
      canR : dda * dir → bool;
               -- canR(p,d) checks if node p has an outgoing arc in direction d
      r    : dda * dir | canR → dda;
               -- the arc from node p in direction d leads to the node r(p,d)
      dirS : dda * dir | canR → dda;
               -- the arc from node p in direction d has the direction dirS(p,d) at node r(p,d)
      canS : dda * dir → bool;
               -- canS(q,e) checks if node q has an incoming arc in direction e
      s    : dda * dir | canS → dda;
               -- the arc to node q with direction e comes from the node s(q,e)
      dirR : dda * dir | canS → dda;
               -- the arc to node q with direction e has the direction dirR(q,e) at node s(q,e)
  end
```

Figure 1: The signature $\mathcal{D}$ for a data dependency algebra. The signature itself has no special interpretation, only the comments explain the intended meaning of the functions.

2.1 Signatures and Algebras

Signatures declare sort names and function names, defining syntactic and type properties. A *sort name* is an identifier for a type, specifically there is a sort name bool (standing for the logical values). A *function name*, declared by $\mathsf{f} : \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_\mathsf{n} \longrightarrow \mathsf{s}$, defines an identifier f, a *domain* $\mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_\mathsf{n}$, and a *codomain* s. The s and s_i are sort names, and the declaration means that the result is of sort s, and that the argument at position i should have sort s_i. This would be expressed by the declaration **function** $\mathsf{f}\ (\mathsf{x}_1 : \mathsf{s}_1,\ \mathsf{x}_2 : \mathsf{s}_2,\ \ldots,\ \mathsf{x}_\mathsf{n} : \mathsf{s}_\mathsf{n}) : \mathsf{s}$ in a language like Ada, where the parameters x_i are formal parameter names. Given that $\mathsf{p} : \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_\mathsf{n} \longrightarrow \mathsf{bool}$ (a predicate declaration), we will sometimes use the form $\mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_\mathsf{n} \mid \mathsf{p}$ for the domain. This means that f is defined for all arguments where p is true. We will use this to declare partial functions, i.e., functions that need not be defined for every argument value. A *constant* is declared as a function with an empty list of arguments.

A signature (see for instance Figures 1 or 6 or the first part of Figure 2) declares a collection of sort and function names, after the keyword **defines**. Parameters to a signature are given as a block of sort and function names after the keyword **parameters**. A signature may contain several parameter blocks before the actual declaration.

An *algebra* associates a meaning to a signature by defining a *carrier* S for each sort name s and a *function* for each function name. A carrier is a set, e.g. defined by a type expression in a program implementation. Given a function name $\mathsf{f} : \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_\mathsf{n} \longrightarrow \mathsf{s}$, the associated function f must be defined on the corresponding sets $f : S_1 \times S_2 \times \ldots \times S_n \to S$. Functions may be defined using program code, which in principle is just expressions formed from previously defined functions.

2.2 Implementations

One way to define the algebra for a specific signature is to provide constructions, commonly known as *implementations*. In Sapphire an implementation is a module with the declaration of a parameterized signature followed by a *code* section. The code section has to define expressions giving appropriate semantical definitions for all entities defined in the **defines** section of the signature declaration. The implementation module does not have global bindings, and all the sorts and functions accessible inside it are declared in the **parameters** part of the signature. When an implementation is instantiated, a meaning will be supplied to the parameters.

The code defining a sort defines its *data structure*, and is formed using type expressions. A type expression is either a *product* of sorts, a *conditional product*, or a *disjoint sum*. A product definition has the form $\mathsf{s} = \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_n$ where the s_i are sort names. Given that S_i is the carrier for s_i, then the carrier for s is the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$. A conditional product $\mathsf{s} = \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_n \mid \mathsf{p}$, where $\mathsf{p} : \mathsf{s}_1 * \mathsf{s}_2 * \ldots * \mathsf{s}_n \rightarrow \mathsf{bool}$ is a predicate, defines the carrier for s to be the set

$$\{(x_1, x_2, \ldots, x_n) \in S_1 \times S_2 \times \ldots \times S_n \mid p(x_1, x_2, \ldots, x_n)\}$$

This is also known as set comprehension. Disjoint sums are not used in this paper and will not be defined here. Functions are defined in the traditional way: $\mathsf{f}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n) = exp$, where *exp* is formed from function applications and the use of the variables x_i. Figure 2 shows a complete implementation module. The module will be explained at the end of this section.

Within the code section of an implementation module the definitions are fully visible. Thus the definition of a sort makes its right hand side interchangeable with its left hand side. This is necessary in order to define the functions, see, e.g., the canR and canS predicates in Figure 2, which rely on the fact that dda is a pair of integers. However, an implementation is an encapsulation mechanism: the defined entities are atomic for a user of the implementation module. This can be seen in Figure 2, where the functions r and s return a pair of integers. A user of the module can only see the declared sort name dda, and will have to consider the result of these functions as atomic values.

For simplicity, we will assume that bool and its usual operations (the constants true and false, operations not : bool → bool, and : bool * bool → bool, and or : bool * bool → bool), as well as the integers (int with carrier $\mathcal{Z}$) and complex numbers (complex) with their common operations are part of the language. In addition we will freely use the following integer operations. (When referring to binary digits (bits), the binary representation of a number, we count the bits from the right, the rightmost being digit 0).

```
digit      : int * int → int;         -- digit(n,i) extracts bit i from n
setdigit   : int * int * int → int;   -- setdigit(n,i,b) sets bit i of n to b
complement : int * int → int;         -- complement(n,i) complements bit i of n,
             -- complement(n,-1)=n
reverse    : int * int → int;         -- reverse(n,d) reverses the bits of the d-bit number n
```

```
implementation Butterfly
  parameters
    operations
      h : → int; −− height of butterfly
  defines
    sort  dda, dir
    operations
      −− operations as in signature $\mathcal{D}$:
      canR : dda * dir → bool;
      r    : dda * dir | canR → dda;
      dirS : dda * dir | canR → dir;
      canS : dda * dir → bool;
      s    : dda * dir | canS → dda;
      dirR : dda * dir | canS → dir;
      −− operations specific for the butterfly data dependency:
      left  : → dir;               −− the two directions for the arcs in the butterfly
      right : → dir;
      B     : → $\mathsf{table}_{\mathsf{int,dda}}$;     −− sequence of bottom nodes, nodes without outgoing arcs
      lower : dda → dda            −− move to the bottom node along the vertical arcs
  local
    operations
      ddinv : int * int → bool −− datainvariant for the dda
      drinv : int → bool,      −− datainvariant for dir
  code
    ddinv(r,c)        = (0<=r<=h) and (0<=c<2**h);
    dda               = int * int | ddinv; −− the set $\{(x, y) \in \mathcal{Z} \times \mathcal{Z} | \ 0 \leq x \leq n$ and $0 \leq y < 2^h\}$
    drinv(n)          = (0 <= n <= 1);
    dir               = int | drinv; −− the set $\{0, 1\}$
    left              = 0;
    right             = 1;
    canR((rf,cf), d)  = (0 < rf); −− there are no nodes below row 0
    canS((rt,ct), d)  = (rt < h); −− there are no nodes above row h
    r((rf,cf), d)     = (rf-1,setdigit(cf,rf-1,d)); −− direction left to lower column numbers,
    s((rt,ct), d)     = (rt+1,setdigit(ct,rt,d)); −− direction right to higher column numbers.
    dirS((rf,cf), d)  = if d = digit(cf,rf-1) → d
                        | d<>digit(cf,rf-1) → 1-d
                        fi; −− crossover arcs have different “r” and “s” directions
    dirR((rt,ct), d)  = if d = digit(ct,rt) → d
                        | d<>digit(ct,rt) → 1-d
                        fi;
    B                 = evolving c : int s-from add(create,0,0) until (c=2**h)
                        in true → (c+1,(0,c)) end;
                        −− using the natural numbers as a linear dda
    lower(rt,ct)      = (0,ct);
  end
```

Figure 2: Code defining a butterfly of height h. left will index the left arc (towards the lower column numbers) and right the right arc (towards the higher column numbers) at a node.

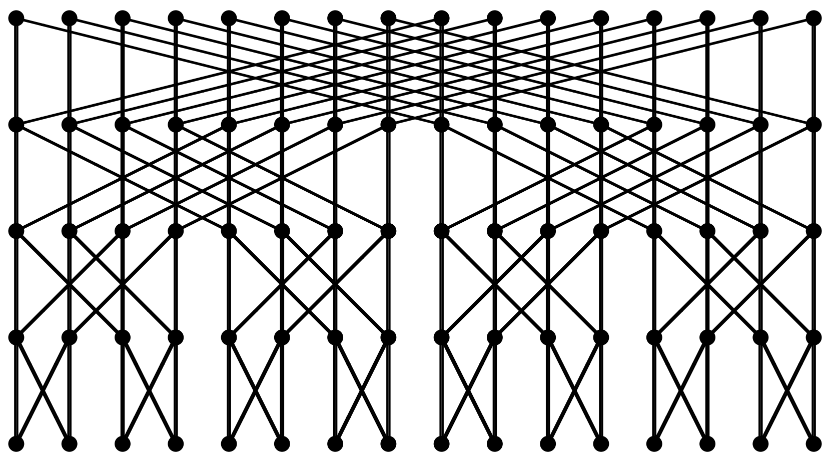


Figure 3: A butterfly of height 4

The module Butterfly (see Figure 2) defines the *butterfly* data dependency graph (Figure 3), basically with signature $\mathcal{D}$. The implementation uses integer pairs, in the range (0,0) through $(\mathsf{h}, 2^{\mathsf{h}} - 1)$, as the carrier. The butterfly has nodes indexed by coordinate pairs (i,j), i being row index and j being column index. A butterfly may be defined recursively on the height parameter h. With height 0, the butterfly contains only one point, indexed by (0,0). A butterfly of height h, is composed (horizontally) from two butterflies of height h-1, such that $2^{\mathsf{h}-1}$ is added to the column numbers of the rightmost subbutterfly. Then a new top row with indices (h,0) through $(\mathsf{h}, 2^{\mathsf{h}} - 1)$ is added. A node (h,j) in the top row is connected by a vertical arc to the node (h-1,j) (a top node in one of the butterflies of height h-1) and by a diagonal arc to the node (h-1,complement(j,h-1)). The r function captures the relationship between a node and the nodes it has arcs leading to, while the s function allows a move in the opposite direction. This implementation also defines a table, actually a sequence, B of bottom nodes (row 0), and an operation lower which moves vertically from a node to the bottom.

2.3 Data Dependency Algebras and Constructive Recursion

A data dependency algebra is the definition of a directed multigraph using program code. A directed multigraph is a set of nodes N and a set of arcs E, and functions $k : E{\rightarrow}N$, returning the source node of an arc, $m : E{\rightarrow}N$, returning the target node of an arc. Given a set D of directions, we may use the elements of this set to enumerate all outgoing arcs at each node. Then a node $n{\in}N$ and a direction $d{\in}D$, the *source representation*, will uniquely identify each arc. The function k on this representation is just $k(n, d) = n$. We may also enumerate the incoming arcs at every node using D, and represent the arcs by node and incoming direction, the *target representation*. Then the function m becomes trivial because $m(n, d) = n$. Note that source and target representations may have different directions associated with the same arc.

In the definition of a data dependency algebra we think in terms of both of these representations for identifying arcs. We have a set of nodes, carrier for the sort name dda, and a set of directions, carrier for the sort name dir. Given the source representation, we define r-functions (for *receive*) r : dda * dir →dda, that return the target node of an arc (corresponding to m above), and dirS : dda * dir →dir (*direction to send*), that returns the direction in the target representation of the same arc. Thus the pair (r(p,d),dirS(p,d)) switches from the source representation (p,d) to the target representation of the same arc. Since only some of the directions dir will be in use at any one node, these functions are partial, and the predicate canR(p,d) (*can receive*) will indicate whether the expression (p,d) actually identifies an arc in the source representation. We do not need to define an equivalent of the k function, as this just returns the first component of the dda * dir pair. Similarly we have the functions s (*send*), dirR (*direction to receive*), and canS (*can send*) on the target representation. This is summed up in Figure 1, where we declare the signature $\mathcal{D}$ with an explaining commentary.

Definition 2.1 [dda] A *data dependency algebra* has signature $\mathcal{D}$ (see Figure 1) and satisfies the axioms (the "⇒" means that the equations are conditional)

```
canR(p,d) ⇒ canS(r(p,d),dirS(p,d));
canR(p,d) ⇒ s(r(p,d),dirS(p,d)) = p;
canR(p,d) ⇒ dirR(r(p,d),dirS(p,d)) = d;
canS(p,d) ⇒ canR(s(p,d),dirR(p,d));
canS(p,d) ⇒ r(s(p,d),dirR(p,d)) = p;
canS(p,d) ⇒ dirS(s(p,d),dirR(p,d)) = d.
```

□

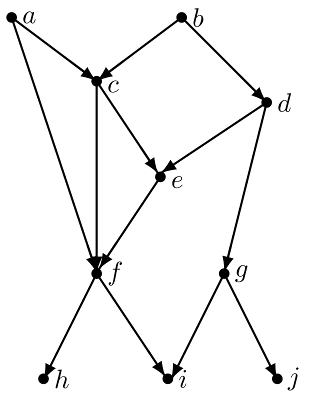
Figures 4 and 5 illustrate the relationship between a graph and the functions declared in $\mathcal{D}$. They also show some expressions using these functions. In Figure 2 we declared the butterfly dda. Another example is the infinite linear graph. This dda is definable by taking the natural numbers as carrier for dda, with only one direction unit. The predicates are canR(n,unit)= 0<n and canS(n,unit)=true. The functions r(n,unit)=n-1 and s(n,unit)=n+1. Both dirS and dirR return unit for all arguments.

In our functional language we express recursion as following the r-connections of a dda. A *constructive recursive function* (see Čyras and Haveraaen [8]) is a function where the argument domain is a dda and that satisfies the following:

Definition 2.2 [constructive recursion] Given a data dependency algebra with signature $\mathcal{D}$ (Figure 1). A function f with signature f : dda → t, where t is any sort, is *constructive recursive* if the program code used to define it has the following syntactic restrictions:

- the definition of f is not recursive, or
- all recursive calls of f in the code for f(p) are of the form f(r(p,d)).

□



−− paths from the top

r(a,2) = r(b,1) = c;

r(a,1) = r(r(a,2),1) = f;

r(r(b,2),1) = r(r(b,1),2) = r(r(a,2),2) = e;

r(r(a,1),1) = r(r(r(a,2),1),1) = h;

r(r(r(b,1),1),2) = r(r(r(b,2),2),1) = i;

−− all paths from the bottom to node c

s(s(h,1),2) = s(s(s(h,1),3),1) = c;

s(s(i,1),2) = s(s(s(i,1),3),1) = c;

−− circular trips

a = s(s(r(a,1),2),1) = s(r(r(a,2),1),1);

b = s(s(r(r(b,1),2),2),1) = s(s(r(r(b,2),1),1),2);

i = r(r(s(s(s(i,1),3),2),2),1);

Figure 4: A simple directed graph illustrating the dda concepts. The nodes are named by letters, and each outgoing arc at a node is enumerated from 1 to 3 from left to right, likewise for each incoming arc. The dda functions for this graph are defined in Figure 5. The right column state some properties of the graph based on these functions.

The use of the r function of the dda in the definition of f may be thought of as if the function call f(p) *receives* the value of the function f at the node r(p,d). From this follows that the value of f at a node q has to be *sent* to the nodes s(q,e) for all directions e such that canS(q,e). This is the reason for the send-receive terminology in the definition of a dda. The dda itself then defines the data dependencies of the function f. The computations of f, its *computational aspect* f', is obtained by replacing the recursive calls f(r(p,d)) with variables. Setting these variables to the appropriate values, i.e., the value of f at r(p,d), will allow the non-recursive function f' to compute the value that f(p) has.

As an example of a constructive recursive function we will define the Broadcast-Sum. It is a very simple function that adds together the values at the bottom node of a butterfly, so that a copy of the sum is placed in all top nodes. The broadcast and the summation is done intermixed. One way to envision this is to look at the butterfly as many juxtaposed trees, each one with a distinct top node as its root, and all trees sharing all bottom nodes as their leaves. The function sum below defines the summation on any one of these trees, and for a butterfly dda the sum is computed on all trees in parallel.

```
sum : dda → complex,
sum(d) = if     canR(d,left) → sum(r(d,left)) + sum(r(d,right))
         | not canR(d,left) → lookup(DV,d)
         fi
```

The sum function defines a value at every node of the butterfly dda. The values at the bottom nodes are given by lookup(DV,d). DV is a distributed table, i.e., every element of

canR(a,1) = true;	r(a,1) = f;	dirS(a,1) = 1;	canS(a,1) = false;		
canR(a,2) = true;	r(a,2) = c;	dirS(a,2) = 1;	canS(a,2) = false;		
canR(a,3) = false;			canS(a,3) = false;		
canR(b,1) = true;	r(b,1) = c;	dirS(b,1) = 1;	canS(b,1) = false;		
canR(b,2) = true;	r(b,2) = d;	dirS(b,2) = 1;	canS(b,2) = false;		
canR(b,3) = false;			canS(b,3) = false;		
canR(c,1) = true;	r(c,1) = f;	dirS(c,1) = 2;	canS(c,1) = true;	s(c,1) = a;	dirR(c,1) = 1;
canR(c,2) = true;	r(c,2) = e;	dirS(c,2) = 1;	canS(c,2) = true;	s(c,2) = b;	dirR(c,2) = 1;
canR(c,3) = false;			canS(c,3) = false;		
canR(d,1) = true;	r(d,1) = e;	dirS(d,1) = 2;	canS(d,1) = true;	s(d,1) = b;	dirR(d,1) = 1;
canR(d,2) = true;	r(d,2) = g;	dirS(d,2) = 1;	canS(d,2) = false;		
canR(d,3) = false;			canS(d,3) = false;		
canR(e,1) = true;	r(e,1) = f;	dirS(e,1) = 3;	canS(e,1) = true;	s(e,1) = c;	dirR(e,1) = 2;
canR(e,2) = false;			canS(e,2) = true;	s(e,2) = d;	dirR(e,2) = 1;
canR(e,3) = false;			canS(e,3) = false;		
canR(f,1) = true;	r(f,1) = h;	dirS(f,1) = 1;	canS(f,1) = true;	s(f,1) = a;	dirR(f,1) = 1;
canR(f,2) = true;	r(f,2) = i;	dirS(f,2) = 1;	canS(f,2) = true;	s(f,2) = c;	dirR(f,2) = 1;
canR(f,3) = false;			canS(f,3) = true;	s(f,3) = e;	dirR(f,3) = 1;
canR(g,1) = true;	r(g,1) = i;	dirS(g,1) = 2;	canS(g,1) = true;	s(g,1) = d;	dirR(g,1) = 2;
canR(g,2) = true;	r(g,2) = j;	dirS(g,2) = 1;	canS(g,2) = false;		
canR(g,3) = false;			canS(g,3) = false;		
canR(h,1) = false;			canS(h,1) = true;	s(h,1) = f;	dirR(h,1) = 1;
canR(h,2) = false;			canS(h,2) = false;		
canR(h,3) = false;			canS(h,3) = false;		
canR(i,1) = false;			canS(i,1) = true;	s(i,1) = f;	dirR(i,1) = 2;
canR(i,2) = false;			canS(i,2) = true;	s(i,2) = g;	dirR(i,2) = 1;
canR(i,3) = false;			canS(i,3) = false;		
canR(j,1) = false;			canS(j,1) = true;	s(j,1) = g;	dirR(j,1) = 2;
canR(j,2) = false;			canS(j,2) = false;		
canR(j,3) = false;			canS(j,3) = false;		

Figure 5: The definition of the dda functions for the graph illustrated in Figure 4. This specification is rather cumbersome because there is no regularity in the graph. Compare this with the definition of the butterfly (Figure 2) which defines a butterfly of any size.

DV is associated with a node d of the butterfly dda. The values at the top nodes of the butterfly will contain the computed sums for all the elements of the table DV.

2.4 Tables and the evolving statement

Tables play an important role in the definition and distribution of data on a computer, so it is one of the few built in types in Sapphire. The signature $\mathcal{T}$ (see Figure 6) declares the table operations. A table may be viewed as an array where the index type need not be an integer, or a symbol table which may be indexed by and store any type of data. The table type is parameterized by an index type ind and element type base. We will sometimes subscript the sort table with its parameters, e.g., $\mathsf{table}_{\mathsf{ind,base}}$ to distinguish

```
signature 𝒯
    parameters
        sort  ind, base
    defines
        sort  table -- a table of entries of type base indexed by ind
        operations
            create  : → table;
                     -- an empty table, i.e., a table with no entries
            legal   : table * ind → bool;
                     -- legal(T,i) checks if i is an index for an entry in T
            legent  : table * ind * base → bool;
                     -- if legent(T,i,b) then it is legal to add b with i as index to T
                     -- note that legent(T,i,b) = not(legal(t,i))
            add     : table * ind * base | legent → table;
                     -- add(T,i,b) returns a table where b has been added to T at index i
            lookup  : table * ind | legal → base;
                     -- lookup(T,i) returns the value indexed by i in T
            remove  : table * ind → table;
                     -- if legal(T,i) is satisfied, then remove(T,i) returns a table where the entry
                     -- indexed by i has been removed from T, otherwise it returns T
            size    : table → int;
                     -- size(T) tells the number of entries in the table T
    end
```

Figure 6: The signature $\mathcal{T}$ for the built in type table. Table is polymorphic, with index and base types as parameters.

different instantiations of the type. With reals as index type and integers as base type for a table instantiation, the expression 0 = size(remove(remove (add(create, 2.1, 45), 3.4), 2.1) is well defined and true.

A table indexed from 1 by consecutive natural numbers is called a *sequence*. We will often use a shorthand of the form [a,b,c,...] to denote specific sequences. This stands for the expression add(...add(add(add(create,1,a),2,b),3,c),...) .

A table where the index type is a dda may be thought of as defining a value distributed over (some of) the nodes of the data dependency graph; A value belongs to the node n if it is indexed by n in the table.

In addition to the operations given in the signature $\mathcal{T}$, Sapphire has an *evolving* statement which also manipulates tables. There are some variations of this statement, of which the **from** and **s-from** variants are relevant to this presentation. The simpler statement is the **from** variant, which has the following following syntactic structure:

evolving *dvar*

from *dtable*

in *guard* → (*tind*,*tbase*) **end**

This is a combined filter, or set comprehension, and *map* statement on the table *dtable*. The *dvar* is a local variable name ranging over the index type of the table *dtable*. The guarded expression of the form *guard* →(*tind*,*tbase*) is just like an alternative in an if-statement, and acts as a filtering statement, only letting the values of *dvar* satisfying the *guard* statement to appear in the resulting table. The resulting table consists of the pairs (*tind*,*tbase*), where the expression *tind* identify the new indices and *tbase* are the base values being indexed in the resulting table. Given a table X : $\mathsf{table}_{\mathsf{int},\mathsf{real}}$, the *evolving* statement:

```
evolving i : int from X
in lookup(X,i)<>0.0 → (i,round(lookup(X,i)*lookup(X,i))) end
```

will return a $\mathsf{table}_{\mathsf{int},\mathsf{int}}$ where all elements equal to 0.0 have been removed (the *filter* part), and the other elements have been squared and rounded to the nearest integer (the *map* part). It is also possible that the result table has different index values, and even different index type, from the input table.

The other variant of the *evolving* statement is the **s-from** variant. It has an additional component, and requires that the index type of the table must be a dda:

```
evolving dvar
s-from dtable until pred
in guard → (tind,tbase) end
```

This is also a filter-map statement, but the table it uses as input is *not* the *dtable*, but a subset TS(*dtable*) of the nodes of the dda. This subset is defined to be the set of nodes n such that every r-path from n leads to a node in *dtable* and no node on this r-path, including n, satisfies the predicate *pred*. An *r-path* is the sequence of nodes we traverse by following the arcs in the r-direction. Referring to Figure 4, we may be given a table T1 with only node i as an index, and T2 with nodes h and i as indices. Then $TS(\mathsf{T1}) = \{i\}$ and $TS(\mathsf{T2}) = \{a, c, e, f, h, i\}$. If the predicate *pred(dvar)* defined a cut at dvar = e, all nodes with r-paths via e would be removed, hence $TS'(\mathsf{T2}) = \{f, h, i\}$. The set TS(*dtable*) may be computed by the algorithm in Figure 7. It generates the set using the s connections, hence the name **s-from** for this variant of the evolving statement.

A useful application of the **s-from** variant is to define a function that replicates a value v a specified number of times. We use the fact that int is the carrier for a linear dda in its definition. Remember that evolving iterates from the index values of input table, and that a sequence with one element has 1 as its only index value (in the expression [n]).

```
replicate(v,n) = evolving i : int s-from [n] until i=n+1 in true → (i,v) end
```

If the expression *tbase* in the **s-from** evolving statement contains a call to a constructive recursive function f, we may compute the value of the function at each node as they are added to the set TS(*dtable*) by the algorithm in Figure 7. When a node is added to TS(*dtable*), the value of the function f at all the r-neighbors of that node have been computed, and we need only compute the non-recursive *computational aspect* f' with the appropriate values of f as inputs. This is a very efficient way to compute a recursive function, especially if the data dependency graph has many paths leading to the same nodes (e.g. the butterfly in Figure 3), as the value of f at any one node will only be

```
$TS(dtable) := \{\ \}$;
move the indices of $dtable$ to the set $DS_0$;
$n := 0$;
repeat
        compute the set $DS_{n+1}$ of all the neighbors in the s-direction of $DS_n$,
                omitting the nodes where $pred$ is satisfied;
        move all nodes in $DS_n$ that have all their r-neighbors in $TS(dtable)$ to $TS(dtable)$;
        move the remaining nodes in $DS_n$ to $DS_{n+1}$;
        $n := n + 1$;
until $DS_n = \{\ \}$;
```

Figure 7: Imperative algorithm for determining the set $TS(dtable)$

computed once. The filter-map part of the evolving statement may be used to retain the evaluated function at all interesting nodes during one expansion of the dda. This proves the following theorem.

Theorem 2.3 (imperative code) *A constructive recursive function* f *defined over a dda may be translated into a recursion free loop program which evaluates the computational aspect of* f *at most once for every node of the dda.*

The constructive recursive function sum previously defined, defines a value at every node of the butterfly dda. We have mentioned that the values at the bottom nodes are values from the distributed table DV. The function SUM below defines the relation between the input table DV and a distributed output table. The output table is just the function value of sum at the top nodes of the butterfly. ok(DV) is a predicate that checks if size(DV) = 2**h, where h is the height of the butterfly.

```
SUM : (table_{dda,complex} | ok) → (table_{dda,complex} | ok),
SUM(DV) = evolving d : dda
          s-from DV until false
          in not(canS(d,left)) → (d,sum(d))
          end;
```

Notice the use of the predicate not(canS(d,left)) in the filter position in order to retain only the sum values computed at the top nodes of the butterfly. To use this function on an input table X : $\text{table}_{\text{int,complex}}$ to compute an output table Y : $\text{table}_{\text{int,complex}}$ we would call:

```
Y = collect(SUM(distribute(X)));
```

The functions distribute and collect are defined below. The function readdress, which is used in the definition of collect, moves all the nodes at the top of the butterfly to the corresponding nodes at the bottom of the butterfly. B is the sequence of bottom nodes.

```
readaddress : table_{dda,complex} → table_{dda,complex}, -- lower the data to the bottom nodes
readdress(R) = evolving d : dda from R in true → (lower(d), lookup(DR,d)) end;
distribute : table_{int,complex} → table_{dda,complex}, -- distribute X at the bottom nodes
distribute(X) = evolving i : int from B in true → (lookup(B,i), lookup(X,i)) end;
collect : table_{dda,complex} → table_{int,complex}, -- collect a distributed table into a sequence
collect(R) = let BR = readdress(R)
             in evolving i : int
                from B
                in true → (i,lookup(BR,lookup(B,i)))
                end
             end;
```

2.5 The Fast Fourier Transform

Another constructive recursive function being defined over the butterfly data dependency is the fast Fourier transform (FFT for short). It is a function from a 2^h element complex vector X to a 2^h element complex vector. The FFT is a standard algorithm presented in most text books on algorithm theory (see, e.g., Manber [14]). The function fft below defines the essentials of the FFT computation on a butterfly. It is defined in a context with a butterfly dda with height h (see Figure 2). We have also used the complex constant c$= -2 \cdot \pi \cdot i$, where i is the complex unit. exp is the complex exponentiation function. The functions fst and snd return the first and second components of a pair of integer pairs, respectively.

```
FFT : (table_{dda,complex} | ok) → (table_{dda,complex} | ok)
FFT(DV) = let fft : dda → complex,
              fft(d) = if canR(d,left) → lookup(DV,d)
                       | not canR(d,left) → exp(c·row/(2**col))·fft(r(d,right)) + fft(r(d,left))
                       fi where { row=fst(pro(d)), col=snd(pro(d)) }
          in evolving d : dda
             s-from DV until false
             in not(canS(d,left)) → (d,fft(d))
             end
          end
```

The declaration of fft, the function from a node of the dda to a complex number, is local to the declaration of FFT, the function from a distributed table to a distributed table. The outer function provides the abstraction of a fast Fourier transform, defining the input to the fft function, and collecting the interesting values being computed by fft. The inner function fft defines the actual computations that have to be done. If we want to compute the FFT of a sequence X : $\text{table}_{\text{int,complex}}$, we will have to distribute it over the bottom nodes of the butterfly before calling the FFT. The result is the distributed table DR : $\text{table}_{\text{dda,complex}}$, which will have to be collected to a sequence Y : $\text{table}_{\text{int,complex}}$ if we do not want a distributed result.

```
DR = FFT(evolving i : int from B in true → (lookup(B,i), lookup(X,reverse(i,h)) ) end);
Y = collect(DR);
```

B is the set of bottom nodes from the butterfly. The expression reverse(log(size(X),i)) is to permute the input vector X correctly for the FFT implementation with the given butterfly.

3 Parallelizing the Programs

A parallel machine consists of a set of *processing elements*, *PEs*, and *communication channels* connecting the PEs. Together this forms a directed graph, which may be defined as a dda. If the machine is a SIMD (Single Instruction, Multiple Data) architecture, we may take a *space-time view* of a parallel program executing on it: it performs computational steps interspersed with blocks of communications. The process at PE p at time-step t may send a message to the process at PE q at time-step $t + 1$ provided there is a communication channel from processor p to processor q. Each processor at every time-step thus becomes a node in a directed graph, and the communication channels between the PEs become directed arcs, from one time-step to the next. This also forms a directed graph, and this special variant of a dda will be called a *space-time algebra.* In Haveraaen [11] it is shown that this model also is useful for other parallel machines such as a MIMD (Multiple Instruction, Multiple Data).

Definition 3.1 [sta] A *space-time algebra* is a data dependency algebra with the following restrictions:

- the *carrier for the nodes of an sta* is (a subset of) the Cartesian product of the processing elements of a parallel machine and a time counter, usually the integers,
- the *carrier for the directions* are the channels going out from and leading into the processors, including a channel for *in-memory* communication, allowing a processor to retain data in memory between computations, and
- the r and s functions define allowable communications from one time-step to the next.

□

An *embedding* of a program's dda in a parallel machine's sta (see Miranker & Winkler [16]) is a mapping of *dda nodes to sta nodes*, and *dda arcs to sta paths.* In order to express the embedding in a convenient way, we will generate the transitive closure of a space-time algebra.

Definition 3.2 [tca] The *trancitive closure algebra* is a data dependency algebra generated from a space-time algebra in the following way

- the *carrier for the nodes of a tca* is the nodes of the sta,
- the *carrier for the directions* are sequences of the channels of the sta, and

```
signature 𝒮
    parameters
        sort  sta, channel
    parameters
        sort  int, table_{int,channel}
    defines
        operations
            canRec : sta * table_{int,channel} → bool;
            rec    : sta * table_{int,channel} | canRec → sta;
            dirSen : sta * table_{int,channel} | canRec → table_{int,channel};
            canSen : sta * table_{int,channel} → bool;
            sen    : sta * table_{int,channel} | canSen → sta;
            dirRec : sta * table_{int,channel} | canSen → table_{int,channel};
    end
```

Figure 8: Signature $\mathcal{S}$ for the transitive closure of a space-time algebra. The directions of this graph are defined by sequences of channels, a channel is a direction in the space-time algebra. The tca is a data dependency algebra. The names canRec and canSen correspond to canR and canS respectively, the function names rec and sen correspond to the r and s functions, and the function names dirSen and dirRec correspond to the dirS and dirR functions.

- the r and s functions define finite paths in the sta.

□

Theorem 3.3 (embedding) *The embedding of a dda in the space-time of a parallel machine may be expressed as an implementation of the dda in a sta's tca.*

Proof Using the sta nodes as carrier for the dda nodes will map the nodes of the dda to sta nodes. The carrier for the directions of the dda will be sequences of channels of the sta, so that arcs between nodes of the dda will be mapped onto paths in the sta by the code for the dda functions. □

We are now able to formulate and prove the main theorem of this paper.

Theorem 3.4 (parallel implementation) *The embedding of a dda in the space-time of a parallel machine defines the execution of a constructive recursive function* f *distributed in the space-time of the parallel machine.*

Proof The embedding assigns a space-time coordinate to the nodes p of the dda, so each call of the computational aspect f' of f at node p assigns a space-time coordinate to this computation. Moreover, each arc of the dda is mapped onto a path in the space-time of the parallel machine, hence all sends (as described in the proof of the imperative code theorem 2.3) will be on physical links in the parallel machine. □

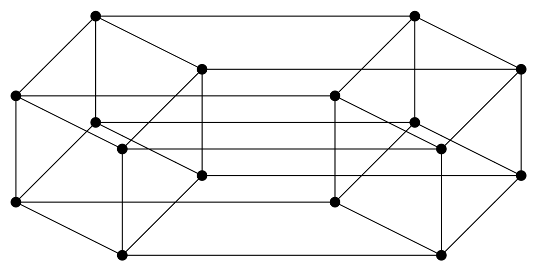


Figure 9: The hypercube connections drawn for a hypercube of dimension 4. Each PE has 4 channels numbered 1 through 4. Channels with the same number are drawn parallel to each other.

The theorem tells us that given a constructive recursive program, and an implementation of its data dependencies in the tca of a parallel machine, we may generate parallel, imperative code for this machine. Yet no part of the language has any reference to the concepts of process nor communication. The only notions we are working with are dependencies and implementations. Since we are allocating each data item on a different processor, this method of programming is best suited for programming massively parallel machines.

Our approach to parallel programming also allows us to validate the correctness of our implementations. The implementations of a dda for different parallel machines must all be isomorphic, and this may be validated whenever a new embedding is implemented.

A table distributed over a dda will become a distributed data structure on the parallel machine if the dda is embedded in the machine's sta. Thus elements of the table will be placed on specific processors at specific time-steps of the computation. This also distributes the computation of expressions like `SUM(readdress(FFT(DX)))` where `DX` is an appropriately distributed table. Since tca's are used when we implement a dda for a parallel machine, we will use the signature $\mathcal{S}$ defined in Figure 8 when referring to tca's.

The following subsections contain examples of embeddings. Each embedding allows both the `SUM` and the `FFT` functions to be executed distributed over the processors of a parallel machine. In many cases it will be possible to generate these embeddings automatically by applying the methods and tools developed for this purpose (see for instance Chen & al [5]). Once generated, these embeddings may be stored in a library of embeddings together with hand coded ones.

3.1 The Butterfly embedded in a Hypercube Computer

Many parallel computers, e.g., Intel's iPSC-2 or Thinking Machine's Connection Machine CM-2, have processors connected in a hypercube, or boolean n-cube, network. A hypercube

```
specification Hypercube
  parameters
    operations
      h : → int; –– dimension of hypercube
  defines
    sort  sta   –– the set of space-time indices {(s,t) ∈ PE × Z}
                –– the channel carrier is the set of directions {d∈Z | 0 ≤ d ≤ h}
    operations
      –– operations as in signature D
      canR  : sta * int → bool;
      r     : sta * int | canR → sta;
      dirS  : sta * int | canR → int;
      canS  : sta * int → bool;
      s     : sta * int | canS → sta;
      dirR  : sta * int | canS → int;
      –– operations specific for this space-time algebra
      time  : sta → int;              –– extract a logical time component
      space : sta → int;              –– extract a logical space component
      retime: sta → sta;              –– change the logical time component to 0
      tcut  : → table_{int,sta};      –– a time cut: all PEs at time-step 0
    equations
      –– the axioms of a dda must be satisfied. In addition we have:
                        canR(p,i) = 0<=i<=h;
                        canS(p,i) = 0<=i<=h;
                        dirS(p,i) = i
                        dirR(p,i) = i
                     time(r(p,d)) = time(p)-1;
                     time(s(p,d)) = time(p)+1;
                  time(retime(p)) = 0;
              time(lookup(tcut,i)) = 0;
         0 <= space(p) < 2**h = true
                    space(r(p,d)) = complement(space(p),d-1);
                    space(s(p,d)) = space(r(p,d));
                 space(retime(p)) = space(p);
             space(lookup(tcut,i)) = i-1;
  end
```

Figure 10: The specification Hypercube for the sta of a hypercube of dimension h, i.e., a machine with 2^h processors. Note that the algebra for the Hypercube is defined by the physical hardware, but the projection operations time and space will interact with the movement functions r and s in the way specified in the axioms.

```
implementation BbyH
  parameters
    operations
      h : → int; −− height of butterfly and dimension of hypercube
  parameters
     −− the tca of the hypercube and the special hypercube sta operations are parameters
  defines
     −− defines exactly the signature for the butterfly dda
  local
    operations
      ddinv : sta → bool;             −− datainvariant for the dda
      drinv : int → bool;             −− datainvariant for dir
      toch  : int * int * dir → int;  −− function to find the channel for a send:
  code
    ddinv(p)    = (0<=time(p)<=h);
    dda         = sta | ddinv; −− the “butterfly subset” of the sta nodes
    drinv(n)    = (0 <= n <= 1);
    dir         = int | drinv; −− the set {0, 1}
    left        = 0;
    right       = 1;
    toch(s,t,d) = if (d = digit(s,t)) → 0
                  | (d<>digit(s,t)) → t+1
                  fi; −− the time index t determines channel number
    canR(p, d)  = (0 < time(p)); −− there are no dda nodes below row 0
    canS(p, d)  = (time(p) < h); −− there are no dda nodes above row h
    r(p,d)      = rec(p,[toch(space(p),time(p)-1,d)]);
    s(p,d)      = sen(p,[toch(space(p),time(p),d)]);
    dirS(p, d)  = if d = digit(space(p),time(p)-1) → d
                  | d<>digit(space(p),time(p)-1) → 1-d
                  fi;
    dirR(p, d)  = if d = digit(space(p),time(p)) → d
                  | d<>digit(space(p),time(p)) → 1-d
                  fi;
    lower(p)    = retime(p);
    B           = evolving i : int from tcut in true → (i,lookup(tcut,i)) end;
  end
```

Figure 11: The code defining the butterfly embedded in the hypercube space-time, i.e., a definition of how to execute programs with butterfly data dependencies distributed on hypercubes. No programs based on butterfly (see Figure 2) will need to be modified when switching to this butterfly implementation.

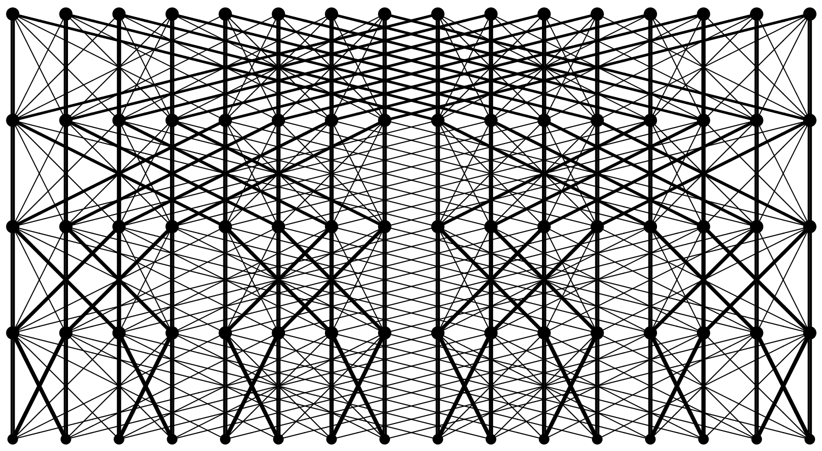


Figure 12: The hypercube space-time pattern drawn for a hypercube of dimension 4. The figure also shows an embedding of the butterfly of height 4 in the hypercube. All hypercube nodes are butterfly nodes (filled circles), and the hypercube paths that are butterfly arcs have been thickened.

of dimension n, has 2^n processors. Each processor has $n + 1$ communication channels (including the in-memory channel). Physical connections are numbered 1 through n, while the in-memory channel has number 0. Figure 9 gives a schematic sketch of a hypercube connected parallel machine.

The hypercube sta (see Figure 10) has some operations in addition to the basic dda functions. These include space and time projection functions, which allow us to form useful expressions when defining the embedding. The figure also provides some axioms specifying the new operations.

The embedding of the butterfly on the hypercube (in Figure 11, illustrated in Figure 12) maps each node at a level of the butterfly to a node in the hypercube at the same time-step. The different rows of the butterfly are simulated using time in the hypercube. The BbyH implementation is efficient as we have h time steps where h is the height of the butterfly, i.e., it is time optimal. Since every node of the hypercube at every time-step also is a node of the butterfly, this implementation is also space optimal.

3.2 The Butterfly embedded in a Linear Array

A machine like the MasPar MP-2 has its processors connected in a 2-dimensional mesh with circular connections from the start of every row (and column) to the end of the row (or column, respectively). MP-2 also has diagonal connections with a toroidal wrap. We will simplify this by defining a linear array (1-dimensional mesh, see Figure 13), and implement

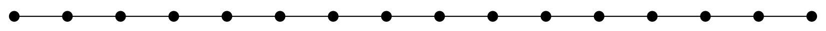


Figure 13: The linear array connections drawn for a linear array with 16 PEs. The direction W is towards the left, and the direction E towards the right.

the butterfly on it. The extension to two-dimensional meshes should be straight forward, although the technical details will be much more involved.

The specification Linear (14) has a sort channel with three distinguished constants naming the communication ports of the linear array (E, W and O), and the integer c standing for the number of processor nodes in the array. The communication port O names the communication of a datum by retaining it in memory. Linear has the same additional operations as Hypercube, however the axioms for the space index manipulations are very different, reflecting the difference in architecture.

In the embedding of the butterfly on the linear array (in Figure 15, illustrated in Figure 16), many of the sta nodes are not part of the embedding. This is because the diagonal connections in the butterfly has to pass through several nodes. This wastes computational resources, and is a drawback of the linear array when compared to the hypercube. We have to skip $2^{\mathsf{i}} - 1$ time steps when moving in the send direction of the graph, where i is the row number. This is encoded using the base 2 (truncating) log : int →int function in the data invariant ddinv. The time consumption for this embedding is exponential in h, but h is logarithmic in the size of the input vector. Hence, the SUM and FFT algorithms will execute in linear time with respect to the size of the input vector.

If the linear array had a different set of connections, we might have been able to implement the butterfly in a pattern resembling the perfect shuffle (shuffle exchange network) and thus reducing the time complexity. The space requirement will always be linear in terms of the input vector, as required by the definition of the butterfly.

4 Comparisons with other approaches

We will compare our approach to parallel programming to the functional approaches of Chen & al, Čyras & al, Trishina, and Tucker & Zucker, and the imperative approach of Bräunl. Comparing the constructive recursive approach to the Crystal approach (Chen & al [5]), we find that a Crystal function $\dot{f} : D \to t$ is expressed in terms of a specific model D for the dda. Instead of abstracting over this model as we do, they use the fact that there is an isomorphism $g : D \to E$ from the model D to the embedding E of D in the target machine's space-time algebra ST. The isomorphism is called a *reshape morphism*, since it reshapes the domain D into E. The reshape morphism is used to reformulate the function $\dot{f}$ into the function $\hat{\dot{f}} : \mathsf{ST} \to t$, defined by the equation $\hat{\dot{f}} = \dot{f} \circ g^{-1}$, where g^{-1} is the inverse

```
specification Linear
    parameters
        operations
            c : → int; -- number of processors
    defines
        sort  sta,       -- the set of space-time indices {(s,t)∈ PE × Z}
              channel    -- the channel is the set of directions {O, E, W}
        operations
            -- operations as in signature D:
            canR  : sta * channel → bool;
            r     : sta * channel | canR → sta;
            dirS  : sta * channel | canR → channel;
            canS  : sta * channel → bool;
            s     : sta * channel | canS → sta;
            dirR  : sta * channel | canS → channel;
            -- operations specific for this space-time algebra:
            O     : → channel;              -- direction origin
            E     : → channel;              -- direction east
            W     : → channel;              -- direction west
            time  : sta → int;              -- extract a logical time component
            space : sta → int;              -- extract a logical space component
            retime: sta → sta;              -- change the logical time component to 0
            tcut  : → table_{int,sta};      -- a time cut: all PEs at time-step 0
        equations
            -- the axioms of a dda must be satisfied. In addition we have:
                         canR(p,W) = space(p) > 0;
                         canR(p,O) = true;
                         canR(p,E) = space(p) < c-1;
                         canS(p,i) = canR(p,i);
                         dirS(p,W) = E
                         dirS(p,O) = O
                         dirS(p,E) = W
                         dirR(p,i) = dirS(p,i);
                      time(r(p,d)) = time(p)-1;
                      time(s(p,d)) = time(p)+1;
                   time(retime(p)) = 0;
              time(lookup(tcut,i)) = 0;
              0 <= space(p) < c = true
                    space(r(p,W)) = space(p)-1;
                    space(r(p,O)) = space(p);
                    space(r(p,E)) = space(p)+1;
                 space(retime(p)) = space(p);
            space(lookup(tcut,i)) = i-1;
    end
```

Figure 14: The specification Linear for the sta of a linear array with c processors. Note that the algebra for Linear is defined by the physical hardware, but the projection operations time and space will interact with the movement functions r and s in the way specified in the axioms.

```
implementation BbyL
  parameters
    operations
      h : → int; -- height of butterfly – we must have that c = 2**h
  parameters
     -- the tca of the linear array and the special linear array sta operations are parameters
  defines
     -- defines exactly the signature for the butterfly dda
  local
    operations
      ddinv : sta → bool;                         -- datainvariant for the dda
      drinv : int → bool                          -- datainvariant for dir
      rech  : int * int * dir → table_{int,channel}; -- the sequence of channels for a receive
      sech  : int * int * dir → table_{int,channel}  -- the sequence of channels for a send
  code
    ddinv(p)    = (0<=time(p)<=2**h) and (0<=space(p)<2**h)
                  and (2**log(time(p)+1)=time(p)+1); -- log is with base 2
    dda         = sta | ddinv; -- the butterfly subset of the sta nodes
    drinv(n)    = (0 <= n <= 1);
    dir         = int | drinv; -- the set {0, 1}
    left        = 0;
    right       = 1;
    rech(s,t,d) = if d=digit(s,log(t+1)-1) → replicate(O,(t+1)/2)
                  |  d<digit(s,log(t+1)-1) → replicate(E,(t+1)/2)
                  |  d>digit(s,log(t+1)-1) → replicate(W,(t+1)/2)
                  fi;
    sech(s,t,d) = if d=digit(s,log(t+1)) → replicate(O,t+1)
                  |  d<digit(s,log(t+1)) → replicate(W,t+1)
                  |  d>digit(s,log(t+1)) → replicate(E,t+1)
                  fi;
    canR(p, d)  = (0 < time(p)); -- there are no dda nodes below row 0
    canS(p, d)  = (time(p) < 2**h); -- there are no dda nodes above row 2^h
    r(p,d)      = rec(p,rech(space(p),time(p),d));
    s(p,d)      = sen(p,sech(space(p),time(p),d));
    dirS(p, d)  = if d = digit(space(p),log(time(p)+1)-1) → d
                  | d<>digit(space(p),log(time(p)+1)-1) → 1-d
                  fi;
    dirR(p, d)  = if d = digit(space(p),log(time(p)+1)) → d
                  | d<>digit(space(p),log(time(p)+1)) → 1-d
                  fi;
    lower(p)    = retime(p);
    B           = evolving i : int from cut in true → (i,lookup(cut,i)) end;
  end
```

Figure 15: The code defining the butterfly embedded in the linear array space-time, i.e., a definition of how to execute programs with butterfly data dependencies distributed on linear arrays. No programs based on butterfly (see Figure 2) will need to be modified when switching to this butterfly implementation.

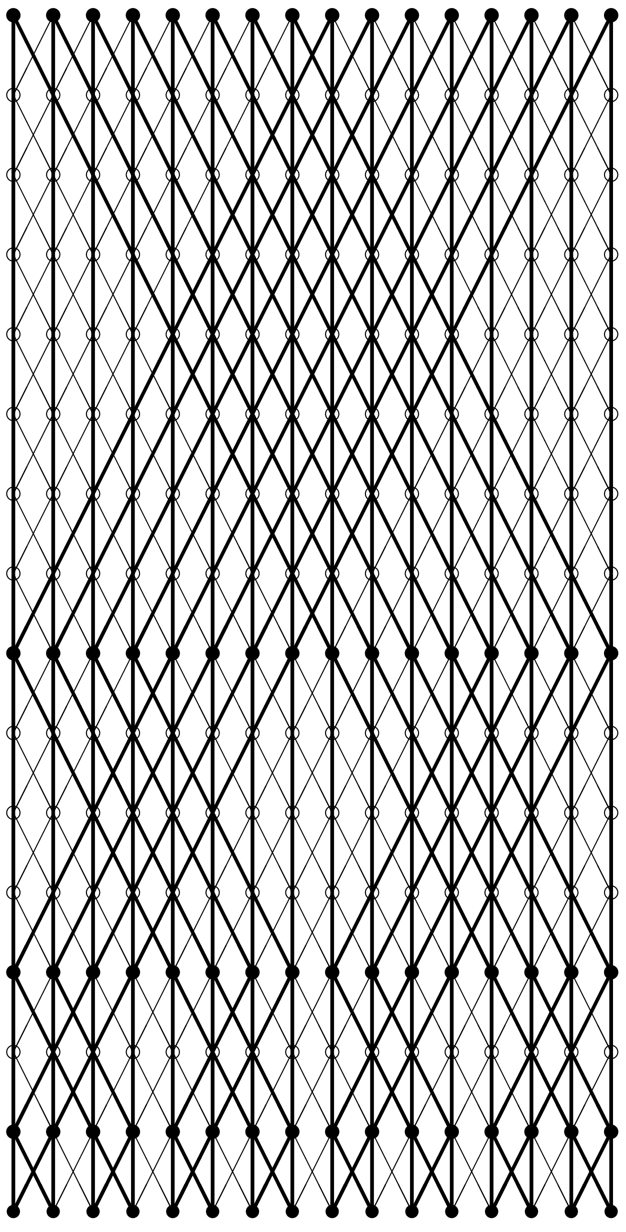


Figure 16: The linear array space-time pattern drawn for a linear array with 16 processors. The figure also shows an embedding of the butterfly of height 4 in the linear array. The linear array nodes that are butterfly nodes are filled, and the linear array paths that are butterfly arcs have been thickened.

function of g. The reshape morphisms g and g^{-1} may be programmer-supplied, but in many cases it is possible to automatically derive them. This is similar to the possibility of automatic generation of space-time embeddings of dda's in the Sapphire approach.

Similar results have been achieved by Čyras & al (see [7] or [10]). They restrict their attention to dda's that are regular grid structures, and do not define the s-functions, only the r-functions, in their code. The s-function is always derivable due to the simple structure of the base domains. A detailed comparison is presented in Haveraaen and Čyras [8]. Trishina [19] generates Occam programs from reccurrence equations over discrete lattice structures by similar automatic transformations.

Tucker & Zucker[20] have developed a general theory on the recursion over structures. Their parallel work, however, is oriented towards structures with space-time properties. This will restrict their work to the subclass of space-time algebras of our approach.

Yet another approach related to ours is that of Parallaxis (Bräunl [3]). This is an imperative programming language for SIMD machines based on Modula-2. In Parallaxis programs are essentially defined over dda's (virtual machine in their terminology) explicitly defined by the programmer. Since the concept is based on SIMD computing, the dda's are restricted to the sta subclass of data dependencies. The Parallaxis compiler is intended to automatically derive a mapping from the user-specified dda to the sta of the target machine.

5 Summary

We have shown that algorithms may be partitioned in a computational part and a data dependency part, and that these may be defined as separate pieces of program text. We have chosen to define the first as a constructive recursive function, the latter as a data dependency algebra. Examples show that the same constructive recursive function may be defined over different dda's, and that several functions may utilize the same dda. The first case with the sum function which works with both trees and butterflies, the other case with the sum and fft functions that both use the butterfly. More important is the fact that different definitions of one data dependency algebra is possible. A special implementation of a dda, in the form of a machine specific space-time algebra, distributes the computation on a parallel machine. This distribution is explicitly controlled by the programmer, yet completely eliminates specification of tasks and other low-level constructs. Since a dda, and hence its embedding on a particular machine, is reusable for many algorithms, this seems like a useful approach to the programming of parallel machines. In many cases these embeddings may be automatically derived, and stored in the form of a module for later use. This is in contrast with many other techniques for parallel functional programming (see Szymanski [18]) which have to derive this information at every compilation.

The space-time algebras intuitively capture the behavior of a SIMD machine, but we have shown (see Haveraaen [11]) that it also carries over to MIMD (multiple instruction multiple data) computers. Due to the fine granularity of the distribution of data and computations this approach seems ideal for massively parallel machines. By combining several logical processors in one physical processor, this approach may be useful for machines with coarser granularity.

We are currently investigating this approach by implementing examples in Sapphire. Andersson [2] has among other problems implemented several versions of LU decomposition with pivoting. We are also developing compilers for several parallel machine architectures.

Acknowledgements

Much of this work was conceived during the author's sabbatical at Yale University, New Haven, Connecticut, in 1989. Thanks especially to Marina Chen for inviting me and Young-il Choo for our fruitful discussions.

References

[1] *The Programming Language Ada. Reference Manual. American National Standards Institute, ANSI/MIL-STD-1815A-1983.IX.* Springer-Verlag, Lecture Notes in Computer Science 155, 1983.

[2] T. Andersson: *Safir, analyse og programmering av dataavhengigheter på parallelle arkitekturer.* Thesis, the Department of informatics, University of Bergen, 1993.

[3] T. Bräunl: Structured SIMD Programming in Parallaxis. *Journal on Structured Programming*, vol. 10, no.2, July 1989, pp. 121-132

[4] L. Cardelli and P. Wegner: On understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, Dec. 1985.

[5] M. Chen, Y. Choo and J. Li: Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991, pp. 255-308.

[6] P.M. Cohn: *Universal Algebra.* D. Reidel, Dordrecht, 1981. ISBN 90-277-1213-1.

[7] V. Čyras: Loop Program Synthesis Using Array Traversing Modules. In Haveraaen & Meldal (ed.): *Proc. of 4th Nordic Workshop on Program Correctness*, University of Bergen Informatics t.r. 78., 1993.

[8] V. Čyras & M. Haveraaen: Modular programming of recurrencies: a Comparison of Two Approaches. *Informatica*, 1995, Vol. 6 no. 4, pp. 397–444.

[9] A.J. Field and P.G. Harrison: *Functional Programming.* Addison Wesley, Wokingham UK, 1988. ISBN 0-201-19249-7.

[10] S.N.Greshnev, E.Z.Lyubimskii, V.A.Chiras. Synthesis of programs on data structures. *Programming and Computer Software*, Vol.11, No.5, 1985, pp.282-291. (translated from Programmirovanie, No.5, pp.44-54, September-October, 1985).

[11] M. Haveraaen: Comparing Some Approaches to Programming Distributed Memory Machines. *Proceedings of the 6th Distributed Memory Computing Conference*, DMCC6, IEEE 1991, pp. 222-227.

[12] M. Haveraaen: How to Create Parallel Programs without Knowing it. In Meldal & Haveraaen: *Proceedings of the 4th Nordic Workshop on Program Correctness*, Reports in Informatics no. 78, Department of Informatics, University of Bergen, Norway, March 1993, pp. 165-176.

[13] P. Hudak: Para-functional Programming in Haskell. In B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991, pp. 159-196.

[14] U. Manber: *Introduction to Algorithms - a Creative Approach.* Addison Wesley, Reading Massachusetts USA, 1989. ISBN 0-201-12037-2.

[15] D. Michie: “Memo” functions and machine learning. *Nature* vol 218, 1968, pp. 19-22.

[16] W.L. Miranker & A. Winkler: Spacetime Representations of Computational Structures. *Computing* vol 32, 1983, pp. 93-114.

[17] V. Sarkar: PTRAN: The IBM Parallel Translation System. In B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991, pp. 309-391.

[18] B.K. Szymanski (ed.): *Parallel Functional Languages and Compilers*, ACM Press, New York / Addison Wesley, Reading, Mass., 1991.

[19] E. Trishina: Development of Occam Programs by Stepwise Algebraic Transformations of Affine Recurrence Equations. In Haveraaen & Meldal (ed.): *Proc. of 4th Nordic Workshop on Program Correctness*, University of Bergen Informatics t.r. 78., 1993.

[20] J.V. Tucker & J.I. Zucker: *Program Correctness over Abstract Data Types, with Error-State Semantics*, North-Holland, Amsterdam, 1988.

[21] M. Wirsing: Algebraic Specification. In J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics*, Elsevier/MIT Press, Amsterdam 1990, pp. 675-788.