Yale University
Department of Computer Science

Synthesis of a Systolic Dirichlet Product Using Non-Linear Domain Contraction

Marina Chen and Young-il Choo

YALEU/DCS/TR-664
December 1988

This work has been supported in part by the Office of Naval Research under Contract N00014-86-K-0310 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

Synthesis of a Systolic Dirichlet Product Using Non-Linear Domain Contraction

Marina CHEN and Young-il CHOO

Department of Computer Science
Yale University
New Haven, CT 06520
chen-marina@yale.edu choo@yale.edu

A systolic array for computing the Dirichlet product is derived, illustrating our methodology for designing parallel programs. We show how a straightforward implementation leads to congested communication channels, and then define a non-linear domain contraction to "fold" the domain and obtain a solution. The methodology consists of stepwise transformations using domain morphisms and refinement. This example reveals the formal aspects of parallel program design that are automatable and the informal aspects which require insight.

1 Introduction

In this paper we derive a systolic array implementation for computing the Dirichlet product in Crystal. Through this example, which has an interesting non-linear communication pattern, we illustrate the program transformation techniques and present our philosophy of designing parallel programs.

The Dirichlet product (or convolution) [1] of two arithmetical functions b and c is the function a defined by

$$a(n) = \sum_{k|n} b(k)c(n/k).$$

Finding a systolic implementation of this function is particularly challenging since the indices in the terms of the sum are related in a non-linear manner. The existence of such a solution enlarges the problem domain known to be amenable to systolic implementations. For comparison, systolic implementations are known for a related, but simpler, Möbius function [11], and for the convolution sum, as used in signal processing (where we have $c(n-k)$ rather than $c(n/k)$) [7]. The first systolic solution we are aware of was by Quinton [10] using an unusual non-linear transformation.

In this paper, we use the design methodology presented in [2,5] to derive an entirely different solution for the Dirichlet product which we believe is much simpler than Quinton's. The two key notions used in the derivation are *fan-reduction refinement* and *domain morphism*. The insight for our solution came from our attempt to distribute two non-linearly related input sequences over a linear and discrete cartesian space. The key step in the program transformation is a non-linear domain contraction that changes the original domain over which the program is defined into two layers.

In Section 2 we introduce the Crystal language and model. In Section 3, the Dirichlet product is defined in Crystal and the standard fan-out reductions are applied leading to a design with communication congestion. In Section 4, we begin again using a non-linear domain contraction. This avoids the congestion and leads to the spacetime mapping. The target implementation can now be described in the this spacetime domain. Finally, we conclude with a few remarks on our view of parallel program design and programming environment.

2 The Crystal Language and Model

The programs in this paper are expressed in Crystal with mathematical symbols for certain operators which otherwise would be written as string tokens. Briefly, Crystal is a functional language where the lambda notation is used for expressing functions. This makes our program transformations algebraic and amenable to automation. Rather than formally defining the language, the key notions will be introduced through an example.

Let N denote the type of natural numbers. The equation

$$f = \lambda(x, y) : N \times N. \left\{ \begin{array}{l} x > 0 \rightarrow 2x + 3y \\ x \leq 0 \rightarrow y - x \end{array} \right\} : N$$

defines a function from pairs of numbers to numbers. The braces represent graphically the conditional expression. The type of the function is indicated by the typing of the arguments and the typing of the body of the function using the colon syntax. In this case, the function has type $[N \times N \rightarrow N]$. A Crystal program is a set of mutually recursive definitions.

The model of Crystal contains basic data types, functions, and the following special objects. An *index domain* is a set of indices representing locations. These indices can be thought of as the addresses of processors, or just abstract indices for accessing composite data structures. A *communication metric* indicates the cost of communication between indices. A simple example of an index domain is the *interval domain*, denoted $l \,..\, r$, which consists of all integers from l to r, inclusive, with the communication metric being the difference in values. The empty domain results if $l > r$.

Given two index domains A and B, the *product* domain, denoted $A \times B$, consists of indices of the form (a, b), a from A and b from B, with the *Manhattan* communication metric defined to be the sum of the communication metric in each component. Dually, the *sum domain*, denoted $A + B$, consists of indices of the form $\langle i, e \rangle$, such that e is in A if $i = 1$, and e is in B if $i = 2$. In general, let I be an index domain and $D(i)$ be an index domain for each i in I. The sum domain of all $D(i)$'s, denoted $\biguplus_{i:I} D(i)$, is an index domain whose elements are $\langle i, e \rangle$, such that the *component index* i is from I and the *element* e is in $D(i)$, for each i in I. When I and each $D(i)$ are interval domains, the function $g : \biguplus_{i:I} D(i) \rightarrow I \times N$ taking $\langle i, e \rangle$ to (i, e) is clearly an injection. We can use this injection to induce the Manhattan metric of $I \times N$ on $\biguplus_{i:I} D(i)$.

In order to use the sum domain construction without having to specify the component index explicitly, we introduce the accompanying notion of the coproduct (or sum) map. Let D_1 and D_2 be index domains, V a value domain, and $f_1 : D_1 \rightarrow V$ and $f_2 : D_2 \rightarrow V$ be a pair of functions. The *coproduct map* of f_1 and f_2, is a function

$$[f_1, f_2] : D_1 + D_2 \rightarrow E,$$

satisfying

$$[f_1, f_2]\langle 1, x \rangle = f_1 x \quad \text{and} \quad [f_1, f_2]\langle 2, x \rangle = f_2 x.$$

In words, it is a function equivalent to f_1 if the argument is from D_1 and equivalent to f_2 if the argument is from D_2.

The central notion in the Crystal model is that of a *data field*, a function from an index domain to some domain of values. The value can again be a data field for representing hierarchical data structures. A data field whose codomain is an index domain is called a *domain morphism*, or just morphism. A domain morphism whose codomain is a sum domain will be called a *domain contraction.* Domain morphisms play a crucial role in reshaping one data field into another and lies at the heart of our program transformation method. A domain morphism which has an inverse is called a *reshape morphism*, while one which is only injective, or one-to-one, is called a *refinement morphism.* Refinement morphisms are used, for example, in fan-in and fan-out reductions, since extra indices are used to make communication local.

3 Dirichlet Product in Crystal—First Attempt

A Crystal program for computing the Dirichlet product of b and c can be written

$$\begin{aligned} N &= 1 \mathinner{..} \infty, \\ D &= \lambda n : N.1 \mathinner{..} n, \\ a &= \lambda n : N. \sum_{k:D(n)} b(k) \times c(\overline{m}(n,k)), \end{aligned} \tag{1}$$

where

$$\overline{m} = \lambda(n,k) : N \times N. \left\{ \begin{array}{c} k|n \to n/k \\ \neg k|n \to \bot \end{array} \right\},$$

and we let $b(\bot) = 0$ and $c(\bot) = 0$.

This program is correct, but its parallel implementation can be highly inefficient. To obtain a systolic implementation we need to eliminate (1) non-local communication, (2) large fan-in, and (3) large fan-out communication patterns. In this first attempt, we use fan-in and fan-out reductions for b and c in the definition of a to eliminate these. For c, however, the number of communication through some index turns out to be a function of its location leading to unacceptable congestion.

3.1 Fan-out Reduction for b

In definition (1), an input $b(k)$, for each $k \in D(n)$, is needed in computing $a(n)$, for all n in N such that $k|n$. If we represent $n = kl$, then for each k in N, $b(k)$ is needed by all kl, where l ranges over N. So, the data field

$$b' = \lambda(k,l) : N \times N. \left\{ \begin{array}{l} l = 1 \to b(k) \\ l > 1 \to b'(k, l-1) \end{array} \right\} \tag{2}$$

distributes $b(k)$ with only local communication, i.e., the communication cost is bound by a constant (equal to 1 in the Manhattan metric). Unfortunately, the domain of b' does not match the domain of a. What we want is a data field b'' defined over the sum domain $U = \biguplus_{n:N} D(n)$, a disjoint union of $D(n)$'s.

First, define the domain morphism and its inverse

$$g = \lambda(k,l) : N \times N.\langle l \times k, k\rangle : U \qquad g^{-1} = \lambda\langle n,k\rangle : U_0.(k, n/k) : N \times N,$$

where $U_0 = g(N \times N)$, the image under g and a proper subset of U. Next, the desired data field b'' over U_0 can be derived from b' using the methodology developed in [5]. We want g'' such that the following diagram commutes:

$$\begin{tikzcd}
N \times N \arrow[r, "b'"] \arrow[d, shift right, "g"'] & V \\
U_0 \arrow[u, shift right, "g^{-1}"'] \arrow[ur, "b''"'] &
\end{tikzcd}$$

or, in equations, $b'' \circ g = b'$. The derivation of the recursive definition of b'' from that of b' is as follows:

1. Substitute $b'' \circ g$ for b' in 1:

$$b'' \circ g = \lambda(k,l) : N \times N. \begin{Bmatrix} l = 1 \rightarrow b(k) \\ l > 1 \rightarrow b'' \circ g(k, l-1) \end{Bmatrix}$$

2. Right compose each side with g^{-1} and simplify:

$$b'' = \left[\lambda(k,l) : N \times N. \begin{Bmatrix} l = 1 \rightarrow b(k) \\ l > 1 \rightarrow b'' \circ g(k, l-1) \end{Bmatrix}\right] \circ g^{-1}$$

3. η-abstract the right hand side with $\langle n,k \rangle : U_0$:

$$b'' = \lambda\langle n,k \rangle : U_0. \left[\lambda(k,l) : N \times N. \begin{Bmatrix} l = 1 \rightarrow b(k) \\ l > 1 \rightarrow b'' \circ g(k, l-1) \end{Bmatrix}\right](g^{-1}\langle n,k \rangle)$$

4. Unfold $g^{-1}\langle n,k \rangle$ and β-reduce the inner redex:

$$b'' = \lambda\langle n,k \rangle : U_0. \begin{Bmatrix} n/k = 1 \rightarrow b(k) \\ n/k > 1 \rightarrow b'' \circ g(k, n/k - 1) \end{Bmatrix}$$

5. Unfold composition and unfold g:

$$b'' = \lambda\langle n,k \rangle : U_0. \begin{Bmatrix} n = k \rightarrow b(k) \\ n > k \rightarrow b''\langle n-k, k \rangle \end{Bmatrix}$$

The occurrence of "$n - k$" on the right-hand-side indicates non-constant communication, so we define a new data field b^* that extends b'' to U and which only has unit communication:

$$b^* = \lambda\langle n,k \rangle : U. \begin{Bmatrix} n = k \rightarrow b(k) \\ n > k \rightarrow b^*\langle n-1, k \rangle \end{Bmatrix} \qquad (3)$$

3.2 Fan-out Reduction for c

Similarly, we need to distribute c. From the definition of a, for a particular j in N, $c(j)$ will be used in computing $a(n)$, for all pairs (n,k) in $N \times N$ such that $j = \overline{m}(n,k)$. To determine all such pairs (n,k) for a given j, let l be a parameter ranging from $-j+1$ to ∞. Since $j = \overline{m}(j^2 + l \times j, j+l)$, we are only interested in the pairs $(j^2 + l \times j, j+l)$, for all l. Hence, the data field

$$c' = \lambda\langle j,l\rangle : W. \left\{ \begin{array}{l} l = 0 \rightarrow c(j) \\ l > 0 \rightarrow c'\langle j, l-1\rangle \\ l < 0 \rightarrow c'\langle j, l+1\rangle \end{array} \right\}, \qquad (4)$$

where $W = \biguplus j : N.L(j)$ and $L(j) = -j+1 \mathinner{..} \infty$, distributes the input sequence c with only local communication.

The next step is to reshape the domain W of c' to the desired domain U. For this, define the domain morphism

$$h = \lambda\langle j,l\rangle : W.\langle j^2 + l \times j, j+l\rangle : U.$$

We can easily check that h is one-to-one, and its inverse is

$$h^{-1} = \lambda\langle n,k\rangle : U_1.\langle n/k, k - n/k\rangle,$$

where $U_1 = h(W)$ is a proper subset of U. Next, using the same derivation used for b'', derive data field c'' such that $c' = c'' \circ h$:

$$\begin{array}{rcl}
c'' \circ h & = & \lambda\langle j,l\rangle : W. \left\{ \begin{array}{l} l = 0 \rightarrow c(j) \\ l > 0 \rightarrow c'' \circ h\langle j, l-1\rangle \\ l < 0 \rightarrow c'' \circ h\langle j, l+1\rangle \end{array} \right\} \\
c'' \circ h \circ h^{-1} & = & \left[\lambda\langle j,l\rangle : W. \left\{ \begin{array}{l} l = 0 \rightarrow c(j) \\ l > 0 \rightarrow c'' \circ h\langle j, l-1\rangle \\ l < 0 \rightarrow c'' \circ h\langle j, l+1\rangle \end{array} \right\}\right] \circ h^{-1} \\
c'' & = & \lambda\langle n,k\rangle : U_1.\left[\lambda\langle j,l\rangle : W. \left\{ \begin{array}{l} l = 0 \rightarrow c(j) \\ l > 0 \rightarrow c'' \circ h\langle j, l-1\rangle \\ l < 0 \rightarrow c'' \circ h\langle j, l+1\rangle \end{array} \right\}\right](h^{-1}\langle n,k\rangle) \\
c'' & = & \lambda\langle n,k\rangle : U_1. \left\{ \begin{array}{l} k - n/k = 0 \rightarrow c(n/k) \\ k - n/k > 0 \rightarrow c'' \circ h\langle n/k, k - n/k - 1\rangle \\ k - n/k < 0 \rightarrow c'' \circ h\langle n/k, k - n/k + 1\rangle \end{array} \right\} \\
c'' & = & \lambda\langle n,k\rangle : U_1. \left\{ \begin{array}{l} k = \sqrt{n} \rightarrow c(k) \\ k > \sqrt{n} \rightarrow c''\langle n - n/k, k-1\rangle \\ k < \sqrt{n} \rightarrow c''\langle n + n/k, k+1\rangle \end{array} \right\}
\end{array}$$

The occurrence of "$n - n/k$" and "$n + n/k$" in the above definition indicates non-local communication. We need further refinement to eliminate them.

3.3 Eliminating Non-local Communication

Eliminating non-local communication means eliminating the occurrences of "$n - n/k$" and "$n + n/k$" in the definition of c'' and using only "$n - 1$" or "$k - 1$". After careful analysis of the communication pattern and some insight, we formulate suitable guards containing only the formal parameters, which, when executed, give the desired behavior.

First, define

$$U' = \biguplus_{n:N} (D(n) \cup \{ n+1 \}),$$

a slight extension of U. The new data field

$$c^* = \lambda\langle n,k\rangle : U. \left\{ \begin{array}{l} k = \sqrt{n} \rightarrow c(k) \\ k > \sqrt{n} \rightarrow \left\{ \begin{array}{l} (k-1)|n \wedge n \neq k(k-1) \rightarrow c^*\langle n, k-1\rangle \\ \neg(k-1)|n \vee n = k(k-1) \rightarrow c^*\langle n-1, k\rangle \end{array} \right\} \\ k < \sqrt{n} \rightarrow c^*\langle n + n/k, k+1\rangle \end{array} \right\} \tag{5}$$

has only local communication when $k > \sqrt{n}$. The extra index is used for transmitting values along the grid orthogonal paths.

This program uses two types of local communication (from $(n-1, k)$ to (n,k) and from $(n, k-1)$ to (n,k)) to approximate the non-local communication from $(n - n/k, k-1)$ to $(n-k)$. That the guards properly control the local communication is shown by the following:

Lemma 1 *For all n, k in the index domain N defined above, if $k > \sqrt{n}$ and $k|n$, then (1) $(k-1)|(n-n/k)$, (2) for all v such that $n - n/k < v < n$, $k > \sqrt{v}$ and $\neg(k-1)|v$, and (3) if $n \neq k(k-1)$, then $\neg(k-1)|n$.*

Proof: Let $l = n/k$.

(1) Since $n - l = l \times k - l = l \times (k-1)$, clearly $k-1$ divides $n - n/k$.

(2) It is obvious that $k > \sqrt{n}$ implies $k > \sqrt{v}$ for $n - n/k < v < n$. For any v in N, satisfying $n - n/k < v < n$,

$$\begin{aligned} v &= l \times k - u \\ &= l \times (k-1) + (l-u) \\ &= l \times (k-1) + r \end{aligned}$$

where $0 < u < l$ and $r = l - u$. Thus $0 < r < l$. Since $k > \sqrt{n}$ implies $l < k$, we have $0 < r < k-1$. Hence $k-1$ cannot divide v.

(3) If $n \neq k(k-1)$, then $l \neq k-1$. But $l < k$, thus l must be less than $k-1$, which implies $\neg(k-1)|n$. $\square$

We next show that the definition of c^* in (5) is correct.

Theorem 2 *The two data fields c^* and c'' defined above agree on all elements of U_1.*

Proof: From the definitions of a and $\overline{m}$, $c^*(n,k)$ will be called by $a(n)$ only when $k|n$. Assume $k > \sqrt{n}$, since this is the only branch that is different from c''.

From the body of $c^*(n,k)$, since $k|n$, either $\neg(k-1)|n$ or $n = k(k-1)$ by the Lemma. Hence a call to $c^*(n-1,k)$ is made.

Again by the Lemma, for all v, $n - n/k < v < n$, $k > \sqrt{v}$ and $\neg(k-1)|v$ will hold, so there will be altogether n/k calls to $c^*(v,k)$, for $n - n/k < v \leq n$. Finally, the condition

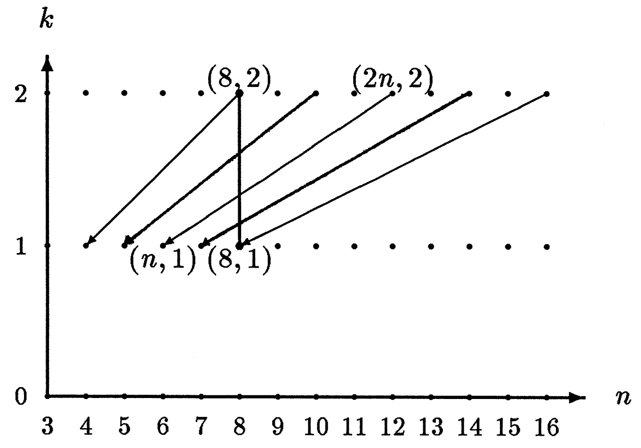

Figure 1: Communication congestion for $4 \le n \le 8$.

$(k-1)|n-n/k$ holds for the call $c^*(n-n/k,k)$, and $n-n/k=(n/k)(k-1)<k(k-1)$, because $k>\sqrt{n}$. Thus the first branch of the conditional will be taken and the subsequent call will be $c^*(n-n/k,k-1)$. Now let $n'=n-n/k$ and $k'=k-1$, we have $k'|n'$. Thus as long as $k'>\sqrt{n'}$, the same sequence of recursive calls will be made. Such sequences will terminate when the boundary condition $k'=\sqrt{n'}$ finally holds. The ratio of the amount decreased in the first component of the index pair to that of the second is always n/k, thus approximating the non-local communication in c''. □

3.4 Communication Congestion

Unfortunately, when $k<\sqrt{n}$, the part of U satisfying this condition becomes too congested for local communication to distribute c. To see this, consider the part $A=1\,..\,u\times 1\,..\,1$, for some fixed u. Without loss of generality, let u be an even number. From the definition of c^*, each value of c^* at $\langle n,1\rangle$ in A, is transmitted from c^* at $\langle 2n,2\rangle$ in U. Figure 1 illustrates the congestion for $4\le n\le 8$.

In the subdomain $B=u/2..u\times 1..2$, there are at least $u/2$ such in-coming values $c^*\langle 2n,2\rangle$, $u/2\le n\le u$. If all these $u/2$ communications are made local (and orthogonal to the axes), we would need at least

$$\sum_{i=1}^{u/2} i=\frac{1}{2}\cdot\frac{u}{2}(\frac{u}{2}+1)$$

distinct index elements for routing. But in this subdomain, there are only $2\cdot(u/2)$ index elements, which is roughly a factor of $u/8$ smaller than what is needed for routing. Thus even with multiple channels for communication for each index element, as long as the number of channels is constant or independent of the size of the domain U, there cannot be an implementation in which all communication is local.

4 Dirichlet Product Using Domain Contraction

The insight on how to reduce the communication congestion lies in the "symmetry" of k and n/k along the curve $k=\sqrt{n}$. Briefly, We associate the index $\langle n,k\rangle$ in U, with $k>\sqrt{n}$, with $\langle n,k'\rangle$, $k'<\sqrt{n}$, such that $k'=n/k$, when k divides n. This "contraction" or "folding" is defined in our methodology as a non-linear domain contraction.

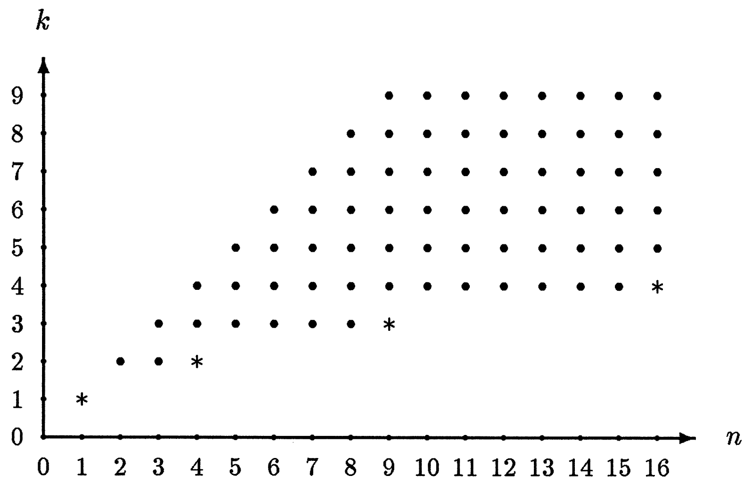

Figure 2: The domains M_1 and M_2 after contraction.

4.1 Non-Linear Domain Contraction

We need these index domains:

$$\begin{aligned}
U &= \biguplus_{n:N} D(n), \\
E &= \lambda n : N.p(n) \mathinner{..} n, \\
F &= \lambda n : N.\{\, \sqrt{n} \mid \text{square?}(n)\,\}, \\
M_1 &= \biguplus_{n:N} E(n), \\
M_2 &= \biguplus_{n:N} (E(n) \cup F(n)),
\end{aligned}$$

where

$$p = \lambda(n) : N.\left\{ \begin{array}{l} \text{square?}(n) \to \sqrt{n} + 1 \\ \neg\text{square?}(n) \to \lceil \sqrt{n} \rceil \end{array} \right\}.$$

Figure 2 illustrates the domain M_1 which consists only of the dots, and M_2 which also contains the asterisks. The domains are bounded on one side by p, which gives the parabolic shape.

The domain contraction that we use is

$$\begin{aligned}
g &= \lambda\langle n,k\rangle : U.\left\{ \begin{array}{l} k > \sqrt{n} \to \langle 1, \langle n,k\rangle\rangle \\ k \le \sqrt{n} \to \langle 2, \langle n, \overline{m}(n,k)\rangle\rangle \end{array} \right\} : M_1 + M_2, \\
g^{-1} &= [(\lambda\langle n,k\rangle : M_1.\langle n,k\rangle : U), (\lambda\langle n,k\rangle : M_2.\langle n, \overline{m}(n,k)\rangle : U)].
\end{aligned}$$

The idea is that g splits the original domain U into two domains, M_1 and M_2, which will be aligned as two layers over the processors with the communication cost between the corresponding elements of the two layers defined to be 0.

Next, rewrite a as a function of f:

$$\begin{aligned}
a &= \lambda n : N. \sum_{k:D(n)} f\langle n,k\rangle \\
f &= \lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k)) \qquad (6)
\end{aligned}$$

The goal is to derive the data field $\hat{f}$ from f that makes the following diagram commute:

$$\begin{tikzcd} U \arrow[r, "f"] \arrow[d, shift right, "g"'] & V \\ M_1 + M_2 \arrow[u, shift right, "g^{-1}"'] \arrow[ur, "\hat{f}"'] & \end{tikzcd}$$

or, in equations, $f = \hat{f} \circ g$ and $\hat{f} = f \circ g^{-1}$.

The derivation is similar to that of b'' and we begin with definition (6):

$$\begin{aligned}
\hat{f} \circ g &= \lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k)) \\
\hat{f} &= (\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k))) \circ g^{-1} \\
\hat{f} &= (\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k))) \circ \\
&\quad [(\lambda\langle n,k\rangle : M_1.\langle n,k\rangle : U), (\lambda\langle n,k\rangle : M_2.\langle n, \overline{m}(n,k)\rangle) : U)] \\
\hat{f} &= [(\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k))) \circ (\lambda\langle n,k\rangle : M_1.\langle n,k\rangle : U), \\
&\quad (\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k))) \circ (\lambda\langle n,k\rangle : M_2.\langle n, \overline{m}(n,k)\rangle) : U)] \\
\hat{f} &= [(\lambda\langle n,k\rangle : M_1.(\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k)))\langle n,k\rangle), \\
&\quad (\lambda\langle n,k\rangle : M_2.(\lambda\langle n,k\rangle : U.b(k) \times c(\overline{m}(n,k)))\langle n, \overline{m}(n,k)\rangle))] \\
\hat{f} &= [(\lambda\langle n,k\rangle : M_1.b(k) \times c(\overline{m}(n,k))), \\
&\quad (\lambda\langle n,k\rangle : M_2.b(\overline{m}(n,k)) \times c(\overline{m}(n, \overline{m}(n,k))))] \\
\hat{f} &= [(\lambda\langle n,k\rangle : M_1.b(k) \times c(\overline{m}(n,k))), (\lambda\langle n,k\rangle : M_2.b(\overline{m}(n,k)) \times c(n,k))]
\end{aligned}$$

We used the fact $\overline{m}(n, \overline{m}(n,k)) = k$ for $\langle n,k\rangle$ in M_2, to simplify at the last step.

Using the following functions

$$\begin{aligned}
\hat{b}_1 &= \lambda\langle n,k\rangle : M_1.b(k) & \hat{b}_2 &= \lambda\langle n,k\rangle : M_2.b(n/k) \\
\hat{c}_1 &= \lambda\langle n,k\rangle : M_1.c(n/k) & \hat{c}_2 &= \lambda\langle n,k\rangle : M_2.c(k)
\end{aligned}$$

we can rewrite $\hat{f}$ to:

$$\begin{aligned}
\hat{f} = [(\lambda\langle n,k\rangle : M_1.&\left\{\begin{array}{l} k|n \rightarrow \hat{b}_1\langle n,k\rangle \times \hat{c}_1\langle n,k\rangle \\ \neg k|n \rightarrow 0 \end{array}\right\}, \\
(\lambda\langle n,k\rangle : M_2.&\left\{\begin{array}{l} k|n \rightarrow \hat{b}_2\langle n,k\rangle \times \hat{c}_2\langle n,k\rangle \\ \neg k|n \rightarrow 0 \end{array}\right\}]
\end{aligned}$$

Note that $\hat{b}_i$ and $\hat{c}_i$ are defined without $\overline{m}$, and so the predicate $k|n$ needs to be used in the definition of $\hat{f}$, just as in the original definition of a. The reason is that $\hat{b}_i$ and $\hat{c}_i$ should not be called by a when $\neg k|n$.

Next, unfolding $\hat{f}$ in a and splitting the domain over which k ranges explicitly into three parts $E(n)$, $F(n)$, and the second copy of $E(n)$, we get

$$\begin{aligned}
\hat{a} = \lambda n : N.\Big(&\sum_{k:E(n)} \left\{\begin{array}{l} k|n \rightarrow \hat{b}_1\langle n,k\rangle \times \hat{c}_1\langle n,k\rangle \\ \neg k|n \rightarrow 0 \end{array}\right\} + \\
&\sum_{k:E(n)} \left\{\begin{array}{l} k|n \rightarrow \hat{b}_2\langle n,k\rangle \times \hat{c}_2\langle n,k\rangle \\ \neg k|n \rightarrow 0 \end{array}\right\} + \sum_{k:F(n)} \left\{\begin{array}{l} k|n \rightarrow \hat{b}_2\langle n,k\rangle \times \hat{c}_2\langle n,k\rangle \\ \neg k|n \rightarrow 0 \end{array}\right\}\Big) \qquad (7)
\end{aligned}$$

4.2 Fan-out Reduction

Just as in Section 3 the $b(k)$'s and $c(n/k)$'s must be distributed for computing $a(n)$ for different instances of $k|n$, they need to be distributed to $\hat{b}_i$ and $\hat{c}_i$. Note that the form in which $\hat{b}_1\langle n,k\rangle$ and $\hat{c}_2\langle n,k\rangle$ require $b(k)$ and $c(k)$ is similar to that for $a(n)$. Thus by using transformations similar to those in the derivation of b^* (Equation (3)), we derive the fan-out reduced versions $\check{b}_1$ and $\check{c}_2$ below. Also, $\hat{b}_2\langle n,k\rangle$ and $\hat{c}_1\langle n,k\rangle$ require $b(n/k)$ and $c(n/k)$ the way $a(n)$ requires $c(n/k)$, thus the transformations for deriving the fan-out reduced versions $\check{b}_2$ and $\check{c}_1$ below are analogous to those in the derivation of c^* (Equation (5)).

Like the domain U' (an extension of U) defined in Section 3, we need to define a new domain

$$M_2' = \biguplus_{n:N} (E(n) \cup \{\, n+1 \,\} \cup F(n)),$$

a slight extension of M_2 that includes an index used for the transmission of values. Fan-out reduction produces:

$$\check{b}_1 = \lambda\langle n,k\rangle : M_1 . \left\{ \begin{array}{l} k = n \rightarrow b(k) \\ k < n \rightarrow \check{b}_1\langle n-1,k\rangle \end{array} \right\}$$

$$\check{b}_2 = \lambda\langle n,k\rangle : M_2' . \left\{ \begin{array}{l} k = \sqrt{n} \rightarrow b(k) \\ k > \sqrt{n} \rightarrow \left\{ \begin{array}{r} (k-1)|n \wedge n \neq k(k-1) \rightarrow \check{b}_2\langle n,k-1\rangle \\ \neg(k-1)|n \vee n = k(k-1) \rightarrow \check{b}_2\langle n-1,k\rangle \end{array} \right\} \end{array} \right\}$$

where we have simplified $b(\overline{m}(n,k))$ to $b(k)$ when $k = \sqrt{n}$.

Note that b appears in both $\check{b}_1$ and $\check{b}_2$. This means that data must be routed from the outside of a systolic array for both cases.

Since we want to minimize such external communication as much as possible, we use the fact that for all k, $n > k > \sqrt{n}$, $\check{b}_1\langle n,k\rangle = b(k)$ and replace the occurrence of $b(k)$ in $\check{b}_2$ with it without changing its behavior:

$$\check{b}_2 = \lambda\langle n,k\rangle : M_2' . \left\{ \begin{array}{l} k = \sqrt{n} \rightarrow \left\{ \begin{array}{l} n = 1 \rightarrow b(1) \\ n > 1 \rightarrow \check{b}_1\langle n-1,k\rangle \end{array} \right\} \\ k > \sqrt{n} \rightarrow \left\{ \begin{array}{r} (k-1)|n \wedge n \neq k(k-1) \rightarrow \check{b}_2\langle n,k-1\rangle \\ \neg(k-1)|n \vee n = k(k-1) \rightarrow \check{b}_2\langle n-1,k\rangle \end{array} \right\} \end{array} \right\}$$

In the new definition of $\check{b}_2$, b occurs only when $n = k = 1$, which is the same condition under which it occurs in $\check{b}_1$. Hence no extra external communication is needed. Similarly,

$$\check{c}_1 = \lambda\langle n,k\rangle : M_2' . \left\{ \begin{array}{l} k = \sqrt{n} \rightarrow \left\{ \begin{array}{l} n = 1 \rightarrow c(1) \\ n > 1 \rightarrow \check{c}_2\langle n-1,k\rangle \end{array} \right\} \\ k > \sqrt{n} \rightarrow \left\{ \begin{array}{r} (k-1)|n \wedge n \neq k(k-1) \rightarrow \check{c}_1\langle n,k-1\rangle \\ \neg(k-1)|n \vee n = k(k-1) \rightarrow \check{c}_1\langle n-1,k\rangle \end{array} \right\} \end{array} \right\}$$

$$\check{c}_2 = \lambda\langle n,k\rangle : M_2 . \left\{ \begin{array}{l} k = n \rightarrow c(k) \\ k < n \rightarrow \check{c}_2\langle n-1,k\rangle \end{array} \right\}$$

Although $\check{c}_1$ is defined over M_2' (in order to make the communication local) it will only be called over the subdomain M_1. Figure 3 illustrates the flow of $\check{b}_2$ and $\check{c}_1$.

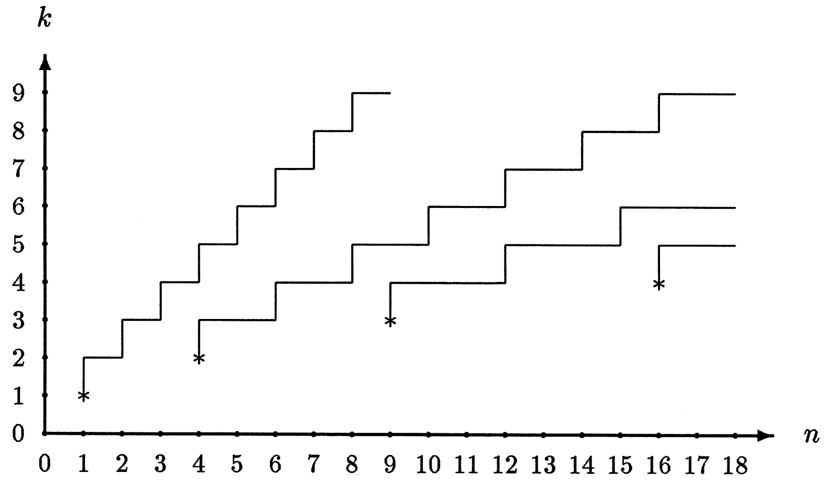

Figure 3: The flow of $\check{b}_2$ and $\check{c}_1$.

4.3 Fan-in Reduction

Next, we perform fan-in reduction on the definition of $\hat{a}$ which contains a summation over many arguments. The idea is to accumulate two pairs of product terms to the partial sum at each time step. Depending on the value of $\langle n,k\rangle$ there are four boundary cases (either k in $F(n)$ or $E(n)$, and either $k|n$ or $\neg k|n$) for initializing the partial sum. Since the case $k \in F(n)$ and $\neg k|n$ never holds, three boundary cases remain. The new data field $\check{a}$ defined below has fan-in degree of at most three:

$$\check{a} \;=\; \lambda\langle n,k\rangle : M_2.$$
$$\left\{\begin{array}{ll} k|n \rightarrow & \left\{\begin{array}{ll} k = \sqrt{n} & \rightarrow \check{b}_2\langle n,k\rangle \times \check{c}_2\langle n,k\rangle \\ k = \lceil\sqrt{n}\rceil & \rightarrow \check{b}_1\langle n,k\rangle \times \check{c}_1\langle n,k\rangle + \check{b}_2\langle n,k\rangle \times \check{c}_2\langle n,k\rangle \\ k > \lceil\sqrt{n}\rceil & \rightarrow \check{a}\langle n,k-1\rangle + \check{b}_1\langle n,k\rangle \times \check{c}_1\langle n,k\rangle + \check{b}_2\langle n,k\rangle \times \check{c}_2\langle n,k\rangle \end{array}\right\} \\ \neg k|n \rightarrow & \left\{\begin{array}{ll} k = \lceil\sqrt{n}\rceil & \rightarrow 0 \\ k > \lceil\sqrt{n}\rceil & \rightarrow \check{a}\langle n,k-1\rangle \end{array}\right\} \end{array}\right\}$$

and is related to the original data field by

$$a(n) = \check{a}\langle n,n\rangle.$$

4.4 Spacetime Mapping

We now apply a spacetime mapping (which is just another domain morphism) to transform a data field into a sequence of operations in an array over time. The technique for finding suitable linear functions for this purpose is well known, see for example [8,9,3]. The choice of the particular domain morphism g below is in order to generate a systolic array with "constant" response time, i.e., each element of the output sequence follows the corresponding element of the input sequence by an amount of time independent of n in N.

To derive the data fields over spacetime, we define two domains, the domain morphism between them, and a few subdomains:

$$X \;=\; -1\,..\,\infty,$$

$$\begin{aligned}
T &= 2\,..\,\infty, \\
h &= \lambda\langle n,k\rangle : M_2'.(n-k, n+k) : X \times T, \\
S_2' &= h(M_2'), \\
h^{-1} &= \lambda(x,t) : S_2'.\langle \tfrac{t+x}{2}, \tfrac{t-x}{2}\rangle : M_2', \\
S_1 &= h(M_1), \\
S_2 &= h(M_2).
\end{aligned}$$

For convenience, we identify the sum domain $\biguplus_{i:X} T$ with the Manhattan communication metric with the product domain $X \times T$. Next, define new data fields $\tilde{b}_i$, $\tilde{c}_i$, and $\tilde{a}$ over the subdomains S_1, S_2', and S_2 of $X \times T$, respectively.

For example, from $\check{b}_1$ we can derive $\tilde{b}_1$ in the same manner as before.

$$\begin{aligned}
\check{b}_1 &= \lambda\langle n,k\rangle : M_1.\left\{\begin{array}{l} k = n \to b(k) \\ k < n \to \check{b}_1\langle n-1,k\rangle \end{array}\right\} \\
\tilde{b}_1 \circ h &= \lambda\langle n,k\rangle : M_1.\left\{\begin{array}{l} k = n \to b(k) \\ k < n \to \tilde{b}_1 \circ h\langle n-1,k\rangle \end{array}\right\} \\
\tilde{b}_1 \circ h \circ h^{-1} &= [\lambda\langle n,k\rangle : M_1.\left\{\begin{array}{l} k = n \to b(k) \\ k < n \to \tilde{b}_1 \circ h\langle n-1,k\rangle \end{array}\right\}] \circ h^{-1} \\
\tilde{b}_1 &= \lambda\langle x,t\rangle : S_1.[\lambda\langle n,k\rangle : M_1.\left\{\begin{array}{l} k = n \to b(k) \\ k < n \to \tilde{b}_1 \circ h\langle n-1,k\rangle \end{array}\right\}](h^{-1}\langle x,t\rangle) \\
\tilde{b}_1 &= \lambda(x,t) : S_1.\left\{\begin{array}{l} \frac{t-x}{2} = \frac{t+x}{2} \to b(\frac{t-x}{2}) \\ \frac{t-x}{2} < \frac{t+x}{2} \to \tilde{b}_1 \circ h(\frac{t+x}{2} - 1, \frac{t-x}{2}) \end{array}\right\} \\
\tilde{b}_1 &= \lambda(x,t) : S_1.\left\{\begin{array}{l} x = 0 \to b(\frac{t}{2}) \\ x > 0 \to \tilde{b}_1(x-1,t-1) \end{array}\right\}
\end{aligned}$$

And for $\tilde{b}_2$ we have:

$$\tilde{b}_2 = \lambda(x,t) : S_2'.\left\{\begin{array}{l}
\frac{t-x}{2} = \sqrt{\frac{t+x}{2}} \to \left\{\begin{array}{l} \frac{t+x}{2} = 1 \to b(\frac{t-x}{2}) \\ \frac{t+x}{2} > 1 \to \tilde{b}_1(x-1,t-1) \end{array}\right\} \\
\frac{t-x}{2} > \sqrt{\frac{t+x}{2}} \to \\
\quad \left\{\begin{array}{l} (\frac{t-x}{2} - 1) | \frac{t+x}{2} \wedge \frac{t+x}{2} \neq \frac{t-x}{2}(\frac{t-x}{2} - 1) \to \tilde{b}_2(x+1,t-1) \\ \neg(\frac{t-x}{2} - 1) | \frac{t+x}{2} \vee \frac{t+x}{2} = \frac{t-x}{2}(\frac{t-x}{2} - 1) \to \tilde{b}_2(x-1,t-1) \end{array}\right\}
\end{array}\right\}$$

In the above, a call $\tilde{b}_2(x,t)$, with $x = 0$ and t even, will start the following sequence of calls: $\tilde{b}_2(-1,t-1)$, $\tilde{b}_2(0,t-2)$, $\tilde{b}_2(-1,t-3)$, $\tilde{b}_2(0,t-4)$, $\ldots$, $\tilde{b}_2(0,2)$. Since this sequence of calls does nothing except copy the same value, it can be simplified to $\tilde{b}_2(0,t-2)$, $\tilde{b}_2(0,t-4)$, $\ldots$, $\tilde{b}_2(0,2)$, in which the same value is copied. This modification does not change the value of any data field element in S_2, but it allows us to restrict $\tilde{b}_2$ to domain S_2, resulting in one less array element (since we don't need processor -1 in X) in the implementation. The final data fields are:

$$\tilde{b}_1 = \lambda(x,t) : S_1.\left\{\begin{array}{l} x = 0 \to b(\frac{t}{2}) \\ x > 0 \to \tilde{b}_1(x-1,t-1) \end{array}\right\}$$

$$\tilde{b}_2 = \lambda(x,t): S_2.$$
$$\left\{ \begin{array}{l} \frac{t-x}{2} = \sqrt{\frac{t+x}{2}} \rightarrow \left\{ \begin{array}{l} \frac{t+x}{2} = 1 \rightarrow b(\frac{t-x}{2}) \\ \frac{t+x}{2} > 1 \rightarrow \tilde{b}_1(x-1,t-1) \end{array} \right\} \\ \frac{t-x}{2} > \sqrt{\frac{t+x}{2}} \rightarrow \\ \quad \left\{ \begin{array}{l} x = 0 \rightarrow \tilde{b}_2(x,t-2) \\ x > 0 \rightarrow \left\{ \begin{array}{r} (\frac{t-x}{2}-1) | \frac{t+x}{2} \wedge \frac{t+x}{2} \neq \frac{t-x}{2}(\frac{t-x}{2}-1) \rightarrow \tilde{b}_2(x+1,t-1) \\ \neg(\frac{t-x}{2}-1) | \frac{t+x}{2} \vee \frac{t+x}{2} = \frac{t-x}{2}(\frac{t-x}{2}-1) \rightarrow \tilde{b}_2(x-1,t-1) \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

$$\tilde{c}_1 = \lambda(x,t): S_2.$$
$$\left\{ \begin{array}{l} \frac{t-x}{2} = \sqrt{\frac{t+x}{2}} \rightarrow \left\{ \begin{array}{l} \frac{t+x}{2} = 1 \rightarrow c(\frac{t-x}{2}) \\ \frac{t+x}{2} > 1 \rightarrow \tilde{c}_2(x-1,t-1) \end{array} \right\} \\ \frac{t-x}{2} > \sqrt{\frac{t+x}{2}} \rightarrow \\ \quad \left\{ \begin{array}{l} x = 0 \rightarrow \tilde{c}_1(x,t-2) \\ x > 0 \rightarrow \left\{ \begin{array}{r} (\frac{t-x}{2}-1) | \frac{t+x}{2} \wedge \frac{t+x}{2} \neq \frac{t-x}{2}(\frac{t-x}{2}-1) \rightarrow \tilde{c}_1(x+1,t-1) \\ \neg(\frac{t-x}{2}-1) | \frac{t+x}{2} \vee \frac{t+x}{2} = \frac{t-x}{2}(\frac{t-x}{2}-1) \rightarrow \tilde{c}_1(x-1,t-1) \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

$$\tilde{c}_2 = \lambda(x,t): S_1. \left\{ \begin{array}{l} x = 0 \rightarrow c(\frac{t}{2}) \\ x > 0 \rightarrow \tilde{c}_2(x-1,t-1) \end{array} \right\}$$

$$\tilde{a} = \lambda(x,t): S_2.$$
$$\left\{ \begin{array}{l} \quad \frac{t-x}{2} | \frac{t+x}{2} \rightarrow \\ \quad \left\{ \begin{array}{lll} \frac{t-x}{2} = \sqrt{\frac{t+x}{2}} & \rightarrow \tilde{b}_2(x,t) \times \tilde{c}_2(x,t) \\ \frac{t-x}{2} = \left\lceil \sqrt{\frac{t+x}{2}} \right\rceil & \rightarrow \tilde{b}_1(x,t) \times \tilde{c}_1(x,t) + \tilde{b}_2(x,t) \times \tilde{c}_2(x,t) \\ \frac{t-x}{2} > \left\lceil \sqrt{\frac{t+x}{2}} \right\rceil & \rightarrow \tilde{a}(x+1,t-1) + \\ & \quad \tilde{b}_1(x,t) \times \tilde{c}_1(x,t) + \tilde{b}_2(x,t) \times \tilde{c}_2(x,t) \end{array} \right\} \\ \neg \frac{t-x}{2} | \frac{t+x}{2} \rightarrow \\ \quad \left\{ \begin{array}{l} \frac{t-x}{2} = \left\lceil \sqrt{\frac{t+x}{2}} \right\rceil \rightarrow 0 \\ \frac{t-x}{2} > \left\lceil \sqrt{\frac{t+x}{2}} \right\rceil \rightarrow \tilde{a}(x+1,t-1) \end{array} \right\} \end{array} \right\}$$

And the output $a(n)$ is obtained at $\check{a}(n,n) = \tilde{a} \circ h(n,n) = \tilde{a}(0,2n)$.

A straightforward interpretation of the above definitions gives precise description of the operations of the systolic array. The domain S_2 (with a proper subset S_1) specifies the number of array elements and the time steps needed. Each processor stores its identity x, and keeps a counter for the time step t for computing all of the predicates in the guards. We can apply standard compiler techniques such as common sub-expression elimination to reduce a great many of the operations in computing the predicates.

Figure 4(a) indicates the flow of b and c for computing $a(12)$. The nodes $h(n,k)$ are the image under h of indices (n,k) in U. Each pair of values ($b(k)$ and $c(k)$) start out at processor 0. The first pair ($b(1)$ and $c(1)$) do not move. The next pair ($b(2)$ and $c(2)$) move across until they reach $h(4,2)$. From this point on they move at half speed. In general, the pair $b(k)$ and $c(k)$ begin to move at a slower rate at coordinate $h(k^2,k)$. This slower rate of propagating $b(k)$ and $c(k)$ is achieved by moving in the $-x$ direction for one unit and in the $+x$ direction for k units as shown in Figure 4(b).

Five registers R_a, R_{b1}, R_{b2}, R_{c1}, and R_{c2} are needed for each array element in order to

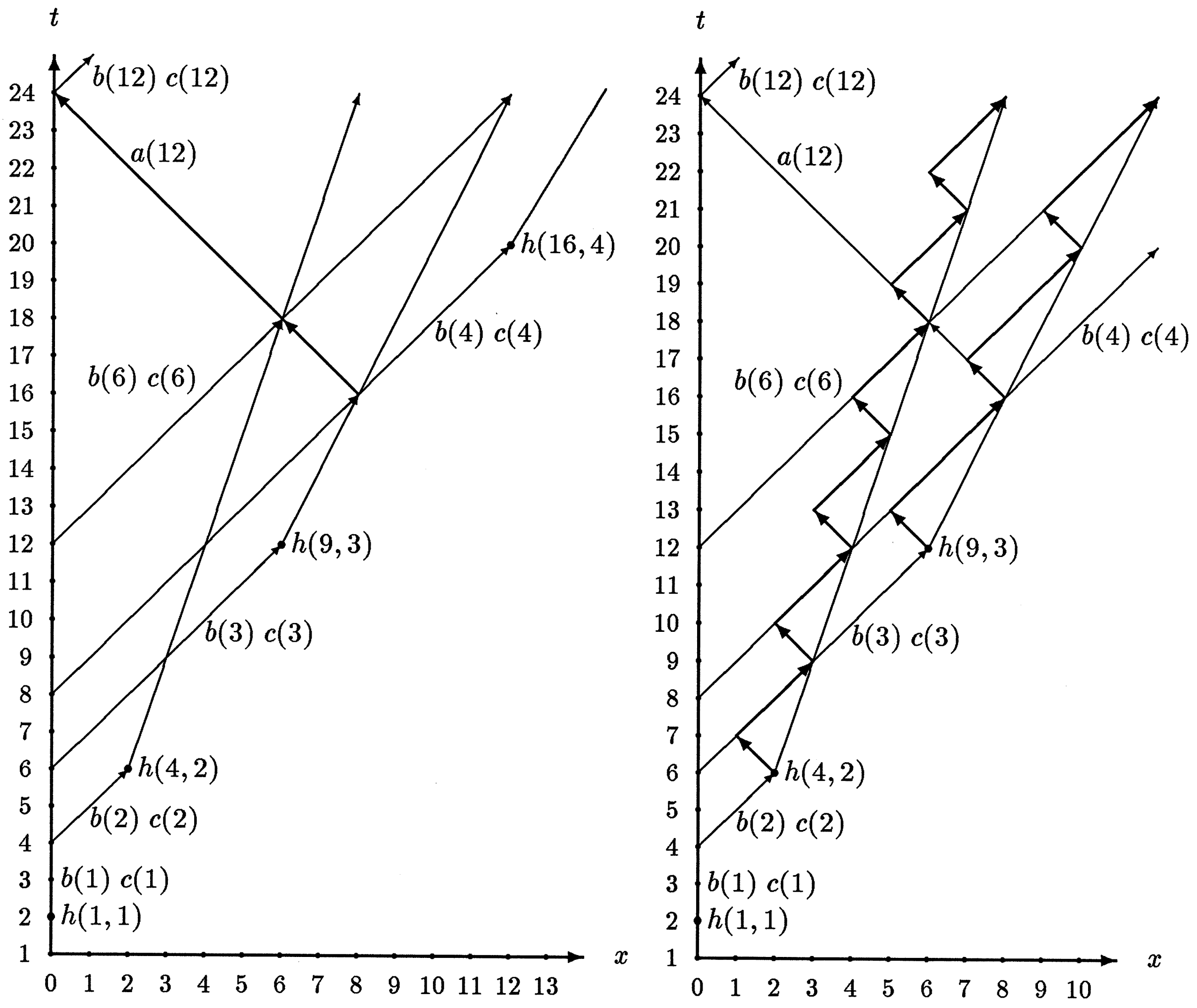

Figure 4: (a) The flow of b and c for computing $a(12)$. (b) Simulation of flows using unit communication.

distribute the data fields $\tilde{a}$, $\tilde{b}_1$, $\tilde{b}_2$, $\tilde{c}_1$, and $\tilde{c}_2$, respectively. In addition, processor 0 has an extra stage of shift register for each of $\tilde{b}_2$ and $\tilde{c}_1$ due to the elimination of processor -1. Figure 4a illustrates the data flow of these registers where x increases towards the right. Flows of R_{b1} and R_{c2} are to the right, and R_a to the left. Flows of R_{b2} and R_{c1} can be in either direction and varies with time as shown in Figure 4(b).

Registers R_{b1} and R_{c2} load their values from external inputs b and c, respectively, in processor 0 at every even time step. Registers R_{b2} and R_{c1} loads their values from external inputs b and c respectively in processor 0 at time step 2, and from registers R_{b1} and R_{c2} respectively in processor x at time step t when $\frac{t-x}{2} = \sqrt{\frac{t+x}{2}}$ and $\frac{t+x}{2} > 1$.

Each element $a(n)$ of the output sequence is obtained at processor 0 at time step $2n$. Note that the output $a(n)$ is obtained at the same time step $2n$ as inputs $b(n)$ and $c(n)$ become available. The delay is the time needed to perform the operations to add the pair of products $b(1) \times c(n)$ and $b(n) \times c(1)$ to the existing partial product.

5 Concluding Remarks

The derivation of the systolic array for the Dirichlet product began from a simple definition through a sequence of program transformations each determined by a domain morphism.

The theory of Crystal consists of the parallel computation model as a collection of data fields defined over index domains with communication metrics, the language constructs which allow algebraic manipulation, and the concepts of domain morphisms and refinements of a data fields, and is applicable to any target implementation. For a particular target implementation, the index domain and communication metric need to be specialized to reflect its network topology, processor power, communication latency, memory size, etc. For systolic implementation, we need special refinement and domain morphisms to obtain index domains which are subdomains of cartesian products of interval domains with the Manhattan metric, and a communication pattern that is local with constant fan-in and fan-out.

The design process has both formal and informal aspects. The formal aspect, such as the program transformations, is mechanizable. For example, once a domain morphism is specified, the derivation of the new data field from the original definition is automatable. Note that there are no restrictions on the shape of the domain nor on the domain morphism itself, as long as it has an inverse.

The informal aspect requires insight into the behavior of the algorithm, sometimes even a lemma or two. For example, finding the appropriate refinement of a data field so as to achieve constant fan degrees and local communications is often an art. Similarly, except for very restricted classes of problems, determining which domain morphism is needed to achieve a given objective requires insight. The non-linear domain contraction is an example where finding the right lemma to prove is critical.

One may ask how much design activity can be formalized and automated. By severely restricting the types of programs, we could automate the process of finding domain morphisms and refinements. For example, for programs falling into a restricted syntactic class, the so called "uniform recurrent equations" [6], there are algorithms for computing the appropriate domain morphisms and spacetime mappings. Real life applications, however, demand more. Short of being able to prove general theorems automatically, the next best thing is to provide a language and programming environment in which the insight of the programmer can be expressed and implemented. For example, the specification of domain morphisms to allow new data fields to be automatically derived, and check the consistency of new data fields obtained by refinement. The objective of the Crystal metalanguage [5] is to provide such capabilities.

The concept of reshaping the index domain of data fields proves extremely useful. In this paper, we have used it throughout the derivation: in the fan-in and fan-out reductions, in the contraction of the domain, and in the spacetime mapping. Ultimately, a parallel computation must be carried out by a finite number of processors. Mapping large computation onto a smaller machine is no more than reshaping the index domain (see [4] for an example). Different strategies such as partitioning a computation into blocks and assigning each block to a processor or scattering the computation over processors connected as a toroid are just examples of different domain morphisms.

Acknowledgment

Our thanks to Tom Verhoeff for raising the question of a systolic implementation for the Dirichlet product and getting us interested in this gem of a non-linear function.

References

[1] Tom M. Apostle. *Introduction to Analytic Number Theory.* Springer-Verlag, 1976.

[2] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[3] M. C. Chen. Synthesizing systolic designs. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 209–215, May 1985.

[4] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1988.

[5] Young-il Choo and Marina Chen. *A Theory of Parallel-Program Optimization.* Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[6] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[7] H.T. Kung. Why systolic architecture? *IEEE Computer*, 37–46, January 1982.

[8] Dan I. Moldovan. On the design of algorithms for VLSI systolic arrays. In *IEEE Transaction on Computer*, 1983.

[9] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of 11th Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[10] Patrice Quinton. *A Systolic Algorithm for the Convolution of Arithmetic Functions.* Technical Report, IRISA, Campus de Beaulieu, 1988.

[11] Tom Verhoeff and Martin Rem. Derivation of a systolic program for Möbius function. Presented at La Jolla Workshop on Concurrency, February 1988.