Processor Lower Bounds for Array Computations with Linear Schedules

Peter Cappello and Ömer Eğecioğlu
Department of Computer Science
University of California at Santa Barbara
$\{omer, cappello\}$@$cs.ucsb.edu$

Abstract

Using a directed acyclic graph (dag) model of algorithms, this paper treats a problem related to precedence-constrained multiprocessor schedules for array computations: Given a dag and a valid linear schedule for it, compute a lower bound on the number of processors required by the schedule. This is an important practical problem: A good schedule uses time efficiently; a good mapping of nodes to processors uses energy efficiently. The lower bounds obtained are good; we know of no case where they are not exactly tight. Actually, a harder problem is solved: Given a parametrized family of dags and a correspondingly parametrized valid linear schedule, symbolically generate a processor lower bound **formula**. To visualize the method for computing this formula, the nodes, representing computational tasks, are viewed as a set of lattice points in a convex polyhedron. The number of tasks that are scheduled for execution during any given time step of a linear schedule is the number of non-negative integer solutions to a set of linear Diophantine equations parametrized by the number of nodes of the dag. An algorithm, based on construction of generating functions, is presented for computing these numbers. This is the first such algorithm to be reported. Using this algorithm and a symbolic algebra package, formulas for processor lower bounds are obtained automatically.

The algorithm has been implemented as a Mathematica program. Example runs for Matrix-Vector Product, and Triangular Matrix Product problems are given. While widely applicable and elegant, it has worst case exponential running time. This is not surprising however; the simpler computation: "Are *any* processors scheduled for a particular time step?", which is equivalent to "Is a particular coefficient of the generating function nonzero?" is already known to be an NP-complete problem.

Keywords: Parallel processing, array computation, lower bound, linear schedule, Diophantine equation, generating function.

1 INTRODUCTION

We consider array computations, often referred to as systems of uniform recurrence equations [9]. Parallel execution of uniform recurrence equations has been studied extensively, from at least as far back as 1966 (see, e.g., [8, 11, 10, 1, 17, 2, 3, 18, 19, 4]). In such computations, the tasks to be computed are viewed as the nodes of a directed acyclic graph, where the data dependencies are represented as arcs. Given a dag $G = (N, A)$, a multiprocessor schedule assigns node v for processing during step $\tau(v)$ on processor $\pi(v)$. A valid multiprocessor schedule is subject to two constraints:

Causality: A node can be computed only when its children have been computed at previous steps:

$$(u, v) \in A \Rightarrow \tau(u) < \tau(v). \tag{1}$$

Non-conflict: A processor cannot compute 2 different nodes during the same time step:

$$\tau(v) = \tau(u) \Rightarrow \pi(v) \neq \pi(u). \tag{2}$$

In what follows, we refer to valid schedules simply as schedules. A schedule is good, if it uses time efficiently; an implementation of a schedule is good, if it used energy efficiently (e.g., uses few processors). Given a parametrized dag family and a correspondingly parametrized linear schedule, we compute a formula for the lower bound on the number of processors required by the schedule. It is a good lower bound; we know of no dag treatable by this method for which the lower bounds are not also upper bounds. This is the first reported algorithm for automatically generating such formulae.

The nodes of the dag typically can be viewed as lattice points in a convex polyhedron. This interpretation is motivated below by two examples.

EXAMPLE 1: $n \times n$ Matrix-Vector Product

An algorithm for $n \times n$ matrix-vector product is given in the following procedure, written in a Pascal-like notation. A is the input matrix, x is the input vector, and $y = A \cdot x$ is the output vector. We index the entries of an n-dimensional vector v by $v[0], v[1], \ldots, v[n-1]$.

```
for i = 0 to n − 1 do:
    y[i] ← 0;
    for j = 0 to n − 1 do:
        y[i] ← y[i] + A[i, j] · x[j];
    endfor;
endfor;
```

Computation is "located" at certain index pairs defined by the *for* loop limits, namely all pairs (i, j) satisfying:

$$\begin{array}{ccccc} 0 & \leq & i & \leq & n-1 \\ 0 & \leq & j & \leq & n-1 \end{array} \tag{3}$$

Clearly, these pairs (i, j) are the lattice points inside the 2-dimensional convex polyhedron whose four faces are defined by the four inequalities above. The faces of the polyhedron are, in turn, constructed from the *for* loop limits. This geometric interpretation of the node set leads to a combinatorial interpretation: solutions to a set of linear Diophantine equations that we describe below. We henceforth are concerned with only *non-negative* integral solutions to Diophantine equations. In this way, the inequalities $0 \leq i$, and $0 \leq j$ are implied, and need not be specified. In order to transform the set of inequalities in (3) to a set of *equations*, we introduce integral slack variables $s_1, s_2 \geq 0$ and write

$$\begin{array}{cccccc} i & & + s_1 & & = & n-1 \\ & j & & + s_2 & = & n-1 \end{array}$$

Now the standard array computation for $n \times n$ matrix-vector product is given by $G_n = (N, A)$, where

- $N = \{(i, j) \mid 0 \leq i, j \leq n-1\}$.
- $A = \{[(i, j), (i', j')] \mid (i, j) \in N,\ (i', j') \in N$ and either $i' = i + 1$, and $j' = j$; or $j' = j + 1$, and $i' = i$.

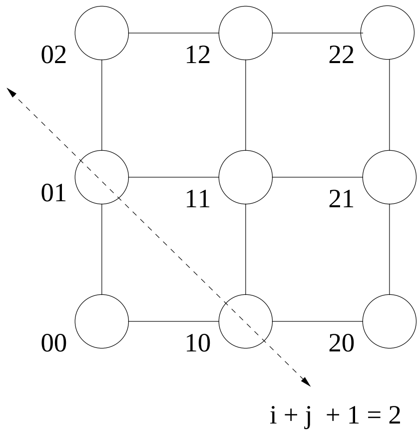

Figure 1: The matrix-vector product dag.

One linear multiprocessor schedule for G_n is to execute node $N(i,j)$ at time $i+j+1$. For the $n \times n$ case, the computation would begin in time step 1 with the computation of $N(0,0)$, and end in time step $2n-1$ with the computation of $N(n-1, n-1)$. At time step τ, all nodes $N(i,j)$, where $i+j+1 = \tau$ are scheduled for execution (See Figure 1). At time step $\tau = n$, there are n nodes scheduled for execution: $N(0, n-1), N(1, n-2), \ldots, N(n-1, 0)$.

Note that if we include the linear schedule $i+j+1 = \tau$ in the set of Diophantine equations describing the loop index ranges, then number of non-negative solutions to the augmented system of linear Diophantine equations is the number of tasks scheduled for execution during time step τ. Thus for any particular τ with $1 \le \tau \le 2n-1$, the number of solutions to the resulting linear Diophantine system is a lower bound on the number of processors necessary for the schedule.

As an example, for $\tau = n$, the augmented system obtained from (3) is

$$\begin{array}{ccccl} i & +j & & & = n-1 \\ i & & +s_1 & & = n-1 \\ & j & & +s_2 & = n-1 \end{array} \tag{4}$$

The number of non-negative integral solutions to (4) is a processor lower bound for the $n \times n$ Matrix-Vector Product problem. Time step $\tau = n$ requires the maximum number of nodes that must be computed concurrently, as τ ranges from 1 to $2n-1$. Thus, to realize this schedule, n processors are necessary.

EXAMPLE 2: $n \times n$ Triangular Matrix Product

An algorithm for matrix product $C = A \cdot B$, where A and B are given $n \times n$ upper triangular matrices is given below. The main body of this algorithm is taken from [7].

for $i = 0$ to $n-1$ do:
  for $j = i$ to $n-1$ do:
    for $k = i$ to j do:
      $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$;
    endfor;
  endfor;
endfor;

The computational nodes are defined by non-negative integral triplets (i,j,k) satisfying

$$i \le n-1$$

$$\begin{array}{ccccc} i & \le & j & \le & n-1 \\ i & \le & k & \le & j\ . \end{array}$$

However, fewer than 5 constraints are needed to define this polyhedron. The first inequality above is a consequence of the two on the next line, for example. In fact, the whole polyhedron is defined by the inequalities

$$i \ \le \ k \ \le \ j \ \le \ n-1.$$

Introducing integral slack variables $s_1, s_2, s_3 \ge 0$, we obtain the equivalent linear Diophantine system

$$\begin{array}{rrrrrrl} & j & & +s_1 & & & = n-1 \\ & -j & +k & & +s_2 & & = 0 \\ i & & -k & & & +s_3 & = 0 \end{array}$$

It can be shown that a linear schedule for the corresponding dag is given by $\tau(i,j,k) = i+j+k+1$. The halfway point in this schedule is time step $\tau = \frac{3}{2}n - 1$. When n is an even number, say $n = 2N$, then this value of τ is $3N-1$. Adding the schedule constraint to system we already have, we obtain the augmented Diophantine system

$$\begin{array}{rrrrrrl} i & +j & +k & & & & = 3N-2 \\ & j & & +s_1 & & & = 2N-1 \\ & -j & +k & & +s_2 & & = 0 \\ i & & -k & & & +s_3 & = 0 \end{array} \tag{5}$$

If $n = 2N+1$ is an odd number, then the augmented system becomes

$$\begin{array}{rrrrrrl} i & +j & +k & & & & = 3N-1 \\ & j & & +s_1 & & & = 2N-1 \\ & -j & +k & & +s_2 & & = 0 \\ i & & -k & & & +s_3 & = 0 \end{array} \tag{6}$$

Therefore, a lower bound for the number of processors needed for the $n \times n$ Triangular Matrix Product problem is the number of solutions of (5) if $n = 2N$, and the number of solutions of (6) if $n = 2N+1$.

In the examples above, the final problem to be solved is the determination of the number of solutions d_n to a linear Diophantine system of the form $\mathbf{a} \cdot \mathbf{z} = n\mathbf{b} + \mathbf{c}$ where $\mathbf{a}$ is some $r \times s$ integral matrix, $\mathbf{b}$ and $\mathbf{c}$ are r-dimensional integral vectors.

2 THE GENERAL FORMULATION

We now generalize these two examples and consider the problem of computing a lower bound for the number of processors needed to satisfy a given linear schedule.

> *That is, we show how to automatically construct a formula for the number of lattice points inside a linearly parameterized family of convex polyhedra by automatically constructing a formula for the number of solutions to the corresponding linearly parameterized system of linear Diophantine equations.*

The algorithm for doing this is our principal contribution. No such algorithm has ever been reported.

Our use of linear Diophantine equations, we believe, is well-motivated: the computations of an inner loop are typically defined over a set of indices that can be described as the lattice points in a convex polyhedron. Indeed, in two languages, SDEF [5] and ALPHA [24], one expressly defines domains of computation as the integer points contained in some programmer-specified convex polyhedron.

The general setting exemplified by the Matrix-Vector Product and the Triangular Matrix Multiply problems we considered above is as follows: Suppose $\mathbf{a}$ is an $r \times s$ integral matrix, and $\mathbf{b}$ and $\mathbf{c}$ are r-dimensional integral vectors. Suppose further that, for every $n \geq 0$, the linear Diophantine system $\mathbf{a} \cdot \mathbf{z} = n\mathbf{b} + \mathbf{c}$, i.e.

$$\begin{array}{ccccccccc} a_{11}z_1 & + & a_{12}z_2 & + & \ldots & + & a_{1s}z_s & = & b_1 n + c_1 \\ a_{21}z_1 & + & a_{22}z_2 & + & \ldots & + & a_{2s}z_s & = & b_2 n + c_2 \\ \vdots & & & & \vdots & & \vdots & = & \vdots \\ a_{r1}z_1 & + & a_{r2}z_2 & + & \ldots & + & a_{rs}z_s & = & b_r n + c_r \end{array} \tag{7}$$

in the non-negative integral variables $z_1, z_2, \ldots, z_s$ has a finite number of solutions. Let d_n denote the number of solutions for n. The generating function of the sequence d_n is $f(t) = \sum_{n \geq 0} d_n t^n$. For a linear Diophantine system of the form (7), $f(t)$ is always a rational function, and we provide an algorithm to compute $f(t)$ symbolically. The Mathematica program implementing the algorithm also constructs a formula for the numbers d_n from this generating function.

Given a nested *for* loop, the procedure to follow is informally as follows:

1. Write down the node space as a system of linear inequalities. That is, the loop bounds must be affine functions of the loop indices. The domain of computation is represented by the set of lattice points inside the convex polyhedron, described by this system of linear inequalities.

2. Eliminate unnecessary constraints by translating the loop indices (so that $0 \leq i \leq n-1$ as opposed to $1 \leq i \leq n$, for example). The reason for this is that the inequality $0 \leq i$ is implicit, whereas $1 \leq i$ introduces an additional constraint.

3. Transform the system of inequalities to a system of equalities by introducing slack variables, one for each inequality.

4. Augment the system with a linear schedule $\tau = \tau(n)$;

5. Run the program `DiophantineGF.m` to construct the generating function, $f(t) = \sum d_n t^n$, that enumerates the number d_n of solutions to the resulting linear system of Diophantine equations.

3 MATHEMATICA RUNS

Once the Mathematica program `DiophantineGF.m` we have written for this computation[1] has been loaded by the command `<< DiophantineGF.m`, the user may request examples and help in its usage. The program essentially requires three arguments $\mathbf{a}, \mathbf{b}, \mathbf{c}$ of the Diophantine system

$$\mathbf{az} = n\mathbf{b} + \mathbf{c} \ . \tag{8}$$

The main computation is performed by executing the command `DiophantineGF[a,b,c]`. The output is the (rational) generating function $f(t) = \sum_{n \geq 0} d_n t^n$, where d_n is the number of solutions $\mathbf{z} \geq \mathbf{0}$ to (8). After the computation of $f(t)$ by the program, the user can execute the command `formula`, which produces formulas for d_n, and the command `formulaN[c]` which evaluates the formula for d_n for $n = c$. If needed, the generating function $f(t)$ computed by the program subsequently can be manipulated by various Mathematica commands, such as `Series[]`.

[1] http://www.cs.ucsb.edu/~omer/personal/abstracts/DiophantineGF.m

Below, we provide sample runs of `DiophantineGF.m`. For the linear schedule of the Matrix-vector product example, the augmented Diophantine system in (4) can be written in the form (8) where

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}. \tag{9}$$

3.1 Sample Run for the $n \times n$ Matrix-Vector Multiplication Example:

```
In[1]:= << DiophantineGF.m
In[2]:= a = {{1, 1, 0, 0},
            {1, 0, 1, 0},
            {0, 1, 0, 1}};

In[3]:= b = {1, 1, 1};
In[4]:= c = {-1, -1, -1};
In[5]:= DiophantineGF[a, b, c]
           t
Out[5]= ---------
                 2
        (-1 + t)

In[6]:= formula;
Binomial Formula : C[n, 1]
Power Formula    : n
In[7]:= formulaN[10]
Out[7]= 10
```

For $n \times n$ Triangular Matrix Product problem the Diophantine system is $\mathbf{az} = N\mathbf{b} + \mathbf{c}$ where

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{10}$$

and $\mathbf{c} = [-2, -1, 0, 0\ ]^T$ for $n = 2N$, and $\mathbf{c} = [-1, -1, 0, 0\ ]^T$ for $n = 2N + 1$.

3.2 Sample Run for the $n \times n$ Triangular Matrix Product Example ($n = 2N$):

```
In[1]:= << DiophantineGF.m
In[2]:= a = {{1, 1, 1, 0, 0, 0},
            {0, 1, 0, 1, 0, 0},
            {0,-1, 1, 0, 1, 0},
            {1, 0,-1, 0, 0, 1}};

In[3]:= b = {3, 2, 0, 0};
In[4]:= c = {-2,-1, 0, 0};
In[5]:= DiophantineGF[a, b, c]


          t
```

```
Out[5]= --------
               3
        (1 - t)

In[6]:= formula
Binomial Formula : C[1 + n, 2]
                   n (1 + n)
Power Formula    : ---------
                       2
```

When $n = 2N+1$, the Mathematica run results in the same generating function computed above for the case $n = 2N$. Therefore a lower bound for the number of processors needed to satisfy a the linear schedule $\tau(i,j,k) = i+j+k+1$ for the $n \times n$ Triangular Matrix Product is

$$\frac{\lfloor n/2 \rfloor (1 + \lfloor n/2 \rfloor)}{2} .$$

4 THE ALGORITHM

We demonstrate the algorithm on a specific instance, and sketch its proof. Consider the linear Diophantine system

$$\begin{array}{ccccc} z_1 & - & 2z_2 & = & n \\ z_1 & + & z_2 & = & 2n \end{array} \tag{11}$$

in which z_1 and z_2 are non-negative integers. Let d_n denote the number of solutions to (11). Associate indeterminates λ_1 and λ_2 to the first and the second equations, respectively, and also indeterminates t_1 and t_2 to the first and the second columns of the system. Consider the the product of the geometric series

$$R = \frac{1}{1-\lambda_1^1\lambda_2^1 t_1}\ \frac{1}{1-\lambda_1^{-2}\lambda_2^1 t_2} = \left(\sum_{\alpha_1 \geq 0} (\lambda_1^1\lambda_2^1 t_1)^{\alpha_1}\right)\left(\sum_{\alpha_2 \geq 0} (\lambda_1^{-2}\lambda_2^1 t_2)^{\alpha_2}\right)$$

where the exponents of λ_1 and λ_2 in the first factor are the coefficients in the first column and the exponents of λ_1 and λ_2 in the second factor are the coefficients in the second column. Individual terms arising from this product are of the form

$$\lambda_1^{\alpha_1 - 2\alpha_2}\ \lambda_2^{\alpha_1+\alpha_2}\ t_1^{\alpha_1} t_2^{\alpha_2}\ , \tag{12}$$

where α_1, α_2 are non-negative integers. Following Cayley, MacMahon [12] makes use of the operator $\underset{=}{\Omega}$ which picks out those terms (12) in the power series expansion whose exponents of λ_1 and λ_2 are both equal to zero (this is the λ-*free* part of the expansion). Thus, the contribution of the term in (12) to $\underset{=}{\Omega}(R)$ is nonzero if and only if the exponents of λ_1 and λ_2 are equal to zero. If this is the case, the contribution is $t_1^{\alpha_1} t_2^{\alpha_2}$ if and only if $z_1 = \alpha_1$ and $z_2 = \alpha_2$ is a solution to the homogeneous system

$$\begin{array}{ccccc} z_1 & - & 2z_2 & = & 0 \\ z_1 & + & z_2 & = & 0 \end{array} \tag{13}$$

This means, in particular, that what MacMahon calls the "crude" generating function of the solutions to the homogeneous system (13) is

$$\frac{1}{1-\lambda_1^1\lambda_2^1 t_1}\ \frac{1}{1-\lambda_1^{-2}\lambda_2^1 t_2}\ ,$$

and

$$\underset{=}{\Omega}\left(\frac{1}{(1-\lambda_1^1\lambda_2^1 t_1)(1-\lambda_1^{-2}\lambda_2^1 t_2)}\right) = \sum t_1^{\alpha_1} t_2^{\alpha_2}$$

where the summation is over all solutions $z_1 = \alpha_1$ and $z_2 = \alpha_2$ of (13). Let $R_n = \lambda_1^{-n}\lambda_2^{-2n}\ R$ where the exponents are the negatives of the right hand sides of (11). Then

$$\underset{=}{\Omega}(R_n) = \sum t_1^{\alpha_1} t_2^{\alpha_2}$$

where now the summation is over all non-negative integral solutions $z_1 = \alpha_1$, $z_2 = \alpha_2$ of (11). If we let $t_1 = t_2 = 1$, then $\underset{=}{\Omega}(R_n)$ specializes to the number of solutions d_n to (11). Let L denote the substitution operator that sets each t_i equal to 1. Then $d_n = L\underset{=}{\Omega}(R_n)$, and the operator $\underset{=}{\Omega}$ commutes both with L operation and addition of series. Thus,

$$\begin{aligned} f(t) &= \sum_{n\geq 0} L\underset{=}{\Omega}(R_n)\ t^n \\ &= \underset{=}{\Omega}\left(\sum_{n\geq 0} L(R_n)\ t^n\right) && (14) \\ &= \underset{=}{\Omega}\left(\frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)} \sum_{n\geq 0} \lambda_1^{-n}\lambda_2^{-2n}\ t^n\right)\ . && (15) \end{aligned}$$

Since

$$\sum_{n\geq 0} \lambda_1^{-n}\lambda_2^{-2n}\ t^n = \frac{1}{1-\lambda_1^{-1}\lambda_2^{-2}\ t}\ ,$$

the generating function $f(t)$ is obtained by applying the operator $\underset{=}{\Omega}$ to the crude generating function

$$F = \frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)}\ . \tag{16}$$

Now, we make use of the identity that appears in Stanley [22] for the computation of the homogeneous case above, namely

$$\frac{1}{(1-A)(1-B)} = \frac{1}{(1-AB)(1-A)} + \frac{1}{(1-AB)(1-B)} - \frac{1}{1-AB}. \tag{17}$$

We demonstrate the usage of this identity on the example at hand. Taking the first two factors of (16) as $(1-A)^{-1}$ and $(1-B)^{-1}$, and using (17),

$$\begin{aligned} F = &\frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \\ &+ \frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \\ &- \frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \end{aligned}$$

which we can write as $F = F_1 + F_2 - F_3$, where F_1, F_2, and F_3 denote the three summands above. By additivity,

$$f(t) = \underset{=}{\Omega}(F) = \underset{=}{\Omega}(F_1) + \underset{=}{\Omega}(F_2) - \underset{=}{\Omega}(F_3)\ .$$

Using the same identity in (17), this time on F_3 with $(1-A)^{-1}$ and $(1-B)^{-1}$ as the two factors, we obtain the expansion

$$F_3 \;=\; \frac{1}{(1-t)(1-\lambda_1^{-1}\lambda_2^2)} + \frac{1}{(1-t)(1-\lambda_1^{-1}\lambda_2^{-2}\,t)} - \frac{1}{(1-t)}$$

Call a product of the form

$$\frac{\pm 1}{(1-A)(1-B)\cdots(1-Z)} \tag{18}$$

that may arise during this process *uniformly-signed* if the exponents of λ_1 that appear in $A, B, \ldots, Z$ are either all non-negative, or all non-positive; the exponents are λ_2 that appear in $A, B, \ldots, Z$ are either all non-negative, or all non-positive;, etc.. Clearly if U is such a uniformly-signed product, then $\underset{=}{\Omega}(U)$ is obtained from U by discarding the factors which are not purely functions of t, as there can be no "cross cancellation" of any of the terms coming from different expansions into geometric series of the factors $(1-A)^{-1}$, $(1-B)^{-1}, \ldots, (1-Z)^{-1}$ of U.

The idea, then, is to use identity (17) repeatedly using pairs of appropriate factors in such a way that in the resulting products of the form (18) that arise are all uniformly-signed. The contribution of a uniformly-signed product to $f(t)$ is simply the product of the terms in it that are functions of t only, and all other factors can be ignored. For the summands of F_3 above, for example,

$$\underset{=}{\Omega}\left(\frac{1}{(1-t)(1-\lambda_1^{-1}\lambda_2^2)}\right) \;=\; \frac{1}{(1-t)}\,,$$

and also

$$\underset{=}{\Omega}\left(\frac{1}{(1-t)(1-\lambda_1^{-1}\lambda_2^{-2}\,t)}\right) \;=\; \frac{1}{(1-t)}\,.$$

Therefore, a pair of $(1-t)^{-1}$'s cancel and

$$\underset{=}{\Omega}(F_3) \;=\; \frac{1}{(1-t)}\,.$$

The problem is to pick the $(1-A)^{-1}$, $(1-B)^{-1}$ pairs at each step appropriately to make sure that the process eventually ends with uniformly-signed products only. This cannot be done arbitrarily, however. We use the following criterion: Given a term of the form (18), pick the λ_i with the smallest i for which a negative and a positive exponent appears among $A, B, \ldots, Z$. Use two extremes of such opposite signed factors $(1-A)^{-1}$ and $(1-B)^{-1}$ of the current term in (18), and apply identity (17) with this choice of A and B. This computational process results in a ternary tree whose leaves are functions of t only, after the application of the operator $\underset{=}{\Omega}$. The generating function $f(t)$ can then be read off as the (signed) sum of the functions that appear at the leaf nodes. The reader can verify that the example at hand results in the generating function

$$f(t) = \frac{1}{(1-t)(1+t+t^2)}$$

after the functions of t at the leaf nodes of the resulting ternary tree are summed up and necessary algebraic simplifications are carried out.

In the case above, $\mathbf{c} = \mathbf{0}$. Now, we consider the a more general case with $\mathbf{c} \neq \mathbf{0}$. These are the instances for which the description and the proof of the algorithm is not much harder, but the extra computational effort required justifies the use of a symbolic algebra package.

As an example, consider the Diophantine system

$$\begin{array}{rcrcl} z_1 & - & 2z_2 & = & n-2 \\ z_1 & + & z_2 & = & 2n+3 \end{array} \tag{19}$$

As before, let d_n be the number of solutions to (19) in nonnegative integers z_1, z_2, and let $f(t)$ be the generating function of the d_n. As in the derivation of the identity (15) for $f(t)$, this time we obtain

$$f(t) = \underset{=}{\Omega} \left(\frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)} \sum_{n\geq 0} \lambda_1^{-n+2}\lambda_2^{-2n-3}\ t^n \right) . \tag{20}$$

Since

$$\sum_{n\geq 0} \lambda_1^{-n+2}\lambda_2^{-2n-3}\ t^n \ = \ \frac{\lambda_1^2\lambda_2^{-3}}{1-\lambda_1^{-1}\lambda_2^{-2}\ t} ,$$

the generating function $f(t)$ is obtained by applying the operator $\underset{=}{\Omega}$ to the crude generating function

$$F = \frac{\lambda_1^2\lambda_2^{-3}}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} . \tag{21}$$

Now, we proceed as before using the identity (17), ignoring the numerator for the time being. It is no longer true that there be no "cross cancellation" of any of the terms coming from different expansions into geometric series of the factors $(1-A)^{-1}$, $(1-B)^{-1}, \ldots, (1-Z)^{-1}$ in a product U of the form (18) even if the term is uniformly-signed. It could be that the exponents of all of the λ_1 that appear in U are negative, and the exponents of all of the λ_2 that appear in U are all positive, but there can be $\lambda-$free terms arising from the expansions of the products that involve λ's, since the numerator $\lambda_1^2\lambda_2^{-3}$ can cancel terms of the form $\lambda_1^{-2}\lambda_2^3 t^k$ that may be produced if we expand the factors into geometric series and multiply. The application of $\underset{=}{\Omega}$ would then contribute t^k from this term to the final result coming from U for example. The important observation is that the geometric series expansion of the terms that involve λ in U need not be carried out past powers of λ_1 larger than 2, and past powers of λ_2 smaller than -3. This means that we need to keep track of only a polynomial in λ_1, λ_2 and t before the application of $\underset{=}{\Omega}$ to find the λ-free part contributed by this leaf node. In this case, this contribution may involve a polynomial in t as well.

5 COMPLEXITY

While general and elegant, the number of leaves in the generated tree is $O(3^n)$, where $n = \sum_{\{a_i\}} |a_i|$, and $\{a_i\}$ is the set of coefficients describing the set of Diophantine equations. However, the set of coefficients describing the Diophantine system is not unique. Translating the polyhedron, and omitting superfluous constraints (i.e., not in their transitive reduction) reduces the algorithm's work. Additional preprocessing may be possible (e.g., via unitary transforms). Moreover, even answering the question "Are *any* processors scheduled for this time step?" is known to be NP-complete [20, 6]. This computational complexity is further ameliorated by the observation that, since a formula can be automatically produced from the generating function, it needs to be constructed only once for a given algorithm. In practice, array algorithms typically have a description that is sufficiently succinct to make this automated formula production feasible.

6 SUMMARY

Given 1) a nested loop program whose underlying computation dag has nodes representable as lattice points in a convex polyhedron, and 2) a multiprocessor schedule for these nodes that is linear in the loop indices, we seek the number of nodes in the convex polyhedron that are scheduled for a particular time step (which is a lower bound on the number of processor needed to satisfy the schedule[2]). This

[2]It is relatively easy to determine which time step maximizes this lower bound.

is done by constructing a system of linear Diophantine equations whose solutions represent the lattice points of interest. Our principal contribution is devising an algorithm and its implementation for constructing the generating function that enumerates these solutions.

Several examples illustrated the relationship between nested loop programs and Diophantine equations, and were annotated with the output of a Mathematica program that implements the algorithm. The algorithmic relationship between the Diophantine equations and the generating function was illustrated with a simple example. Proof of the algorithm's correctness was sketched (space limitations preclude a more detailed exposition) while illustrating its steps. The algorithm's exponential computational complexity should be seen in light of two facts:

- Deciding if a time step has *any* nodes associated with it is NP-complete; we construct a *formula* for the number of such nodes;
- This formula is a processor lower bound, not just for one instance of a scheduled computation but for parameterized family of such computations.

In bounding the number of processors needed to satisfy a linear multiprocessor schedule for a nested loop program, we actually derived a solution to a more general linear Diophantine problem of the type given by Eq. 7. It may be of independent interest that the number of lattice points inside an arbitrary convex polyhedron can be easily computed as a byproduct of the algorithm.

References

[1] Peter R. Cappello. *VLSI Architectures for Digital Signal Processing.* PhD thesis, Princeton University, Princeton, NJ, Oct 1982.

[2] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI array design with geometric transformations. In H. J. Siegel and Leah Siegel, editors, *Proc. Int. Conf. on Parallel Processing*, pages 448–457, Bellaire, MI, Aug. 1983.

[3] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI array design with linear transformations of space-time. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 2: VLSI theory, pages 23–65. JAI Press, Inc., Greenwich, CT, 1984.

[4] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is close to optimal. In José Fortes, Edward Lee, and Teresa Meng, editors, *Application Specific Array Processors*, pages 37–46. IEEE Computer Society Press, August 1992.

[5] Bradley R. Engstrom and Peter R. Cappello. The SDEF programming system. *J. of Parallel and Distributed Computing*, 7:201–231, 1989.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, CA, 1979.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, 2nd Edition, The Johns Hopkins University Press, Baltimore, 1990.

[8] Richard M. Karp, Richard E. Miller, and Shmuel Winograd. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl. Math*, 14:1390–1411, 1966.

[9] Richard M. Karp, Richard E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14:563–590, 1967.

[10] H.-T. Kung and Charles E. Leiserson. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*, pages 271–292. Addison-Wesley Publishing Co, Menlo Park, CA, 1980.

[11] Leslie Lamport. The parallel execution of Do-Loops. *Comm. of the ACM*, 17(2):83–93, Feb. 1974.

[12] Percy A. MacMahon. Memoir on the Theory of the Partitions of Numbers- Part II, in *Collected Papers, Vol. I, Combinatorics*, George E. Andrews, Editor, The MIT Press, Cambridge, MASS, 1978, pp. 1138–1188.

[13] Percy A. MacMahon. Memoir on the Theory of the Partitions of Numbers- Part IV, in *Collected Papers, Vol. I, Combinatorics*, George E. Andrews, Editor, The MIT Press, Cambridge, MASS, 1978, pp. 1292–1314.

[14] Percy A. MacMahon. Application of the Partition Analysis to the study of the properties of any system of consecutive integers. in *Collected Papers, Vol. I, Combinatorics*, George E. Andrews, Editor, The MIT Press, Cambridge, MASS, 1978, pp. 1189–1211.

[15] Percy A. MacMahon. The Diophantine Inequality $\lambda x \geq \mu y$. in *Collected Papers, Vol. I, Combinatorics*, George E. Andrews, Editor, The MIT Press, Cambridge, MASS, 1978, pp. 1212–1232.

[16] Percy A. MacMahon. Note on the The Diophantine Inequality $\lambda x \geq \mu y$. in *Collected Papers, Vol. I, Combinatorics*, George E. Andrews, Editor, The MIT Press, Cambridge, MASS, 1978, pp. 1233–1246.

[17] Dan I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, Jan. 1983.

[18] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Symp. on Computer Architecture*, pages 208–214, 1984.

[19] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In K. V. Nori, editor, *Lecture Notes in Computer Science*, number 241: Foundations of Software Technology and Theoretical Computer Science, pages 488–503. Springer Verlag, December 1986.

[20] Sartaj Sahni. Computational related problems.
SIAM J. Comput., 3:262–279, 1974.

[21] Chris Scheiman *Mapping Fundamental Algorithms onto Multiprocessor Architectures.* Ph.D. Thesis, UC Santa Barbara, Dept. of Computer Science, Dec., 1993.

[22] Richard P. Stanley Linear homogeneous diophantine equations and magic labelings of graphs. Duke Math. J., 40:607-632, 1973.

[23] Richard P. Stanley *Enumerative Combinatorics, Volume I*, Wadsworth & Brooks/Cole, Monterey, CA 1986.

[24] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.