

PROCESSOR LOWER BOUND FORMULAS FOR ARRAY COMPUTATIONS AND PARAMETRIC DIOPHANTINE SYSTEMS

PETER CAPPELLO and ÖMER EĞECIOĞLU

*Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106, USA
{omer, cappello}@cs.ucsb.edu*

Received 25 July 1997

Revised 20 October 1997

Communicated by Oscar H. Ibarra

ABSTRACT

Using a directed acyclic graph (dag) model of algorithms, we solve a problem related to precedence-constrained multiprocessor schedules for array computations: Given a sequence of dags and linear schedules parametrized by n , compute a lower bound on the number of processors required by the schedule as a function of n . In our formulation, the number of tasks that are scheduled for execution during any fixed time step is the number of non-negative integer solutions d_n to a set of parametric linear Diophantine equations. We present an algorithm based on generating functions for constructing a formula for these numbers d_n . The algorithm has been implemented as a Mathematica program. Example runs and the symbolic formulas for processor lower bounds automatically produced by the algorithm for *Matrix-Vector Product*, *Triangular Matrix Product*, and *Gaussian Elimination* problems are presented. Our approach actually solves the following more general problem: Given an arbitrary $r \times s$ integral matrix \mathbf{A} and r -dimensional integral vectors \mathbf{b} and \mathbf{c} , let d_n ($n = 0, 1, \dots$) be the number of solutions in non-negative integers to the system $\mathbf{Az} = n\mathbf{b} + \mathbf{c}$. Calculate the (rational) generating function $\sum_{n \geq 0} d_n t^n$ and construct a formula for d_n .

Keywords: Parallel algorithm, array computation, lower bound, Diophantine equation, lattice point, generating function.

1. Introduction

We consider array computations, often referred to as systems of uniform recurrence equations²⁵. Parallel execution of uniform recurrence equations has been studied extensively, from at least as far back as 1966 (e.g.,^{24,27,26,17,33,18,19,34,35,20}). 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:

$$(u, v) \in A \Rightarrow \tau(u) < \tau(v).$$

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

$$\tau(v) = \tau(u) \Rightarrow \pi(v) \neq \pi(u).$$

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 uses few processors. This view prompted several researchers to investigate processor-time-minimal schedules for families of dags. These are time-minimal schedules that in addition use as few processors as possible. Processor-time-minimal schedules for various fundamental problems have been proposed in the literature: Scheiman and Cappello ^{4,3,13,10} examine the dag family for matrix product; Louka and Tchunte ⁹ examine the dag family for Gauss-Jordan elimination; Scheiman and Cappello ^{11,12} examine the dag family for transitive closure; Benaini and Robert ^{2,1} examine the dag families for the algebraic path problem and Gaussian elimination. Clauss, Mongenet, and Perrin ⁵ developed a set of mathematical tools to help find a processor-time-minimal multiprocessor array for a given dag. Another approach to a general solution has been reported by Wong and Delosme ^{15,16}, and Shang and Fortes ¹⁴. They present methods for obtaining optimal linear schedules. That is, their processor arrays may be suboptimal, but they get the best linear schedule possible. Darte, Khachiyan, and Robert ²⁰ show that such schedules are close to optimal, even when the constraint of linearity is relaxed.

In ¹⁰, a lower bound on the number of processors needed to satisfy a schedule for a particular time step was formulated as the number of solutions to a linear Diophantine equation, subject to the linear inequalities of the convex polyhedron that defines the dag's computational domain. Such a geometric/combinatorial formulation for the study of a dag's task domain has been used in various other contexts in parallel algorithm design as well (e.g., ^{24,25,27,33,34,8,7,35,5,14,42,16}; see Fortes, Fu, and Wah ⁶ for a survey of systolic/array algorithm formulations.) The maximum such bound for a given linear schedule, taken over all time steps, is a lower bound for the number of processors needed to satisfy the schedule for the dag family. Here, we present a more general and uniform technique for deriving such lower bounds:

*Given a parametrized dag family and a correspondingly parametrized linear schedule, we compute a **formula** for a lower bound on the number of processors required by the schedule.*

This is much more general than the analysis of an optimal schedule for a given *specific* dag. The lower bounds obtained are good; we know of no dag treatable by this method for which the lower bounds are not also upper bounds. We believe this to be the first reported algorithm and its implementation for automatically generating such formulae.

The nodes of the dag typically can be viewed as lattice points in a convex polyhedron. Adding to these constraints the linear constraint imposed by the schedule itself results in a linear Diophantine system of the form

$$\mathbf{A}\mathbf{z} = n\mathbf{b} + \mathbf{c} , \tag{1}$$

where the matrix \mathbf{A} and the vectors \mathbf{b} and \mathbf{c} are integral, but not necessarily non-negative. The number d_n of solutions in non-negative integers $\mathbf{z} = [z_1, z_2, \dots, z_s]^t$ to this linear system is a lower bound for the number of processors required when the dag corresponds to parameter n . Our algorithm produces (symbolically) the generating function for the sequence d_n , and from the generating function, a formula for the numbers d_n . We do not make use of any special properties of the system that reflects the fact that it comes from a dag. Thus in (1), \mathbf{A} can be taken to be an arbitrary $r \times s$ integral matrix, and \mathbf{b} and \mathbf{c} arbitrary r -dimensional integral vectors. As such we actually solve a more general combinatorial problem of constructing the generating function $\sum_{n \geq 0} d_n t^n$, and a formula for d_n given a matrix \mathbf{A} and vectors \mathbf{b} and \mathbf{c} , for which the lower bound computation is a special case. There is a large body of literature concerning lattice points in convex polytopes and numerous interesting results: see for example Stanley³⁹ for Ehrhart polynomials, and Sturmfels^{40,41} for vector partitions and other mathematical treatments. Our results are based mainly on MacMahon^{31,32}, and Stanley³⁸.

The outline of this paper is as follows. In Section 2, we use the examples of *Matrix-Vector Product*, *Triangular Matrix Product*, and *Gaussian Elimination* problems to describe the lattice point interpretation of parametric dags. Section 3 describes the general formulation of the problem and the sequence of steps to go from a dag to a parametric linear Diophantine system. In Section 4, we present sample runs of the Mathematica implementation: these include the array computation examples of Section 2, and three others. In Section 5 we describe the main points of the algorithm to construct the generating function and the ideas behind its proof. In Section 6 we present a high level description of the implementation, remark on the complexity of the algorithm, and summarize our results.

2. Examples from Array Computations

2.1. 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. M is the input matrix, x is the input vector, and $y = M \cdot x$ is the output vector. We index the entries of an n -dimensional vector v by $v[0], v[1], \dots, v[n-1]$.

```

for  $i = 0$  to  $n - 1$  do:
     $y[i] \leftarrow 0$ ;
    for  $j = 0$  to  $n - 1$  do:
         $y[i] \leftarrow y[i] + M[i, j] \cdot x[j]$ ;
    endfor;
endfor;

```

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

$$\begin{aligned} 0 &\leq i \leq n-1 \\ 0 &\leq j \leq n-1 \end{aligned} \tag{2}$$

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 (2) to a set of *equations* (which turn out to be easier to work with), we introduce integral slack variables $s_1, s_2 \geq 0$ and write

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

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

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

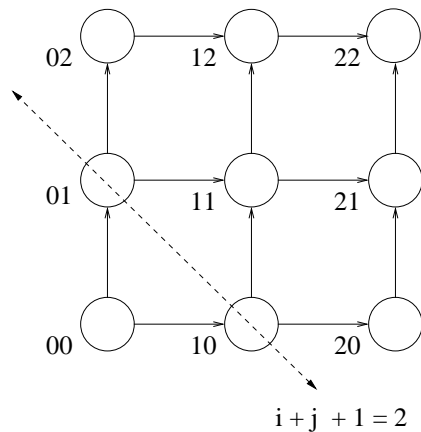


Figure 1: The matrix-vector product dag for $n = 2$.

The standard, time-minimal 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 parallel execution (see Figure 1). At time step $\tau = n$, there are n nodes scheduled for execution: $N(0, n-1), N(1, n-2), \dots, N(n-1, 0)$. 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 $1 \leq \tau \leq 2n - 1$, 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 (2) is

$$\begin{array}{rcl} i & + & j & & & = & n - 1 \\ i & & & + & s_1 & & = & n - 1 \\ & & j & & & + & s_2 & = & n - 1 \end{array} \quad (3)$$

The number of non-negative integral solutions to (3) is a processor lower bound for the $n \times n$ *Matrix-Vector Product* problem. For this example, time step $\tau = n$ obviously requires the maximum number of nodes that must be computed concurrently, as τ ranges from 1 to $2n - 1$. Thus, as is well known, to realize this schedule, n processors are necessary (and clearly sufficient).

2.2. Example 2: $n \times n$ Triangular Matrix Product

An algorithm for the computation of the 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 Golub and Van Loan²³.

```

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

$$\begin{array}{l} i \leq n - 1 \\ i \leq j \leq n - 1 \\ i \leq k \leq j. \end{array}$$

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

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

Note that as before we assume from the outset that the variables are non-negative. Introducing integral slack variables $s_1, s_2, s_3 \geq 0$, we obtain the equivalent linear Diophantine system

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

A linear schedule for the corresponding dag is given by $\tau(i, j, k) = i + j + k + 1$. Since τ ranges from 1 to $3n - 2$, we can augment the system by adding the constraint $i + j + k + 1 = \alpha(3n - 2)$ for any rational number α between 0 and 1. In particular the halfway point in this schedule is time step $\tau \approx \frac{3}{2}n - 1$. When n is an even number, say $n = 2N$, then we can take τ to be $3N - 1$. Adding the schedule constraint to the system we already have, we obtain the augmented Diophantine system

$$\begin{array}{rcccccl}
i & + j & + k & & & = 3N - 2 \\
& j & & + s_1 & & = 2N - 1 \\
- j & + k & & + s_2 & & = 0 \\
i & & - k & & + s_3 & = 0
\end{array} \tag{4}$$

If $n = 2N + 1$ is an odd number, then the exact midpoint of the schedule is $\tau = 3N + 1$. The augmented system now becomes

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

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 (4) if $n = 2N$, and the number of solutions of (5) if $n = 2N + 1$.

2.3. Example 3: Gaussian Elimination without Pivoting

The algorithm for performing Gaussian elimination on an $n \times n$ matrix M below is taken from Golub and Van Loan ²³.

```

for  $i = 0$  to  $n - 1$  do:
  for  $j = i + 1$  to  $n - 1$  do:
     $w_j \leftarrow M[i, j]$ ;
  endfor;
  for  $j = i + 1$  to  $n - 1$  do:
     $\eta \leftarrow M[j, i] / M[i, i]$ ;
    for  $k = i + 1$  to  $n - 1$  do:
       $M[j, k] \leftarrow M[j, k] - \eta \cdot w_j$ ;
    endfor;
  endfor;
endfor;

```

We are interested in the triply-nested *for* loop, the heart of the computation. The computational nodes are defined by non-negative integral triplets (i, j, k) satisfying the constraints

$$\begin{aligned}
i &\leq n-1 \\
i+1 &\leq j \leq n-1 \\
i+1 &\leq k \leq n-1
\end{aligned}$$

Note that as before we assume that the variables are non-negative. Since the first inequality is superfluous, introducing integral slack variables $s_1, s_2, s_3, s_4 \geq 0$, we obtain the equivalent linear Diophantine system

$$\begin{aligned}
i - j &+ s_1 &= -1 \\
&j &+ s_2 &= n - 1 \\
i &- k &+ s_3 &= -1 \\
&k &+ s_4 &= n - 1
\end{aligned}$$

A linear schedule for the corresponding dag is given by $\tau(i, j, k) = i + j + k + 1$. Since τ ranges from 1 to $3n - 2$, we can augment the system by adding the constraint at the halfway point: $\tau \approx \frac{3}{2}n - 1$. When n is an even number, say $n = 2N$, then we can take τ to be $3N - 1$. Adding the schedule constraint to system we already have, we obtain the augmented Diophantine system

$$\begin{aligned}
i + j + k &= 3N - 2 \\
i - j &+ s_1 &= -1 \\
&j &+ s_2 &= 2N - 1 \\
i &- k &+ s_3 &= -1 \\
&k &+ s_4 &= 2N - 1
\end{aligned} \tag{6}$$

Here $\mathbf{b} = [3, 0, 2, 0, 2]^t$ and $\mathbf{c} = [-2, -1, -1, -1, -1]^t$. The system for Gaussian elimination for $n = 2N + 1$ is

$$\begin{aligned}
i + j + k &= 3N - 1 \\
i - j &+ s_1 &= -1 \\
&j &+ s_2 &= 2N \\
i &- k &+ s_3 &= -1 \\
&k &+ s_4 &= 2N
\end{aligned} \tag{7}$$

which differs from the even case only in the vector \mathbf{c} .

Therefore, a lower bound for the number of processors needed to implement the schedule of the algorithm for Gaussian elimination without pivoting of an $n \times n$ matrix is the number of solutions of (6) if $n = 2N$, and the number of solutions of (7) if $n = 2N + 1$.

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

3. The General Formulation

We now generalize these 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 and its implementation are our principal contributions.

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, SDEF²¹ and ALPHA⁴², one expressly defines domains of computation as the integer points contained in some programmer-specified convex polyhedron.

The general setting exemplified by *Matrix-Vector Product*, *Triangular Matrix Product*, and *Gaussian Elimination* problems is as follows: Suppose \mathbf{a} (also denoted by \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 $\mathbf{a}\mathbf{z} = n\mathbf{b} + \mathbf{c}$, i.e.

$$\begin{aligned} a_{11}z_1 + a_{12}z_2 + \dots + a_{1s}z_s &= b_1n + c_1 \\ a_{21}z_1 + a_{22}z_2 + \dots + a_{2s}z_s &= b_2n + c_2 \\ \vdots & \\ a_{r1}z_1 + a_{r2}z_2 + \dots + a_{rs}z_s &= b_rn + c_r \end{aligned} \tag{8}$$

in the non-negative integral variables z_1, z_2, \dots, 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 (8), $f(t)$ 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. 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 in our formulation, whereas $1 \leq i$ introduces an additional constraint.
3. Transform the system of inequalities to a system of equalities by introducing non-negative slack variables, one for each inequality.
4. Augment the system with a linear schedule for the associated dag, “frozen” in some intermediate time value: $\tau = \tau(n)$;

5. Run the program `DiophantineGF.m` on the resulting data. The program calculates the rational generating function $f(t) = \sum d_n t^n$, where d_n is the number of solutions to the resulting linear system of Diophantine equations, and produces a formula for d_n .

4. Mathematica Runs

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

$$\mathbf{a}\mathbf{z} = n\mathbf{b} + \mathbf{c} . \quad (9)$$

The main computation is performed by the call `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 (9). After the computation of $f(t)$ by the program, the user can execute the command `formula`, which produces formulas for d_n in terms of binomial coefficients (with certain added divisibility restrictions), and in terms of the ordinary power basis in n when such a formula exists. The command `formulaN[c]` evaluates 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[]`.

Below, we provide sample runs of `DiophantineGF.m`. The first three are processor lower bound computations for the array computation problems formulated, the rest are examples from combinatorics.

4.1. Sample Run 1: $n \times n$ Matrix-Vector Multiplication

For the linear schedule of the *Matrix-vector Product* example, the augmented Diophantine system in (3) can be written in the form (9) where

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} . \quad (10)$$

```
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}; c = {-1, -1, -1};
In[4]:= DiophantineGF[a, b, c]
```

^a<http://www.cs.ucsb.edu/~omer/personal/abstracts/DiophantineGF.m>

```

Out[4]= -----
          t
          2
        (-1 + t)

```

```

In[5]:= formula;
Binomial Formula : C[n, 1]
Power Formula    : n

```

In the output, $C[x, k]$ denotes the binomial coefficient $\binom{x}{k} = \frac{x!}{k!(x-k)!}$ when x is a non-negative integer, and zero otherwise.

4.2. Sample Run 2: $n \times n$ Triangular Matrix Product ($n = 2N$)

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

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 2 \\ 0 \\ 0 \end{bmatrix} \quad (11)$$

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

```

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

```

```

Out[4]= -----
          t
          3
        (1 - t)

```

```

In[5]:= formula
Binomial Formula : C[1 + n, 2]
                n (1 + n)
Power Formula    : -----
                    2

```

Since the n in this formula is our N , substituting $n/2$, we find that a lower bound for the number of processors needed to satisfy the linear schedule $\tau(i, j, k) = i + j + k + 1$ for the $n \times n$ *Triangular Matrix Product* is

$$\frac{n(n+2)}{8}$$

when n is even. When $n = 2N + 1$, the Mathematica run for the $n \times n$ problem results in the generating function $(1 + t^2)/(1 - t)^3(1 + t)$. This time the formula for d_n depends on whether or not N is even. It is found to be $2m^2 + 2m + 1$ if n is of the form $4m + 1$, and $2(1 + m)^2$ if n is of the form $4m + 3$.

To summarize, 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

$$\begin{array}{lll} 2m^2 + m & \text{if} & n = 4m, \\ 2m^2 + 3m + 1 & \text{if} & n = 4m + 2, \\ 2m^2 + 2m + 1 & \text{if} & n = 4m + 1, \\ 2m^2 + 4m + 2 & \text{if} & n = 4m + 3. \end{array}$$

In particular, a lower bound that holds for all cases for this problem is

$$\lfloor \frac{n}{4} \rfloor (2 \lfloor \frac{n}{4} \rfloor + 1).$$

4.3. Sample Run 3: Gaussian Elimination

For *Gaussian Elimination* without pivoting of an $n \times n$ matrix the Diophantine system is $\mathbf{a}\mathbf{z} = N\mathbf{b} + \mathbf{c}$ where

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (12)$$

Here $\mathbf{b} = [3, 0, 2, 0, 2]^t$ and $\mathbf{c} = [-2, -1, -1, -1, -1]^t$, for $n = 2N$. The generating function computed is

$$\frac{t^2(3 + t)}{(1 - t)^3(1 + t)}.$$

The actual formula `DiophantineGF.m` produces for the coefficient of t^N in the expansion of this function is

$$\begin{aligned} & (3C[(N - 2)/2, 0] - C[(N - 4)/2, 0] - 2C[(N - 3)/2, 0])/8 + \\ & (C[N - 3, 2] + 3C[(N - 2)/2, 0] - C[N - 2, 2] - 5C[N - 1, 2] + 21C[N, 2])/8 \end{aligned} \quad (13)$$

However, note that $C[x, 0] = 0$ unless x is an integer. This means that

$$3C[(N - 2)/2, 0] - C[(N - 4)/2, 0] - 2C[(N - 3)/2, 0] = \begin{cases} 2 & N \text{ even,} \\ -2 & N \text{ odd.} \end{cases}$$

Simplifying the other binomial coefficients in (13), we get the lower bound for $n = 2N$ as

$$\frac{2N^2 - N}{2} \quad \text{if } N \text{ is even,} \quad \frac{2N^2 - N - 1}{2} \quad \text{if } N \text{ is odd,}$$

which can be combined into $\lfloor \frac{2N^2-N}{2} \rfloor$ for $n = 2N$. The system for Gaussian elimination for $n = 2N + 1$ is given in (7). In this case $\mathbf{c} = [-1, -1, 0, -1, 0]^t$ and \mathbf{a} and \mathbf{b} are the same as above. The generating function computed by the program is

$$\frac{t(1+3t)}{(1-t)^3(1+t)}.$$

Simplifying the automatically produced formula as before,

$$(C[(N-1)/2, 0] - 3C[(N-3)/2, 0] + 2C[(N-2)/2, 0])/8 + (3C[N-2, 2] - 11C[N-1, 2] + 17C[N, 2] + 7C[N+1, 2])/8,$$

we obtain

$$\frac{2N^2 - N}{2} \text{ if } N \text{ is even, } \quad \frac{2N^2 - N - 1}{2} \text{ if } N \text{ is odd.}$$

Therefore the lower bound for $n = 2N + 1$ is also $\lfloor \frac{2N^2-N}{2} \rfloor$. Combining with the previous case, we obtain the processor lower bound

$$\lfloor \frac{\lfloor \frac{n}{2} \rfloor (2 \lfloor \frac{n}{2} \rfloor - 1)}{2} \rfloor$$

for $n \times n$ Gaussian elimination without pivoting for arbitrary n .

Next we present examples of sample runs for a few problems that do not arise from array computations.

4.4. Sample Run 4

Consider the inequalities

$$\begin{aligned} z_1 &\leq n-1 \\ z_2 &\leq n-1 \\ z_3 &\leq n-1 \end{aligned}$$

in non-negative integers z_1, z_2, z_3 . The number of solutions is $d_n = n^3$ for $n \geq 1$ since each z_i can be independently picked from $\{0, 1, \dots, n-1\}$. To verify this using `DiophantineGF.m`, we first introduce integral slack variables $s_1, s_2, s_3 \geq 0$ and write the corresponding system of equalities

$$\begin{array}{rclcl} z_1 & & + s_1 & & = n-1 \\ & z_2 & & + s_2 & = n-1 \\ & & z_3 & & + s_3 = n-1 \end{array}$$

with

$$\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}. \quad (14)$$

The Mathematica run gives the formula for d_n in the power basis as well as in terms of binomial coefficients, resulting in

$$d_n = n^3 = \binom{n}{3} + 4 \binom{n+1}{3} + \binom{n+2}{3}.$$

In[1]:= << DiophantineGF.m

In[2]:= a = {{1, 0, 0, 1, 0, 0},
 {0, 1, 0, 0, 1, 0},
 {0, 0, 1, 0, 0, 1}};

In[3]:= b = {1, 1, 1}; c = {-1, -1, -1};

In[4]:= DiophantineGF[a, b, c]

2
 t (1 + 4 t + t)

Out[4]= -----

4
 (-1 + t)

In[5]:= formula

Binomial Formula : C[n, 3] + 4 C[1 + n, 3] + C[2 + n, 3]
 3

Power Formula : n

4.5. Sample Run 5

For this example consider the linear Diophantine system

$$\begin{aligned} z_1 - z_2 + 2z_3 &= n + 1 \\ z_2 + z_4 &= n - 2 \\ 2z_1 + z_3 + z_5 &= n + 3 \end{aligned} \tag{15}$$

DiophantineGF.m gives the generating function of the number of solutions d_n of this system in non-negative integers as

2 2 3 4
 t (-1 - 2 t - t + t + 2 t)

Out[5]= -----

3 2
 (-1 + t) (1 + t) (1 + t + t)

2 3 4 5 6 7 8 9
 = t + 3 t + 5 t + 7 t + 9 t + 12 t + 14 t + 0[t]

In particular, we see from the Taylor series expansion of the generating function above that $d_8 = 14$, and thus there are 14 solutions to (15) when $n = 8$. This result can be checked in another way as follows: We can first substitute $n = 8$ in (15),

resulting in the system

$$\begin{aligned} z_1 - z_2 + 2z_3 &= 9 \\ z_2 + z_4 &= 6 \\ 2z_1 + z_3 + z_5 &= 11 \end{aligned} \tag{16}$$

This system then should have 14 solutions as well. Running `DiophantineGF.m` with the corresponding input

$$\mathbf{a} = \begin{bmatrix} 1 & -1 & 2 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 9 \\ 6 \\ 11 \end{bmatrix}$$

we obtain

$$\text{Out}[5] = \frac{14}{1-t} = 14 + 14t + 14t^2 + 14t^3 + 14t^4 + 14t^5 + 0[t^6]$$

which again implies that the number of solutions to (16) is 14, independently of n .

4.6. Sample Run 6

For the linear Diophantine system

$$\begin{aligned} z_1 - 2z_2 - z_3 &= n - 4 \\ z_1 + z_2 - z_3 &= 2n + 3 \\ z_1 - 2z_3 &= 2n - 2 \end{aligned} \tag{17}$$

`DiophantineGF.m` calculates the generating function of the number of solutions as

$$\text{Out}[6] = t^2 + t^5 + t^8$$

Thus (17) has unique solutions in non-negative integers for $n = 2, 5$, and 8 , and no other solutions.

5. The Algorithm

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

$$\begin{aligned} z_1 - 2z_2 &= n \\ z_1 + z_2 &= 2n \end{aligned} \tag{18}$$

in which z_1 and z_2 are non-negative integers. Let d_n denote the number of solutions to (18). 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 product of the geometric series

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

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

$$\lambda_1^{\alpha_1-2\alpha_2} \lambda_2^{\alpha_1+\alpha_2} t_1^{\alpha_1} t_2^{\alpha_2}, \quad (19)$$

where α_1, α_2 are non-negative integers. Following Cayley, MacMahon²⁸ makes use of the operator $\underline{\underline{\Omega}}$ which picks out those terms (19) 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 (19) to $\underline{\underline{\Omega}}(R)$ is non-zero 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^b to the homogeneous system

$$\begin{aligned} z_1 - 2z_2 &= 0 \\ z_1 + z_2 &= 0 \end{aligned} \quad (20)$$

This means, in particular, that what MacMahon calls the “crude” generating function of the solutions to the homogeneous system (20) is

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

and

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

where the summation is over all solutions $z_1 = \alpha_1$ and $z_2 = \alpha_2$ of (20). Let $R_n = \lambda_1^{-n} \lambda_2^{-2n} R$, where the exponents of λ_1 and λ_2 are the negatives of the right hand sides of first and the second equations of (18), respectively. Then

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

where now the summation is over all non-negative integral solutions $z_1 = \alpha_1, z_2 = \alpha_2$ of (18), since generic terms arising from the expansion of R are now of the form

$$\lambda_1^{\alpha_1-2\alpha_2-n} \lambda_2^{\alpha_1+\alpha_2-2n} t_1^{\alpha_1} t_2^{\alpha_2}.$$

If we let $t_1 = t_2 = 1$, then $\underline{\underline{\Omega}}(R_n)$ specializes to the number of solutions d_n to (18). Let \mathcal{L} denote the substitution operator that sets each t_i equal to 1. Then $d_n = \mathcal{L} \underline{\underline{\Omega}}(R_n)$, and the operator $\underline{\underline{\Omega}}$ commutes both with \mathcal{L} operation and addition of series. Thus,

$$\begin{aligned} f(t) &= \sum_{n \geq 0} \mathcal{L} \underline{\underline{\Omega}}(R_n) t^n \\ &= \underline{\underline{\Omega}} \left(\sum_{n \geq 0} \mathcal{L}(R_n) t^n \right) \end{aligned} \quad (21)$$

$$= \underline{\underline{\Omega}} \left(\frac{1}{(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-2} \lambda_2)} \sum_{n \geq 0} \lambda_1^{-n} \lambda_2^{-2n} t^n \right). \quad (22)$$

^bThere is only a single solution to (20) in this case, but this does not effect the general nature of the demonstration of the algorithm on this example.

Since

$$\sum_{n \geq 0} \lambda_1^{-n} \lambda_2^{-2n} t^n = \frac{1}{1 - \lambda_1^{-1} \lambda_2^{-2} t}, \quad (23)$$

the generating function $f(t)$ can be obtained by applying the operator $\underline{\Omega}$ to the crude generating function

$$F = \frac{1}{(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-2} \lambda_2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)}. \quad (24)$$

Now, we make use of the identity that appears in Stanley ³⁸ for the computation of the homogeneous case above, namely

$$\frac{1}{(1 - A)(1 - B)} = \frac{1}{(1 - AB)(1 - A)} + \frac{1}{(1 - AB)(1 - B)} - \frac{1}{1 - AB}. \quad (25)$$

We demonstrate the usage of this identity on the example at hand. Taking the first two factors of (24) as $(1 - A)^{-1}$ and $(1 - B)^{-1}$ (i.e. $A = \lambda_1 \lambda_2$, $B = \lambda_1^{-2} \lambda_2$), and using (25),

$$\begin{aligned} F &= \frac{1}{(1 - \lambda_1^{-1} \lambda_2^2)(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)} \\ &\quad + \frac{1}{(1 - \lambda_1^{-1} \lambda_2^2)(1 - \lambda_1^{-2} \lambda_2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)} \\ &\quad - \frac{1}{(1 - \lambda_1^{-1} \lambda_2^2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)} \end{aligned} \quad (26)$$

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,

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

Continuing this way by using the identity (25), this time on F_3 with $(1 - A)^{-1}$ and $(1 - B)^{-1}$ as the two factors, we obtain the expansion

$$\begin{aligned} F_3 &= \frac{1}{(1 - \lambda_1^{-2} t)(1 - \lambda_1^{-1} \lambda_2^2)} + \frac{1}{(1 - \lambda_1^{-2} t)(1 - \lambda_1^{-1} \lambda_2^{-2} t)} - \frac{1}{(1 - \lambda_1^{-2} t)} \\ &= F_{31} + F_{32} - F_{33}. \end{aligned} \quad (27)$$

Call a product of the form

$$\frac{\pm 1}{(1 - A)(1 - B) \cdots (1 - Z)} \quad (28)$$

that may arise during this process *uniformly-signed* if the exponents of λ_1 that appear in A, B, \dots, Z are either all non-negative, or all non-positive; the exponents of λ_2 that appear in A, B, \dots, Z are either all non-negative, or all non-positive, etc.. Clearly if U is such a uniformly-signed product, then $\underline{\Omega}(U)$ 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 $(1 - A)^{-1}, (1 - B)^{-1}, \dots, (1 - Z)^{-1}$ of U .

The idea, then, is to use identity (25) repeatedly using pairs of appropriate factors in such a way that the resulting products of the form (28) 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. Each of the summands of F_3 given in (27) above, for example, are uniformly signed. Since neither term contains a factor which is a pure function of t , the contribution of each is zero.

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. For example in the application of the identity (25) to

$$\frac{1}{(1 - \lambda_1^{-1}\lambda_2^1)(1 - \lambda_1^2\lambda_2^1)(1 - \lambda_1^1\lambda_2^{-1})} \quad (29)$$

with $1 - A = 1 - \lambda_1^{-1}\lambda_2^1$ and $1 - B = 1 - \lambda_1^2\lambda_2^1$ (in which the λ_1 exponents have opposite sign), one of the three terms produced by the identity to be further processed is

$$\frac{1}{(1 - \lambda_1^{-1}\lambda_2^1)(1 - \lambda_1^1\lambda_2^2)(1 - \lambda_1^1\lambda_2^{-1})}.$$

Continuing with the choice $1 - A = 1 - \lambda_1^1\lambda_2^2$, and $1 - B = 1 - \lambda_1^1\lambda_2^{-1}$ (in which the λ_2 exponents have opposite sign), one of the three terms produced is

$$\frac{1}{(1 - \lambda_1^{-1}\lambda_2^1)(1 - \lambda_1^2\lambda_2^1)(1 - \lambda_1^1\lambda_2^{-1})},$$

which is identical to (29). In particular the weight argument in Stanley³⁸ does not result in an algorithm unless the λ_i are processed to completion in a fixed ordering of the indices i (e.g. first all exponents of λ_1 are made same signed, then those of λ_2 , etc.)

Accordingly, we use the following criterion: Given a term of the form (28), pick the λ_i with the smallest i for which a negative and a positive exponent appears among A, B, \dots, Z . Use two extremes (i.e. maximum positive and minimum negative exponents) of such opposite signed factors $(1 - A)^{-1}$ and $(1 - B)^{-1}$ of the current term in (28), and apply identity (25) 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 $\underline{\Omega}$. 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 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

$$\begin{aligned} z_1 - 2z_2 &= n - 2 \\ z_1 + z_2 &= 2n + 3 \end{aligned} \tag{30}$$

As before, let d_n be the number of solutions to (30) in non-negative integers z_1, z_2 , and let $f(t)$ be the generating function of the d_n . As in the derivation of the identity (22) for $f(t)$, this time we obtain

$$f(t) = \underline{\underline{\Omega}} \left(\frac{1}{(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-2} \lambda_2)} \sum_{n \geq 0} \lambda_1^{-n+2} \lambda_2^{-2n-3} t^n \right). \tag{31}$$

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 $\underline{\underline{\Omega}}$ to the crude generating function

$$F = \frac{\lambda_1^2 \lambda_2^{-3}}{(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-2} \lambda_2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)}. \tag{32}$$

Now, we proceed as before using the identity (25), ignoring the numerator for the time being. It is no longer true that there can be no ‘‘cross cancellation’’ of any of the terms coming from different expansions into geometric series of the factors $(1 - A)^{-1}, (1 - B)^{-1}, \dots, (1 - Z)^{-1}$ in a product U of the form (28) 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 λ -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 $\underline{\underline{\Omega}}$ 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 $\underline{\underline{\Omega}}$ to find the λ -free part contributed by this leaf node. In this case, this contribution may involve a polynomial in t as well. Therefore when $\mathbf{c} \neq \mathbf{0}$, we need to calculate with truncated Taylor expansions at the leaf nodes of the computation tree. It is this aspect of the algorithm that is handled most efficiently (in terms of coding effort) by a symbolic algebra package such as Mathematica.

6. Implementation, Complexity, and Remarks

The main computational effort of the algorithm is the construction of the ternary tree (Figure 2), where each internal node is expanded according to identity (25), until uniform-signed leaf expressions are reached. For simplicity, we consider the case $\mathbf{c} = \mathbf{0}$. Carrying out this portion of the computation symbolically is unwise:

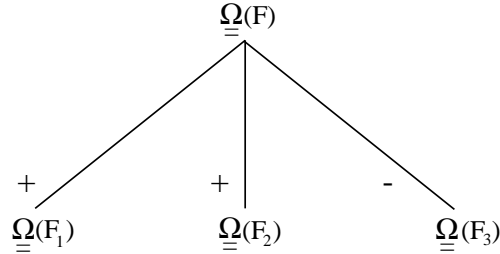


Figure 2: Generation of the ternary tree from identity (25).

instead, we represent each expression F that $\underline{\Omega}$ operates on as a $(r + 1) \times (s + 1)$ matrix M_F of integers. The 0^{th} row is reserved for the exponents of t in each factor in the denominator of F . The i^{th} row is the exponents of λ_i in F . For example, the initial F in (24) is encoded by the matrix

$$F = \frac{1}{(1 - \lambda_1 \lambda_2)(1 - \lambda_1^{-2} \lambda_2)(1 - \lambda_1^{-1} \lambda_2^{-2} t)} \quad \longleftrightarrow \quad M_F = \begin{bmatrix} 0 & 0 & 1 \\ 1 & -2 & -1 \\ 1 & 1 & -2 \end{bmatrix}.$$

Identity (25) applied to F now turns into column operations on $M = M_F$: Let C and C' be two columns of M . Then the three matrices corresponding to the summands F_1 , F_2 and F_3 in the application of (25) are obtained from M by

1. M_1 : Replace C by $C + C'$ in M ,
2. M_2 : Replace C' by $C + C'$ in M ,
3. M_3 : Replace C' by $C + C'$, and C by the zero vector in M .

The calculation in (26) followed by (27) results in the portion of the computation tree represented as matrices with sign in Figure 3. When the computation is continued, it can be seen that the middle subtree in this example produces a leaf node

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \longleftrightarrow \quad \frac{1}{(1 - \lambda_1 \lambda_2)(1 - \lambda_2 t)(1 - t^2)}.$$

The contribution of this to the generating function is the term

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

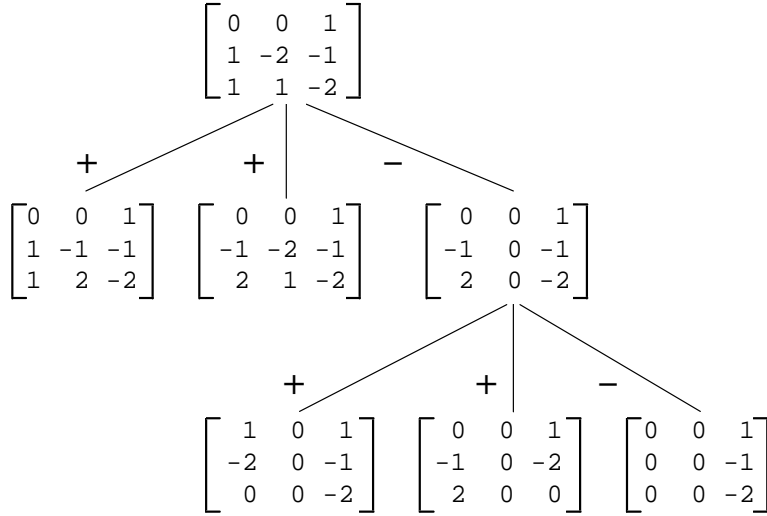


Figure 3: The start of the computational tree.

In general, when we arrive at a leaf node (i.e. when all rows of the current matrix are uniformly-signed), suppose d_1, d_2, \dots, d_l are the positive elements in the 0^{th} row, with the added property that the entries below each d_i in the leaf matrix are all zeros. The contribution of this leaf node to the generating function is then

$$\frac{\pm 1}{(1 - t^{d_1})(1 - t^{d_2}) \dots (1 - t^{d_l})} \quad (33)$$

with the appropriate sign. It is immediate that the resulting generating function is rational with a common denominator of the form (33).

We give a high level description of the algorithm using matrices to represent the coefficients. We assume that the given system is of the form (8) where $\mathbf{c} = \mathbf{0}$. The input is of the form of a matrix $M = M_{initial}$ given in (34). The rows of M are indexed 0 through r , and the columns are indexed 0 through s .

$$M_{initial} = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ a_{11} & a_{12} & \dots & a_{1s} & -b_1 \\ a_{21} & a_{22} & \dots & a_{2s} & -b_2 \\ \vdots & & \vdots & & \vdots \\ a_{r1} & a_{r2} & \dots & a_{rs} & -b_r \end{bmatrix}, \quad (34)$$

with $sign_{initial} = +1$. Assume the following functions

Zerocolumn $[M, j]$: Returns **true** iff the entries of M in the j -th column in rows 1 through r are all zeros.

Uniformsigned $[M]$: Returns **true** iff rows of M are all nonnegative or all nonpositive.

Minindex[M, i]: Returns the index of a smallest element in row i of M .

Maxindex[M, i]: Returns the index of a largest element in row i of M .

Firstnonuniform[M]: Returns the index of the first non uniformly-signed row in M .

Addcolumn[M, u, v]: Returns the matrix which is obtained from M by adding its v -th column to its u -th column.

Zapcolumn[M, v]: Returns the matrix obtained by replacing the v -th column of M by zeros.

Update[$gf, M, sign$]:

begin

Calculate $S = \{j \mid \text{Zerocolumn}[M, j]\}$;

Let $\{d_1, d_2, \dots, d_l\}$ be the multiset of positive values among $M[0, j]$, $j \in S$;

$gf = gf + sign * 1 / (1 - t^{d_1})(1 - t^{d_2}) \dots (1 - t^{d_l})$;

end Update

The main recursion for the $\mathbf{c} = \mathbf{0}$ case that generates the ternary tree consists of the basic steps given in Figure 4.

The number of leaves in the generated ternary tree is exponential in $n = \sum_{\{a_i\}} |a_i|$, where $\{a_i\}$ is the set of coefficients describing the set of Diophantine equations. The depth of recursion can be reduced somewhat, when the columns to be used are picked carefully. It is also possible to prune the tree when the input vector \mathbf{c} determines that there can be no λ -free terms resulting from the current matrix (e.g., some row is all strictly positive or all negative with $\mathbf{c} = \mathbf{0}$, or the row elements are weakly negative but the corresponding c_i is positive, etc.). Furthermore, the set of coefficients describing the Diophantine system coming from an array computation 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 some unitary transform).

The fact that the algorithm has worst case exponential running time 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 series expansion of the generating function non-zero?" is already known to be an NP-complete problem^{36,22}. 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.

To summarize the main ideas of this paper: given a nested loop program whose underlying computation dag has nodes representable as lattice points in a convex

```

Program Main[M]:
  begin
    gf = 0;
    M = Minitial;
    sign = signinitial = +1;

    Recurse[M, sign] :
      begin
        if Uniformsigned[M] then Update[gf, M, sign] and return;

        else begin
          i = Firstnonuniform[M];
          u = Minindex[M, i];
          v = Maxindex[M, i];

          M1 = Addcolumn[M, u, v];
          M2 = Addcolumn[M, v, u];
          M3 = Zapcolumn[M1, v];

          Recurse[M1, sign];
          Recurse[M2, sign];
          Recurse[M3, -sign];
        end;
      end Recurse;

  end Main

```

Figure 4: Basic description of the algorithm for $\mathbf{c} = \mathbf{0}$.

polyhedron, and a multiprocessor schedule for these nodes that is linear in the loop indices, we produce a formula for the number of lattice points in the convex polyhedron that are scheduled for a particular time step (which is a lower bound on the number of processors needed to satisfy the schedule). This is done by constructing a system of parametric 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 from which a formula for the number of these solutions is produced.

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 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 a 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 (8). This leaves open some interesting combinatorial questions of rationality and associated algorithm design: e.g. the analogue of (23) when the right hand side of the system in (8) consists of higher degree polynomials in n .

References

1. A. Benaini and Yves Robert. Space-time-minimal systolic arrays for Gaussian elimination and the algebraic path problem. *Parallel Computing*, 15:211–225, 1990.
2. Abdelhamid Benaini and Yves Robert. Spacetime-minimal systolic arrays for Gaussian elimination and the algebraic path problem. In *Proc. Int. Conf. on Application Specific Array Processors*, 746–757, Princeton, September 1990. IEEE Computer Society.
3. Peter Cappello. A processor-time-minimal systolic array for cubical mesh algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):4–13, January 1992. (Erratum: 3(3):384, May, 1992).
4. Peter R. Cappello. A spacetime-minimal systolic array for matrix product. In John V. McCanny, John McWhirter, and Earl E. Swartzlander, Jr., editors, *Systolic Array Processors*, 347–356, Killarney, IRELAND, May 1989. Prentice-Hall.
5. Ph. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In *Proc. Int. Conf. on Application Specific Array Processors*, 4–18, Princeton, September 1990. IEEE Computer Society.
6. José A. B. Fortes, King-Sun Fu, and Benjamin W. Wah. Systematic design approaches for algorithmically specified systolic arrays. In Veljko M. Milutinović, editor, *Computer Architecture: Concepts and Systems*, chapter 11, 454–494. North-Holland, Elsevier Science Publishing Co., New York, NY 10017, 1988.
7. José A. B. Fortes and Dan I. Moldovan. Parallelism detection and algorithm transformation techniques useful for VLSI architecture design. *J. Parallel Distrib. Comput.*, 2:277–301, Aug. 1985.
8. José A. B. Fortes and F. Parisi-Presicce. Optimal linear schedules for the parallel execution of algorithms. In *Int. Conf. on Parallel Processing*, 322–328, Aug. 1984.
9. Basile Louka and Maurice Tchuenta. An optimal solution for Gauss-Jordon elimination on 2D systolic arrays. In John V. McCanny, John McWhirter, and Earl E. Swartzlander Jr., editors, *Systolic Array Processors*, 264–274, Killarney, IRELAND, May 1989. Prentice-Hall.
10. Chris Scheiman and Peter Cappello. A processor-time minimal systolic array for the 3d rectilinear mesh. In *Proc. Int. Conf. on Application Specific Array Processors*, 26–33, Strasbourg, FRANCE, July 1995.
11. Chris Scheiman and Peter R. Cappello. A processor-time minimal systolic array for transitive closure. In *Proc. Int. Conf. on Application Specific Array Processors*, 19–30, Princeton, September 1990. IEEE Computer Society.

12. Chris Scheiman and Peter R. Cappello. A processor-time minimal systolic array for transitive closure. *IEEE Trans. on Parallel and Distributed Systems*, 3(3):257–269, May 1992.
13. Chris J. Scheiman and Peter Cappello. A period-processor-time-minimal schedule for cubical mesh algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):274–280, March 1994.
14. Weijia Shang and José A. B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6):723–742, June 1991.
15. Yiwan Wong and Jean-Marc Delosme. Optimization of processor count for systolic arrays. Dept. of Computer Sci. RR-697, Yale Univ., May 1989.
16. Yiwan Wong and Jean-Marc Delosme. Space-optimal linear processor allocation for systolic array synthesis. In V.K. Prasanna and L. H. Canter, editors, *Proc. 6th Int. Parallel Processing Symposium*, 275–282. IEEE Computer Society Press, Beverly Hills, March 1992.
17. Peter R. Cappello. *VLSI Architectures for Digital Signal Processing*. PhD thesis, Princeton University, Princeton, NJ, Oct 1982.
18. 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*, 448–457, Bellaire, MI, Aug. 1983.
19. 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, 23–65. JAI Press, Inc., Greenwich, CT, 1984.
20. 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*, 37–46. IEEE Computer Society Press, August 1992.
21. Bradley R. Engstrom and Peter R. Cappello. The SDEF programming system. *J. of Parallel and Distributed Computing*, 7:201–231, 1989.
22. 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.
23. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, 2nd Edition, The Johns Hopkins University Press, Baltimore, 1990.
24. 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.
25. Richard M. Karp, Richard E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14:563–590, 1967.
26. H.-T. Kung and Charles E. Leiserson. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*, 271–292. Addison-Wesley Publishing Co, Menlo Park, CA, 1980.
27. Leslie Lamport. The parallel execution of Do-Loops. *Comm. of the ACM*, 17(2):83–93, Feb. 1974.
28. 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, 1138–1188.
29. 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, 1292–1314.
30. 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, 1189–1211.
 31. 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, 1212–1232.
 32. 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, 1233–1246.
 33. Dan I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, Jan. 1983.
 34. Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Symp. on Computer Architecture*, 208–214, 1984.
 35. 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, 488–503. Springer Verlag, December 1986.
 36. Sartaj Sahni. Computational related problems. *SIAM J. Comput.*, 3:262–279, 1974.
 37. Chris Scheiman. *Mapping Fundamental Algorithms onto Multiprocessor Architectures*. Ph.D. Thesis, UC Santa Barbara, Dept. of Computer Science, Dec., 1993.
 38. Richard P. Stanley Linear homogeneous diophantine equations and magic labelings of graphs. *Duke Math. J.*, 40:607–632, 1973.
 39. Richard P. Stanley. *Enumerative Combinatorics, Volume I*, Wadsworth & Brooks/Cole, Monterey, CA 1986.
 40. Bernd Sturmfels. *Gröbner Bases and Convex Polytopes*, AMS University Lecture Series, Providence, RI 1995.
 41. Bernd Sturmfels. *On Vector Partition Functions*, J. Combinatorial Theory, Series A, 72:302–309, 1995.
 42. 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.