# Processor-Time-Optimal Systolic Arrays

Peter Cappello, Ömer Eğecioğlu, and Chris Scheiman

**Abstract**

Minimizing the amount of time and number of processors needed to perform an application reduces the application's fabrication cost and operation costs. A directed acyclic graph (dag) model of algorithms is used to define a time-minimal schedule and a processor-time-minimal schedule. We present a technique for finding a lower bound on the number of *processors* needed to achieve a given schedule of an algorithm. The application of this technique is illustrated with a tensor product computation. We then apply the technique to the free schedule of algorithms for matrix product, Gaussian elimination, and transitive closure. For each, we provide a time-minimal processor schedule that meets these processor lower bounds, including the one for tensor product.

**Keywords:** algebraic path problem, dag, Diophantine equation, Gaussian elimination, matrix product, optimal, systolic array, tensor product, transitive closure.

## 1 Introduction

We consider regular array computations, often referred to as systems of uniform recurrence equations [26]. Parallel execution of uniform recurrence equations has been studied extensively, from at least as far back as 1966 (e.g., [25, 31, 29, 7, 37, 9, 10, 38, 39, 13]). 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.

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

In what follows, we refer to a valid schedule simply as a schedule. A *time-minimal* schedule for an algorithm completes in $S$ steps, where $S$ is the number of nodes on a longest path in the dag. A time-minimal schedule exposes an algorithm's maximum parallelism. That is, it bounds the number of *time steps* the algorithm needs, when infinitely many processors may be used. A *processor-time minimal* schedule is a time-minimal schedule that uses as few *processors* as any time-minimal schedule for the algorithm. Although only one of many performance measures, processor-time-minimality is useful because it measures the *minimum processors* needed to extract the *maximum parallelism* from a dag. Being machine-independent, it is a more fundamental measure than those that depend on a particular machine or architecture. This view prompted several researchers to investigate processor-time-minimal schedules for families of dags. Processor-time-minimal systolic arrays are easy to devise, in an ad hoc manner, for 2D systolic algorithms. This apparently is not the case for 3D systolic algorithms. There have been several publications regarding processor-time-minimal systolic arrays for fundamental 3D algorithms. Processor-time-minimal schedules for various fundamental problems have been proposed in the literature: Scheiman and Cappello [8, 5, 47, 44] examine the dag family for matrix product; Louka and Tchuente [33] examine the dag family for Gauss-Jordan elimination; Scheiman and Cappello [45, 46] examine the dag family for transitive closure; Benaini and Robert [3, 2] examine the dag families for the algebraic path problem and Gaussian elimination. Each of the algorithms listed in Table 1 has the property that, in its dag representation, every node is on a longest path: Its free schedule is its *only* time-minimal schedule. A processor lower bound for achieving it is thus a processor lower bound for achieving maximum parallelism with the *algorithm*.

Table 1: *Some 3D algorithms for which processor-time-minimal systolic arrays are known.*

| **Algorithm** | **Citation** | **Time** | **Processors** |
|---|---|---|---|
| Algebraic Path Problem | [3] | $5n-2$ | $n^2/3+O(n)$ |
| Gauss-Jordan elimination | [33] | $4n$ | $5n^2/18+O(n)$ |
| Gaussian elimination | [3] | $3n-1$ | $n^2/4+O(n)$ |
| Matrix product | [8, 5] | $3n-2$ | $\lceil 3n^2/4 \rceil$ |
| Transitive closure | [45] | $5n-4$ | $\lceil n^2/3 \rceil$ |
| Tensor product | this article | $4n-3$ | $(2n^2+n)/3$ |

Clauss, Mongenet, and Perrin [11] 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 [57, 58], and Shang and Fortes [48]. 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 [13] show that such schedules are close to optimal, even when the constraint of linearity is relaxed. Geometric/combinatorial formulations of a dag's task domain have been used in various contexts in parallel algorithm design as well (e.g., [25, 26, 31, 37, 38, 18, 17, 39, 11, 48, 54, 58]; see Fortes, Fu, and Wah [16] for a survey of systolic/array algorithm formulations.)

In §2, we present an algorithm for finding a lower bound on the number of *processors* needed to achieve a given schedule of an algorithm. The application of this technique is illustrated with a tensor product computation. Then, in §3, we apply the technique to the free schedule of algorithms for matrix product, Gaussian elimination, and transitive closure. For each, we provide a compatible processor schedule that meets these processor lower bounds (i.e., a processor-time-minimal schedule) including the one for tensor product. We finish with some general conclusions and mention some open problems.

One strength of our approach centers around the word *algorithm*: we are finding processor lower bounds not just for a particular dag, but for a linearly parameterized family of dags, representing the infinitely many problem sizes for which the algorithm works. Thus, the processor lower bound is not a number but a *polynomial* (or finite set of polynomials) in the linear parameter used to express different problem sizes. The formula then can be used to optimize the implementation of the algorithm, not just a particular execution of it. The processor lower bounds are produced by:

- formulating the problem as finding, for a particular time step, the number of processors needed by that time step as a formula for the number of integer points in a convex polyhedron,
- representing this set of points as the set of solutions to a linearly parameterized set of linear Diophantine equations,
- computing a generating function for the number of such solutions,
- deriving a formula from the generating function.

The ability to compute formulae for the number of solutions to a linearly parameterized set of linear Diophantine equations has other applications for nested loops [12], such as finding the number of instructions executed, the number of memory locations touched, and the number of I/O operations.

The strength of our algorithm—its ability to produce formulae—comes, of course, with a price: The computational complexity of the algorithm is exponential in the size of the input (the number of bits in the coefficients of the system of Diophantine equations). The algorithm's complexity however is quite reasonable given the complexity of the *problem*: Determining if there are *any* integer solutions to the system of Diophantine equations is NP-complete [19]; we produce a formula for the number of such solutions.

Clauss and Loechner independently developed an algorithm for the problem based on *Ehrhart polynomials*. In [12], they sketch an algorithm for the problem, using the "polyhedral library" of Wilde [56].

## 2 Processor Lower Bounds

We present a general and uniform technique for deriving 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

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

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 $\mathbf{z} = [z_1, z_2, \ldots, z_s]^t$ 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 integral vectors of length $r$. 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 Stanley [50] for Ehrhart polynomials (Clauss and Loechner [12] use these), and Sturmfels [51, 52] for vector partitions and other mathematical treatments. Our results are based mainly on MacMahon [34, 36], and Stanley [49].

## 2.1 Example: Tensor product

We examine the dag family for the 4D mesh: the $n \times n \times n \times n$ directed mesh. This family is fundamental, representing a communication-localized version of the standard algorithm for Tensor product (also known as Kronecker product). The Tensor product is used in many mathematical computations, including multivariable spline blending and image processing [20], multivariable approximation algorithms (used in graphics, optics, and topography) [28], as well as many recursive algorithms [27].

The Tensor product of an $n \times m$ matrix $A$ and a matrix $B$ is:

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ \dots & & & \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{bmatrix}$$

The 4D mesh also represents other algorithms, such as the least common subsequence problem [23, 22], for 4 strings, and matrix comparison (an extension of tuple comparison [24]). An example dag is shown in Figure 1, for $n = 3$.

### 2.1.1 The dag

The 4D mesh can be defined as follows. $G_n = (N_n, A_n)$, where

- $N_n = \{(i, j, k, l) \mid 0 \leq i, j, k, l \leq n - 1\}$.

- $A_n = \{[(i,j,k,l),(i',j',k',l')] \,|(i,j,k,l) \in N_n,\ (i',j',k',l') \in N_n$ and exactly 1 of the following conditions holds:

  1. $i' = i+1,\ j' = j,\ k' = k,\ l' = l$;
  2. $j' = j+1,\ i' = i,\ k' = k,\ l' = l$;
  3. $k' = k+1,\ i' = i,\ j' = j,\ l' = l$;
  4. $l' = l+1,\ i' = i,\ j' = j,\ k' = k.\}$

### 2.1.2 The parametric linear Diophantine system of equations

The computational nodes are defined by non-negative integral 4-tuples $(i,j,k,l)$ satisfying

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

Introducing nonnegative integral slack variables $s_1, s_2, s_3, s_4 \geq 0$, we obtain the equivalent linear Diophantine system describing the computational nodes as

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

A linear schedule for the corresponding dag is given by $\tau(i,j,k,l) = i+j+k+l+1$. For this problem $\tau$ ranges from 1 to $4n-3$. The computational nodes about halfway through the completion of the schedule satisfy the additional constraint

$$i+j+k+l = 2n-2$$

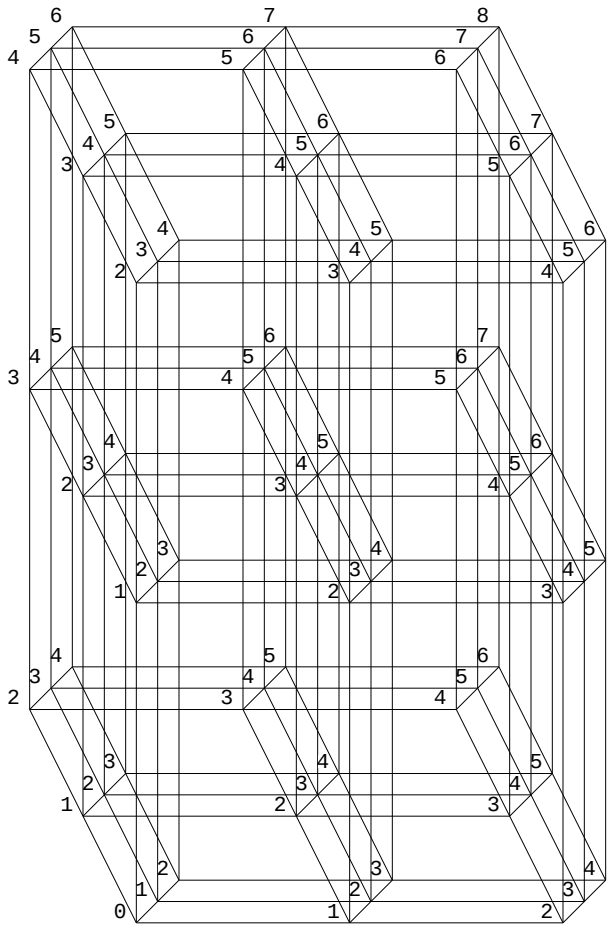

Figure 1: *The 4-dimensional cubical mesh, for* $n = 3$.

Adding this constraint we obtain the augmented Diophantine system

$$\begin{array}{rcl} i + j + k + l & = & 2n-2 \\ i + s_1 & = & n-1 \\ j + s_2 & = & n-1 \\ k + s_3 & = & n-1 \\ l + s_4 & = & n-1 \end{array} \tag{2}$$

Therefore, a lower bound for the number of processors needed for the *Tensor Product* problem is the number of solutions to (2). The corresponding Diophantine system is $\mathbf{az} = n\mathbf{b} + \mathbf{c}$ where

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

### 2.1.3 The Mathematica program input & output

Once the Mathematica program `DiophantineGF.m` 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 (1). 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 (1). 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[]`.

The `DiophantineGF` run on the data for the Tensor product problem gives (verbatim)

```
In[1]:= << DiophantineGF.m
Loaded. Type "help" for instructions, "example" for examples.
In[2]:=  a = {{1,1,1,1,0,0,0,0},
              {1,0,0,0,1,0,0,0},
```

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

```
              {0,1,0,0,0,1,0,0},
              {0,0,1,0,0,0,1,0},
              {0,0,0,1,0,0,0,1}};
In[3]:= b = {2,1,1,1,1}; c = {-2,-1,-1,-1,-1};
In[4]:= DiophantineGF[a,b,c]
                  2
         t (1 + t)
Out[4]= ----------
                 4
         (-1 + t)
In[5]:= formula
Binomial Formula : C[n, 3] + 2 C[1 + n, 3] + C[2 + n, 3]
                             2
                    n (1 + 2 n )
Power Formula    : ------------
                         3
```

Therefore

$$\frac{2n^3}{3} + \frac{n}{3}$$

is a processor lower bound for the Tensor Product problem.

## 2.2 General Formulation

We now generalize this example 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 is, we believe, a significant contribution.

The use of linear Diophantine equations 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 [14] and ALPHA [54], one expressly defines domains of computation as the integer points contained in some programmer-specified convex polyhedron.

The general setting exemplified by the *Tensor Product* problem is as follows: Suppose $\mathbf{a}$ is an $r \times s$ integral matrix, and $\mathbf{b}$ and $\mathbf{c}$ are integral vectors of length $r$. Suppose further that, for every

$n \geq 0$, the linear Diophantine system $\mathbf{az} = n\mathbf{b} + \mathbf{c}$, i.e.

$$\begin{array}{ccccccccc} a_{11}z_1 & + & a_{12}z_2 & + & \ldots & + & a_{1s}z_s & = & b_1 n + c_1 \\ a_{21}z_1 & + & a_{22}z_2 & + & \ldots & + & a_{2s}z_s & = & b_2 n + c_2 \\ \vdots & & & & \vdots & & \vdots & = & \vdots \\ a_{r1}z_1 & + & a_{r2}z_2 & + & \ldots & + & a_{rs}z_s & = & b_r n + c_r \end{array} \tag{4}$$

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 (4), $f(t)$ 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: $\tau = \tau(n)$; In other words, we augment the linear system with a parameterized constraint that specifies a hyperplane. Points that are in the polyhedron and also are in this hyperplane are scheduled to execute on the same time step, and hence require different processors. This number of processors is a lower bound on the number of processors needed for the entire computation.

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 $d_n$.

## 2.3 The Algorithm

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

$$\begin{array}{rcrcl} z_1 & - & 2z_2 & = & n \\ z_1 & + & z_2 & = & 2n \end{array} \tag{5}$$

in which $z_1$ and $z_2$ are non-negative integers. Let $d_n$ denote the number of solutions to (5). Associate indeterminates $\lambda_1$ and $\lambda_2$ 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

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

where the exponents of $\lambda_1$ and $\lambda_2$ in the first factor are the coefficients in the first column and the exponents of $\lambda_1$ and $\lambda_2$ in the second factor are the coefficients in the second column. Individual terms arising from this product are of the form

$$\lambda_1^{\alpha_1-2\alpha_2}\ \lambda_2^{\alpha_1+\alpha_2}\ t_1^{\alpha_1}t_2^{\alpha_2}\ , \tag{6}$$

where $\alpha_1, \alpha_2$ are non-negative integers. Following Cayley, MacMahon [35] makes use of the operator $\underset{=}{\Omega}$ which picks out those terms (6) in the power series expansion whose exponents of $\lambda_1$ and $\lambda_2$ are both equal to zero (this is the $\lambda$-*free* part of the expansion). Thus, the contribution of the term in (6) to $\underset{=}{\Omega}(R)$ is non-zero if and only if the exponents of $\lambda_1$ and $\lambda_2$ 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[2] to the homogeneous system

$$\begin{array}{rcrcl} z_1 & - & 2z_2 & = & 0 \\ z_1 & + & z_2 & = & 0 \end{array} \tag{7}$$

This means, in particular, that what MacMahon calls the "crude" generating function of the solutions to the homogeneous system (7) is

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

[2] There is only a single solution to (7) in this case, but this does not effect the general nature of the demonstration of the algorithm on this example.

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 (7). Let $R_n = \lambda_1^{-n}\lambda_2^{-2n}R$, where the exponents of $\lambda_1$ and $\lambda_2$ are the negatives of the right hand sides of first and the second equations of (5), respectively. 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 (5), 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 $\underset{=}{\Omega}(R_n)$ specializes to the number of solutions $d_n$ to (5). Let $\mathcal{L}$ denote the substitution operator that sets each $t_i$ equal to 1. Then $d_n = \mathcal{L}\,\underset{=}{\Omega}(R_n)$, and the operator $\underset{=}{\Omega}$ commutes both with $\mathcal{L}$ operation and addition of series. Thus,

$$\begin{aligned} f(t) &= \sum_{n\geq 0} \mathcal{L}\,\underset{=}{\Omega}(R_n)\ t^n \\ &= \underset{=}{\Omega}\left(\sum_{n\geq 0}\mathcal{L}(R_n)\ t^n\right) \qquad (8) \\ &= \underset{=}{\Omega}\left(\frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)}\sum_{n\geq 0}\lambda_1^{-n}\lambda_2^{-2n}\ t^n\right)\ . \qquad (9) \end{aligned}$$

Since

$$\sum_{n\geq 0}\lambda_1^{-n}\lambda_2^{-2n}\ t^n\ =\ \frac{1}{1-\lambda_1^{-1}\lambda_2^{-2}\ t}\ , \qquad (10)$$

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

$$F = \frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)}\ . \qquad (11)$$

Now, we make use of the identity that appears in Stanley [49] for the computation of the homogeneous case above, namely

$$\frac{1}{(1-A)(1-B)}\ =\ \frac{1}{(1-AB)(1-A)}\ +\ \frac{1}{(1-AB)(1-B)}\ -\ \frac{1}{1-AB}. \qquad (12)$$

We demonstrate the usage of this identity on the example at hand. Taking the first two factors of (11) as $(1-A)^{-1}$ and $(1-B)^{-1}$ (i.e. $A=\lambda_1\lambda_2$, $B=\lambda_1^{-2}\lambda_2$ ), and using (12),

$$\begin{aligned} F &= \frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \\ &\quad + \frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \\ &\quad - \frac{1}{(1-\lambda_1^{-1}\lambda_2^2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} \end{aligned} \tag{13}$$

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,

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

Continuing this way by using the identity (12), this time on $F_3$ with $(1-A)^{-1}$ and $(1-B)^{-1}$ as the two factors, we obtain the expansion

$$\begin{aligned} F_3 &= \frac{1}{(1-\lambda_1^{-2}\ t)(1-\lambda_1^{-1}\lambda_2^2)} + \frac{1}{(1-\lambda_1^{-2}\ t)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} - \frac{1}{(1-\lambda_1^{-2}\ t)} \\ &= F_{31} + F_{32} - F_{33}\ . \end{aligned} \tag{14}$$

Call a product of the form

$$\frac{\pm 1}{(1-A)(1-B)\cdots(1-Z)} \tag{15}$$

that may arise during this process *uniformly-signed* if the exponents of $\lambda_1$ that appear in $A, B, \ldots, Z$ are either all non-negative, or all non-positive; the exponents of $\lambda_2$ 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 $\underline{\underline{\Omega}}(U)$ 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 $(1-A)^{-1}, (1-B)^{-1}, \ldots, (1-Z)^{-1}$ of $U$.

The idea, then, is to use identity (12) repeatedly using pairs of appropriate factors in such a way that the resulting products of the form (15) 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 (14) 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 (12) to

$$\frac{1}{(1-\lambda_1^{-1}\lambda_2^1)(1-\lambda_1^2\lambda_2^1)(1-\lambda_1^1\lambda_2^{-1})} \tag{16}$$

with $1-A = 1-\lambda_1^{-1}\lambda_2^1$ and $1-B = 1-\lambda_1^2\lambda_2^1$ (in which the $\lambda_1$ 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 $\lambda_2$ 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 (16). In particular the weight argument in Stanley [49] does not result in an algorithm unless the $\lambda_i$ are processed to completion in a fixed ordering of the indices $i$ (e.g. first all exponents of $\lambda_1$ are made same signed, then those of $\lambda_2$, etc.)

Accordingly, we use the following criterion: Given a term of the form (15), pick the $\lambda_i$ with the smallest $i$ for which a negative and a positive exponent appears among $A$, $B, \ldots, Z$. 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 (15), and apply identity (12) 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 $\underline{\underline{\Omega}}$. 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

$$\begin{array}{rcrcl} z_1 & - & 2z_2 & = & n-2 \\ z_1 & + & z_2 & = & 2n+3 \end{array} \tag{17}$$

As before, let $d_n$ be the number of solutions to (17) 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 (9) for $f(t)$, this time we obtain

$$f(t) = \underset{=}{\Omega} \left( \frac{1}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)} \sum_{n\geq 0} \lambda_1^{-n+2}\lambda_2^{-2n-3}\ t^n \right) . \tag{18}$$

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

$$F = \frac{\lambda_1^2\lambda_2^{-3}}{(1-\lambda_1\lambda_2)(1-\lambda_1^{-2}\lambda_2)(1-\lambda_1^{-1}\lambda_2^{-2}\ t)} . \tag{19}$$

Now, we proceed as before using the identity (12), 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 $(1-A)^{-1}$, $(1-B)^{-1}, \ldots, (1-Z)^{-1}$ in a product $U$ of the form (15) even if the term is uniformly-signed. It could be that the exponents of all of the $\lambda_1$ that appear in $U$ are negative, and the exponents of all of the $\lambda_2$ that appear in $U$ are all positive, but there can be $\lambda-$free terms arising from the expansions of the products that involve $\lambda$'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 $\lambda$ in $U$ need not be carried out past powers of $\lambda_1$ larger than 2, and past powers of $\lambda_2$ smaller than $-3$. This means that we need to keep track of only a polynomial in $\lambda_1, \lambda_2$ and $t$ before the application of $\underset{=}{\Omega}$ to find the $\lambda$-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.

### 2.4 Complexity and remarks

Some detailed remarks concerning implementation are given in [6], which, for reasons of space, we omit here.

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 $\lambda$-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 [42, 19]. 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.

## 3 Processor Upper Bounds

Minimizing the amount of time and number of processors needed to perform an application reduces the application's fabrication cost and operation costs. This can be important in applications where minimizing the size and energy of the hardware is critical, such as space applications and consumer electronics. In this section, we present schedules that are processor-time-minimal. They are exact,

not just asymptotic.

Again, for each of the dags discussed below, the free schedule is its *only* time-minimal schedule. A processor lower bound for achieving it is thus a processor lower bound for achieving maximum parallelism with the *algorithm*, not just one schedule for it, much less one instance of one schedule for it (i.e., one value of $n$ for one parameterized schedule)

## 3.1 Matrix Product

### 3.1.1 Problem

**Program fragment:** The standard program fragment for computing the matrix product $C = A \cdot B$, where $A$ and $B$ are given $n \times n$ matrices is given below.

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

**Dag:** The cube-shaped 3D mesh (i.e., the $n \times n \times n$ mesh) can be defined as follows.

$G_{n \times n \times n} = (N, A)$, where

- $N = \{(i, j, k) \mid 1 \leq i, j, k \leq n\}$.
- $A = \{[(i, j, k), (i', j', k')] \mid$ where exactly 1 of the following conditions holds
  1. $i' = i + 1$
  2. $j' = j + 1$
  3. $k' = k + 1$

  for $1 \leq i, j, k \leq n\}$.

### 3.1.2 Lower bound

**Parameterized linear Diophantine system of equations:** The computational nodes are defined by non-negative integral triplets $(i, j, k)$ satisfying

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

**Mathematica input/output:** The `DiophantineGF` run for even $n$ gives

```
In[1]:=  << DiophantineGF.m
Loaded. Type "help" for instructions, "example" for examples.
In[2]:=  a = {2,2,2,0,0,0},
             {1,0,0,1,0,0},
             {0,1,0,0,1,0},
             {0,0,1,0,0,1}};
In[3]:=  b = {3,1,1,1}; c = {-2,-1,-1,-1};
In[4]:= DiophantineGF[a, b, c]
              2       2
           -3 t  (1 + t )
Out[4]= ------------------
                3        3
        (-1 + t)  (1 + t)
In[5]:= formula
                                  -9 + n                -7 + n
Binomial Formula : (-3 (3 C[2 + ------, 2] - 7 C[2 + ------, 2] +
                                   2                     2
                  -5 + n               -4 + n                -3 + n
>        5 C[2 + ------, 2] - 8 C[2 + ------, 2] + 15 C[2 + ------, 2] -
                    2                    2                     2
                  -2 + n
>        8 C[2 + ------, 2] - 3 C[-4 + n, 2] + 9 C[-3 + n, 2] -
                    2
>        11 C[-2 + n, 2] + 9 C[-1 + n, 2] - 8 C[n, 2])) / 16
```

Recall that $C[x, k]$ denotes the binomial coefficient $\binom{x}{k} = \frac{x!}{k!(x-k)!}$ when $x$ is a non-negative integer, and zero otherwise. This means that in the above formula $C[2 + (n-9)/2, 2]$ vanishes for even $n$, for example. In this way the formula simplifies to

$$\frac{3}{4}n^2, \qquad (n \text{ even}).$$

For odd $n$ we obtain the generating function and the formula below:

```
                  3
              -6 t
Out[4]= ------------------
                3         3
        (-1 + t)  (1 + t)
```

and a similar formula, which simplifies to

$$\frac{3}{4}n^2 - \frac{3}{4}, \qquad (n \text{ odd}).$$

These cases can be combined to obtain the processor lower bound $\lfloor \frac{3}{4}n^2 \rfloor$.

### 3.1.3 Upper bound

**Schedule:** The map $m : N \mapsto \mathbf{Z}^3$ (i.e., from nodes to processor-time) can be defined formally as follows.

$$\begin{pmatrix} time \\ space_1 \\ space_2 \end{pmatrix} = \begin{pmatrix} t(i,j,k) \\ s_1(i,j,k) \\ s_2(i,j,k) \end{pmatrix}, \text{where}$$

$$t(i,j,k) = i + j + k - 2$$

$$s_1(i,j) = (i + j - \lceil n/2 \rceil - 1) \bmod n$$

$$s_2(i,j) = \begin{cases} i - j, & \text{if } n \text{ is even or } \lceil n/2 \rceil + 1 \le i + j \le \lceil 3n/2 \rceil \\ i - j + 1, & \text{if } n \text{ is odd and } \lceil n/2 \rceil + 1 > i + j \\ i - j - 1, & \text{if } n \text{ is odd and } i + j > \lceil 3n/2 \rceil \end{cases}$$

**Proof of optimality:** [5].

## 3.2 Gaussian Elimination

### 3.2.1 Problem

We examine the dag family for the Gaussian elimination algorithm. One use of this algorithm is to transform a linear system $Ax = b$ into an upper-triangular system $Ux = c$. In this algorithm, there is no pivoting and the vector $b$ is transformed as well as the matrix $A$. This dag is a subgraph of an $n \times (n+1) \times n$ rectilinear mesh. An example dag is shown in Figure 2, for $n = 5$.

The Gaussian elimination dag can be defined as follows. $G_n = (N, A)$, where

- $N = \{(i,j,k) \mid 0 \le i \le n-1,\ 0 \le j \le n,\ 0 \le k \le \min(i,j)\}$.
- $A = \{[(i,j,k),(i',j',k')] \mid (i,j,k) \in N,\ (i',j',k') \in N$ and exactly 1 of the following conditions holds:
  1. $i' = i+1,\ j' = j,\ k' = k$;
  2. $j' = j+1,\ i' = i,\ k' = k$;
  3. $k' = k+1,\ i' = i,\ j' = j.\}$

### 3.2.2 Lower Bound

The longest directed path in the dag clearly has $(n+1)+(n-1)+(n-1) = 3n-1$ nodes. Any time-minimal schedule of the Gaussian elimination dag $G_n$ requires at least $\lceil n^2/4 + n/2 \rceil$ processors.

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

$$\mathbf{a} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{20}$$

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 that `DiophantineGF.m` produces for the coefficient of $t^N$ in the expansion of this function is

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

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

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

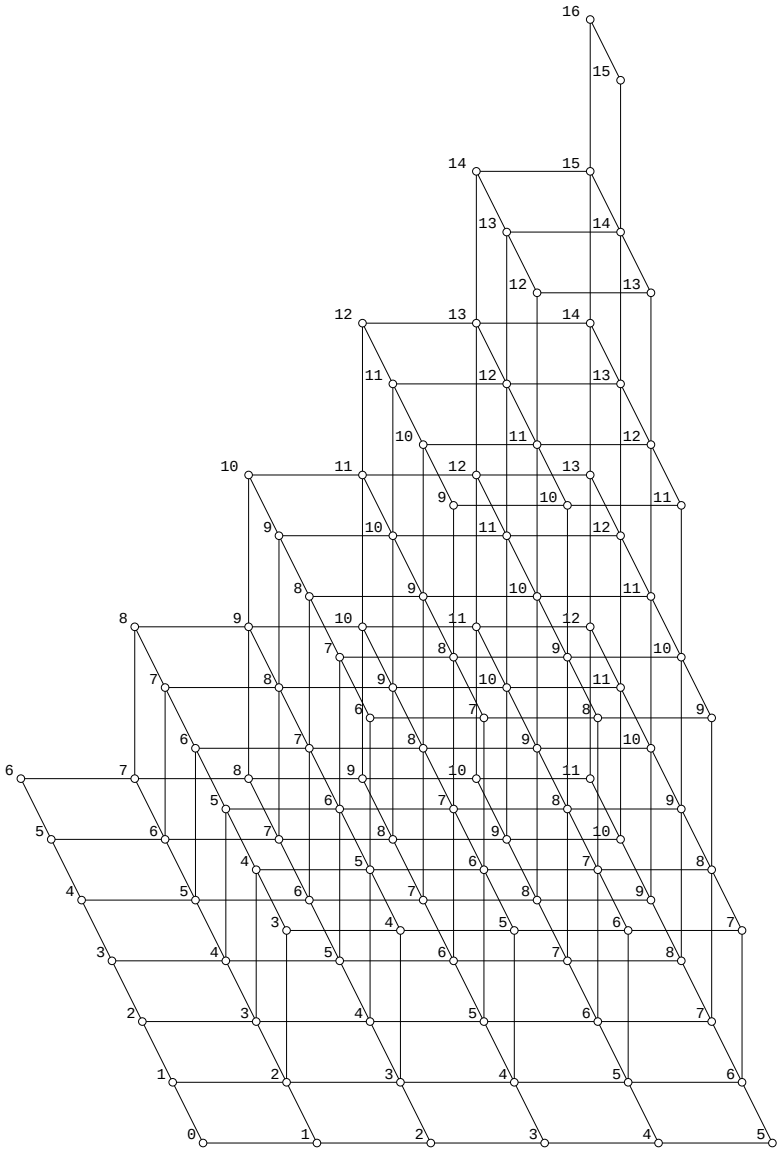

Figure 2: *The Gaussian elimination dag, for* $n = 5$.

Simplifying the other binomial coefficients in (21), we get the lower bound for $n = 2N$ as

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

which can be combined into $\lfloor \frac{2N^2-N}{2} \rfloor$ for $n = 2N$. When $n = 2N + 1$, $\mathbf{c} = [-1, -1, 0, -1, 0]^t$ 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,

$$\begin{aligned}(C[(N-1)/2, 0] - 3C[(N-3)/2, 0] + 2C[(N-2)/2, 0])/8 \quad + \\ (3C[N-2, 2] - 11C[N-1, 2] + 17C[N, 2] + 7C[N+1, 2])/8,\end{aligned}$$

we obtain

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

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 $n$.

### 3.2.3 Upper Bound

A. Benaini and Yves Robert [3, 2] presented a processor-time-minimal solution for Gaussian elimination via an example solution for $n = 9$, which is generalizable for odd $n$. The mapping below is a formal generalization of their approach, and is valid for all $n \in \mathcal{N}$.

We map the 3D mesh onto a 2D mesh of processors with map $m : N_n \mapsto \mathcal{N}^3$. Given a mesh node $(i, j, k) \in N_n$, $m(i, j, k)$ produces a time step (its first component), and a processor location (its last 2 components). The map $m$ can be defined as follows.

$$m \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} time \\ \pi_1 \\ \pi_2 \end{pmatrix} = \begin{pmatrix} \tau(i,j,k) \\ \pi_1(i,j) \\ \pi_2(i,j) \end{pmatrix}, \text{where}$$

$$\tau(i,j,k) = i + j + k,$$

$$\pi_1(i,k) = \begin{cases} \lfloor n/2 \rfloor - i + k & \text{if } i < \lceil n/2 \rceil - 1 \\ i - \lceil n/2 \rceil & \text{if } i \geq \lceil n/2 \rceil - 1 \text{ and } k \geq \lceil n/2 \rceil \text{ and } n \text{ even} \\ i - \lceil n/2 \rceil + 1 & \text{if } i \geq \lceil n/2 \rceil - 1 \text{ and } (k < \lceil n/2 \rceil \text{ or } n \text{ odd}) \end{cases}$$

$$\pi_2(i,k) = \begin{cases} i + 1 & \text{if } i < \lceil n/2 \rceil - 1 \\ k \bmod \lceil n/2 \rceil & \text{if } i \geq \lceil n/2 \rceil - 1 \end{cases}$$

The geometrical interpretation of this mapping is as follows:

Each $j$-column of nodes is computed by the same processor. Each processor computes 1 to 3 columns of nodes.

We divide the columns into three regions: two triangles and a rectangle as shown in Figure 3. The rectangular region defines the processor space: That is, every column of nodes in the rectangle will be computed by a distinct processor. We then map the remaining columns onto these processors.

The top triangle is mapped using a simple mod function, shown in the term $\pi_2(i,k) = k \bmod \lceil n/2 \rceil$. The other triangle, labeled $ABC$ in Figure 3, is fitted into the top right portion of the rectangle after linearly transforming it. In Figure 3, triangle $abc$ is the transformed triangle, with column $A$ mapping to the same processor as column $a$, etc.

## 3.3 Transitive Closure

### 3.3.1 Problem

Aho, Hopcroft, and Ullman [1] define transitive closure as follows:

> *"Suppose our cost matrix $C$ is just the adjacency matrix for the given digraph. That is, $C[i,j] = 1$ if there is an arc from $i$ to $j$, and 0 otherwise. We wish to compute the matrix $A$ such that $A[i,j] = 1$ if there is a path of length one or more from $i$ to $j$, and 0 otherwise. $A$ is often called the* transitive closure *of the adjacency matrix."*

Perhaps the best known parallel algorithm for transitive closure is by Guibas, Kung, and Thompson [21, 53], operating on a toroidally connected mesh. This problem has seen progress in the research of Rote [41], Robert and Trystram [40], Benaini, Robert, and Tourancheau [4], and Kung, Lo, and Lewis [30]. This last algorithm (the KLL algorithm) is a clever reindexing of the Floyd-Warshall [55, 15] algorithm (see also [32]). It is the KLL algorithm which is analyzed below.

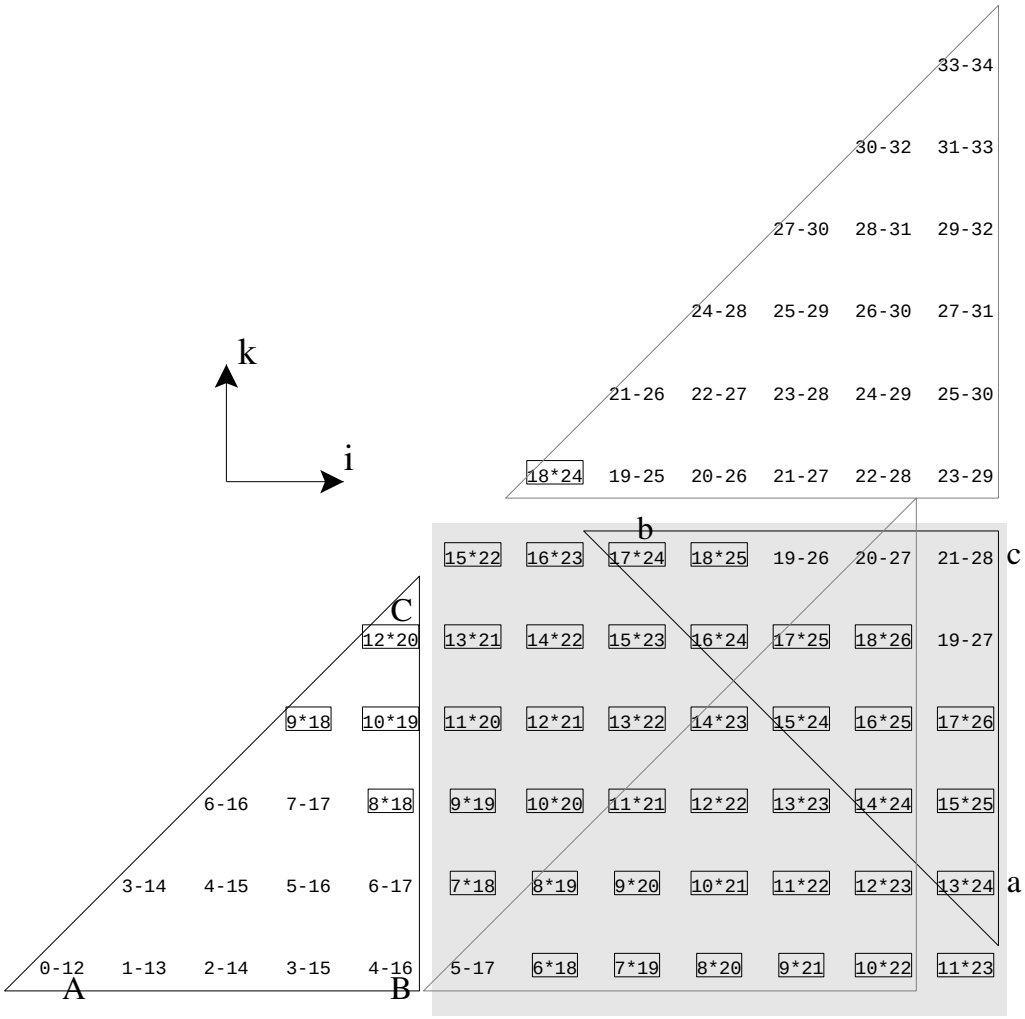


Figure 3: *The Gaussian elimination dag for $n = 12$, projected along the $j$-axis. The rectangular region defines the processor space. The remaining columns, which form two triangles, are mapped into the rectangular region to complete the processor allocation.*

The KLL dependence dag [30] for computing the transitive closure, illustrated in Fig. 4, can be defined as follows.

$G_{tc(n)} = (N, A)$, where

- $N = \{(i,j,k) \mid 1 \leq i,j,k \leq n\}$.
- $A =$

$$\{[(i,j,k),(i',j',k)] \mid \text{where } i' = i+1 \text{ exclusive-or } j' = j+1, \text{ for } 1 \leq i,j,k < n\}$$
$$\bigcup \{[(i,j,k),(i',j',k')] \mid i' = i-1, j' = j-1, k' = k+1 \text{ for } 1 < i,j \leq n,\ 1 \leq k < n\}$$
$$\bigcup \{[(i,j,k),(i',j',k')] \mid i' = i-1, j' = j = n, k' = k+1 \text{ for } 1 < i \leq n,\ 1 \leq k < n\}$$
$$\bigcup \{[(i,j,k),(i',j',k')] \mid i' = i = n, j' = j-1, k' = k+1 \text{ for } 1 < j \leq n,\ 1 \leq k < n\}.$$

### 3.3.2 Lower Bound

The longest directed path in this dag has $5n-4$ nodes (see [30]). Any time-minimal schedule of the $G_{tc}(n)$ dag requires at least $\left\lceil \frac{n^2}{3} \right\rceil$ processors.

### 3.3.3 Upper Bound

**Schedule:** The schedule depends on $n$ mod 3. We show the schedule for the case where $n$ mod $3 = 0$. The remaining cases, as well as the proofs of optimality are found in [46].

The map $m_0 : N \mapsto \mathbf{Z}^3$ (i.e., from nodes to spacetime) can be defined formally as follows.

$$\begin{pmatrix} time \\ space_1 \\ space_2 \end{pmatrix} = \begin{pmatrix} \tau(i,j,k) \\ \pi_1(j,k) \\ \pi_2(j) \end{pmatrix}, \text{where}$$

$$\tau(i,j,k) = 3(k-1) + i + j - 1$$
$$\pi_1(j,k) = k' = \lceil n/2 \rceil - \lfloor \tfrac{j-1}{3} \rfloor + (k - (\lceil n/2 \rceil - \lfloor \tfrac{j-1}{3} \rfloor)) \bmod \tfrac{n}{3}$$
$$\pi_2(j) = j' = j$$

For this mapping, $\tau(i,j,k)$ is found by examining the dag of Fig. 4. It is the earliest time a particular node can be processed. $\pi_1(j,k)$ is the mod $n/3$ function, with an offset to assure that the first processor of the top row is located in the middle $k$ column.

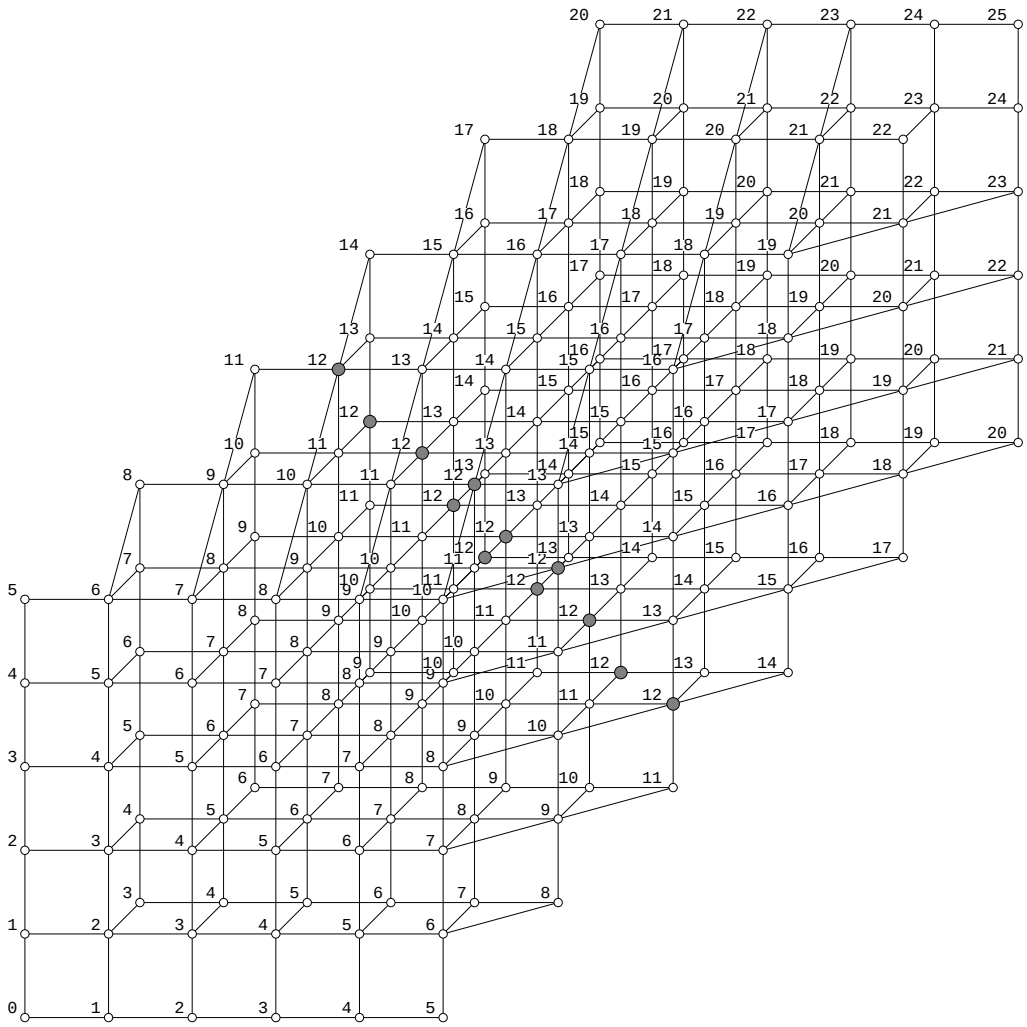

Figure 4: *The dag* $G_{tc}(6)$. *Each node is labeled with its time step in a time minimal schedule. Maximal concurrent sets (with 12 nodes) are associated with time steps 9–18. Such time steps are referred to as* processor maximal. *The black nodes comprise the maximal concurrent set for processor maximal time step 12.*

The mapping is done with a simple mod $\frac{n}{3}$ function applied along the $k$ axis. There are $n/3$ real processors for each $k$-row. For a particular row, the remaining $2n/3$ columns map to the $n/3$ real processors such that $k_{real} \bmod n/3 = k_{remaining} \bmod n/3$. Thus, each real processor handles 3 columns: its first column finishes execution *just before* its second column begins execution, and its second column finishes execution *just before* its third column begins execution (i.e., scheduling constraints are met). For the example in Fig. 5, we use a (mod12/3) function so that each of the remaining $2 \cdot 12/3$ columns per row are mapped to a processor. The connectivity implied by this mapping requires, for example, that the processor assigned to column **A** must communicate directly to the processor assigned to column **D**. To realize these boundary connections, we map the array of Fig. 5 onto the surface of a cylinder.

**Proof of optimality:** [43].

**Processor layout:** The processor layout is shown in Figure 5, as described above.

## 3.4 Tensor Product

### 3.4.1 Upper Bound

The processor lower bound for any time-minimal schedule for this computation was presented in §2: directed mesh clearly has $4n-3$ nodes. Any time-minimal schedule of the 4D mesh requires at least $(2/3)n^3 + n/3$ processors.

We now show that there is a systolic array that achieves this lower bound for processor-time-minimal multiprocessor schedules.

We map the 4D mesh onto a 3D mesh of processors with map $m : N_n \mapsto \mathcal{N}^4$. Given a mesh node $(i,j,k,l) \in N_n$, $m(i,j,k,l)$ produces a time step (its first component), and a processor location (its last 3 components). The map $m$ is defined as follows.

$$m \begin{pmatrix} i \\ j \\ k \\ l \end{pmatrix} = \begin{pmatrix} time \\ \pi_1 \\ \pi_2 \\ \pi_3 \end{pmatrix} = \begin{pmatrix} \tau(i,j,k,l) \\ \pi_1(i,j,k) \\ \pi_2(i,j,k) \\ \pi_3(i,j,k) \end{pmatrix}, \text{where}$$

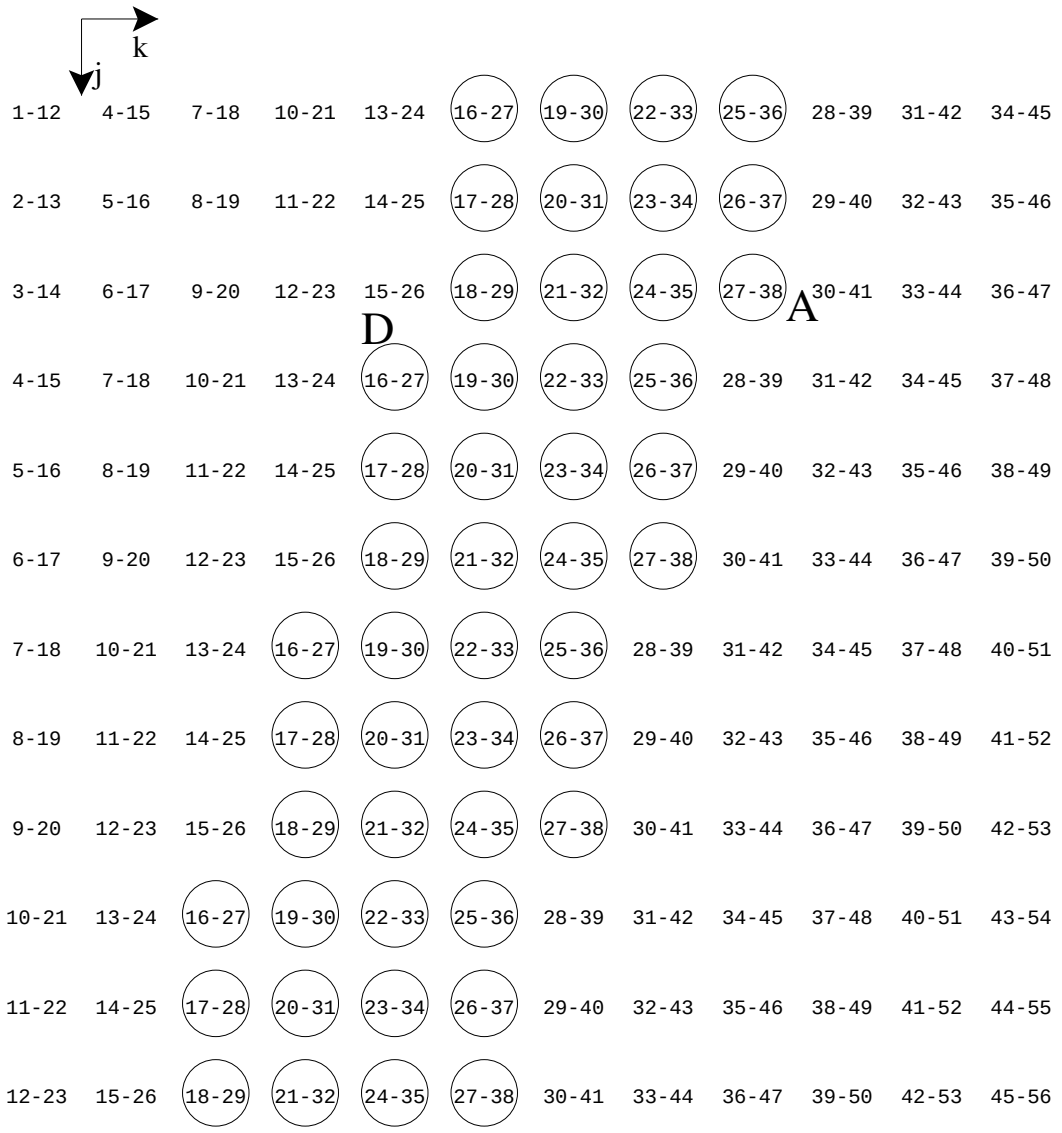


Figure 5: $G_{tc}(12)$, *viewed along the i axis. Each entry (i.e., i-column) is indicated by the interval of time steps for its 12 nodes. Circled i-columns contain a processor maximal time step,* $t_m = 27$. *These circled columns also represent the processors.*

$$\tau(i,j,k,l) = i+j+k+l,$$

$$\pi_1(i,j,k) = i+j+k-1 \bmod n$$

$$\pi_2(i,j,k) = \begin{cases} 2j+k-n+2 & \text{if } i+j+k < n-1 \\ i-j & \text{if } n-1 \le i+j+k \le 2n-2 \\ 2j+k-2n+1 & \text{if } 2n-2 < i+j+k \end{cases}$$

$$\pi_3(i,j,k) = \begin{cases} i+j & \text{if } i+j+k < n-1 \\ k & \text{if } n-1 \le i+j+k \le 2n-2 \\ i+j-n+1 & \text{if } 2n-2 < i+j+k \end{cases}$$

.

We now discuss how this mapping was derived, and also give a geometrical interpretation.

One way to find a processor-time-minimal schedule for a 4-dimensional cubical mesh is as follows:

- First, we assume that the $n$ points $(i,j,k,0)$ to $(i,j,k,n-1)$ (for every fixed $i,j,k$) are mapped to the same processor. This gives us $n^3$ columns of nodes to be mapped.

- As shown previously, there are at least $(2/3)n^3 + n/3$ nodes which must be computed at the same time step, in a time-minimal schedule. Therefore, there are $(2/3)n^3 + n/3$ columns containing these nodes (since the time steps corresponding to each node in a column are unique.)

  We choose these $(2/3)n^3 + n/3$ columns as our processor space. This accomplishes 2 things: (1) It assigns each of these columns of nodes to the processor that computes it, and (2) It determines the shape of the 3-dimensional processor array (before any topological changes).

- We then map all of the remaining $n^3 - (2/3)n^3 + n/3$ columns to the processors, without violating the scheduling constraints.

The mapping $m$ was derived by following the 3 steps above. The last step has many valid solutions. We have chosen one that is convenient.

The mapping $m$ has the following geometrical interpretation:

As previously mentioned, we collapse the 4-dimensional space to 3 dimensions, by only concerning ourselves with $(i,j,k)$ columns, where each of the $n$ points in these columns is mapped to the same processor. For that reason, $l$ is not a factor in the space mapping.

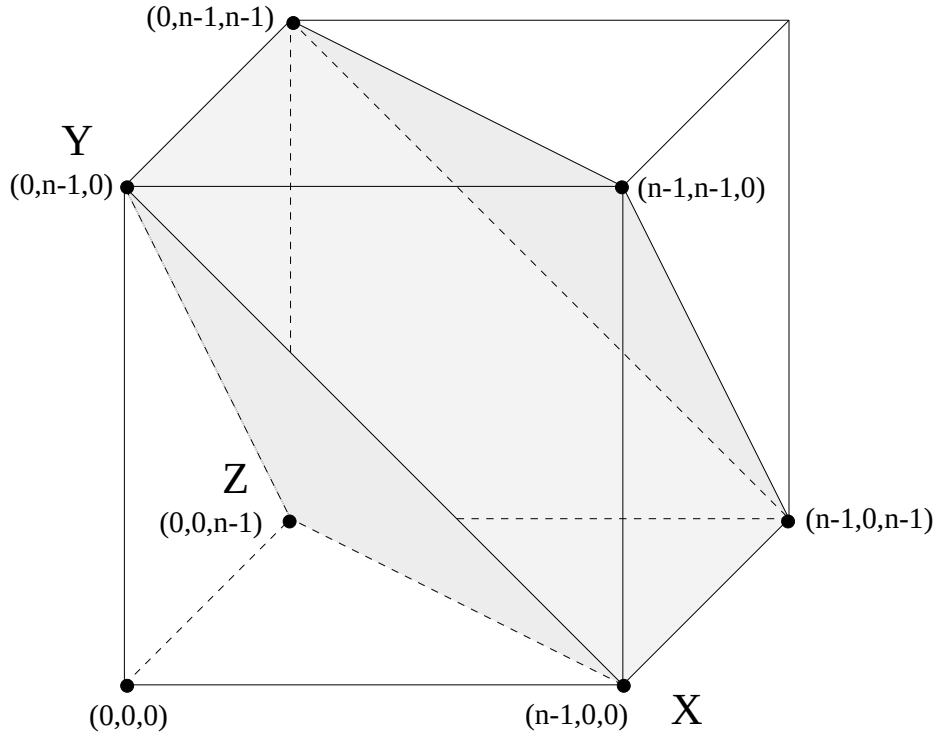


Figure 6: *The 3 regions of the 4D-mesh (after 1 dimension has been collapsed). The middle region, shaded here, is also the processor space. It is inclusively bounded by the hyperplanes* $i+j+k = n-1$ *and* $i+j+k = 2n-2$. *The nodes outside of the shaded region are mapped into it.*

We divide this 3-dimensional cube of columns into 3 regions: the tetrahedron formed by [the convex full of] the columns below the hyperplane $i+j+k = n-1$, the tetrahedron formed by the columns above the hyperplane $i+j+k = 2n-2$, and the remaining middle region. The middle region is the processor space. These 3 regions are shown in Figure 6.

The middle region has 2 triangular faces, along the planes $i+j+k = n-1$ and $i+j+k = 2n-2$. These 2 faces become the bottom and top layers of processors. The planes between these, defined by $i+j+k = c,\ n-1 < c < 2n-2$, make up the remaining layers of processors, for a total of $n$ layers. Columns are mapped to the correct layer by the mapping function $\pi_1(i,j,k)$.

We map the lower tetrahedron into the processor space by first translating it along the vector $(1,1,1)$ until its upper face, defined by the plane $i+j+k=n-2$, lies in the same plane as the middle region's upper face, defined by the plane $i+j+k=2n-2$. We then rotate the tetrahedron 180 degrees about the line $i=j=k$, and translate it again so that the integer points of the tetrahedron lie on integer points of the middle region. This second translation is actually done on a plane by plane basis for the planes $i+j+k=c$ to assure a convenient mapping. These transformations are done by the mapping functions $\pi_2(i,j,k)$ and $\pi_3(i,j,k)$.

The upper tetrahedron is mapped to the processor space similarly, except that it is translated down to the lower part of the middle region.

**Proof of optimality:** [43].

**Processor layout:** The processor layout is shown in Figure 6, as described above.

# 4 Conclusion

Given a nested loop program whose underlying computation dag has nodes representable as lattice points in a convex 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 to lower bounds is the algorithm and its implementation for constructing the generating function from which a formula for the number of these solutions is produced.

Several examples illustrate the relationship between nested loop programs and Diophantine equations, and are annotated with the output of a Mathematica program that implements the algorithm. The algorithmic relationship between the Diophantine equations and the generating function is illustrated with a simple example. Proof of the algorithm's correctness is sketched, while illustrating its steps. The algorithm's time complexity is exponential. However this 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. This leads to some interesting combinatorial questions of rationality and algorithm design based on more general system of Diophantine equations.

Another direction of research concerns optimizing processor-time-minimal schedules: finding a processor-time-minimal schedule with the highest throughput: a *period-processor-time-minimal* schedule. While such a schedule has been found and proven optimal in the case of square matrix product [47], this area is open otherwise. Another area concerns k-dimensional meshes. We have generalized the square mesh lower bounds, yielding a bound for a square mesh of any fixed dimension, $k$. We have not however generalized our upper bound: We have no generalized square mesh schedule that is processor-time-minimal for every $k$ (i.e., for meshes of the form $n^k$, where both $n$ and $k$ are parameters).

# References


[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co, Reading, Mass, 1974.

[2] A. Benaini and Yves Robert. Space-time-minimal systolic arrays for gaussian elimination and the algebraic path problem. *Parallel Computing*, 15:211–225, 1990.

[3] 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*, pages 746–757, Princeton, September 1990. IEEE Computer Society.

[4] Abdelhamid Benaini, Yves Robert, and B. Tourancheau. A new systolic architecture for the algebraic path problem. In John V. McCanny, John McWhirter, and Earl E. Swartzlander Jr., editors, *Systolic Array Processors*, pages 73–82, Killarney, IRELAND, May 1989. Prentice-Hall.

[5] 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).

[6] Peter Cappello and Ömer Eğecioğlu. Processor lower bound formulas for array computations and parametric diophantine systems. *International Journal of Foundations of Computer Science*, 9(4):351–375, 1998.

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

[8] 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*, pages 347–356, Killarney, IRELAND, May 1989. Prentice-Hall.

[9] 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.

[10] 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.

[11] 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*, pages 4–18, Princeton, September 1990. IEEE Computer Society.

[12] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, 1998.

[13] 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.

[14] Bradley R. Engstrom and Peter R. Cappello. The SDEF programming system. *J. of Parallel and Distributed Computing*, 7:201–231, 1989. Listed as "submitted" before Nov. 1987.

[15] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5, June 1962.

[16] 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, pages 454–494. North-Holland, Elsevier Science Publishing Co., New York, NY 10017, 1988.

[17] 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.

[18] José A. B. Fortes and F. Parisi-Presicce. Optimal linear schedules for the parallel execution of algorithms. In *Int. Conf. on Parallel Processing*, pages 322–328, Aug. 1984.

[19] 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.

[20] J. Granata, M. Conner, and R. Tolimieri. Recursive fast algorithm and the role of the tensor product. *IEEE Transactions on Signal Processing*, 40(12):2921–2930, December 1992.

[21] Leonidas J. Guibas, H.-T. Kung, and Clark D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Caltech Conf. on VLSI*, pages 509–525, 1979.

[22] Daniel S. Hirschberg. Recent results on the complexity of common-subsequence problems. In David Sankoff and Joseph B. Kruskal, editors, *Time warps, string edits, and macromolecules : the theory and practice of sequence comparison.* Addison-Wesley, Reading, Mass, 1983.

[23] Oscar H. Ibarra and Michael Palis. VLSI algorithms for solving recurrence equations and applications. *IEEE Trans. on Acoust., Speech, and Signal Processing*, ASSP-35(7):1046–1064, July 1987.

[24] Guo jie Li and Benjamin W. Wah. The design of optimal systolic algorithms. *IEEE Trans. Comput.*, C-34(1):66–77, 1985.

[25] 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.

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

[27] E.V. Krishnamurthy, M. Kunde, M. Schimmler, and H. Schroder. Systolic algorithm for tensor products of matrices: implementation and applications. *Parallel Computing*, 13(3):301–308, March 1990.

[28] E.V. Krishnamurthy and H. Schroder. Systolic algorithm for multivariable approximation using tensor products of basis functions. *Parallel Computing*, 17(4-5):483–492, July 1991.

[29] 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.

[30] Sun-Yuan Kung, Sheng-Chun Lo, and Paul S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Trans. Comput.*, 36(5):603–614, May 1987.

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

[32] Wei-Ming Lin and V. K. Prasanna Kumar. A note on the linear transformation method for systolic array design. *IEEE Trans. Comput.*, 39(3):393–399, 1990.

[33] Basile Louka and Maurice Tchuente. 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*, pages 264–274, Killarney, IRELAND, May 1989. Prentice-Hall.

[34] Percy A. MacMahon. The diophantine inequality $\lambda x \geq \mu y$. In George E. Andrews, editor, *Collected Papers, Vol. I, Combinatorics*, pages 1212–1232. The MIT Press, Cambridge, MASS, 1979.

[35] Percy A. MacMahon. Memoir on the theory of the partitions of numbers- part ii. In George E. Andrews, editor, *Collected Papers, Vol. I, Combinatorics*, pages 1138–1188. The MIT Press, Cambridge, MASS, 1979.

[36] Percy A. MacMahon. Note on the the diophantine inequality $\lambda x \geq \mu y$. In George E. Andrews, editor, *Collected Papers, Vol. I, Combinatorics*, pages 1233–1246. The MIT Press, Cambridge, MASS, 1979.

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

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

[39] 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.

[40] Yves Robert and D. Trystram. An orthogonal systolic array for the algebraic path problem. In *Int. Workshop Systolic Arrays*, 1986.

[41] G. Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219, 1985.

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

[43] Chris Scheiman. *Mapping Fundamental Algorithms onto Multiprocessor Architectures.* PhD thesis, UC Santa Barbara, Dept. of Computer Science, chriss@cs.ucsb.edu, December 1993.

[44] 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*, pages 26–33, Strasbourg, FRANCE, July 1995.

[45] Chris Scheiman and Peter R. Cappello. A processor-time minimal systolic array for transitive closure. In *Proc. Int. Conf. on Application Specific Array Processors*, pages 19–30, Princeton, September 1990. IEEE Computer Society.

[46] 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.

[47] 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.

[48] 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.

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

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

[51] Bernd Sturmfels. *Gröbner Bases and Convex Polytopes.* AMS University Lecture Series, Providence, RI, 1995.

[52] Bernd Sturmfels. On vector partition functions. *J. Combinatorial Theory, Series A*, 72:302–309, 1995.

[53] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Inc, Rockville, MD 20850, 1984.

[54] 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.

[55] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9, Jan 1962.

[56] D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Corvallis, Oregon, December 1993. Also published as IRISA technical report PI 785, Rennes, France, Dec 1993.

[57] Yiwan Wong and Jean-Marc Delosme. Optimization of processor count for systolic arrays. Dept. of Computer Sci. RR-697, Yale Univ., May 1989.

[58] 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*, pages 275–282. IEEE Computer Society Press, Beverly Hills, March 1992.