Automatic Processor Lower Bound Formulas for Array Computations

Peter Cappello
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106, USA
cappello@cs.ucsb.edu

Ömer Eğecioğlu
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106, USA
omer@cs.ucsb.edu

Abstract

In the directed acyclic graph (dag) model of algorithms, consider the following problem for precedence-constrained multiprocessor schedules for array computations: Given a sequence of dags and linear schedules parameterized by n, compute a lower bound on the number of processors required by the schedule as a function of n. This problem is formulated so that 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. Generating function methods are then used for constructing a formula for the numbers d_n. We implemented this algorithm as a Mathematica program. This paper is an overview of the techniques involved and their applications to well-known schedules for Matrix-Vector Product, Triangular Matrix Product, and Gaussian Elimination dags. Some example runs and automatically produced symbolic formulas for processor lower bounds by the algorithm are given.

1 Introduction

Parallel execution of array computations (uniform recurrence equations) [17] has been studied extensively [18, 14, 15, 19, 20, 16]). In such computations, the computational tasks are viewed as the nodes of a dag; arcs represent data dependencies. 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 the constraints:

Causality: A node is computed only after its children have been computed: $(u, v) \in A \Rightarrow \tau(u) < \tau(v)$.

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 [3, 11, 9] examine the dag family for matrix product; Louka and Tchuente [8] examine the dag family for Gauss-Jordan elimination; Scheiman and Cappello [10] 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 [4] 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 [13], and Shang and Fortes [12]. 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 [16] show that such schedules are close to optimal, even when the constraint of linearity is relaxed.

In [9], 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., [17, 18, 19, 7, 6, 20, 4, 12, 13]; see Fortes, Fu, and Wah [5] 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. [23, 22] present a more general and uniform technique for deriving such lower bounds: Given a parameterized dag

family and a correspondingly parameterized linear schedule, a *formula* for a lower bound on the number of processors required by the schedule is computed. 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.

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 $\mathbf{Az} = n\mathbf{b} + \mathbf{c}$, 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 the linear system above, $\mathbf{A}$ 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 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.

2 Examples from Array Computations

2.1 $n \times n$ Matrix-Vector Product

An algorithm for $n \times n$ matrix-vector product is given in the following procedure. 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 $v[0], v[1], \ldots, v[n-1]$.

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:

$$\begin{array}{rcl} 0 \leq & i & \leq n-1 \\ 0 \leq & j & \leq n-1 \end{array} \qquad (1)$$

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

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

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

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

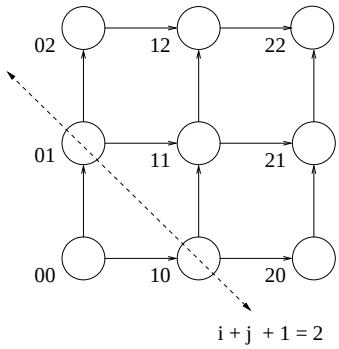

Figure 1. The matrix-vector product dag for $n = 2$.

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 begins in time step 1 with the computation of $N(0, 0)$, and ends 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: $N(0, n-1), N(1, n-2), \ldots, N(n-1, 0)$. 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 (1) is

$$\begin{array}{rcl} i + j & = & n-1 \\ i + s_1 & = & n-1 \\ j + s_2 & = & n-1 \end{array} \qquad (2)$$

The number of non-negative integral solutions to (2) is a processor lower bound for the $n \times n$ *Matrix-Vector Product* problem.

2.2 $n \times n$ 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.

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

$$\begin{array}{rcl} i & \leq & n-1 \\ i \leq & j & \leq n-1 \\ i \leq & k & \leq j\,. \end{array}$$

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

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

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 $\tau \approx \frac{3}{2}n - 1$. For simplicity, assume 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

$$\begin{array}{rrrrrrr} i & +j & +k & & & = & 3N-2 \\ & j & & +s_1 & & = & 2N-1 \\ & -j & +k & & +s_2 & = & 0 \\ i & & -k & & +s_3 & = & 0 \end{array} \qquad (3)$$

Therefore, for $n = 2N$ a lower bound for the number of processors needed for the $n \times n$ *Triangular Matrix Product* problem is the number of solutions of (3).

2.3 Gaussian Elimination without Pivoting

An algorithm for performing Gaussian elimination on an $n \times n$ matrix M is given below.

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 (i,j,k) satisfying the constraints

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

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

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

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 at the halfway point: $\tau \approx \frac{3}{2}n-1$. Again, we only present the case of even n because of space considerations. For$n = 2N$, we can take τ to be $3N-1$. Adding the schedule constraint to system we already have, we obtain the augmented Diophantine system

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

Here $\mathbf{b} = [3,0,2,0,2]^t$ and $\mathbf{c} = [-2,-1,-1,-1,-1]^t$. 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 with $n = 2N$ is the number of solutions of the above system.

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 (parametrized by n) Diophantine system of the form $\mathbf{A}\mathbf{z} = n\mathbf{b}+\mathbf{c}$ where $\mathbf{A}$ is some $r \times s$ integral matrix, $\mathbf{b}$ and $\mathbf{c}$ are r-dimensional integral vectors.

3 The General Formulation

Next, we describe how to use a Mathematica program to to automatically construct a formula for the number of lattice points inside a linearly parameterized family of convex polyhedra. The algorithm for doing this and its implementation have been reported in detail in [our paper].

First of all, the general setting exemplified by the problems in the preceding section 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. 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 above form $f(t)$ is always a rational function, and can be computed symbolically [23, 22]. The Mathematica program `DiophantineGF.m` we have written for this computation[1] 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 also produces a formula for the numbers d_n.

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

4 Mathematica Runs

The program `DiophantineGF.m` requires three arguments $\mathbf{a}, \mathbf{b}, \mathbf{c}$ of the Diophantine system

$$\mathbf{a}\mathbf{z} = n\mathbf{b} + \mathbf{c} \tag{4}$$

as input. 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 (4). 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` on the the array computation problems formulated in Section 2.

4.1 $n \times n$ Matrix-Vector Multiplication

For the linear schedule of the *Matrix-vector Product* example, the augmented Diophantine system in (2) can be written in the form (4) where

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

```
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]

             t
Out[4]= ---------
                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 $n \times n$ 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

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

and $\mathbf{c} = [-2, -1, 0, 0\,]^T$ for $n = 2N$. In this 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]
           t
Out[4]= --------
                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 $n(n + 2)/8$ where $n = 2N$. Considering the case $n = 2N + 1$ as well, we find that 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

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

4.3 Gaussian Elimination

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

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

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

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

From this, the program produces the the lower bound for $n = 2N$ as $\lfloor \frac{2N^2 - N}{2} \rfloor$ for $n = 2N$. This also happens to be the formula produced for $n = 2N + 1$. Combining the results, for $n \times n$ Gaussian elimination without pivoting for arbitrary n, we obtain the processor lower bound

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

5 Complexity

The algorithm for the computation of $f(t)$ generates a ternary tree and accumulates the symbolic information obtained at the leaves [our paper]. The number of leaves is exponential in $n = \sum_{\{a_i\}} |a_i|$, where $\{a_i\}$ is the set of coefficients describing the set of Diophantine equations. However, 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 $f(t)$ non-zero?" is already known to be an NP-complete problem [21]. 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: 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 can 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.

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 algorithm's exponential computational complexity should be seen in light of two facts: (1) Deciding if a time step has *any* nodes associated with it is NP-complete; we construct a *formula* for the number of such nodes; (2) 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 leaves open some interesting combinatorial questions of rationality and associated algorithm design: e.g. how to compute the associated generating function of the number of solutions when the right hand side of the main system consists of higher degree polynomials in n, and not just linear.

For details of this presentation, please refer to [23, 22].

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] B. Louka and M. 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*, 264–274, Killarney, IRELAND, May 1989.

[9] 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.

[10] 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.

[11] 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.

[12] W. Shang and J. A. B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6):723–742, 1991.

[13] 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.

[14] P. Cappello and K. 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.

[15] P. Cappello and K. 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, 1984.

[16] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is close to optimal. In J. Fortes, E. Lee, and T. Meng, eds., *Application Specific Array Processors*, 37–46. IEEE Computer Society Press, 1992.

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

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

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

[20] S. V. Rajopadhye, S. Purushothaman, and R. 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, Dec. 1986.

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

[22] P. Cappello, Ö. Eğecioğlu, and C. Scheiman. Processor-time-optimal systolic arrays. *Parallel Algorithms and Applications.*, 15:167–199, 2000.

[23] P. Cappello and Ö. Eğecioğlu. Processor Lower Bound Formulas for Array Computations and Parametric Diophantine Systems. *Int. J. of Foundations of Computer Science.*, 9(4):351-375, 1998.

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