Optimization of Automatically Generated Parallel Programs

ALESSANDRO MARONGIU[1] and PAOLO PALAZZARI[2]
[1]Department of Electronic Engineering
University "La Sapienza" of Rome
Via Eudossiana 18 - 00184 ROME
ITALY

[2]HPCN Project
ENEA - C.R. Casaccia
Via Anguillarese 301, S.P.100, 00060 Santa Maria di Galeria (ROME)
ITALY

Abstract: - The automatic synthesis of parallel programs, starting from high level specifications of an iterative algorithm, requires a space time transformation assigning to each statement the time when and the processor where it is executed. Time space transformations are based on not singular matrices. An additional restriction on these matrices, unimodularity, ensures a simpler code generation and avoids run time overhead due to control code. In this paper we show that unimodular matrices are equivalent to not-unimodular ones in the parallelism extraction and we give an optimization method which, exploring the space of unimodular matrices, find a (near) optimal space time transformation for a program expressed through a System of Affine Recurrence Equations (SARE).

Key-Words: - Automatic Parallelization, polytope model, SARE, Simulated Annealing, Unimodular matrices. *IMACS/IEEE CSCC'99 Proceedings,* Pages:2141-2146

1 Introduction

In recent years the automatic synthesis of parallel programs from iterative algorithms has received considerable attention. A lot of parallelization techniques have been developed which, starting from an high level description of an iterative algorithm, generate parallel versions of the original program (see [1] and [2] for an extensive bibliography). Almost all these synthesis techniques are based on the polytope model [3],[4]. In this model the iteration space is represented through a bounded convex subset of the integer lattice Z^N (i.e. a polytope) and the dependencies between statements are represented through dependencies vectors. The parallelization is then attained by a couple of affine transformation functions [1]:

- the *timing function,* which gives information about the time execution of a statement;
- the *allocation function,* which gives information about the processor that will execute a statement.

Timing and allocation function form the space-time transformation which is characterized by an integer and not singular matrix. A further restriction, unimodularity [5] [8] [11], may be applied to the trasformation matrix. This property, although considered not essential, allows a simpler code generation and a smaller control run time overhead (which is high desiderable) [6]. However using unimodular matrices has two main disadvantages:

- the set of applicable space-time transformations is restricted;
- the generation of an unimodular matrix is more difficult because more constraints have to be considered.

In this paper we will:

- show that the unimodularity constraint does not affect the parallelism extracted from an algorithm being unimodular and not unimodular matrices equivalent in parallelism extraction.
- give a method to build a space-time transformation using unimodular matrices;
- give a method to explore the space of the unimodular admissible matrices allowing the searching for the (near) optimal space-time transformation.

Throughout the paper we will assume the theoretical framework presented in [1].

2 Algorithm Model

We briefly introduce the iterative algorithm model described through SARE (System of Affine Recurrence Equations). More detailed information can be found in [1], [14]. A SARE is described by a certain number of equations E:

$$X(z)=f(...,Y[\rho(z)],...) \text{ with } z\in I_E \quad (1)$$

where:

- X and Y are multi-dimensional array.
- $z\equiv(i\ j\ ...\ k)^T$ is the $N\times1$ iteration vector. It represents the set of indices of X: $X(z)$ means $X[i,j,...k]$. Moreover z represents the statement $S_X(z)$ which computes $X(z)$;
- I_E is the iteration space related to (1). I_E is a convex subset of Z^N (i.e. a polytope). The convex union of all I_E related to all equations forms the global iteration space polytope I.
- $\rho(z)$ is an affine index mapping function. It defines a flow dependence between the points $\rho(z)$ and z (computation of $X(z)$ requires $Y[\rho(z)]$). In the SARE model $\rho(z)$ is affine, i.e. $\rho:Z^N\rightarrow Z^{N'}\equiv Rz+r$, being R an $N\times N$ matrix and r an $N\times1$ vector.
- f is a function used to compute $X(z)$.

The main advantage of SARE model is that analysis of algorithm dependencies, along with extraction of parallelism and mapping optimization (memory and scheduling), can be done automatically at compile time. Such an analysis is performed through dependence vectors defined as

$$d_{Y,\rho(z)}=z-\rho(z) \quad (2)$$

3 Space Time Transformation

Parallelization of a SARE implies a space-time transformation which, for each point $z\in I\subset Z^N$, gives the time and the processor where the corresponding statement will be executed. The space-time transformation is composed by [1]

- the timing function $\tau(z)$, which returns when each statement $S_X(z)$ will be executed, and
- the allocation function $\pi(z)$ which returns the processor on which $S_X(z)$ will be executed.

These two functions must be:

- *admissible*: they must guarantee the semantics of the algorithm maintaining dependence relations, and
- *compatible*: no more than one statement can be executed on a processor at the same time.

Timing function is chosen in the set of n-dimensional affine functions [1]:

$$\tau(z)\equiv\Lambda z+\alpha \quad (3)$$

where Λ (*timing matrix*) is an integer $n\times N$ matrix and α is an integer $n\times1$ vector. $\tau(z)$ assigns to every statement $S_X(z)$ $\forall z\in I$ a value $\tau(z)\in Z^n$ which gives the time scheduling of $S_X(z)$. In order to determine univocally the time allocation of a given $S_X(z)$, values $\tau(z)$ have to be totally ordered. We define the lexicographical ordering on $\tau(z)$ values: given two $n\times1$ vectors x and y, they are (strictly) lexicographically ordered $x\underline{<<}y$ ($x<<y$) if exists a k so that $x(i)=y(i)$ for $0\le i\le k<n$ and $x(k+1)\le y(k+1)$ ($x(k+1)<y(k+1)$). k is the ordering depth.

When $\tau(z)$ assigns the same value to several points $z\in I$, the corresponding statements will be executed simultaneously. In [1] is shown that the set of points with a same $\tau(z)$ value belong to a *timing surface TS* defined as $TS=\{z\in I|\tau(z)=\tau_0\}$.

TS are m-dimensional sets generated by the kernel of matrix Λ, $Ker(\Lambda)$. Given $\tau(z)$, m is the degree of parallelism extracted by $\tau(z)$ from the algorithm. Two timing functions Λ_1 and Λ_2 extract the same parallelism if they have the same timing surfaces, i.e. if $Ker(\Lambda_1)\equiv Ker(\Lambda_2)$; in such a case the number of concurrent operations for the two timing function is the same.

In order to guarantee admissibility, $\Lambda d_{Y,\rho(z)}>>0$ must be verified for each dependence vector [1].

Allocation function $\pi(z)$ returns information about processor executing $S_X(z)$. Given an n-dimensional timing function, statements to be executed concurrently, i.e. on different processors, belong to an m-dimensional set; so we choose as multiprocessor architecture a set of processors placed on an m-dimensional grid. Each processor is uniquely identified by a set of coordinates $(p_1\ p_2\ ...\ p_m)^T$ so, given a point $z\in I$, $\pi(z)$ returns the coordinates of the processor executing $S_X(z)$. $\pi(z)$ is chosen in the set of m-dimensional affine functions:

$$\pi(z)\equiv\Sigma z+\beta \quad (4)$$

where Σ (*allocation matrix*) is an integer $m\times N$ matrix and β is an integer $m\times1$ vector. In [1] is shown that the set of points projected onto a given processor are found through the kernel of matrix Σ $Ker(\Sigma)$.

By composing $\tau(z)$ and $\pi(z)$ we have the space-

time transformation T(z) [1]:

$$T(z) = \begin{pmatrix} \tau \\ \pi \end{pmatrix} = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix} z + \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = Tz + \gamma \qquad (5)$$

where $T = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix}$ and $\gamma = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$

Compatibility of transformation is assured if matrix T is not singular [6].

The new set of coordinates $(\tau\ \pi)^T$ introduced by T(z) gives when and where a statement $S_X(z)$ is executed. Because $T(z): z \rightarrow (\tau\ \pi)^T$, we have $S_X(z) \rightarrow S_X(\tau,\pi)$, i.e. the statement $S_X(z)$ is executed on processor with coordinates π and scheduled at geometrical time τ. The space-time coordinate system also distributes variables among processors. Because $T(z): z \rightarrow (\tau\ \pi)^T$,we have $X(z) \rightarrow X(\tau,\pi)$, i.e. variable $X(z)$ is stored in the local memory of processor π and addressed through vector τ: $X(\tau)$.

The set of statements to execute on a given processor π is generated through the nesting of n sequential loops [3] [4].

In the space-time coordinates system dependence vectors are transformed too according as in the following:

$$d_{T,Y,\rho(z)} = T d_{Y,\rho(z)} = (\Lambda d_{Y,\rho(z)}, \Sigma d_{Y,\rho(z)}) = (d_\tau, d_\pi) \quad (6)$$

where d_τ is a $n \times 1$ vector and d_π is a $m \times 1$ vector.

It is easy to see that d_τ is lexicographical positive due to the admissibility condition.

While in the original space a dependence vector d defines a flow dependence between points $\rho(z)=z_0$ and z_0+d, in the transformed space it defines a couple of dependencies on time components and processor components. In fact, given $T(z): z \rightarrow (\tau\ \pi)^T$, we have $z_0 \rightarrow (\tau_0\ \pi_0)^T$ and $z_0 + d \rightarrow (\tau_0\ \pi_0)^T + (d_\tau\ d_\pi)^T = (\tau_0 + d_\tau\ \ \pi_0 + d_\pi)^T$.

Communications are generated by the processor part of the dependence vector. If d_π is not null, operand required by statement $S_X(\tau_0 + d_\tau, \pi_0 + d_\pi)$, computed in processor $\pi_0 + d_p$, is stored in the processor π_0. So communications have to be executed in order to transfer operands from storage processor π_0 to computing processor $\pi_0 + d_\pi$. If d_π is null, no communication is performed because statement and operand are in the same processor. Being SARE a single assignment computational model, it has the effect to produce a great waste of memory due to the impossibility to reuse memory locations containing values not more used. Automatic memory optimization can be performed by analyzing the time part of the transformed dependence vector as shown in [1] and [7].

Completion time, communications to be performed and memory requirements of the generated parallel program depend on the matrix T of transformation function.

4 Choosing Matrix *T*

If matrix T is chosen as a generic integer not singular matrix, T(z) projects original iteration space I, defined on the Z^N lattice, onto a target iteration space I_T defined on a lattice L which generally differs from Z^N [8]. Dealing with L causes run time overhead because extra code must be inserted to handle transformed loop indices [6] and to avoid the memory wasting caused by a not dense iteration space. On the contrary, if T is an unimodular not singular, T(z) projects iteration space I onto the target iteration space I_T which are defined on the same lattice Z^N avoiding the before mentioned run time overhead causes.

Moreover the using of a not unimodular matrix T enlarges the set of space time transformation but this enlargement is useless. In fact, through the Hermite Normal Form (HNF) [8], we can write $T=HT_U$ where

- H is a non singular lower triangular matrix with positive diagonal elements
- T_U is an unimodular matrix.

Matrix T and T_U are equivalent as explained in the following

Theorem: Given a matrix T and the corresponding matrix T_U defined from the HNF we have that:

1. *they extract the same parallelism;*
2. *they have the same admissibility properties.*

Proof

Being T the composition of a timing and an allocation matrix, we rewrite the HNF as:

$$T = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix} = \begin{pmatrix} H_{11} & 0 \\ H_{21} & H_{22} \end{pmatrix} \begin{pmatrix} \Lambda_U \\ \Sigma_U \end{pmatrix} \qquad (7)$$

where $T_U = \begin{pmatrix} \Lambda_U \\ \Sigma_U \end{pmatrix}$, H_{11} (H_{22}) is an $n \times n$ ($m \times m$) lower triangular not singular matrix with positive diagonal elements and H_{21} is an $m \times n$ matrix.

From (7) we have:

$$\Lambda = H_{11} \Lambda_U \qquad (8)$$

$$\Sigma = H_{21} \Lambda_U + H_{22} \Sigma_U \qquad (9)$$

Being the degree of parallelism individuated by

the $Ker(\Lambda)$, we show that $Ker(\Lambda) \equiv Ker(\Lambda_U)$.

On the basis of kernel definition, $\Lambda k=0$ $\forall k \in Ker(\Lambda)$. From expression (8) we have $\Lambda k = H_{11}\Lambda_U k = 0 \quad \forall k \in Ker(\Lambda)$. As H_{11} is not singular, $H_{11}\Lambda_U k=0 \Leftrightarrow \Lambda_U k=0 \ \forall k \in Ker(\Lambda)$. So $\Lambda k=0 \Leftrightarrow \Lambda_U k=0 \ \forall k \in Ker(\Lambda)$ and hence $Ker(\Lambda) \equiv Ker(\Lambda_U)$.

Given a dependence vector d, on the basis of admissibility condition $d_\tau = \Lambda d >> 0$ with a lexicographical ordering depth k. We show that $\Lambda d >> 0 \Leftrightarrow \Lambda_U d >> 0$.

From (8) $d_\tau = \Lambda d = H_{11}\Lambda_U d = H_{11} d_{\tau U} >> 0$ where $d_{\tau U} = \Lambda_U d$. Being H_{11} lower triangular with positive diagonal elements and $d_\tau(1)=d_\tau(1)=...= d_\tau(k)=0$ and $d_\tau(k+1)>0$, we have:

$d_\tau(1)=H_{11}(1,1)d_{\tau U}(1)=0 \Rightarrow d_{\tau U}(1)=0;$

$d_\tau(2)=H_{11}(2,1)d_{\tau U}(1)+H_{11}(2,2)d_{\tau U}(2)=0$
$\Rightarrow d_{\tau U}(2)=0;$
...

$d_\tau(k)=H_{11}(k,1)d_{\tau U}(1)+H_{11}(k,2)d_{\tau U}(2)+...+$
$+H_{11}(k,k)d_{\tau U}(k)=0 \Rightarrow d_{\tau U}(k)=0;$

$d_\tau(k+1)=H_{11}(k+1,1)d_{\tau U}(1)+H_{11}(k+1,2)d_{\tau U}(2)+..+$
$+H_{11}(k+1,k)d_{\tau U}(k)+H_{11}(k+1,k+1)d_{\tau U}(k+1)>0$
$\Rightarrow d_{\tau U}(k+1)>0;$

So the admissibility is preserved.

T and T_U generally induce different mapping on processor space because, due to (9), Σ and Σ_U may generate different kernel: $Ker(\Sigma) \neq Ker(\Sigma_U)$. From (9) it is easy to prove that T and T_U generate the same mapping ($Ker(\Sigma) = Ker(\Sigma_U)$) on processor space if $H_{21}=0$. In fact we have: $\Sigma k = 0 \quad \forall k \in Ker(\Sigma)$. From expression (9), if $H_{21}=0$, we have $\Sigma k = H_{22}\Sigma_U k = 0$ $\forall k \in Ker(\Sigma)$. As H_{22} is not singular, $H_{22}\Sigma_U k=0 \Leftrightarrow \Sigma_U k=0 \ \forall k \in Ker(\Sigma)$. So $\Sigma k=0 \Leftrightarrow \Sigma_U k=0$ $\forall k \in Ker(\Sigma)$ and hence $Ker(\Sigma) \equiv Ker(\Sigma_U)$.

5 Unimodular Admissible Matrices

Hermite developed an algorithm [11] to generate an $N \times N$ unimodular matrix from:

- an $(N-1)\times(N-1)$ unimodular matrix M_{N-1};
- N integer numbers $t_{1,1}, t_{1,2}, ..., t_{1,N}$ with Greatest Common Divisor (GCD) $GCD(t_{1,1}, t_{1,2}, ..., t_{1,N})=1$. These numbers will be the first row of the unimodular matrix.

Furthermore Hermite demonstrated that the space of unimodular matrices is closed with respect to his algorithm: all unimodular matrices can be built with this algorithm.

As particular case of Hermite algorithm, we report the following procedure to build up an unimodular matrix.

Procedure BuildUnimodular(
  input: x integer N-vector with $GCD(x)=1$
  output: T $N \times N$ unimodular integer matrix)
begin
  for k:=1 **to** N **do** $T(1,k)=x(k)$
  compute a_1, a_2 and π_1, being
    $a_1 t_{1,1} - a_2 t_{1,2} = \pi_1 = GCD(t_{1,1}, t_{1,2})$
  $T(2,1):=a_1$
  $T(2,2):=a_2$
  for k:=3 **to** N **do** $T(2,k)=0$
  for i:=3 **to** N **do**
    compute b_{i-2}, a_i and π_{i-1}, being
      $b_{i-2}T(1,i) - a_i\pi_{i-2} = \pi_{i-1} = GCD(T(1,i),\pi_{i-2})$
    for k:=1 **to** i-1 **do** $T(i,k)=b_{i-2}(T(1,k)/\pi_{i-2})$
    $T(i,i):=a_i$
    for k:=i+1 **to** N **do** $T(i,k):=0$
  end for (i)
end.

The extended Euclid's algorithm [12] can be used to compute integer values a, b, c in the integer expression $ax+by = c = GCD(x,y)$.

Given two unimodular $N \times N$ matrices T_1 and T_2, we can always transform T_1 into T_2 through a sequence of the following u-elementary transformations [11]:

- exchange two rows (columns);
- multiply a row (column) by -1;
- sum of row (column) i to row (column) $j \neq i$;
- transpose.

In order to generate admissible transformation matrices T, we demonstrate the following

Theorem: *given a set of dependence vectors d_i $(i=1,2,...,k)$, we solve, in the x unknown, the system $xd_i \geq 1$. The matrix T obtained from procedure* BuildUnimodular(x,T) *is admissible.*

Proof: the first component of Td_i is positive (≥ 1) for each d_i, so surely $\Lambda d_i >> 0$ and hence the transformation is admissible.

The system $xd_i \geq 1$ can be solved through the Polyhedral Library [15].

6 Optimization of the Mapping

Starting from an admissible unimodular matrix, through the u-elementary transformations we are able to explore the space of all the unimodular matrices. In order to optimize the mapping with respect to memory usage and completion time, we must associate a cost to a given mapping matrix T and give an algorithm to explore the mapping matrix space.

We use the Simulated Annealing (SA) [9] algorithm to optimize the mapping. As required by SA, we introduce the New(T) function which returns a new admissible matrix $T'=\begin{pmatrix}\Lambda'\\ \Sigma'\end{pmatrix}$ adjacent (i.e. derived through anyone of the u-elementary operations) to T. The structure of New is the following:

procedure New(
  input: T, d_i (dependence vectors)
  output T')
begin
  repeat
    random select an u-elementary operation;
    T':=u-elementary transformation of T;
  until $\Lambda' d_i >> 0\ \forall i$;
end.

We underline that New procedure always exits because at least one of the u-elementary transformations ensures the maintaining of admissibility. In fact it is easy to verify that matrix T' is admissible if it is obtained from an admissible T by adding the first row to any other row. Indicating with EvaluateCost(T) the function associating to T its cost, the SA algorithm can be written as in the following

procedure SA(
  input: d_i (dependence vectors)
  output T matrix with the minimal cost)
Begin
  x:=solution of system of inequalities $xd_i \geq 1$ [15]
  determine the cooling schedule
        (t_0,r,k, *frozen* condition) as in [10];
  t:=t_0
  T=BuildUnimodular(x)
  cost:=EvaluateCost(T);
  while not *frozen*
    for i:=1 **to** k **do**
      New(T,d_i,T');
      costnew=EvaluateCost(T');
      accept=0,
      if (costnew<cost) accept=1;
      else if exp((cost-costnew)/t)>p accept=1;
      if accept=1 T=T'; cost=costnew;
    end for
    t:=r*t;
  end while
end

The cost returned by the EvaluateCost(T) function must take into account both the execution time and the memory required by the parallelized program. We have chosen the following expression:

$$\text{cost} = w_s \frac{T_{exe}}{\min\left(T_{exe}(k)\right)} + w_m \frac{Mem}{\min\left(Mem(k)\right)}$$

where

- w_S and w_M are weighting coefficients,
- T_{exe} is the (estimated) execution time expressed as weighted sum of the computation time and communication time required by the parallelized algorithm.
- $min(T_{exe}(k))$ and $min(Mem(k))$ are the minimum values of completion time and memory requirements found until optimization step k, i.e. they are the actual minimum values which are updated whenever a new minimum is found.

Notice that $min(T_{exe}(k))$ and $min(Mem(k))$ may not correspond to the same projection. The cost function varies during the first phase of the optimization process, but it 'freezes' as the optimization process goes on.

Coefficients w_S and w_M are used to balance the relative weight of execution time and memory allocation: they can be used to move the search from mappings very efficient in time (but with high memory requirements) to mappings which need few memory space (but spend much time in computations). Usually we adopt w_S and w_M values which give and intermediate behavior, i.e. which achieve a trade off between efficiency in time and in memory. The lower bound for the cost function is $w_S + w_M$.

A typical optimization through the presented SA algorithm requires about 5 minutes on a Pentium II based machine.

7 Results

Using a parallelizing tool developed by us on the basis of [1], we successfully implemented our optimization technique to automatically generate parallel codes for the APE100 Quadrics SIMD machine [13] (we used the configuration with 128 processing nodes).

In order to show the experimental results of the optimization technique we use the SARE expression of block matrix-matrix multiplication as test example. Given a two matrices A and B represented through $q \times q$ blocks of $r \times r$ elements, the generic elements of matrix A and B is A(i,j,ii,jj) and B(i,j,ii,jj) where i and j indicate the block and ii and jj indicate the element within the block. The product C=AB is expressed through the six-dimensional SARE:

- C(i,j,k,ii,jj,kk)=0
  for $1 \leq i \leq q, 1 \leq j \leq q, k=1, 1 \leq ii \leq r, 1 \leq jj \leq r, kk=0$

- C(i,j,k,ii,jj,kk)=0
  for $1\le i\le q$,$1\le j\le q$,$k=0$,$1\le ii\le r$,$1\le jj\le r$ $kk=r+1$
- C(i,j,k,ii,jj,kk)=C(i,j,k,ii,jj,kk-1)+
  +A(i,k,ii,kk)B(k,j,kk,jj)
  for $1\le i\le q$,$1\le j\le q$,$1\le k\le q$,$1\le ii\le r$,$1\le jj\le r$,$1\le kk\le r$
- C(i,j,k,ii,jj,kk)=C(i,j,k-1,ii,jj,kk)+
  +C(i,j,k,ii,jj,kk-1)
  for $1\le i\le q$,$1\le j\le q$,$1\le k\le q$,$1\le ii\le r$,$1\le jj\le r$,$kk=r+1$
- output C(i,j,k,ii,jj,kk)
  for $1\le i\le q$,$1\le j\le q$,$k=q$,$1\le ii\le r$,$1\le jj\le r$,$kk=r+1$

In Fig. 1 we report in abscissa the value of the cost function obtained during the optimization process with w_S=2 and w_M=0.1 (we put more emphasis on execution time) for the parallelized matrix multiplication SARE example: the solid line represents the true execution time on the parallel machine versus the cost function, while the dotted line represents the memory requirement per processor versus the cost function. For our test example we achieve an execution time of 164 ms and a memory allocation of 21952 word/processor.

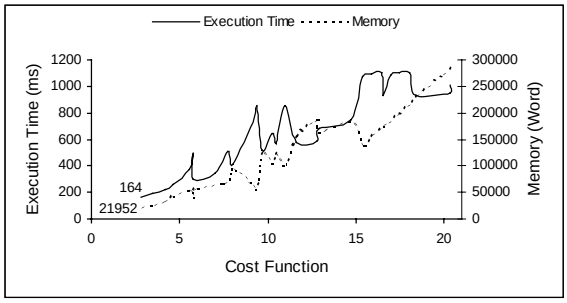

Fig. 1

In Fig. 2 we show the optimization process with w_S=1 and w_M=0.1 (we put less emphasis on time than the previous optimization process): again the solid line represents the true execution time on the parallel machine versus the cost function, while the dotted line represents the memory requirement per processor vs the cost function.

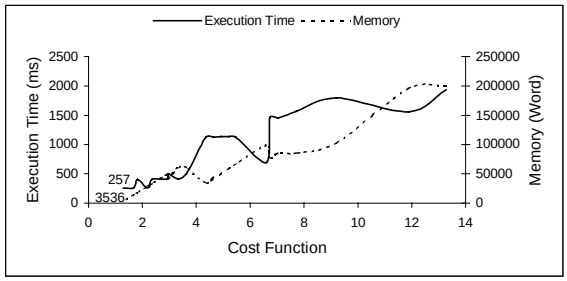

Fig. 2

Because of the less emphasis on time, we expect to have a bigger execution time but with more contained memory requirements. In fact in this case we achieve an execution time of 257 ms with a memory allocation of 3536 word/processor.

In a similar manner the optimization process for memory minimization (w_S=1 and w_M=20) lead to a memory allocation of 2820 word/processor but with an execution time of 1015 ms.

References:

[1] Marongiu A., Palazzari P.,"A New Memory Saving Technique to Map System of Affine Recurrence Equations (SARE) onto Distributed Memory Systems", *Proc. of 13th IPPS*, Apr. 1999 Puerto Rico.

[2] Zimmermann K.H., "Linear Mapping of *n*-dimensional Uniform Recurrences onto *k*-dimensional Systolic Arrays", *Journal of VLSI Signal Processing*, No.12, 1996, pp.187-202.

[3] Feautrier P., "Automatic Parallelization in the Polytope Model", *Les Menuires*, Vol. LNCS 1132, 1996, pp. 79-100.

[4] Lengauer C., "Loop Parallelization in the Polytope Model", *CONCUR*, Vol. LNCS 715, 1993, pp. 398-416.

[5] Dowling M. L., "Optimal code parallelization using unimodular transformations", *Parallel Computing*, 16, 1990, pp. 157-171.

[6] Li W., Pingali K., "A Singular Loop Transformation Framework Based on Non-Singular Matrices", *Proceedings of Fifth Workshop of Languages and Compilers for Parallel Computing*, 1992, pp. 391-405.

[7] Lefebvre V., Feautrier P., "Automatic Storage Management for Parallel Programs", *Parallel Computing*, Vol. 24, 1998, pp.649-671.

[8] Schrijver A., *Theory of Linear and Integer Programming*, John Wiley and Sons, 1986.

[9] Kirkpatrick S., Gelatt C.D., Vecchi M.P., "Optimization by Simulated Annealing", *Science*, Vol. 220, N. 4589, 1983.

[10] Deckers A., Aarts E., "Global Optimization and Simulated Annealing", *Mathematical Programming*, Vol. 50, 1991.

[11] MacDufee,C.C. *The Theory Matrices* Springer 1933.

[12] Knuth D., *The Art of Computer Programming*, Addison Wesley, 1969, pp. 302-4

[13] Bartoloni et al, "A Hardware implementation of the APE100 architecture", *Int. Journal of Modern Physics*, C4, 1993.

[14] Moneget C., Clauss P., Perrin G.R., "Geometrical Tools to Map System of Affine Recurrence Equations on Regular Arrays", *Acta Informatica*, Vol. 31, No.2, 1994, pp. 137-160.

[15] Wilde D.K., "A Library for Doing Polyhedral Operations", *IRISA - Internal Report*, No. 785, 1993.