A New Memory-Saving Technique to Map System of Affine Recurrence Equations (SARE) onto Distributed Memory Systems

Alessandro *Marongiu*[1] and Paolo *Palazzari*[2]
[1]Electronic Engineering Department - University "La Sapienza" – Via Eudossiana, 18
00184 Rome – E-mail marongiu@tce.ing.uniroma1.it
[2]ENEA HPCN project – C.R.Casaccia – Via Anguillarese, 301 S.P.100
00060 S.Maria di Galeria – Rome – E-mail palazzari@casaccia.enea.it

Abstract

In this work we present a procedure for automatic parallel code generation in the case of algorithms described through Set of Affine Recurrence Equations (SARE); starting from the original SARE description in an N-dimensional iteration space, the algorithm is converted into a parallel code for an m-dimensional distributed memory parallel machine (m < N). The used projection technique is based on the polytope model. Some affine transformations are introduced to project the polytope from the original iteration space onto another polytope, preserving the SARE semantic, in the processor-time (t,p) space. Along with polytope transformation, we give a methodology to generate the code within processors and a technique to avoid the memory wasting typical of SARE implementations. Finally a cost function, used to guide the heuristic search for the polytope transformation and derived from the actual implementation of the method on an MPP SIMD machine, is introduced.

1 Introduction

In this work we refer to the automatic parallelization of a particular class of algorithms expressible as System of Affine Recurrence Equations (SARE) ([10], [12], [23]) and including a great number of linear algebra and signal processing problems. A lot of studies have been devoted to this topic ([1]-[3],[6]-[12],[17]-[19], [23]). A detailed discussion on these works is done in paragraph 4, after the introduction of some basic definitions and concepts necessary to understand the used terminology.

Our main original contributions are the following:

- we perform the code generation after a projection of the N-dimensional SARE algorithm onto an m-dimensional processor space ($m < N$); usually other works adopt $m = N - 1$;
- we simply assume that the actual parallel architecture can be represented as a (possibly virtual) m-dimensional distributed memory machine;
- we demonstrate some theorems which give sufficient conditions to obtain admissible projection (i.e. semantically correct);
- we show how to generate the iterative loops within the single processor;
- we demonstrate two theorems to reduce the memory requirements of the SARE (single assignment) model;
- we introduce a cost function (experimentally validated on an actual MPP system) to guide the heuristic search for the best projection.

In the paper all definitions and theorems refer to the N-dimensional rational Cartesian space Q^N unless otherwise specified. The following notations will be used:

Given an $n\times1$ vector x and an $m\times1$ vector y (x,y) indicate the $(n+m)\times1$ vector obtained through the composition of x and y. *Id* is the identity matrix. Iff means if and only if.

2 SARE Computational Model: a Review

This section briefly presents SARE computational model. Detailed information can be found in [10], [12] and [23]. For geometrical details and definitions see [14].

A SARE is described by several equations E:

$X(z)=f(\ldots,Y[\rho(z)],\ldots)$ with $z\in I_E$ (E)

where:

- X and Y are multi-dimensional array variables.
- $z\equiv(i\ j\ \ldots\ k)^T$ is the $N\times1$ iteration vector. $X(z)$ represents $X[i,j,\ldots k]$. $S_X(z)$ is an instance of (E) for a given z.
- I_E is the iteration space related to (E), i.e. the set of indices z where equation (E) is defined. I_E is described through a parameterized polytope [14]. The convex union of all the I_E is the global iteration space I.
- $\rho(z)=Rz+r$ is the affine index mapping function. It defines a flow dependence between $\rho(z)$ and z.

- f is a general function used to compute $X(z)$.

Analysis of algorithm dependencies is performed through dependence vectors:

Definition 2.1 (Dependence Vector): Given a SARE equation (E), the dependence vector associated to Y is:

$d_{Y,\rho(z)}=z-\rho(z)=z-Rz-r$; E 2.1

$d_{Y,\rho(z)}$ gives the dependence between $S_X(z)$ and $S_Y[\rho(z)]$.

$d_{Y,\rho(z)}$ is uniform if it does not depend on z. Dependence vector is uniform if $R\equiv Id$. In this case from E2.1 we have $d_{Y,\rho(z)}=-r$. Even if the results achieved are still valid with non-uniform dependencies, in this work we consider algorithms with uniform dependence vectors because we use, as target machine, the SIMD MPP APE100/Quadrics [15] supporting only uniform communications. Throughout the paper we use the SARE expression of matrix-matrix multiplication algorithm as example.

Given two (q×q) square matrices a and b, the generic element of their product is $c(i,j)=\Sigma a(i,k)*b(k,j)$. The equivalent SARE is:

$A(i,j,k)=a(i,k)$ with $1 \le i \le q$, $j = 0$, $1 \le k \le q$ (*E*1)

$B(i,j,k)=b(k,j)$ with $i = 0$, $1 \le j \le q$, $1 \le k \le q$ (*E*2)

$C(i,j,k)=0$ with $1 \le i \le q$, $1 \le j \le q$, $k = 0$ (*E*3)

$A(i,j,k)=A(i,j-1,k)$ with $1\le i\le q$, $1\le j\le q-1$, $1\le k\le q$ (*E*4)

$B(i,j,k)=B(i-1,j,k)$ with $1\le i\le q-1$, $1\le j\le q$, $1\le k\le q$ (*E*5)

$C(i,j,k)=C(i,j,k-1)+A(i,j-1,k)*B(i-1,j,k)$
with $1\le i\le q$, $1\le j\le q$, $1\le k\le q$ (*E*6)

$c(i,j)=C(i,j,k)$ with $1\le i\le q$, $1\le j\le q$, $k=q$ (*E*7)

where *E*1 and *E*2 are the input equations, *E*3 is an initialization equation, *E*4, *E*5 and *E*6 are computational equations and finally *E*7 is the output equation. Iteration vector is $z\equiv(i,j,k)$. By analyzing index bounds, the iteration space polytope for each equation is easily found.

SARE *(E1-E7)* introduces three dependence vectors derived from equations *E*4, *E*5 and *E*6. Equation *E*4 has one index mapping function related to *A*:

$A(i,j-1,k) \Rightarrow \rho(z) \equiv Idz+(0\ -1\ 0)^T \Rightarrow d_{A,\rho(z)}=-r=(0\ 1\ 0)^T=d_1$.

Equation *E*5 has one index mapping function related to *B*:

$B(i-1,j,k) \Rightarrow \rho(z)\equiv Idz+(-1\ 0\ 0)^T \Rightarrow d_{B,\rho(z)}=-r=(1\ 0\ 0)^T=d_2$.

Equation *E*6 has three index mapping functions related to *A*, *B*, and *C*:

$C(i,j,k-1) \Rightarrow \rho(z)\equiv Idz+(0\ 0\ -1)^T \Rightarrow d_{A,\rho(z)}=-r=(0\ 0\ 1)^T=d_3$.

$A(i,j-1,k) \Rightarrow \rho(z)\equiv Idz+(0\ -1\ 0)^T \Rightarrow d_{A,\rho(z)}=-r=(0\ 1\ 0)^T=d_1$.

$B(i-1,j,k) \Rightarrow \rho(z)\equiv Idz+(-1\ 0\ 0)^T \Rightarrow d_{A,\rho(z)}=-r=(1\ 0\ 0)^T=d_2$.

Parallelization of a SARE implies a space-time transformation which, $\forall z\in I\subset Z^N$, gives the time instant and the processor where the $S_X(z)$ will be executed. The space-time transformation is composed by

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

These two functions must preserve the semantics of the algorithm, i.e. dependence relations have to be maintained, and must be compatible, i.e. no more than one statement can be executed on a processor at the same time.

3 Timing Function

Timing function t(z) gives the relative time scheduling of $S_X(z)$, $\forall z\in I$: $t(z): I \to \Gamma$, being $\Gamma=\{t(z) \mid z\in I\}$. The value $t(z)\in\Gamma$ is called *geometrical time*. Γ must be totally ordered to determine univocally the time allocation of a given $S_X(z)$. So an instruction $S_X(z_1)$ is executed before (after) $S_X(z_2)$ if $t(z_1) \prec (\succ)\ t(z_2)$ $(t(z_1),t(z_2)\in\Gamma)$.

Definition 3.1 (Timing Surface): Given a timing function t(z), we define timing surface (*TS*) the set of points with the same t(z): $TS=\{z\in I \mid t(z) = t_0\in\Gamma\}$. Points belonging to *TS* are executed simultaneously.

TS are *m*-dimensional sets, with $m\le N$. Given t(z), *m* is the actual degree of parallelism extracted by t(z) from the algorithm.

Definition 3.2 (Timing Function Dimension): We define timing function dimension the value $n = N-m$.

We choose t(z) as an affine integer function because the set of convex polyhedra is closed under application of affine functions: $t(z) \equiv \Lambda z + \underline{\alpha}$, being Λ an integer $n\times N$ matrix and $\underline{\alpha}$ an integer $n\times1$ vector.

As t(z) is an integer affine function, we have $\Gamma = \{t\in Z^n \mid t = \Lambda z + \underline{\alpha}$ with $z\in I\}$. Γ must be totally ordered, so we assume the lexicographical ordering on it.[1] Most of the recent works ([1],[2],[7]-[9],[11],[12],[19]) related to algorithm parallelization use one-dimensional affine timing functions. All these methods are mainly based on the Lamport's hyperplane work [16] which adopts the timing function $t(z)=\lambda z+\alpha$, where λ is a $1\times N$ vector, called *timing vector*, and α is a scalar value. Lamport-based timing surfaces are *N*-1 dimensional hyperplanes. Several extensions have been proposed in the literature. In the case of one-dimensional timing function the concepts of affine by statement ([17],[18]) and affine by variable ([23]) are introduced. Other authors adopt *n*-dimensional timing functions (*n*>1); for example in ([3], [10]) a general timing function is introduced without a clear investigation on the theoretical concepts involved. In [6] the special case of the projection of N nested loops onto a *1*-dimensional linear array through two timing hyperplanes is presented.

In this paper we introduce a new class of *n*-dimensional timing functions as an extension of Lamport functions (which are obtained when *n*=1). The *n*-dimensional timing functions allow the generation of *m*-dimensional timing surfaces, so *m*-dimensional machines can be used. *n*-dimensional timing functions extend the class of parallel programs achievable from an algorithm and overcome

[1] Given two $n\times1$ vectors x and y, they are (strictly) lexicographically ordered $x\underline{\ll}y$ ($x\ll y$) 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 dept.

some limitations of one-dimensional timing functions. Lamport demonstrated the existence of one-dimensional scheduling for uniform dependencies algorithms (*R*=*Id*); this is not more true for general affine dependencies (*R*≠*Id*). In fact, due to dependence relations, an algorithm may have a parallelism degree less then *N*-1, so points belonging to *N*-1 dimensional hyperplanes (the only achievable through *1*-dimensional t(z)) cannot be executed in parallel. Furthermore one-dimensional timing functions require at least *N*-1 dimensional machines.

The basic idea underlying our *n*-dimensional timing function is derived from geometrical considerations. While Lamport t(z) introduces timing surfaces that are *N*-1-dimensional hyperplanes, our *n*-dimensional t(z) generates *m*-dimensional timing surfaces that are the intersection of *n* non-parallel *N*-1 dimensional hyperplanes (*m*=*N*-*n*). In fact the intersection of two non parallel hyperplanes is an *N*-2 dimensional set; the intersection of three non parallel hyperplanes is an *N*-3 dimensional set, …, the intersection of *n* non parallel hyperplanes is an *N*-*n*=*m* dimensional set.

In the following we introduce the general form of an *n*-dimensional timing function, demonstrate its properties and give admissibility (necessary and sufficient) conditions to preserve the semantics of the algorithm.

First we define a set of *n* one-dimensional affine timing functions θ_i:

$$\begin{cases} \cdots \\ \theta_i = \lambda_i z + \alpha_i \quad \text{with } z\in I \text{ and } i=1,2,\ldots,n \quad \text{E 3.1} \\ \cdots \end{cases}$$

where λ_i are $1\times N$ linearly independent integer timing vectors and α_i are scalar values (offset).

Definition 3.3 (*n-dimensional Timing Function*)
We define the *n*-dimensional timing function as the composition of the *n* one-dimensional timing functions θ_i:

$$t(z)=(\theta_1\ \theta_2 \ldots \theta_n)^T = \Lambda z + \underline{\alpha} \qquad \text{E 3.2}$$

where Λ is an $n\times N$ matrix with rows λ_i and $\underline{\alpha}$ is a $n\times 1$ vector with components α_i.

Timing Surfaces Theorem: Timing Surfaces related to an *n*-dimensional t(z) are *m* = *N*-*n* dimensional sets. These sets can be found from the Kernel of matrix Λ (*Ker*(Λ)).

Proof. Given an *n*-dimensional t(z) and two points $z_1,z_2\in I$ belonging to a same timing surface, from definition 3.1 they have the same geometrical time $t(z_2) = t(z_1)$. From E3.2, we have $\Lambda z_2+\underline{\alpha}=\Lambda z_1+\underline{\alpha}\Rightarrow\Lambda z_2=\Lambda z_1\Rightarrow$ $\Lambda(z_2-z_1)=0\Rightarrow(z_2-z_1)\in Ker(\Lambda)$. Because of last relation, a timing surface *TS* can be expressed as $TS = \{z_2\in I \mid z_2 = z_1 + v,\ v\in Ker(\Lambda),\ z_1\in I\}$ so *TS* is derived from the *N*-*n* = *m*-dimensional set *Ker*(Λ). As a consequence *TS* is an *m*-dimensional set.□

Corollary to the Timing Surface Theorem: The statements related to points $z_2,z_1\in I$ are scheduled at different time iff $(z_2-z_1)\notin Ker(\Lambda)$.

We show that *n*-dimensional t(z) partitions the vector space Q^N in equivalence classes: each class is labeled by a t(z) value and it corresponds to a timing surface.

Definition 3.4 (Modulus W Congruence) [5]: Given a subspace $W\subseteq Q^N$, two vectors $u,v\in Q^N$ are modulus *W* congruent iff $u-v\in W$. This is an equivalence relation that partitions Q^N in equivalence classes denoted by [u]: two vectors u and v belong to the same class iff $u-v\in W$.

According to definition 3.4, we introduce the modulus *Ker*(Λ) congruence. Points z belonging to the same timing surface fall into the same equivalence class denoted by [z] (timing surface theorem). So there is a duality between equivalence classes introduced by modulus *Ker*(Λ) congruence and timing surfaces. Moreover, because a geometrical time is associated to each timing surface, we can individuate an equivalence class from its geometrical time: $[z] \equiv t(z)$.

For the set of classes [z] we define the algebraic structure of vector space introducing the following operations [5]:

$[z_1]+[z_2]=[z_1+z_2]$	sum of classes
$\gamma[z_1]=[\gamma z_1]$ with $\gamma\in Q^N$	multiply a class by a scalar

Equivalence class vector space is called *quotient space* and it is denoted as $Q^N/Ker(\Lambda)$. Now we can state:
Property 3.1 (Equivalence Classes Induced by a Timing Function): An *n*-dimensional timing function induces a set of equivalence classes denoted by $[z] \equiv t(z)$. The vector space defined by these classes is the quotient space $Q^N/Ker(\Lambda)$.
Quotient Space Theorem [5]: If $Q^N = U \oplus W$ then Q^N/W and *U* are isomorphic.
In our case, being *W*=*Ker*(Λ), any supplementary subspace of *Ker*(Λ) is isomorphic to $Q^N/Ker(\Lambda)$.

An *n*-dimensional timing function is admissible for a given algorithm if it preserves the precedence relations among dependent statements.
n-dimensional Timing Function Admissibility Theorem:
An *n*-dimensional t(z) is admissible iff, for every dependence vector, $\Lambda d_{Y,\rho(z)} >> 0$.
Proof(⇒). Given two dependent statements $S_X(z)$ and $S_Y[\rho(z)]$, to preserve algorithm semantics $S_Y[\rho(z)]$ must be executed before $S_X(z)$. So $t(z) >> t(\rho(z))$ must result (hypothesis). Let be $\rho(z) = z_0$. From dependence vector definition E2.1, $z = z_0 + d_{Y,\rho(z)}$. So $t(z) >> t(\rho(z)) \Rightarrow t(z_0+d_{Y,\rho(z)}) >> t(z_0) \Rightarrow t(z_0+d_{Y,\rho(z)}) - t(z_0) >> 0$. From t(z) definition 4.3 and E4.2 we have:

$$\Lambda(z_0+d_{Y,\rho(z)}) + \underline{\alpha} - \Lambda(z_0) - \underline{\alpha} >> 0 \Rightarrow \Lambda d_{Y,\rho(z)} >> 0 \qquad \text{E 3.3}$$

Proof(⇐). The demonstration is trivial by reading previous part in inverse sequence and reversing the imply-arrows. □

Corollary to Timing Function Admissibility Theorem
An *n*-dimensional t(z) is admissible only if no dependence vector belongs to *Ker*(Λ).

Proof. If a dependence vector $d_{Y,\rho(z)}$ belongs to $Ker(\Lambda)$ we have $\Lambda d_{Y,\rho(z)} = 0$. This violates the admissibility condition E3.3 of previous theorem.□

Admissibility theorem gives the necessary and sufficient condition ($\Lambda d_{Y,\rho(z)} >> 0$ – E3.3) to have an admissible t(z).

In order to explain the introduced concepts, we use again the matrix multiplication example.

Case 1: one-dimensional timing function.
We choose an one-dimensional timing function with timing vector λ_1=(1 1 1) giving the three admissibility conditions (all verified):
$\lambda_1 d_1 = (1\ 1\ 1)(0\ 1\ 0)^T = 1 >> 0$;
$\lambda_1 d_2 = (1\ 1\ 1)(1\ 0\ 0)^T = 1 >> 0$;
$\lambda_1 d_3 = (1\ 1\ 1)(0\ 0\ 1)^T = 1 >> 0$.

Case 2: two-dimensional timing function (1).
We choose a two-dimensional timing function with timing vectors λ_1=(1 1 1) and λ_2=(0 1 1). This leads to the following three admissibility conditions (all verified):

$$\Lambda d_1 = \begin{pmatrix}1 & 1 & 1\\ 0 & 1 & 1\end{pmatrix}\begin{pmatrix}0 & 1 & 0\end{pmatrix}^T = \begin{pmatrix}1\\1\end{pmatrix} >> 0;$$
$$\Lambda d_2 = \begin{pmatrix}1 & 1 & 1\\ 0 & 1 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\end{pmatrix}^T = \begin{pmatrix}1\\0\end{pmatrix} >> 0;$$
$$\Lambda d_3 = \begin{pmatrix}1 & 1 & 1\\ 0 & 1 & 1\end{pmatrix}\begin{pmatrix}0 & 0 & 1\end{pmatrix}^T = \begin{pmatrix}1\\1\end{pmatrix} >> 0.$$

4 Allocation Function

Allocation function p(z) gives processor executing S_X(z).

Given an *n*-dimensional timing function, statements to be executed concurrently (on different processors) belong to an *m*-dimensional set (timing surface theorem); so we choose an *m*-dimensional grid as parallel architecture topology. Each processor is uniquely identified by a set of coordinates $(\pi_1\ \pi_2\ ...\ \pi_m)^T$. Given a point $z\in I$, p(z) gives the coordinates of the processor executing S_X(z).

p(z) is found by projecting the *N*-dimensional iteration space *I* along *n* directions in order to collapse *I* onto the *m*-dimensional processor space. Every projecting direction is characterized by a $N\times 1$ *projection vector* ξ_i. Projection vectors must be linearly independent and constitute the basis of the *projection subspace X.*

Projection directions must be compatible with t(z), i.e. statements assigned to the same processor have to be scheduled at different times. Compatibility is assured by the following

Compatibility Theorem between Projection Vectors and Timing Vectors: Projection vectors ξ_i and t(z) are compatible iff the set of *N* vectors ($\xi_1\ \xi_2\ ...\ \xi_n\ k_1\ k_2\ ...\ k_m$) are a basis for Q^N, being vectors k_i a basis of $Ker(\Lambda)$.

Proof (⇒). Given an equation *E* and two points z_0 and $z_1 = z_0 + \sum_{i=1}^n a_i\xi_i$ ($z_1 - z_0 = \sum_{i=1}^n a_i\xi_i$), $z_0,z_1\in I$, by definition of projection vectors the statements $S_X(z_0)$ and $S_X(z_1)$ will be projected onto the same processor and executed by the same processor; as a consequence, points z_0 and z_1 have to be scheduled at different times, i.e. $t(z_1)\neq t(z_0)$. From corollary to timing surface theorem we know that $t(z_1)\neq t(z_0) \Rightarrow z_1 - z_0 = \sum_{i=1}^n a_i\xi_i \notin Ker(\Lambda)$.
Previous relation has to be verified $\forall a_i$ because of generality of z_1.

$$\sum_{i=1}^n a_i\xi_i \notin Ker(\Lambda) \Rightarrow \sum_{i=1}^n a_i\xi_i \neq \sum_{i=1}^m b_i k_i\ \ \forall a_i,b_i$$

(vectors k_i are a basis of $Ker(\Lambda)$). So we have
$(\xi_1\ \xi_2\ ...\ \xi_n)(a_1\ a_2\ ...\ a_n)^T \neq (k_1\ k_2\ ...\ k_m)(b_1\ b_2\ ...\ b_n)^T \Rightarrow$
$(\xi_1\ \xi_2\ ...\ \xi_n\ k_1\ k_2\ ...\ k_m)(a_1\ a_2\ ...\ a_n\ -b_1\ -b_2\ ...\ -b_n)^T \neq 0$.
Previous relation is valid $\forall a_i,b_i$ if the $N\times N$ matrix ($\xi_1\ \xi_2\ ...\ \xi_n\ k_1\ k_2\ ...\ k_m$) has full rank, so its columns are a basis of Q^N.
Proof(⇐). Being vectors ($\xi_1\ \xi_2\ ...\ \xi_n\ k_1\ k_2\ ...\ k_m$) a basis of Q^N, *X* subspace is a supplementary subpsace of $Ker(\Lambda)$ in Q^N: $Q^N = X\oplus Ker(\Lambda)$. From quotient space theorem the set of classes induced by t(z) are isomorphic to *X*. So *X* does not contain two or more points with the same t(z) value.□

We choose p(z) in the set of affine functions. By indicating a processor through its coordinates $(\pi_1\ \pi_2\ ...\ \pi_m)^T$, we can write p(z) as:
$p(z)=(\pi_1\ \pi_2\ ...\ \pi_m)^T=(\sigma_1\ \sigma_2\ ...\ \sigma_m)^T z+(\beta_1\ \beta_2\ ...\beta_m)^T$ with $z\in I$
where σ_i are $1\times N$ vectors and β_i are scalar values (offset). p(z) can be compactly written as $p(z)=\Sigma z+\underline{\beta}$, where Σ is the $m\times N$ allocation matrix whose rows are σ_i and $\underline{\beta}$ is a $N\times 1$ vector whose components are β_i.

Allocation matrix is determined through projection vectors. Given an equation *E* and two points z_0 and $z_1 = z_0 + \sum_{i=1}^n a_i\xi_i$ ($z_1 - z_0 = \sum_{i=1}^n a_i\xi_i$), $z_0,z_1\in I$, by definition of projection vectors the statements $S_X(z_0)$ and $S_X(z_1)$ are projected onto the same processor, i.e. $p(z_1) = p(z_0) \Rightarrow \Sigma z_1 + \underline{\beta} = \Sigma z_0 + \underline{\beta} \Rightarrow \Sigma(z_1 - z_0) =$
$\Sigma(\sum_{i=1}^n a_i\xi_i) = 0$. Because of the generality of z_1, previous relation has to be verified $\forall a_i$, we have $\Sigma\xi_i = 0$ $\forall i$. Allocation matrix is so an $m\times N$ matrix whose kernel is generated by projection vectors.

In order to clarify previous concepts, we use our example.

Case 1 (one-dimensional timing function and two-dimensional allocation function):

As timing function is one-dimensional, we need one projection vector, for example $\xi_1=(1\ 0\ 0)^T$. From compatibility theorem we have to find a basis of $Ker(\Lambda)$. Being $\lambda_1\equiv(1\ 1\ 1)$ we have, for example, $k_1\equiv(1\ 0\ -1)^T$ and

$k_2 \equiv (0\ 1\ {-1})^T$. As determinant of matrix ($\xi_1\ k_1\ k_2$) is equal to 1, λ_1 and ξ_1 are compatible.

Matrix Σ is found among the matrices having as kernel the vector ξ_1. It easy to verify that $\Sigma=\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is valid.

Case 2 (two-dimensional timing function and one-dimensional allocation function):

As timing function is two-dimensional, we need two projection vectors, for example $\xi_1 \equiv (1\ 0\ 0)^T$ and $\xi_2 \equiv (0\ 1\ 0)^T$. From compatibility theorem we have to find a basis of $Ker(\Lambda)$. Being $\lambda_1 \equiv (1\ 1\ 1)$ and $\lambda_2 \equiv (0\ 1\ 1)$ we have, for example, $k_1 \equiv (0\ 1\ {-1})^T$. As determinant of matrix ($\xi_1\ \xi_2\ k_1$) is equal to 1, $\lambda_{1,2}$ and $\xi_{1,2}$ are compatible.

Matrix Σ is then found in the set of matrices having as kernel vectors ξ_1 and ξ_2. It easy to verify that a valid Σ is $\Sigma=(0\ 0\ 1)$.

5 Space-Time Transformation

By composing timing and allocation function we obtain a transformation function T(z):

$$T(z)=(\theta_1 \theta_2 \dots \theta_n \pi_1\ \pi_2\ \dots\ \pi_m)^T=(\lambda_1 \lambda_2 \dots \lambda_n\ \sigma_1\ \sigma_2\ \dots\ \sigma_m)^T+$$
$$+(\alpha_1 \alpha_2 \dots \alpha_n\ \beta_1\ \beta_2\ \dots\ \beta_m)^T \text{ with } z\ I \qquad \text{E 5.1}$$

T(z) can be rewritten in more compact form as:

$$T(z)=\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix} z + \begin{pmatrix} \alpha \\ \underline{\beta} \end{pmatrix} = Tz+\underline{\gamma}$$

On the basis of compatibility theorem it is easy to demonstrate that transformation matrix T is invertible.

By applying T(z) to the iteration space polytope I we obtain a target iteration space polytope I_T. I_T has the same points of I, but in a different coordinate system, i.e. in space-time coordinates. Being T invertible every point z of Q^N has an unique image that is individuated through a double set of coordinates: the first n coordinates, i.e. *geometrical time* t vector, and the second m coordinates, i.e. the p vector. Matrix T must be unimodular with integer elements [19]; unimodularity ensures that I and I_T have the same number of points with integer coordinates.

6 Code Generation

The theory developed in previous paragraphs allows the generation of SPMD codes, because no hypothesis was made on the target parallel machine. Processors set is organized as a (possibly virtual) m-dimensional regular grid array in which each processor is addressed through its m coordinates p.

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

The set of statements to execute on a given processor p^* is generated through the nesting of n sequential loops which serially scan time indices θ_i. In fact, being t=($\theta_1, \theta_2, \dots, \theta_n$), the scanning of θ_i generates a sequence of t vectors {$t_1\ t_2\ \dots\ t_i\ \dots$} which serially scans statements $S_X(t_1,p^*)\ S_X(t_2,p^*)\ \dots\ S_X(t_i,p^*)\ \dots$

Instruction scanning must respect the semantics of the original algorithm: the vector t sequence {$t_1\ t_2\ \dots\ t_i\ \dots$} has to generate a strictly lexicographically ordered sequence of scheduling vectors $t_1 << t_2 << \dots << t_i << \dots$

It is easy to verify that the following n nested loops generate a lexicographical ordered sequence of vector t

```
for $\theta_1=L_1$ to $U_1$
  for $\theta_2=L_2(\theta_1)$to $U_2(\theta_1)$
    ...
    for $\theta_n=L_n(\theta_1,..,\theta_{n-1})$ to $U_n(\theta_1,..,\theta_{n-1})$
      ...
      Sx(t)
      ...
    endfor $\theta_n,\dots,\theta_1$
```

The lower and upper bounds of nested loops are found applying the technique described in [13].

In the space-time coordinates system, dependence vectors are transformed according to the following

Property 6.1 (*Transformed Dependence Vector*)

Uniform dependence vectors $d_{Y,\rho(z)}$ are transformed into $d_{T,Y,\rho(z)} = Td_{Y,\rho(z)} = (\Lambda d_{Y,\rho(z)}, \Sigma d_{Y,\rho(z)}) = (d^t, d^p)$ where d^t is a $n\times1$ vector and d^p is a $m\times1$ vector.

The proof is obtained applying the definition of T(z).

It is easy to see that d^t is lexicographical positive due to the admissibility theorem.

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: the first refers to time components and the second to processor components. In fact, given T(z):z→(t,p), we have $z_0 \rightarrow (t_0,p_0)$ and $z_0 + d \rightarrow (t_0,p_0) + (d^t,d^p) = (t_0 + d^t,\ p_0 + d^p)$.

Communications are generated by the processor part of the dependence vector. If d^p is not null, operand required by statement $S_X(t_0 + d^t, p_0 + d^p)$, computed in processor $p_0 + d^p$, is stored in the processor p_0. So communications must be executed to transfer operands from storage processor p_0 to computing processor $p_0 + d^p$. If d^p is null, no communication is needed.

Generation of parallel code depends on the target parallel environment. The following steps are the core of the code generation process:

- find an admissible timing function $t(z)$ and a compatible allocation function p(z) which form the T transformation; the T matrix can be found with an heuristic procedure which explores the space of the admissible unimodular matrices with integer coefficients. In order to do this, a (machine dependent) cost function has to be defined.
- an admissible scanning sequence is generated (pseudo-code in this paragraph).
- generate the loop body according to the SARE original equations.

7 Automatic Memory Optimization

We indicate with

- $X(t)=X(\theta_1, \theta_2, ..., \theta_n)$ a logic array variable;
- $V_X(t)=V_X(\theta_1, \theta_2, ..., \theta_n)$ the value of $X(t)$;
- $M_X(t)=M_X(\theta_1, \theta_2, ..., \theta_n)$ the memory location of $X(t)$;

We define the lexicographical maximal vector on S as $x = \max\limits_{x_i \in S}{}_{\ll}(x_i)$, i.e. $x \underline{\gg} x_i \ \forall \ x_i \in S$.

Given two lexicographically ordered $n\times 1$ vectors $x_{min} \underline{\ll} x_{max}$, we define the range set $[x_{min},x_{max}] = \{y \mid x_{min} \underline{\ll} y \underline{\ll} x_{max}\}$

SARE is a single assignment computational model. This implies that the value $V_X(t)$, related to a logic variable $X(t)$, is permanently stored in a physical memory location $M_X(t)$. No other values can be stored in $M_X(t)$, even if $V_X(t)$ is not more used. This produces a great waste of memory.

In this paragraph we show how to reduce memory wasting due to the single assignment constraint. This is accomplished by allowing the (partial) reuse of locations containing values no longer needed.

Given $t_0=(\theta_{0,1}, \theta_{0,2}, ... , \theta_{0,n})$, value $V_X(t_0)$, related to $X(t_0)$, is generated by the statement $S_X(t_0)$ at geometrical time t_0 and will be used later by one or more statements.

Statements that will use the value $V_X(t_0)$ are those depending on X. The geometrical times in which the value $V_X(t_0)$ will be used are obtained by the time part of the transformed dependence vectors. Suppose that the parallel program has w transformed dependence vectors related to the variable X; the times in which the value $V_X(t_0)$ is used are: $t_1=t_0+d^t_1 \gg t_0$, $t_2=t_0+d^t_2 \gg t_0$, ..., $t_w=t_0+d^t_w \gg t_0$. The lexicographical $\gg$ relation is due to the property 7.1.

The memory location $M_X(t_0)$ can be reused after the last use of $V_X(t_0)$.

In the following the concept of life cycle of a value is used [4].

Life Cycle of a Value Theorem: The last using of a value $V_X(t_0)$ occurs at time $t_0+d^t_{\max}$ where $d^t_{\max} = \max\limits_{i=1,..,w}{}_{\gg}(d^t_i)$. So the life cycle of a value $V_X(t_0)$ ($LC_{X(t0)}$) is $LC_{X(t0)}=[t_0,t_0+d^t_{\max}]$.

Proof. Suppose the last using of $V_X(t_0)$ to occur at time $t_k \gg t_0+d^t_{\max}$. In this case a dependence vector d_k exists so that $t_k = t_0+d^t_k \gg t_0+d^t_{\max} \Rightarrow d^t_k \gg d^t_{\max}$. This violates the hypothesis $d^t_{\max} = \max\limits_{i=1,..,w}{}_{\gg}(d^t_i)$, so the last using occurs at geometrical time $t_0+d^t_{\max}$. The life cycle of $V_X(t_0)$ is the elapsed time from its creation time (t_0) to the last using time ($t_0+d^t_{\max}$): $LC_{X(t0)} = [t_0,t_0+d^t_{\max}]$.□

Memory optimization of a logic variable X implies the using of a compressed variable X_C with dimensionality lower than X. We demonstrate two theorems which give sufficient conditions to store the n-dimensional logic variable $X(t_0)$ into an $(n-k)$-dimensional compressed variable (1st theorem) or into an auxiliary scalar variable and an $(n-1)$-dimensional compressed variable (2nd theorem). These two theorems can be used in sequence.

1st Compressed Variable Theorem.

A logic variable $X(t_0)$, generated at geometrical time $t_0=(\theta_{0,1}, \theta_{0,2}, ... , \theta_{0,n})$, can be stored as a compressed logic variable $X_C(\theta_{0,k+1}, \theta_{0,k+2}, ... , \theta_{0,n})$ if the first k components of $d^t_{\max}$ are null, i.e. $d^t_{\max}=(0 \ ... \ 0 \ 0 \ d^t_{k+1} \ ... \ d^t_n)$.

Proof. Under hypothesis of this theorem, we show that a statement $S_X(t_1)$ never overwrites a value $V_X(t_0)$ $(t_1\neq t_0)$ if $t_1\in LC_{X(t0)}$. Without compressed variables, $V_X(t_0)$ generated by $S_X(t_0)$ is stored in $M_X(t_0)$ and $S_X(t_1)$ writes into the memory location $M_X(t_1) \neq M_X(t_0)$.

Using the compressed variable X_C, value $V_X(t_0)$ is stored in $M_X(\theta_{0,k+1}, \theta_{0,k+2}, ... , \theta_{0,n})$ and $S_X(t_1)$ writes the memory location $M_X(\theta_{1,k+1}, \theta_{1,k+2}, ..., \theta_{1,n})$. We prove that if $t_1\in LC_{X(t0)}$ and $t_1\neq t_0$ $\Rightarrow$ $M_X(\theta_{0,k+1}, \theta_{0,k+2}, ... , \theta_{0,n}) \neq M_X(\theta_{1,k+1}, \theta_{1,k+2}, ... , \theta_{1,n})$.

$t_1\in LC_{X(t0)}$ and $t_1 \neq t_0$ $\Rightarrow$ $t_0 \ll t_1 \underline{\ll} t_0+d^t_{\max}$. Previous relation can be rewritten as: $t_0 \ll t_0+\Delta t_1 \underline{\ll} t_0+d^t_{\max} \Rightarrow 0 \ll \Delta t_1 \underline{\ll} d^t_{\max}$. Being $0 \ll \Delta t_1 \underline{\ll} d^t_{\max}$, Δt_1 has the first k components null and at least a non null component in a position from $k+1$ to n. So $S(t_1)$ writes in the memory location

$M_X(\theta_{1,k+1}, \theta_{1,k+2}, ... , \theta_{1,n}) = M_X(\theta_{0,k+1}+\Delta t_{1,k+1}, \theta_{0,k+2}+\Delta t_{1,k+2}, ... , \theta_{0,n}+\Delta t_{1,n}) \neq M_X(\theta_{0,k+1}, \theta_{0,k+2},... ,\theta_{0,n})$.□

Previous theorem implies that $V_X(t_0)$ is stored in the memory location $M_X(\theta_{0,k+1}, \theta_{0,k+2}, ... , \theta_{0,n})$.

2nd Compressed Variable Theorem.

A logic variable $X(t_0)$, generated at geometrical time $t_0=(\theta_{0,1}, \theta_{0,2}, ... , \theta_{0,n})$, can be stored as a logic variable $X_C(\theta_{0,2}, \theta_{0,3}, ... , \theta_{0,n})$ and a logic scalar variable X_T if $d^t_{\max}=(1 \ 0 \ ... \ 0 \ 0)$.

Proof. We show that, under hypothesis of this theorem, a statement $S_X(t_1)$ never overwrites a value $V_X(t_0)$ $(t_1\neq t_0)$ if $t_1\in LC_{X(t0)}$.

$V_X(t_0)$ is generated by $S_X(t_0)$ and stored in $M_X(t_0)$; $S_X(t_1)$ writes in the memory location $M_X(t_1) \neq M_X(t_0)$.

Using the compressed variable X_C, value $V_X(t_0)$ is stored in $M_X(\theta_{0,2}, \theta_{0,3}, ... , \theta_{0,n})$ and $S_X(t_1)$ writes in the memory location $M_X(\theta_{1,2}, \theta_{1,3}, ... , \theta_{1,n})$. We have to prove that, if $t_1 \in LC_{X(t0)}$ and $t_1 \neq t_0$, $M_X(\theta_{0,2}, \theta_{0,3}, \dots, \theta_{0,n}) \neq M_X(\theta_{1,2}, \theta_{1,3}, \dots, \theta_{1,n})$.

$t_1 \in LC_{X(t0)}$ and $t_1 \neq t_0 \Rightarrow t_0 << t_1 \underline{<<} t_0 + d^t_{\max}$. Previous relation can be rewritten as: $t_0 << t_0 + \Delta t_1 \underline{<<} t_0 + d^t_{\max} \Rightarrow 0 << \Delta t_1 \underline{<<} d^t_{\max}$. If $0 << \Delta t_1 << d^t_{\max}$, Δt_1 has the first component equal to 0 and at least a non null component in a position from 2 to n. So $S(t_1)$ writes the memory location

$M_X(\theta_{1,2}, \theta_{1,3}, \dots, \theta_{1,n}) = M_X(\theta_{0,2}+\Delta t_{1,2}, \theta_{0,3}+\Delta t_{1,3}, \dots, \theta_{0,n} + \Delta t_{1,n}) \neq M_X(\theta_{0,2}, \theta_{0,3}, \dots, \theta_{0,n})$.

If $\Delta t_1 = d^t_{\max} = (1\ 0\ \dots\ 0\ 0)$, the last using of $V_X(t_0)$ corresponds to the generation time of the new value $V_X(t_1)$; moreover, the storage location is the same. In order to avoid overwriting, we read the old value $V_X(t_0)$ and store it in the scalar variable X_T which will be used for any reference to $V_X(t_0)$. Then we overwrite $V_X(t_0)$ with $V_X(t_1)$ in memory location $M_X(\theta_{0,2}, \theta_{0,3}, \dots, \theta_{0,n})$.□

2nd Compressed Variable Theorem implies that $V_X(t_0)$ can be stored in $M_X(\theta_{0,2}, \theta_{0,3}, \dots, \theta_{0,n})$. Scalar variable X_T is used to keep value $V_X(t_0)$ before it is overwritten.

8 Choice of the Cost Function

We successfully implemented our technique on the APE100 Quadrics SIMD machine [15] using the parallelizing tool developed by us on the basis of the works [13], [14], [20]-[22]. Starting from a SARE, our tool is able to generate a parallel code for the Quadrics machine. During the experimental work, our main issue was to determine a good cost function to guide the searching for T matrix. On the basis of the actual measures on the SIMD machine we adopted a load balancing cost function.

9 Conclusions

In this work we have considered the mapping of N-dimensional SARE algorithms onto m-dimensional systems ($m<N$).

After a brief review about definitions on SARE, we introduced affine n-dimensional timing function and m-dimensional allocation function which perform a space-time transformation of the $N = n+m$-dimensional iteration space I. We demonstrated compatibility conditions between timing and allocation functions and we gave conditions to avoid memory wasting. We showed scheme outlining the steps to generate a parallel code which is actually implemented on the APE100/Quadrics machine.

References

[1] Clauss P., “An Efficient Allocation Strategy for Mapping Affine Recurrences into Space and Time Optima Regular Processor Arrays”, *Parcella*, Sep. 1994.

[2] Clauss P., Perrin G.R., “Optimal Mapping of Systolic Algorithms by Regular Instruction Shifts”, *IEEE International Conference on Application - Specific Array Processors, ASAP*, pp. 224-235, Aug. 1994.

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

[4] Lefebvre V., Feautrier P., "Automatic storage management for parallel programs",*Parallel Computing*, vol. 24, pp. 649-671, 1998.

[5] Herstein I.N., “Topics in Algebra”, *John Wiley and Sons* 1975.

[6] Lee P., Kedem Z., “Synthesizing Linear Array Algorithms from Nested For Loop Algorithms”, *IEEE Transaction on Computers*, Vol. C-37, No. 12, pp. 1578-1598, Dec. 1988.

[7] Lengauer Christian, “Loop Parallelization in the Polytope Model”, *CONCUR*, Vol. LNCS 715, pp. 398-416, 1993.

[8] Loechner V., Mongenet C., “OPERA: A Toolbox for Loop Parallelization”, *International Workshop on Software Engineering for Parallel and Distribuited Systems, PDSE*, 1996.

[9] Loechner V., Mongenet C., “A Toolbox for Affine Recurrence Equations Parallelization”,*HPCN ’95*, Milan, Vol. LNCS 919. pp. 263-268, May 1995.

[10] Loechner V., Mongenet C., “Solutions to the Communication Minimization Problem for Affine Recurrence Equations”, *EUROPAR ‘97*, Passau, Germany, Vol. LNCS 1300, pp. 328-337 August 1997.

[11] Mongenet C., “Data Compiling for System of Affine Recurrence Equations”, *IEEE Int. Conf. on Application - Specific Array Processors, ASAP*, pp. 212-223, Aug. 1994.

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

[13] Le Verge H., Van Dongen V. , Wilde D.K., “Loop Nest Synthesis Using the Polyhedral Library”, *IRISA - Internal Report*, No. 830, May 1994.

[14] Wilde D.K., “A Library for Doing Polyhedral Operations”, *IRISA - Internal Report*, No. 785, Dec. 1993.

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

[16] Lamport L., "The parallel execution of DO loops", *Comm. Of the ACM*, 17 (2):83-93, Feb. 1974.

[17] Darte A., Robert Y., "Affine-by-Statement scheduling of uniform and affine loop nests over parametric domains",*J. of Par. and Distr. Comp.*, Vol. 29, pp 43-59, 1995

[18] Darte A., Robert Y., "Mapping uniform loop nests onto distributed memory architectures", *Research Report n° 93-03 LIP -ENS Lyon*, Jan. 1993.

[19] Dowling M.L., "Optimal code parallelization using unimodular transformations", *Par. Comp.*,16,pp. 157-171. 1990.

[20] Loechner V., Wilde D.K., "Parameterized polyhedra and their vertices", *Int. J. of Par. Prog.*, Vol.25, No. 6, Dec. 1997.

[21] Clauss P., Loechner V., Wilde D., "Derinving formulae to count solutions to parameterized linear systems using Ehrhart polynomials: applications to the analysis of nested-loop programs", http://www.ee.byu.edu/~wilde/pubs.html.

[22] Clauss P., Loechner V., "Parametric analysis of polyhedral iteration spaces", *IEEE International conference on Application Specific Array Processors, ASAP*, Aug. 1996.

[23] Mongenet C., "Affine dependence classification for communications minimization", *ICPS Research Report No. 96-07*, http://icps.u-strasbg.fr/pub-96