Transformation of Systolic Algorithms for Interleaving Partitions

A.Fernández, J.M.Llabería, J.J.Navarro and M.Valero-García

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
Gran Capità s/n Mòdul D-4, 08034 - Barcelona. SPAIN

Abstract

A systematic method to map systolizable problems onto multicomputers is presented in this paper. A systolizable problem is a problem for which it is possible to design a Systolic Algorithm. This method selects and transforms the Systolic Algorithm into a parallel algorithm with high granularity. The communications requirements are reduced and thus the performance can be increased. The proposed scheme requires a classification of dependences, and it is based in the interleaved execution of several partitions of the Systolic Algorithm. The code to be executed in a processing element of the multicomputer system is obtained through application of the proposed systematic transformations to the original sequential code. By applying this method to the APP we illustrate their main features, and several performance measures for a torus of transputers system are presented, considering the various algorithms which are unified by the APP.

1. INTRODUCTION

Every Processing Element (PE) in a multicomputer comprises a processor with memory and communication links. This memory is only addressable by its associated processor, and the communication is performed through message passing along the point-to-point links interconnecting the PEs. Programming a multicomputer requires partitioning and distributing both computations and data among the PEs. Normally static scheduling is used in order to reduce the communication needs.

An attractive starting point to design parallel algorithms for multicomputer resides in the locality and regularity of communication between the cells in a Systolic Algorithm (SA) [KunL79]. On the other side, the SA can be systematically designed starting from a formal specification of the algorithm (a system of recurrences [Quin84], a nested loops structure [Mold83], etc.).

In order to generate code for each PE in a systolic processor, Lengauer [Leng89] takes an algorithm execution trace and searches for sequence patterns, and them uses iterative-like statements to do the final specification. In shared-memory multiprocessors Wolf and Lam [WolL90] presents a work where the specific operations of a sequential algorithm are reordered to increase parallelism, by means of unimodular transformation matrices.

Now, the fine granularity of the SA does not make it adequate to program the multicomputer in a straighforward manner due to the high costs of communication. Ibarra [IbaS89] and Fernández [FerL89] present two different proposals for reducing this communication costs. Ibarra and Sohn, in [IbaS89] accumulate the state information between adjacent regions in the time-space graph and send the collected symbols in packets. In [FerL89] block algorithms are used; one SA is selected for each class of block and them, respecting dependencies, several blocks are executed concurrently to reduce the number of messages, and to increase accordingly the granularity.

In this paper we propose a systematic method for: a) to select a SA for a systolizable problem from the different algorithms which can be obtained by means of known design techniques, b) to transform the SA in order to increase the granularity, and c) to transform the sequential code into the code to be executed on each PE of a multicomputer. The proposed systematic method is applied in this paper to the Algebraic Path Problem (APP).

The original problem to be solved should be specified using a nested loop structure like those considered in [Mold83]. From this specification, we design a SA for the problem. The SA is partitioned in order to adapt it to the size of the available multicomputer. In order to increase the granularity of the parallel algorithm, an analysis of dependencies among partitions is carried out, and it is used to derive a dependence graph among partitions. These partitions can be executed in a interleaved way when a) they are independent, or b) the dependences among these partitions are due to variables broadcasting; in this case interleaving is possible because the transmission of these variables is pipelined when designing the SA. This pipelining is quite usual in SA, and it allows a greater reduction in the number of transmitted messages. Interleaving means to us the concurrent execution of several partitions, when the scheduling among par-

TH0382-2/91/0000/0056$01.00 © 1991 IEEE

```
for k=0, N-1
 for i=k+1, N+k
  for j=k+1, N+k
   if i=j=k then
S1:  a[k,j,i]=f1(a[i,j,k-1])                    SEG1
   else if i=k then
S2:  a[k,j,i]=a[i,j,k-1]                        SEG2
   else if j=k then
S3:  a[k,j,i]=a[k,j,i-1]
S4:  a[i,k,j]=f3(a[i,j,k-1],a[k,j,i])           SEG3
   else
S5:  a[i,k,j]=a[i,k,j-1]
S6:  a[k,j,i]=a[k,j,i-1]
S7:  a[i,j,k]=f4(a[i,j,k-1],a[i,k,j],a[k,j,i])  SEG4
```

Problem	A^{-1}	transitive closure	shortest path
x	1	true	0
y	-1	true	0
z	0	false	∞
f1(a)	a	true	0
f3(a,b)	a/b	$a \wedge b$	a + b
f4(a,b,c)	a - b * c	$a \vee (b \wedge c)$	min(a,b+c)

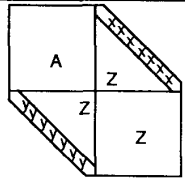


Figure 1. Algorithmic specification of the APP.

titions operations is done statically. Finally, we show that a further partitioning of partitions permits to increase the granularity of the parallel algorithm.

In section 2, the type of algorithm specification is explained, and a revision of the technique used to design and to do a first partitioning of the SA is presented, following the works in [Mold83] and [MolF86]. Section 3 presents the intuitive idea utilized here to increase the system granularity. Section 4 present the guides used to select the SA and the partitioning of the SA. Besides it is shown the way to transform the sequential code in order to obtain the code to be executed on each PE of the systolic processor. Section 5 describes the proposed method to increase the system granularity, as well as the transformations of sequential code which allow the obtention of the code to be executed on each PE. In section 6, an additional partitioning, adequate to increase the ability for partitions interleaving, is proposed. Section 7, finally present performance measures for a transputer-based multicomputer executing the different algorithms which are unified by the APP.

2. SA DESIGN AND PARTITIONING

We consider here algorithms which may be specified in the form of nested loops. The iterations space can be determined as an point lattice with the integer values of the loops indices. Each point in the lattice represent one computation (operations performed in one instance of the loop body) and the data dependences force a partial ordering of the integer points. The dependencies graph at the level of iterations, is composed of the lattice points and the dependence arcs between them.

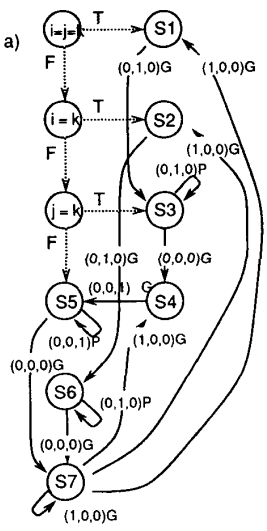
The class of algorithms considered in this paper are characterized by the presence of uniform data dependences, that is dependencies are independent of the points connected by them. The dependencies will be represented by means of vectors [Mold83] whose components correspond to the iterations space indices.

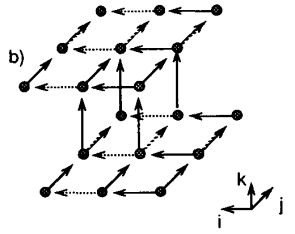
The first step to design a SA will be to make uniform the data dependencies. The problem has been studied by several authors [ForM84], [VanQ88], [WonD88], but no general solution has been given yet. However, for the majority of the Linear Algebra operations (Matrix Multiplication, Triangular Systems of Equations, LU Decomposition, etc.), the uniformization is possible.

Figure 1 shows the uniform algorithm specification for the APP. To solve the different algorithms unified by the APP, we must vary in a convenient manner the f1, f3 and f4 functions as well as the initial data of matrix A [Rot85]. A[i,j,-1] {$0 \le i, j \le N-1$} holds the initial data, and the result is stored in A[i,j,N-1] {$N \le i,j \le 2N-1$}. Figure 2a shows the APP dependence graph at the level of statements (SDG) and figure 2b show the dependence graph at the level of iteration space (DG). The dashed lines in the SDG represent control flow dependencies.

To design the SA we shall use the algebraic method proposed by Moldovan [Mold83] and to do the first partitioning we shall use the technique in [[MolF86]. Let us make a brief review of these methods.

We can obtain the SA by means of a linear transformation on the dependencies matrix D, whose columns (di) are the dependence vectors of the problem. This linear transformation is represented by the matrix





c)

statement	segment	dependence vector	dependence type
S1	SEG_1	$d_1=(0, 1, 0)^T$	G
S2	SEG_2	$d_2=(0, 1, 0)^T$	G
S3	SEG_3	$d_3=(0, 1, 0)^T$	P
S4	SEG_3	$d_4=(0, 0, 1)^T$	G
S7	SEG_4	$d_5=(1, 0, 0)^T$	G
S6	SEG_4	$d_6=(0, 1, 0)^T$	P
S5	SEG_4	$d_7=(0, 0, 1)^T$	P

Figure2. Dependence graphs for the APP. a) dependence graph at the level of statements, b) dependence graph at the level of iteration space and c) summary of the dependencies.

T, sized m-by-m, where m is the dimension of the problem indices space (the number of nested loops).

Matrix T will be divided into two parts. One row vector Π establishing a time ordering of the operations (temporal mapping). Vector Π is normal to the constant time hyperplanes. The points in the indices space which belong to the same hyperplane are computed in the same time instant. The remaining m-1 row vectors (submatrix S), are used to define the spatial assignment of indices space points to cells in the resulting SA (spatial mapping). Submatrix S can be represented by the projection vector which is normal to the plane formed by the m-1 rows in S. Calculations performed in the same cycle must be assigned to different cells. Consequently, the projection vector and the vector establishing the temporal mapping cannot be perpendicular. That is, matrix T must be non-singular.

To preserve dependencies between computations, vector Π must satisfy the condition $\Pi \cdot d_i > 0$, $\forall$ d_i. Accordingly, one point of the indices space, v, is calculated in the cell $S \cdot v$ during cycle $\Pi \cdot v$.

The spatial mapping S determines the topology of communication among SA cells (columns of $S \cdot D$). This submatrix can be selected taking into account a topology of cells interconnection. Let P be a matrix whose columns represent the previously fixed interconnection topology. Matrix S is chosen among the possible solutions to the diophantic equation $S \cdot D = P \cdot K$ ($P_{ij} = \{-1, 0, 1\}$) where matrix K indicates how to use the communication links between available cells. Elements in matrix K must satisfy the following condition:

$$\sum_j k_{ji} \leq \Pi \cdot d_i \quad \text{with } k_{ji} \geq 0$$

This expression indicates that the communication of data associated with dependence d_i must be done using $\sum k_{ji}$ times the communication links.

The method proposed in [MolF86] is used to partition the SA. A grouping of adjacent points in the indices space serves to determine one such partition. These partitions are called bands. The whole computation is carried out by chaining the execution of the different bands in the fixed size Systolic Processor (SP).

The space of indices is partitioned in m-1 of their dimensions. These dimensions are those corresponding to the spatial mapping. Each row in matrix S which is used in the spatial mapping, is observed as a vector normal to a cutting plane in the space of indices.

It is assumed that any band is executed completely before initiating the execution of the following one. So, if band Bt is executed after band Bs, no computations in Bs should depend on computations included in Bt. This requirement imposes a condition to the partitioning hyperplanes, represented by the rows of S:

$$Sj \cdot di \geq 0 \qquad (\forall Sj) \,\&\, (\forall\ di)$$

Each of these bands can be executed by a SA of a given fixed size. Specifically, for the case of a 3D index space, computation v is assigned to band BBr,Bs where:

$$Br = \lfloor S1 \cdot v / W \rfloor$$
$$Bs = \lfloor S2 \cdot v / W \rfloor$$

S1 and S2 are rows of matrix S and W-by-W is the number of PEs of the SP. Moreover, this computation will be performed by PEpq where:

$$p = (S1 \cdot v) \bmod W$$
$$q = (S2 \cdot v) \bmod W$$

It can be seen that this partitioning scheme assigns computations in the boundary of the bands to PEs in the boundary of the SP. So, in order to chain the execution of the bands we need some links between the PEs in the boundary (wraparound) of the SP and a buffer to store data.

3. INCREASE OF GRANULARITY USING BAND INTERLEAVING

The partitioned SA can be executed in a multicomputer by assigning each cell of the SA to one PE of the multicomputer. The parallel algorithm, obtained as explained in last section, has a fine granularity. Because the multicomputer executes a series of SAs, PEs exchange short messages and very frequently (a single matrix element for each link and computation). So, the important communication requirements can lead to a poor performance when executing the algorithm in the multicomputer. Our objective is to increase systematically the granularity of the algorithm, reducing in this way the communication requirements. On this section an intuitive idea of the proposed method, concerned to reach goal, is provided.

The communication time for a message in a multicomputer is modeled by two terms with the following expression:

$$t = tstart + n\ tbyte$$

where tstart is the time required to establish the communication, n is the message size in bytes, and tbyte is the time required to transmit one byte. The total communication time is proportional to these two terms and, consequently, it is interesting to reduce their effect.

The influence of the term tstart can be reduced by transmitting several data items in a single message. This can be achieved with independent bands. Data corresponding to independent bands, and traversing a link, can be packed into a single message passing through that link.

Both the influence of tstart and tbyte can be reduced by minimizing the number of transmitted messages. This features can be achieved by detecting bands which use same input data. This is as common characteristic in SA, typical in systolizable problems with broadcasting variables. When the algorithm is uniformed, the dependencies are pipelined, due to the existence of broadcasting variables and are thus converted to constant dependencies. From now on, we shall denote these dependencies as propagated dependencies. If all the dependencies between the Br and Bs bands are due to propagated dependencies, then the bands can be executed in any order.

On the other hand, in order to avoid the storage of data in the PEs, which will be later used when executing other bands, either independent or with propagated dependencies between them, we propose an interleaved execution of the bands. We understand this interleaving as the concurrent execution of several bands, where the scheduling of operations is performed in a static way.

Now, let us explain the meaning of this interleaving through a simple example where propagated dependences appear. Figure 3 shows two bands Bs and Bs+1 belonging to a given DG. If these bands are exe-

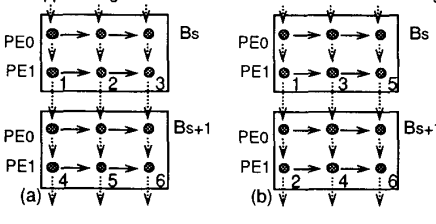


Figure 3. An example of band interleaving.
(a) Execution ordering for operations assigned to PE1 if bands Bs and Bs+1 are executed one after the other.
(b) Execution ordering if band are interleaved.

cuted one after the other then PE1 executes its computations in the order indicated in figure 3a. To perform each of these computations PE1 must receive a data item from PE0. Because the dependence between bands Bs and Bs+1 is propagate, the data item received by PE1 to perform operation (1) is the same that the one received later to perform (4). The same stands for computations (2) and (5), (3) and (6) and so on. The number of messages from PE0 to PE1 can be reduced if PE1 executes operation (4) immediately after operation (1), using in both operations the received data item. Analogously, operation (5) is executed immediately after operation (2) and so on. The new execution ordering is show in figure 3b. In that case we say that bands Bs and Bs+1 are executed in an interleaved way.

4. DESIGN AND PARTITIONING OF THE SA

This section describes the technique used to obtain a parallel algorithm to be executed on a multicomputer, starting from a systolizable problem. The method consist in the following stages: a) dependence classification and decomposition into segments, b) SA selection and partitioning, and c) code generation.

4.1 Dependence classification and decomposition into segments

We identify two types of dependences: propagated dependences (P) and generated dependences (G). Dependences of type P arise from statements of the form V(I) = V(I-d), where d is one of the dependence vectors. This kind of dependences have been introduced during the uniformization of the original code. The rest of dependences are of type G.

After the uniformization of the code, each statement is surrounded by nested loops and its execution can be subjected to some conditions on the indices of the nested loops (guard commands). We define a segment as the set of consecutive statements with the same guard commands. We shall assume that the intersection between the guard commands of segments is empty. This can be achieved by reordering, and creating new segments, if needed. By doing so, in one point of the indices space just the statements of a single segment are executed.

Figure 1 shows the segments for the APP. Figure 2b shows the graph dependencies between statements, SDG. Figure 2c shows a summary of the dependencies in the APP. For each dependence we identify the statement in which the dependence is originated, the segment that statement belongs to, the associated dependence vector and the type of the dependence.

4.2 SA Selection and Partitioning

In our method, the selection of matrix T will be guided by the following criterion: data flows moving from cell to cell in the resulting SA should be, if possible, those associated with propagated dependences. This criterion favours band interleaving.

For the APP we are interested in a SA such that the data associated with the generated dependence (1,0,0) remain static in their PEs. Data associated with the other (propagated) dependencies (0,1,0) and (0,0,1) are transmitted through links. Provided that the available multicomputer has a mesh topology, matrix P is the identity matrix. The selected vector Π is Π=[1,1,1], and the vectors for partitioning are S1=[0,1,0] and S2=[0,0,1]. The spatial mapping matrix is:

$$S = \begin{pmatrix} S1 \\ S2 \end{pmatrix}$$

4.3 Code generation

For the code generation we assume an asynchronous system, that is, there is not a global clock which determines the computation cycles. Communication and synchronization among PEs is done by means of the send and receive primitives with a rendez-vous protocol.

The code to be executed in each PE can be obtained by transforming the original code using the following matrix:

$$T_S = \begin{pmatrix} S \\ \Pi_0 \end{pmatrix}$$

where $\Pi 0$ is a vector such that $\Pi\, d_i \geq \Pi 0\, d_i \geq 0$, the matrix Ts is non-singular and a matrix T_Π such that $T = T_\Pi \cdot Ts$, exists. The Ts transformation arranges computations in such a way that the calculations assigned to a PE are easily identified. Note that, in essence, one spatial transformation has been applied. The T_Π transformation is implicitly obtained when the communication and synchronization primitives are placed.

The transformation Ts, and any matrix which is a permutation of its rows, is valid because any row vector multiplied by di is greater than (or equal to) zero. When applying Ts to m perfectly nested loops we obtain the loop structure as follows:

```
for I1=a,b
   for I2=f2L,f2U,step2
      ...
      for Im=fmL,fmU,stepm
         ...
      endfor
      ...
   endfor
endfor
```

where a and b are constants, f_{iL} and f_{iU} are functions of variables I_1 ,...., I_{i-1} (which, in some cases, may be non-linear, i.e. ceiling, floor, mod...) and $step_i$ is the step of the for and it depends on the determinants of submatrices in Ts. If Ts is a unimodular matrix, functions f_{iL} and f_{iU} are linear, and $step_i = 1$ for any i. Each indexing function of the algorithm variables is transformed by applying Ts^{-1}. Let g be the original indexing function, then $g\ Ts^{-1}$ is the new indexing function.

4.3.1 Problem-size Dependent Code Generation

After application of Ts, the resultant m-1 external loops identify the PEs (processor space) and parametrize the embraced code. The parametrized code to be executed on a PE is the most internal loop, which establishes the local timing ordering for operations inside the PE. One PE becomes unnecessary if the corresponding internal loop does not perform any iteration.

When the code is parametrized, the number of variables dimensions (processors space) to be stored inside the PEs is reduced in m-1 (they become one-dimensional vectors). The vectors maximum size is determined by the magnitude of the component in the remaining dimension of the dependence vector.

The global timing ordering of operations (Π) is done by means of the send / receive communication and synchronization primitives. We use the value $\Pi \cdot d_i$ (with $S \cdot d_i \neq 0$) to identify the size of the buffer between PEs and the value $S \cdot d_i$ to identify the direction of the data flow. The communication primitives are placed at the beginning (receive) and at the end (send) of the loop body. These primitives have tree parameters; the first one identifies the link to do the communication, the second one identifies the data flow between two statements (dependence identifiers), and the third one is the data item to be transmitted.

Communication primitives directly manage buffers of size one. If the buffer size is greater than one, array variable (bufc) is used in the PE which is the source of the dependence. The data item is stored in bufc, instead of being sent after this calculation, and its transmission is delayed a number of iterations (systolic cycles) equal to the delay ($\Pi \cdot d_i$) minus one. Bufc may be also placed in the destination cell of the dependence.

The send and receive statements could be distributed inside the loop body, in order to increase parallelism between communication and computation in the PE. This is possible when the transmitted and the received data item are identical. Besides, the bounded iterations space specified by the program must be taken into account. In some boundary points of the iterations space possibly no data must be sended or received. The information needed to transform the code consists of the input data placement, the SDG, and the guard commands of the segments.

The assignment statements which are source of a dependence such that $S \cdot d_i \neq 0$ are substituted by send primitives. Before those statements which are destination of a dependence such that $S \cdot d_i \neq 0$, a receive statement is placed. It suffices one send or receive statement for each identical dependence vector having the same dependence identifier. Let us remember that, by construction, only one code segment is executed on each point of the iteration space.

In the APP, we have:

$$Ts = \begin{pmatrix} S1 \\ S2 \\ \Pi 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{matrix} \Pi=(1\,1\,1) \\ \Pi\cdot(0\,0\,1)^T = 1 \end{matrix} \qquad \Pi\cdot(0\,1\,0)^T = 1$$

Figure 4a shows the code obtained by applying transformation Ts. In figure 4b the parametrized code for a PE, together with communication primitives, is shown. The send and receive primitives have only two parameters, because the communication links between PEs support just one dependence. Figure 4c shows the PEs and the interconnection topology between PEs, for the APP with size N=2. In this figure, those PEs performing no iteration have been removed.

```
for r=0, 2N-1
  for s=0, 2N-1                                    (a)
    for t=max(0,r-N,s-N), min(N-1,r,s)
      if r=s=t then
        a[t,s,r]=f1(a[r,s,t-1])
      else if r=t then
        a[t,s,r]=a[r,s,t-1]
      else if s=t then
        a[t,s,r]=a[t,s,r-1]
        a[r,t,s]=f3(a[r,s,t-1],a[t,s,r])
      else
        a[r,t,s]=a[r,t,s-1]
        a[t,s,r]=a[t,s,r-1]
        a[r,s,t]=f4(a[r,s,t-1],a[r,t,s],a[t,s,r])
```

```
PE(r,s) (0 ≤ r,s ≤ 2N-1)
for t=max(0,r-N,s-N),min(N-1,r,s)
  if r=s=t then
    send(@(1,0),f1(a))
  else if r=t then
    send(@(1,0),a)
  else if s=t then
    receive(@(1,0),y)
    if r≠N+t then send(@(1,0),y)
    send(@(0,1),f3(a,y))
  else
    receive(@(0,1),x)
    if s≠N+t then send(@(0,1),x)
    receive(@(1,0),y)
    if r≠N+t then send(@(1,0),y)
    a=f4(a,x,y)
```

(b)

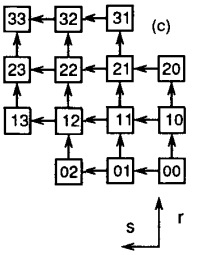


Figure 4. Problem-size dependent code.

4.3.2 Generation of Partitioned Code

The algorithm partitioning is achieved by dividing the processor space in equal size bands. This space is represented by the external loops of the transformed code. Each loop is transformed into two loops and one auxiliary variable. The external loop identifies the band according one of its dimensions, and the internal loop identifies the PE in one of its dimensions. The auxiliary variable depends upon the iteration variables of the loops and follows the same sequence as the iterations variable of the original loop.

Let us consider the iteration statement "for I_i=f_L,f_U,stepi", where f_L and f_U are non-linear functions of the iterations variables in the most external loops, and stepi is the for step. Figure 5a shows the transformation of this statement, where f_L<stepi, W>stepi is assumed, and the auxiliary variable is I_i=B_iW+p. The resulting transformed code is parametrized for a PE. The iteration statements "for p = inner, upper, stepi" are substituted by one conditional statement "if (inner≤p≤upper and (p-inner) mod stepi = 0) then", where p is the parameter. Figure 5b shows the parametrized code for the PE(...p...) along the i dimension.

The dimensions of variables stored in one PE is now m-1, where m is the dimension size of the iterations space. Now, N_i / W is the magnitude of variables in each dimension, N_i being the original magnitude along dimension I_i. Thus, the auxiliary variable now used is I_i'=B_i. The resulting code executed the bands in a lexicographical order.

Figure 6a shows the code obtained by partitioning the APP code seen in figure 4a. Figure 6b shows the parametrized code for each PE. As the bands are executed in a lexicographical order, the buffer between boundary PE is a FIFO queue.

5. TECHNIQUE TO INCREASE GRANULARITY

The code obtained in section 4 exhibits low granularity. In the present section, the code is to be transformed, in order to increase its granularity, by using band interleaving. To identify which bands may be interleaved, the type of dependences between bands must be determined and the directions of interleaving must be chosen. Typical directions to do the interleaving analysis are linear functions of the directions numbering the bands. One possible choice of bands with propagated dependencies is based upon the study on the directions in which the propagated dependencies have been transformed. This paper restrict the analysis to the ortogonal directions numbering the bands.

5.1 Identification the type of dependences between bands

As seen before, the dependencies of an algorithm can be associated with the segment where they are originated. As an example, dependence d2 in our example, of type G, is originated in SEG2. Therefore,

```
for Bi=⌊ fL/W ⌋⌊ fU/W ⌋                          (a)
  if Bi=⌊ fU/W ⌋ then upper=fU mod W
  else upper=W-1
  inner=⌈ (Bi W-fL)/stepi ⌉stepi+fL) mod W
  for p=inner,upper,stepi
    Ii=BiW+p
    ...
```

```
for Bi=⌊ fL/W ⌋⌊ fU/W ⌋                          (b)
  if Bi=⌊ fU/W ⌋ then upper=fU mod W
  else upper=W-1
  inner=⌈ (Bi W-fL)/stepi ⌉stepi+fL) mod W
  if (inner≤p≤upper)and((p-inner)mod stepi = 0) then
    Ii=BiW+p
    ...
```

Figure 5. a) Code after partitioning. b) Parametrized code for the PE(...p...).

```
for Br=0⌊2N-1/W⌋                                        (a)
  if Br=⌊2N-1/W⌋ then pu=(2N-1) mod W
  else pu=W-1
  for p=0,pu
    r=BrW+p
    for Bs=0⌊2N-1/W⌋
      if Bs=⌊2N-1/W⌋ then qu=(2N-1) mod W
      else qu=W-1
      for q=0,qu
        s=BsW+q
        for t=max(0,r-N,s-N),min(N-1,r,s)
          if r=s=t then
            a[t,s,r]=f1(a[r,s,t-1])
          else if r=t then
            a[t,s,r]=a[r,s,t-1]
          else if s=t then
            a[t,s,r]=a[t,s,r-1]
            a[r,t,s]=f3(a[r,s,t-1],a[t,s,r])
          else
            a[r,t,s]=a[r,t,s-1]
            a[t,s,r]=a[t,s,r-1]
            a[r,s,t]=f4(a[r,s,t-1],a[r,t,s],a[t,s,r])
```

```
PE(r,s) (0 ≤ p,q ≤ W-1)                                 (b)
for Br=0⌊2N-1/W⌋
  if Br=⌊2N-1/W⌋ then pu=(2N-1) mod W else pu=W-1
  if (0 ≤ p ≤ pu) then
    r=BrW+p
    for Bs=0⌊2N-1/W⌋
      if Bs=⌊2N-1/W⌋ then qu=(2N-1) mod W else qu=W-1
      if (0 ≤ q ≤ qu) then
        s=BsW+q
        for t=max(0,r-N,s-N),min(N-1,r,s)
          if r=s=t then
            send(@(1,0),f1(a[Br,Bs]))
          else if r=t then
            send(@(1,0),a[Br,Bs])
          else if s=t then
            receive(@(1,0),y)
            if r≠N+t then send(@(1,0),y)
            send(@(0,1),f3(a[Br,Bs],y))
          else
            receive(@(0,1),x)
            if s≠N+t then send(@(0,1),x)
            receive(@(1,0),y)
            if r≠N+t then send(@(1,0),y)
            a[Br,Bs]=f4(a[Br,Bs],x,y)
```

Figure 6. Problem-size independent code.

any band containing computations belonging to SEG2 is a band that generates the dependence d2.

In order to identify the type of dependences between bands we must determine which segments are executed in every band. The dependencies between bands satisfy S·di ≥ 0. We say that there is a generated dependence between bands Br and Bs if any segment executed in Br produces a type G dependence, whose destination is one point in the iteration space belonging to Bs. Similarly, there is a propagated dependence between Br and Bs, if the dependence between any point of the iterations space belonging to Bs and depending upon Br, is a propagated dependence. Now, we illustrate how to carry out this analysis in APP using segment SEG2.

The input data for this analysis are the transformation Ts and the iterations space region where the segment is executed: a) the guard command, b) the loop bounds. For SEG2 this region is as follows:

$$i = k$$
$$0 \le k \le N-1$$
$$0 \le i \le N-1$$
$$k \le j \le N+k$$

To identify which bands in a segment are executed, we distinguish a five steps algorithm:

1) Representation of a generic point in the segment. In SEG2 this point is:

$$v = (k, k, j)^T \text{ with } i = k.$$

2) Obtention of the generic band BBr,Bs in which the segment is executed. This is achieved by applying partitioning to the generic point v;

$$Br = \lfloor S1\ v / W \rfloor = \lfloor t / W \rfloor \qquad (a)$$
$$Bs = \lfloor S2\ v / W \rfloor = \lfloor s / W \rfloor$$

3) Determination of the dimensions values interval identifying a band BBr,Bs. It is done by partitioning of the iteration space transformed by Ts.

$$0 \le r \le N-1 \qquad 0 \le Br \le \lfloor N-1 / W \rfloor$$
$$t \le s \le N+t \qquad \lfloor t / W \rfloor \le Bs \le \lfloor N+t / W \rfloor \qquad (b)$$

4) Searching for a relationship between the dimensions identifying a band BBr,Bs, if it exists. In our example, we can derive from (a) and (b):

$$Br \le Bs$$

5) Determination of a constant value interval for the dimensions identifying a band. In the example of SEG2, the modified range is (by taking into account the bounds $0 \le t \le N-1$ and Ts) :

$$0 \le Br \le \lfloor 2N-1 / W \rfloor$$

Once this analysis has been applied to all the segments, we obtain the following information:

SEG1: $Br = Bs \qquad 0 \le Br \le \lfloor N-1 / W \rfloor \qquad 0 \le Bs \le \lfloor N-1 / W \rfloor$

SEG2: $Br \leq Bs$ $0 \leq Br \leq \lfloor N\text{-}1 / W \rfloor$ $0 \leq Bs \leq \lfloor 2N\text{-}1 / W \rfloor$

SEG3: $Br \geq Bs$ $0 \leq Br \leq \lfloor 2N\text{-}1 / W \rfloor$ $0 \leq Bs \leq \lfloor N\text{-}1 / W \rfloor$

SEG4: $\forall$ Br, Bs $0 \leq Br \leq \lfloor 2N\text{-}1 / W \rfloor$ $0 \leq Bs \leq \lfloor 2N\text{-}1 / W \rfloor$

Intersection of the intervals where each segment is executed leads to identify, in terms of the executed segments, the following types of bands:

A) $(Br=Bs) \& (0 \leq Br \leq \lfloor N\text{-}1/W \rfloor) \& (0 \leq Bs \leq \lfloor N\text{-}1/W \rfloor) \rightarrow$ SEG1, SEG2, SEG3 and SEG4

B) $(Br<Bs) \& (0 \leq Br \leq \lfloor N\text{-}1/W \rfloor) \& (0 \leq Bs \leq \lfloor 2N\text{-}1/W \rfloor) \rightarrow$ SEG2 and SEG4

C) $(Br>Bs) \& (0 \leq Br \leq \lfloor 2N\text{-}1/W \rfloor) \& (0 \leq Bs \leq \lfloor N\text{-}1/W \rfloor) \rightarrow$ SEG3 and SEG4

D) $(\forall Br,Bs) \& (\lfloor N\text{-}1/W \rfloor < Br \leq \lfloor 2N\text{-}1/W \rfloor) \& (\lfloor N\text{-}1/W \rfloor < Bs \leq \lfloor 2N\text{-}1/W \rfloor) \rightarrow$ SEG4

This means to us that type A bands generate dependencies in the communication directions (1,0) and (0,1), because all the segments are executed there. Type B bands generate dependencies in direction (1,0) and propagate dependencies in direction (0,1); type C bands propagates dependencies along (1,0) and generate dependencies along (0,1); and the remaining (type D) bands propagate dependencies along (1,0) and (0,1).

In the APP, type A bands must be executed individually because there are generated dependencies. Execution of types B and C bands may be interleaved, along directions Br and Bs respectively, because their dependencies are propagated. Moreover, as types B and C bands are relatively independent, their execution can be performed in a interleaved way. Type D bands can be interleaved along directions Br and Bs.

Due to the new execution ordering for the bands (interleaved execution), the buffer between boundary PEs is not a FIFO queue.

5.2 Generation of interleaved code

Interleaving the execution of bands with propagated dependences requires to reorder code. The iteration statement specifying the local timing ordering must embrace the iteration statements identifying the directions along which the bands are interleaved. The code travels now over the bands interleaved in a consecutive way (sec. 5.2.1). On the other hand, the placement of statements transmitting data due to propagated dependencies, must be modified. Data due to generated dependencies, from different bands, transmitted along directions where dependencies are not propagated, must also be grouped into a single message (sec. 5.2.2).

5.2.1 Band Scheduling

If the dependencies between bands, along all the directions numbering them, are propagate dependencies, then the code will have the structure of iteration statements as shown in figure 7a. In this code the most external loop specifies the temporal ordering. In cases where the above stated condition does not hold, it is required to isolate explicitly the direction (s) along which the dependencies between bands are propagated. Figure 7b shows the structure of the iteration statements in the code, where BI3 and BI4 are the directions of interleaving. Along BI2 a sequential execution is performed, and along BI1 the values has been fixed.

Let us present now a systematic method to construct the interleaved code.

The initial code is obtained through transformation T_S where the row $\Pi 0$ has been permuted with the rows of S. This code specifies the temporal ordering in its external loop, and the PEs space in their m-1 internal loops. The code is now partitioned and parametrized according to the same criteria explained in section 4.3.2. Once these transformations have been applied to the original code, the resulting code is shown in figure 8a.

```
(a)  for t=0,N-1
       for BI1= fI1L,fI1u
         for BI2= fI2L,fI2u
           ...

(b)  ...
     BI1= constant
     for BI2= fI2L,fI2u
       for t=a,b
         for BI3= fI3L,fI3u
           for BI4= fI4L,fI4u
             ...
```

Figure 7. Iteration statements structure: a) All dependencies are propagated. b) There are propagated and generated dependencies.

Now, the interleaved code can be obtained through repetitive application of two steps. The first step isolates band zones where dependencies of the same type along all the analyzed directions exist. The second step isolates directions along which the dependencies are generated, as well as directions along which the dependencies are propagated.

Two different situations may arise when search for adjacent band zones where, along all the directions numbering the analyzed bands are of the same type:

a) Dependencies along all directions are generated dependencies. In this case the bands must be sequentially executed. Doing so requires interchanging the loops associated to bands with that of the temporal ordering. Performing this interchange may require to start from the original transformation Ts in order to determine the loop bounds.

b) Dependencies along all directions are propagated dependencies. The bands are executed in interleaved way, and no modification of the iteration statements structure is needed.

The zones of adjacent bands are determined by means of intervals intersection. From these intervals, convex band zones are built, and used to generate the code. The achieved code is a sequence of groups with iteration statements travelling over all the bands. The sequence ordering must respect the dependencies between bands. Possibly some zones cannot be separated because dependencies between bands exist.

In the APP, propagated dependencies along all directions are detected in the zone identified by the following range: ($\lfloor$N-1/W$\rfloor$<Br$\leq$ $\lfloor$2N-1/W$\rfloor$) & ($\lfloor$N-1/ W$\rfloor$<Bs$\leq\lfloor$2N-1/ W$\rfloor$). The generated code is composed by two sets of loops. The first set travels over all the bands and the zone with propagated dependencies is excluded by means of a conditional statement (if (Br$\leq\lfloor$N-1/W$\rfloor$) or (Bs$\leq\lfloor$N-1/W$\rfloor$) then). The second set of loops travels over the bands with propagated dependencies along all the directions (figure 8b).

In the second step the directions with generated dependencies are isolated from the directions with propagated dependencies. Given a group of loops, we shall distinguish several situations:

a) Along some directions, the dependencies between bands are always generated, and along the remaining directions are always propagated dependencies. The bands are executed serially along the first set of directions, and the bands are executed interleaved along the second set. To do so, the ordering of the iteration statements must be modified. First we must place those loops whose directions shows generated dependencies, and later the loops whose direction shows propagated dependencies.

b) The dependencies between bands may be propagated or generated, along all the directions. We

```
PE(p,q) (0 ≤ p,q ≤ W-1)                                    (a)
for t=0,N-1
  for Br=⌊t/W⌋,⌊t+N/W⌋
    if Br=⌊t/W⌋ then pi=t mod W else pi=0
    if Br=⌊t+N/W⌋ then pu=(t+N) mod W else pu=W-1
    if (pi ≤ p ≤ pu) then
      r=BrW+p
      for Bs=⌊t/W⌋,⌊t+N/W⌋
        if Bs=⌊t/W⌋ then qi=t mod W else qi=0
        if Bs=⌊t+N/W⌋ then qu=(t+N) mod W else qu=W-1
        if (qi ≤ q ≤ qu) then
          s=BsW+q
          if r=s=t then
            send(@(1,0),f1(a[Br,Bs]))
          else if r=t then
            send(@(1,0),a[Br,Bs])
          else if s=t then
            receive(@(1,0),y)
            if r≠N+t then send(@(1,0),y)
            send(@(0,1),f3(a[Br,Bs],y))
          else
            receive(@(0,1),x)
            if s≠N+t then send(@(0,1),x)
            receive(@(1,0),y)
            if r≠N+t then send(@(1,0),y)
            a[Br,Bs]=f4(a[Br,Bs],x,y)
```

```
for t=0,N-1                                                (b)
  for Br=⌊t/W⌋,⌊t+N/W⌋
    ...
    for Bs=⌊t/W⌋,⌊t+N/W⌋
      ...
      if (Br ≤ ⌊N-1/W⌋) or (Bs ≤ ⌊N-1/W⌋) then
        ...
    endfor
  endfor
endfor
for t=0,N-1
  for Br=⌊N/W⌋,⌊t+N/W⌋
    ...
    for Bs=⌊N/W⌋,⌊t+N/W⌋
      ...
    endfor
  endfor
endfor
```

Figure 8. a) Transformed code ready to begin the interleaving. b) Structure of iteration and conditional statements after applying the first step of the systematic method.

propose to build a loop structure travelling over the bands, from the most external ones, in slices. This can be done because S·di≥0 holds. This way to travel over the bands requires the identification of a pattern, and the knowledge of the number of repetitions. The code structure is as follows:

```
for counter =0,MAX
  ...
  BIi= fi(counter)
  if (fIiL ≤ BI1 ≤ fIiu) then
     for t= c , d
        for BIj= fIjL, fIju
           ...
        end for
        ...
     end for
```

where, once the value in one dimensions is fixed, the remaining directions are traversed with iterations statements. The iterations variable counter, establishes how many times the pattern is repeated.

Successive steps may continue the code modification by applying the same method to new sets of loops, until the dependencies between bands, along the directions which specify the for statements, are propagated dependencies

In the APP, this second step must be applied to the first set of loops. The pattern is obtained by following the diagonal in the plane which identifies the bands Br=Bs and is repeated $\lfloor N-1/W \rfloor$ times. Figure 9a shows the structure of the iterations statements for this case.

This code specifies, in first place, the execution of a type A band, and the subsequent execution of type B bands (direction Bs) free of dependencies, and the later execution of type C bands (direction Br) which are also dependence-free.

The code corresponding to segments to be executed is the only one to be included as loops body in each type of bands. Figure 9b shows, as an example, the code sequence for type B bands.

5.2.2 Communication messages

After ordering the iteration statements, the information transmission statements must be placed in turn. Both directions, those of propagated and those of generated dependencies, must be taken into account in order to do so. Along the former ones, the number of messages and the number of transmitted data are reduced. Besides to determine where the transmission and reception statements must be placed, another requirement is to calculate the size of an auxiliary vector to store the information, until is used in the computation of all the interleaved bands. Along the latter ones, the number of transmitted messages is reduced by grouping several data items, belonging to different bands, into a single message.

In general, the guard commands determining the transmission or reception of a message are not, function of the propagates dependencies direction. Therefore, we can place the send and receive statements before the iteration statement which specifies the direction of interleaving. The possible guard commands dependent of the interleaving direction are due to boundary bands which does not use all the PEs. The guard commands can be rewrited as a function of the PEs.

(a)

```
for c=0,⌊N-1/W⌋
  Br=c
  Bs=c
  for t=0,BrW+W-1
    ...
  endfor
  Br=c
  for t=0,BrW+W-1
    ...
    for Bs=c+1,⌊t+N/W⌋
      ...
    endfor
  endfor
  Bs=c
  for t=0,BrW+W-1
    ...
    for Br=c+1,⌊t+N/W⌋
      ...
    endfor
  endfor
endfor
```

(b)

```
Br=c
for t=0,BrW+W-1
  if Br=⌊t/W⌋ then pi=t mod W else pi=0
  if (pi ≤ p ≤ W-1) then
    r=BrW+p
    for Bs=c+1,⌊t+N/W⌋
      if Bs=⌊t+N/W⌋ then qu=(t+N) mod W else qu=W-1
      if (0 ≤ q ≤ qu) then
        s=BsW+q
        if r=t then
          send(@(1,0),a[Br,Bs])
        else
          receive(@(0,1),x)
          if s≠N+t then send(@(0,1),x)
          receive(@(1,0),y)
          send(@(1,0),y)
          a[Br,Bs]=f4(a[Br,Bs],x,y)
```

Figure 9. a) Pattern to travel over zones with propagate and generate dependencies. b) Complete code for type B bands.

```
Br=c
for t=0,BrW+W-1                    (a)
  ...
  if r≠t then
    receive(@(0,1),x)
    if p≠W-1 then send(@(0,1),x)
  for Bs=c+1,⌊t+N/W⌋
    ...
  endfor
endfor
```

```
Br=c
for t=0,BrW+W-1                                              (b)
  ...
  if r≠t then
    receive(@(0,1),x)
    if p≠W-1 then send(@(0,1),x)
  if q ≤ (t+N) mod W then receive(@(1,0),Y[c+1:⌊t+N/W⌋])
  else receive(@(1,0),Y[c+1:⌊t+N/W⌋-1])
  for Bs=c+1,⌊t+N/W⌋
    ...
    if r=t then Y[Bs]=a[Br,Bs]
    else a[Br,Bs]=f4(a[Br,Bs],x,Y[Bs])
  endfor
  if q ≤ (t+N) mod W then send (@(1,0),Y[c+1:⌊t+N/W⌋])
  else send (@(1,0),Y[c+1:⌊t+N/W⌋-1])
endfor
```

Figure 10. Placement of send and receive statements in type B bands: a) propagate dependencies, b) generate dependencies.

The size of the auxiliary vector required to store propagated data, is given by the values of the component, related to the temporal ordering of the operation of the transformed vector of propagated dependencies ($\Pi o \cdot d_i$). When this value is equal to zero, the reception buffer suffices because the data is used in all the operations, before a new data is received.

In the APP, the transformed code when interleaving is applied along direction Bs, has the structure shown in figure 10a. Observe that it can be transmitted immediately after its reception, because the value is not modified.

Along the directions when dependencies are not propagated, several data must be grouped into a message. The code transformation must isolate the reception of all the data from the computation to be performed, and both operations must be isolate from the sending of all results. The code part which performs the computations by travelling over all the bands, instead of sending the results, stores them into a vector. Later, they are transmitted in a single message with all the results. Similarly, the received data are stored in a vector, which is accessed once the computations have been performed. Both the receive and the send statements are executed depending upon some guard commands which are determined from the original guard commands associated to such statements. The message size is calculated from the number of repetitions of the iteration statement and from the guard commands.

In the APP, the structure of the code part shown before becomes that shown in figure 10b, after grouping messages.

6. FURTHER PARTITIONING OF BANDS

Further partitioning of bands into sub-bands permits to increase the number of sub-bands that can be interleaved, increasing in this way the granularity of the algorithm.

A band that has been identified as a generator of a dependence may actually generate that dependence only during a part of the computations. Coming back to APP, during the first computations, band Bii just propagates d6 and d7 (because all these computations belong to SEG4). Only during the final computations, the band executes SEG1, SEG2, SEG3 and SEG4 and so, generates d1, d2 and d4. Similar situations arise with other bands.

If we isolate that part of the computations where the band just propagate dependences these parts could be interleaved. This fact suggests a further decomposition of bands into sub-bands.

6.1 Partitioning Identification

In order to determine the new partitioning, we have to identify, for every segment, which part of each band contains computations belonging to that segment. We now illustrate how to carry out this analysis in the case of SEG2 from our example.

When one type of band has been selected, the initial information are the guard commands of the segment and the bands area containing that segment. For example, this information are as follows, for SEG2:

$$\begin{array}{c} Br \le Bs \\ 0 \le Br \le \lfloor n-1 \,/\, W \rfloor \\ 0 \le Bs \le \lfloor 2n-1 \,/\, W \rfloor \\ r = t \end{array} \qquad \text{band area } (Br, Bs): \begin{cases} BrW \le r \le BrW+W-1 \\ BsW \le s \le BsW+W-1 \\ \max(0, r-n, s-n) \le t \le \min(0, r, s) \end{cases}$$

We must determine in which band zone are satisfied the segment guard commands. We have r=t for seg-

ment SEG2. By intersecting the i and k ranges in the band area, we obtain that SEG2 is executed when t≥BrW.

After application of this analysis to every segment, we obtain the following informations:

SEG1:	Br = Bs	$0 \le Br \le \lfloor n-1 / W \rfloor$	$0 \le Bs \le \lfloor n-1 / W \rfloor$	$t \ge BrW$
SEG2:	Br ≤ Bs	$0 \le Br \le \lfloor n-1 / W \rfloor$	$0 \le Bs \le \lfloor 2n-1 / W \rfloor$	$t \ge BrW$
SEG3:	Br ≥ Bs	$0 \le Br \le \lfloor 2n-1 / W \rfloor$	$0 \le Bs \le \lfloor n-1 / W \rfloor$	$t \ge BsW$
SEG4:	∀ Br, Bs	$0 \le Br \le \lfloor 2n-1 / W \rfloor$	$0 \le Bs \le \lfloor 2n-1 / W \rfloor$	∀ t

The partitioning plane S0 = (100) allows to separate the zone of propagated dependencies from the zone of generated dependencies, inside every band. That hyperplane satisfies the condition S0·di ≥ 0 (∀ di). The size of the sub-band along the direction of partitioning is obtained by intersecting the intervals from the boundary bands down to the inner.

In the APP, the size of the boundary bands is equal to W along the partitioning direction, and these bands generate dependencies along some directions. Now, inner sub-bands with this size do propagate dependencies along all the communication directions. Accordingly, the size of sub-bands is selected to be W.

By intersecting the intervals in which every segment is executed, we can distinguish the following types of sub-bands, as a function of their executed segments:

for $t \le \min(BrW, BsW)$

A1) (Br=Bs) & $(1 \le Br \le \lfloor N-1/W \rfloor)$ & $(1 \le Bs \le \lfloor N-1/W \rfloor) \rightarrow$ SEG4

B1) (Br<Bs) & $(1 \le Br \le \lfloor N-1/W \rfloor)$ & $(1 \le Bs \le \lfloor 2N-1/W \rfloor) \rightarrow$ SEG4

C1) (Br>Bs) & $(1 \le Br \le \lfloor 2N-1/W \rfloor)$ & $(1 \le Bs \le \lfloor N-1/W \rfloor) \rightarrow$ SEG4

D1) (∀ Br, Bs) & $(\lfloor N-1/W \rfloor < Br \le \lfloor 2N-1/W \rfloor)$ & $(\lfloor N-1/W \rfloor < Bs \le \lfloor 2N-1/W \rfloor) \rightarrow$ SEG4

for $t > \min(BrW, BsW)$

A2) (Br=Bs) & $(0 \le Br \le \lfloor N-1/W \rfloor)$ & $(0 \le Bs \le \lfloor N-1/W \rfloor) \rightarrow$ SEG1, SEG2, SEG3 and SEG4

B2) (Br<Bs) & $(0 \le Br \le \lfloor N-1/W \rfloor)$ & $(0 \le Bs \le \lfloor 2N-1/W \rfloor) \rightarrow$ SEG2 and SEG4

C2) (Br>Bs) & $(0 \le Br \le \lfloor 2N-1/W \rfloor)$ & $(0 \le Bs \le \lfloor N-1/W \rfloor) \rightarrow$ SEG3 and SEG4

D2) (∀ Br, Bs) & $(\lfloor N-1/W \rfloor < Br \le \lfloor 2N-1/W \rfloor)$ & $(\lfloor N-1/W \rfloor < Bs \le \lfloor 2N-1/W \rfloor) \rightarrow$ SEG4

For the type B bands, we see that the SEG2 and SEG4 segments are to be executed. The B2 type sub-bands generate dependencies along the direction (1,0) for those index point such that t ≥ BrW, and the B1 type sub-bands propagate dependencies along the direction (1,0) for those index point when t<BrW. For the C type bands, we see that the SEG3 and SEG4 segments are to be executed. The C1 type sub-bands propagate dependencies along the direction (0,1) for those index point satisfying t<BsW, and the C2 type sub-bands generate dependencies along the direction (0,1) for those index point satisfying t ≥ BsW. For the A type bands, we see that the SEG1, SEG2, SEG3 and SEG4 segments are to be executed. The A2 type sub-bands generate dependencies along the directions (1,0) and (0,1) for those index points with t≥BrW ; and the A1 type sub-bands propagate dependencies along both directions for the index points with t<BsW=BrW.

6.2 Code Transformation

To find a repetitive pattern along the dimension used to make the partitioning, is required in order to generate code. The steps explained in the previous section will then be applied to this pattern.

In the APP, the pattern is obtained by travelling over all the sub-bands according to the order specified by the following sequence of iteration statements:

```
for Bt=0,⌊N-1/W⌋
  ...
```

Given a value of Bt, we see that those sub-bands satisfying $\lfloor t/W \rfloor+1 \le Br, Bs \le \lfloor t+N/W \rfloor$ propagate dependencies along both directions. Therefore, the code is restructured in a way that allows the interleaved execution of these sub-bands (first step). The remaining sub-bands propagate dependencies along one dimension (second step). The sub-bands satisfying $\lfloor t/W \rfloor+1 \le Br \le \lfloor t+N/W \rfloor$ and Bt=Bs propagate dependencies along the dimension (0,1), and the sub-bands satisfying $\lfloor t/W \rfloor+1 \le Bs \le \lfloor t+N/W \rfloor$ and

```
for Bt=0,⌊N-1/W⌋                                    (a)
  Br=Bt
  Bs=Bt
  for t=BtW,min(N-1, BtW+W-1)
    ...
  endfor
  Br=Bt
  for t=BtW,min(N-1, BtW+W-1)
    ...
    for Bs=Bt+1,⌊t+N/W⌋
      ...
    endfor
  endfor
  Bs=Bt
  for t=BtW,min(N-1, BtW+W-1)
    ...
    for Br=Bt+1,⌊t+N/W⌋
      ...
    endfor
  endfor
  for t=BtW,min(N-1, BtW+W-1)
    for Br=Bt+1,⌊t+N/W⌋
      ...
      for Bs=Bt+1,⌊t+N/W⌋
        ...
      endfor
    endfor
  endfor
endfor
```

```
for t=BtW,min(N-1, BtW+W-1)
  for Br=Bt+1,⌊t+N/W⌋                               (b)
    if Br=⌊t+N/W⌋ then pu=(t+N) mod W else pu=W-1
    if (0 ≤ p ≤ pu) then
      r=BrW+p
      for Bs=Bt+1,⌊t+N/W⌋
        if Bs=⌊t+N/W⌋ then qu=(t+N) mod W else qu=W-1
          if (0 ≤ q ≤ qu) then
            s=BsW+q
            receive(@(0,1),x)
            if s≠N+t then send(@(0,1),x)
            receive(@(1,0),y)
            if r≠N+t then send(@(1,0),y)
            a[Br,Bs]=f4(a[Br,Bs],x,y)
      endfor
  endfor
endfor
```

Figure 11. a) Pattern to travel over the sub-bands. b) Complete code for type D1 sub-bands.

Bt=Br propagate dependencies along (1,0). Therefore, the structure of the iteration statements allowing the interleaving along these directions is as shown in figure 11a.

The complete code for the zones where dependencies are propagated along both directions, can be seen in figure 11b.

In the APP, once this transformation has been applied, we only need to place the transmission and reception statements in order to reduce the number of transmitted messages. The technique explained in section 5.2.2 serves this purpose. The above shown complete code parts becomes them as follows:

```
for t=BtW,min(N-1, BtW+W-1)
    if p ≤ (t+N) mod W then receive(@(0,1), X[Bt+1:⌊t+N/W⌋])
    else receive(@(0,1), X[Bt+1:⌊t+N/W⌋- 1])
    if q ≤ (t+N) mod W then receive(@(1,0), Y[Bt+1:⌊t+N/W⌋])
    else receive(@(1,0), Y[Bt+1:⌊t+N/W⌋-1])
    if p≠W-1 then
        if p ≤ (t+N) mod W then send(@(0,1), X[Bt+1:⌊t+N/W⌋])
        else send(@(0,1), X[Bt+1:⌊t+N/W⌋-1])
    if q≠W-1 then
        if q ≤ (t+N) mod W then send(@(1,0), Y[Bt+1:⌊t+N/W⌋])
        else send(@(1,0), Y[Bt+1:⌊t+N/W⌋-1])
    for Br=Bt+1,⌊t+N/W⌋
        for Bs=Bt+1,⌊t+N/W⌋
            if Br=⌊t+N/W⌋ then pu=(t+N) mod W else pu=W-1
            if Bs=⌊t+N/W⌋ then qu=(t+N) mod W else qu=W-1
            if (0 ≤ p ≤ pu) and (0 ≤ q ≤ qu) then
                a[Br,Bs]=f4(a[Br,Bs],X[Bs],Y[Br])
        endfor
    endfor
```

Note that, in this case, the message is transmitted immediately after its reception, because it is not modified.

7. EXPERIMENTAL RESULTS

The code generated in section 6 has been optimized by hand, with a reduction in the number of conditional statements, and has been executed on a 4-by-4 PEs multicomputer based on Transputers [Whit85] T414B and T800C. The results are plotted in figure 12. This figure shows the efficiency obtained varying

the size of matrix A (N_by_N) for the matrix inversion, transitive closure and shortest path problems. The efficiency has been measured using the following expression:

$$\text{Efficiency} = \frac{T_{seq}}{P\ T_{par}}$$

where Tseq is the time required to solve the problem in a single PE, using the best serial algorithm, and Tpar is the time required by P PEs.

The efficiency of the parallel algorithms with 640-by-640 matrices approaches the 75%, 75% and 65% for the matrix inversion, transitive closure and shortest path problems, respectively, using T800C processors. The efficiency of the parallel algorithms with 300-by-300 matrices approaches the 95%, 75% and 65% for the matrix inversion, transitive closure and shortest path problems, respectively, using T414B processors.

8. CONCLUSIONS

In this paper we have presented a systematic method for mapping systolizable problems onto multicomputer. The inputs to the method are a nested loop based specification of the problem to be solved and the interconnection topology of the multicomputer.

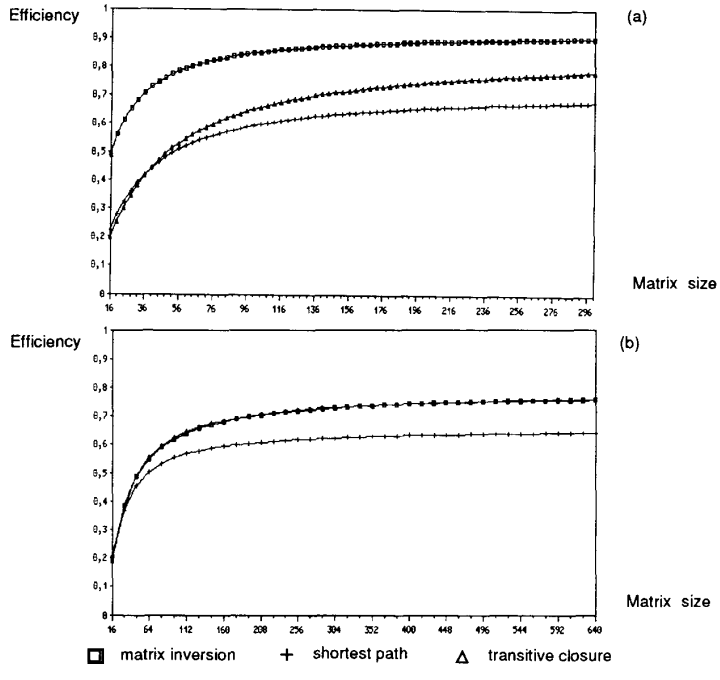


Figure 12. Efficiencies of the problems of the APP with different processors: a) Transputer T414B and b) Transputer T800C.

For the first stages of the method (design and partitioning of a SA for the problem) we use techniques proposed by other authors. The main contribution of the paper is the procedure to increase the granularity of the partitioned SA. This increase of granularity is necessary in order to reduce the communication cost of the algorithm. This issue is specially important when designing parallel algorithms for multicomputers, where the effects of communication between PEs cannot be neglected.

A dependence between bands analysis is performed, in order to increase the granularity. The bands which are either independent or with propagated dependences, are executed in an interleaved way. The possibility of finding bands with propagated dependencies is common in SA, and serves to significantly reduce the number of messages.

Also, we have presented here a systematic method to transform code in order to obtain the program to be executed in PE, starting from the original sequential specification.

The method has been illustrated with an example consisting in the design of a parallel algorithm to solve the APP on a multicomputer with a torus topology. We have obtained some promising results running the code on a ring of Transputers for the different algorithms unified by the APP.

The method has been applied to map others problems like Matrix Multiplication or LU Decomposition on a multicomputer with ring and torus topologies.

Future works are envisaged towards the distribution of buffers in the boundary PEs, inside the local PE memories, and towards increasing the generated code quality.

Acknowledgments

This work is supported by the Ministry of Education of Spain (CICYT TIC-299/89). Also this work is partially supported by ESPRIT's Parallel Computing Action (#4146).

References

[FerL89] A.Fernández, J.M.Llabería, J.J.Navarro, M.Valero-García and M.Valero. "On the Use of Systolic Algorithms for Programming Distributed Memory Multiprocessors", Proc. Int'l. Conference on Systolic Arrays, Prentice Hall, pp. 631.640, 1989.

[ForM84] J.A.B.Fortes and D.I.Moldovan. "Data Broadcasting in Linearly Scheduled Array Processors", Proc. 11th Int'l Annual Symp on Computer Architecture, pp. 224-231, 1984.

[IbaS89] O.H.Ibarra and S.M.Sohn. "On Mapping Systolic Algorithms onto the Hypercube". Int'l Conf. on Parallel Processing, Vol.I, pp 121-124, 1989.

[KunL79] H.T.Kung and C.E.Leiserson. "Systolic Arrays (for VLSI)", Sparse Matrix Proc. 1978, 1979, Society for Industrial and Applied Mathematics (SIAM), pp.256-282.

[Leng89] C.Lengauer. "Towards Systolizing Compilation: an Overview". Proc. Parallel Architectures and Languages Europe, PARLE'89. Lecture Notes in Computer Science 366, Springer-Verlag, pp. 253-272, 1989.

[Mold83] D.I.Moldovan. "On the Design of Algorithms for VLSI Systolic Arrays", Proc. of the IEEE, vol 71, no. 1, 1983. pp. 113-120.

[MolF86] D. I. Moldovan and J. A. B. Fortes. "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", IEEE Trans. on Computers, Vol. 35, n. 1, 1986, pp. 1-12.

[Quin84] P.Quinton. "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations", 11th Int'l Annual Symp. on Computer Architecture, pp.208-214, 1984.

[VanQ88] V.VanDongen and P.Quinton. "Uniformization of Linear Recurrence Equations: a Step Towards the Automatic Synthesis of Systolic Arrays", Proc. Int'l Conf. on Systolic Arrays, pp. 473-482, 1988.

[Rot85] G.Rote. "A Systolic Array for the Algebraic Path Problem (Shortest Paths; Matrix Inversion), Computing 34, pp. 191-219, 1985.

[Whit85] C. Whitby-Stevens. "The Transputer". Proc 12th Int'l Symp. of Computer Architecture, 1985, pp. 292-300.

[WolL90] M.E.Wolf and M.S.Lam. "Maximizing Parallelism via Loop Transformations". 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine-California, August 1990.

[WonD88] Y.Won and J.M.Delosme. "Broadcast Removal in Systolic Algorithms", Proc Int'l Conf. on Systolic Arrays, pp. 403-412, 1988.