A BLOCK ALGORITHM FOR THE ALGEBRAIC PATH PROBLEM AND ITS EXECUTION ON A SYSTOLIC ARRAY

Fernando J. Núñez and Mateo Valero
Departamento de Arquitectura de Computadores
Facultad de Informática de Barcelona, Universidad Politécnica de Cataluña
Pau Gargallo 5, 08028 Barcelona, SPAIN

This work was supported by the Ministry of Education and Science of Spain (CAICYT) under contract PA85-0314.

ABSTRACT

The solution of the Algebraic Path Problem (APP) for arbitrarily sized graphs by a fixed-size systolic array processor (SAP) is addressed. The APP is decomposed in two subproblems. A SAP is designed for each one, both SAPs combined produce a highly implementable versatile SAP (VSAP). The proposed VSAP has $p \times p$ processing elements (PEs), solving the APP of an N-vertex graph in $N^3/p^2 + N^2/p + 3p-2$ cycles. With slight modifications in the operations performed by the PEs the problem is optimally solved in $N^3/p^2 + 3p-2$ cycles.

1. INTRODUCTION

Important problems such as the Transitive Closure (TC), the All-Pairs Shortest Path (SP), and the Gauss-Jordan Elimination are particular cases of the more general APP. Its solution is a computing intensive task, it has cubic complexity with respect to the size of the problem.

Using $(N+1)^2$ PEs in a hexagonally connected array the APP is solved in 7N cycles in [1]. An array that solves the APP in 5N cycles with $N \times (N+1)$ PEs can be found in [2]. In [3] a dependence graph based design method obtains new and already proposed arrays for the APP. There, optimum arrays obtaining the result in 5N cycles with $N \times N$ PEs are proposed. In fact, many SAPs devoted to special cases of the APP can be generalized for solving it. For example, SAPs for the TC and the SP [4], [5], [6], [7].

There is always a computation too large for a given array. Mapping larger sized computations into smaller arrays is of great practical interest. The original problem is decomposed into subproblems whose sizes fit the available SAP size. This paper is devoted to decompose the APP and to design an array for solving them and combining their partial results. APP instances like the TC have been partitioned [8], [9], but to our knowledge there are no works about the partitioning of the APP in the literature.

In the next section the basics of the APP are briefly reviewed. Section 3 is devoted to describe an iterative block algorithm for the APP. Then, in section 4 this algorithm is rearranged for solving the APP using only two matrix operations, i. e., the APP is decomposed into two subproblems. The design of a SAP for solving each subproblem, and their combination to form the resulting VSAP is commented in section 5. Block I/O details, changes for achieving an optimal execution, and performance expressions are given in section 6. Finally, in section 7 the outstanding points addressed in the paper are discussed.

2. THE ALGEBRAIC PATH PROBLEM

Consider a weighted directed graph $G = <V,E,w>$, where V is its N-vertex set, E $N \times N$ its edge set, and $w:E \rightarrow S$ an edge weighting function whose values are taken from the set S. It belongs to a path algebra $<S,+,\times,*,0,1>$ together with two binary operations, "addition" $+:S \times S \rightarrow S$, and "multiplication" $\times:S \times S \rightarrow S$, and a unary operation called closure $()^*:S \rightarrow S$. Constants 0 and 1 belong to S. Hereby, + and $\times$ will be used in infix notation, whereas the closure of an element a will be noted as a^*. Additionally, * will have precedence over $\times$ and this over +. This algebra is a closed semi-ring, i. e., an algebra fulfilling the following axioms [10]: + is associative, commutative, with 0 as neutral; $\times$ is associative with 1 as neutral, it distributes over $+$. Element 0 is absorptive with respect to $\times$. The equality $a^* = 1 + a \times a^* = 1 + a^* \times a$ must hold.

The weight w(p) of a path $p = (e1, e2, \ldots, em)$, $ei \in E$, is defined as $w(p) := w(e1) \times w(e2) \times \ldots \times w(em)$. The APP is the determination of the sum of weights of all the possible paths between each pair of vertices (i,j). If P(i,j) is the set of all the possible paths from i to j, the APP is to find the values [11]:

$$d(i,j) := \sum_{p \in P(i,j)} w(p)$$

We associate a weight matrix $A = \{a(i,j)\}$, $1 \leqq i,j \leqq N$ with graph G, where $a(i,j) := w((i,j))$ if $(i,j) \in E$, and $a(i,j) := 0$ otherwise. This matrix representation permits us to formulate the APP as the computation of a sequence of matrices $A^{(k)} = \{a(i,j)^{(k)}\}$ $0 \leqq k \leqq N$. The value $a(i,j)^{(k)}$ is the weight of all the possible paths from vertex i to j with intermediate vertices v, $1 \leqq v \leqq k$. Initially, $A^{(0)} := A$, then $A^* := A^{(N)}$, where $A^* := \{a(i,j)^*\}$ is an $N \times N$ matrix satisfying $a(i,j)^* = d(i,j)$ if $(i,j) \in P(i,j)$, and $a(i,j)^* := 0$ otherwise.

The algorithms proposed to solve path problems formulated in matrix terms are known as matrix-methods; most are referenced in [11]. Among them we find algorithms for the TC [12], [13]; the SP [14]; and the classic Gauss and Gauss-Jordan eliminations to compute the inverse of a matrix.

It was observed that the same program schemes could solve these problems. A program scheme is a program with fixed control but where the sets over which the variables take their values, and the meaning of the algebraic operations is left uninterpreted [10]. The majority of the program schemes useful for the APP come from Linear Algebra, like Jacobi and Gauss-Seidel methods, and the mentioned Gauss and Jordan eliminations [15]. The Gauss-Jordan elimination has been recently introduced for solving the APP [1]. The array described in [2] is based on it.

There are many interesting problems that are specializations of the APP [15]. If the algebra is boolean: $S = \{0,1\}$; "+"=OR, "x"=AND, $0^* = 1$, $1^* = 1$, "0"=0, and "1"=1; the APP finds the TC. For the SP: $S = [0, +\infty) \cup \{+\infty\}$, "+"=min, "x"=+, "0"=$+\infty$, "1"=0; closure is the constant operation 0. For matrix inversion: $S = \mathbb{R}$, "+" and "$\times$" are the ordinary addition and multiplication with respective neutrals 0 and 1. The closure is $a^* = 1/(1-a)$ for $a \neq 1$.

3. APP BLOCK DECOMPOSITION

Block algorithms are a useful tool for extracting parallelism from problems. For example, when solving matrix problems, both, input and output matrices can be split into blocks or submatrices. Different block subproblems can be allocated to different processors. Processors must combine their partial results in order to attain the desired global solution. This can be named as interblock parallelism.

Block algorithms are also used for solving problems in parallel using systolic arrays. However, the approach is completely different. In this case, blocks must fit the size of the available array. Subproblems are solved one after the other, with a proper sequencing to combine their partial solutions efficiently. By doing so, the original problem is solved. In the context of systolic computing, these are known as size-independent algorithms, and problem decomposition is named partitioning [16], [17]. Systolic arrays explote intrablock parallelism. An array can be seen as a single processor running a sequential algorithm, but operating on blocks instead of elements.

Normally, problem partitioning generates subproblems of different nature. For example, the LU-decomposition is split into smaller matrix products with accumulation, linear systems of equations, and LU-decompositions [18]; the TC requires solving smaller TCs and performing boolean matrix multiplications with accumulation [9]. We are interested in solving all the resulting subproblems using one single array.

Suppose that in the process of solving the APP by obtaining the matrix sequence $A^{(0)}, \ldots, A^{(N)}$, matrix $A^{(k)}$ has been already computed. Also assume we know how to solve the APP for matrices with $p \times p$ elements or smaller. The point is how to obtain

$A^{(k+p)}$ through block operations. The involved blocks are shown in figure 1. Block A1 is a p×p block on the diagonal. Note that A2 and A3 have p×(N-p) and (N-p)×p non-zero elements respectively, while A4 has (N-p)×(N-p). Then, Theorem 1 indicates how to obtain $A^{(k+p)}$ from $A^{(k)}$ through block operations.

Operator ()* represents the APP of a block, + and × are the natural extension to matrices of addition and multiplication of the underlying algebra. Figure 2 illustrates a block operation.

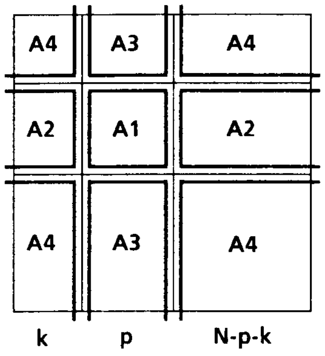


Figure 1. The blocks referenced in Theorem 1.

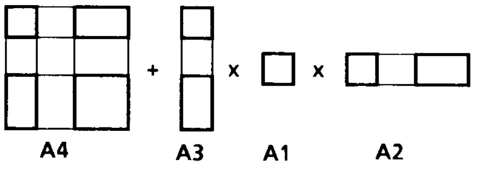


Figure 2. A block operation illustrated.

Theorem 1: Under the preceding assumptions, $A^{(k+p)}$ can be obtained from $A^{(k)}$ as follows:

$$A1^{(k+p)} := A1^{(k)*} \qquad (3.a)$$
$$A2^{(k+p)} := A1^{(k)*} \times A2^{(k)} \qquad (3.b)$$
$$A3^{(k+p)} := A3^{(k)} \times A1^{(k)*} \qquad (3.c)$$
$$A4^{(k+p)} := A4^{(k)} + A3^{(k)} \times A1^{(k)*} \times A2^{(k)} \qquad (3.d)$$

Proof: A k-path in G is defined as a path whose intermediate vertices must belong to the set {1..k}.With each matrix $A^{(k)}$ we can associate a graph $G^{(k)} = <V,E^{(k)},w^{(k)}>$ whose edge weights are the sum of all the possible k-paths in G between each ordered pair of vertices. A (k+p)-path in G is either a k-path (an edge in $G^{(k)}$) or a path in $G^{(k)}$ whose intermediate vertices are taken from the set {k+1..k+p}. An example of the latter path is depicted in figure 3-a. If i,j∈{k+1..k+p} then (3.a) follows trivially. Consider now (3.d), then i,j∈{1..N}-{k+1..k+p}. The general form of this path is represented in figure 3-b. Note that $a(v1,vm)^{(k+p)}$, an element of $A1^{(k)*}$, is the weight of all the (k+p)-paths from v1 to vm. Thus, the weight of this path from i to j is $a(i,v1)^{(k)} \times a(v1,vm)^{(k+p)} \times a(vm,j)^{(k)}$. If vertices are any in the specified sets, the overall operations can be expressed in matrix terms as (3.d). The term $A4^{(k)}$ adds the contributions of the k-paths. Cases (3.b) and (3.c) are simplifications of (3.d).□

It is worth mentioning that block schemes for the APP can also we derived by extending to blocks the properties fulfilled by elements, and then applying any valid program scheme to blocks. Nevertheless, we have decided to present Theorem 1 because it is more general. It permits to design parallel algorithms using different block sizes for the same problem. In addition, by employing graph instead of algebraic terms, it provides insight in the effect of the underlying operations.

4. TWO-PRIMITIVE APP PARTITIONING

As said, all subproblems must be solved by a single array. In this paper, we are concerned with 2-D SAPs although the results also apply to one-dimensional machines. For the moment, let us advance that the fixed-size array developed in the next section has p×p processing elements (PEs).

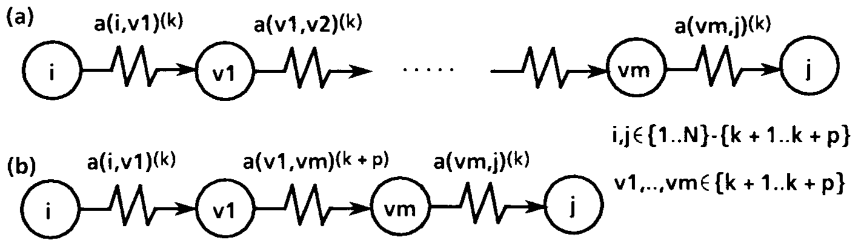


Figure 3. (a) A (k + p)-path in $G^{(k)}$ with intermediate vertices in {k + 1..k + p}; (b) a general form of these paths.

Hence, it is natural to partition the N×N matrices to be computed into square blocks with p×p elements. Without loss of generality, assume p divides N evenly. The remaining of this section will deal with blocks, or groups of them. Some matrix notation conventions are mandatory.

Suppose an N×N matrix A that is split into N/p×N/p blocks of size p×p. A(i,j) is one of these blocks placed at the i-th block-row, j-th block-column. The i-th block row, and the j-th block column are respectively denoted as A(i,&) and A(&,j). A(i,&-j) refers to the i-th row without block A(i,j). Under this convention, A(&-i,&-j) is obtained from A, eliminating its i-th and j-th row and column. Subranges also need to be specified. For instance, A(i,j..k) is composed of blocks in the i-th row from column j to column k.

Now, we will present Algorithm 1, that computes the APP of an N×N matrix from the basic assumption that it is known how to obtain the APP of a p×p block. Algorithm 1 is a direct outcome of Theorem 1. It is row oriented. A column oriented version could be obtained by simply interchanging block indexes, and inverting the order in which blocks are multiplied.

```
Algorithm 1
$B^{(0)} := A$
for k := 1 to N/p
  $B(k,k)^{(k)} := B(k,k)^{(k-1)*}$                                          (4.a)
  $B(k,\&-k)^{(k)} := B(k,k)^{(k)} \times B(k,\&-k)^{(k-1)}$                    (4.b)
  $B(\&-k,\&)^{(k)} := B(\&-k,\&-k)^{(k-1)} + B(\&-k,k)^{(k-1)} \times B(k,\&)^{(k)}$   (4.c)
end for
$A^* := B^{(N/p)}$
```

Note that $A^{(kp)} = B^{(k)}$, thus, after N/p iterations results $A^* = A^{(N)} = B^{(N/p)}$.

At this point, we have seen that the APP is decomposable into smaller APPs and matrix multiplications (with or without accumulation), observing the rules of the underlying algebra. The next stage is to modify Algorithm 1 for making it more suitable of systolization. It is interesting to minimize the number of primitives executable by the VSAP. Only two primitives, named P1 and P2, suffice for the APP. Let us define them. Consider three matrices X, Y, Z with sizes p×p, p×m, and p×m respectively. We define P1 and P2 as: $P1(X,Y) = X^* \times Y$; and $P2(X,Y,Z) = X \times Y + Z$.

Algorithm 2 solves the APP using only P1 and P2.

```
Algorithm 2
$B^{(0)} := A$
for k := 1 to N/p
  $B(k,\&)^{(k)} := P1(B(k,k)^{(k-1)}, B(k,1..k-1)^{(k-1)}|Ip|B(k,k+1..N)^{(k-1)})$   (4.d)
  for i := 1 to N/p with $i \neq k$
    $B(i,\&)^{(k)} := P2(B(i,k)^{(k-1)}, B(k,\&)^{(k)}, B(i,\&-k)^{(k-1)})$          (4.e)
  end for
end for
$A^* := B^{(N/p)}$
```

Lines (4.a) and (4.b) obtain $A(k,\&)^{(k)}$. Using P1 they can be grouped into line (4.d),where the symbol | denotes matrix juxtaposition, and Ip is the p×p identity matrix. On the other hand, line (c) is equivalent to:

```
for i := 1 to N/p with $i \neq k$
    $B(i,\&)^{(k)} := B(i,\&-k)^{(k-1)} + B(i,k)^{(k-1)} \times B(k,\&)^{(k)}$        (4.f)
end for
```

allowing to use P2 for computing $A(i,\&)^{(k)}$ in line (4.e).

5. A VERSATILE SAP FOR THE APP

When designing versatile SAPs, the additional complexity caused by partitioning must be minimized in order to attain an economic and implementable machine. More specifically, it is mandatory to reduce the overhead caused by partitioning in the external hardware and communications, as well as in the required control. The additional delays, and PE complexity have to be minimized too [19].

Intuitively, the VSAP can be obtained by overlapping, in some sense, two SAPs, one for P1 and the other for P2. Moreover, both candidate SAPs must be as similar as possible. An efficient sequencing between consecutive subproblems also has direct influence on performance. Most of these points are taken into account in papers dealing with the mapping of partitioned systolic algorithms into fixed-size arrays.

In order to attain a highly implementable design, additional constraints could be observed. For example, by forcing equal input and output data formats, there is no need for a data rearranging network between the VSAP and the external memory modules. In other words, a single storage scheme can be used for all the matrices.

Another interesting restriction is to have memory modules only along one side of the SAP's polygon. This fact avoids having different algorithms to perform the same operation over matrices entering the array from different points.

Algorithm 2 has been designed to use the same block in a (maybe long) sequence of operations. Hence, in order to be used, this block is preloaded in the VSAP. This helps to reduce the internal and the external I/O bandwidth requirements. All these constraints have influenced the design of the VSAP.

The Jordan algorithm for the quasi-inversion of a matrix can be extended for computing P1 [15]. The following equation must hold: $A^* = A \times A^* + Ip$; then by postmultiplying both sides by B we can write: $P1(A,B) = A \times P1(A,B) + B$. In the latter equation P1(A,B) is the unknown. The Jordan algorithm finds it by computing a sequence of matrices $A^{(k)}$, $B^{(k)}$, and $M^{(k)}$, $1 \leq k \leq p$, with respective dimensions p×p, p×m, and p×p. Initially $A^{(0)} := \{a(i,j)^{(0)}\} = A$, and $B^{(0)} := \{b(i,j)^{(0)}\} = B$; then $A^{(k)} := M^{(k)} \times A^{(k-1)}$; $B^{(k)} := M^{(k)} \times B^{(k-1)}$; $1 \leq k \leq p$. $M^{(k)}$ is obtained from the k-th column of $A^{(k-1)}$. Figure 4 shows its structure. It can be shown that $B^{(p)} = A^* \times B = P1(A,B)$.

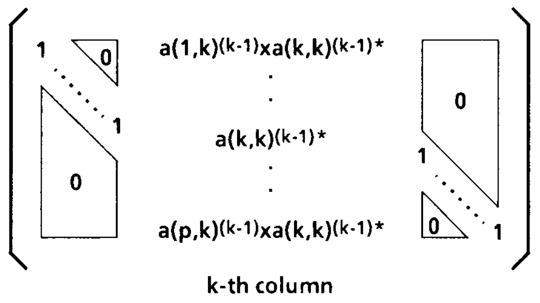


Figure 4. The structure of matrix $M^{(k)}$.

This recurrent formulation suggests to use a pipeline with p stages for computing P1. Matrices will enter into the pipeline by rows. Hence, p processing elements (PEs) per stage will be required. Stage k performs the k-th recurrence. It receives first columns k to p of $A^{(k-1)}$ and delivers columns k + 1 to p of $A^{(k)}$. In this section we use $A^{(k)}$ to refer to a submatrix formed with columns k + 1 to p of this matrix.

Figure 5 illustrates the structure and operation of the third stage when p=4. In this example, columns 3 and 4 of $A^{(2)}$ enter skewed. Rows are circularly reordered, they reach the third stage in the order 3, 4, 1, 2. The first element reaching the stage is $a(3,3)^{(2)}$, then $a(3,3)^{(3)} = a(3,3)^{(2)*}$ is computed and stored in the bottom PE. The other elements in the third column are stored without modification. A stage is initialized when all its PEs have stored their corresponding values. Once initialized, stage k premultiplies by $M^{(k)}$ any entering matrix with p rows and an arbitrary number of columns. Specifically, in the example of figure 5, the fourth column of $A^{(2)}$ and matrix $B^{(2)}$ are premultiplied by $M^{(3)}$. Thus, in the general case, stage k receives $A^{(k-1)}$ followed by $B^{(k-1)}$, sending $A^{(k)}$ and $B^{(k)}$ to stage k + 1.

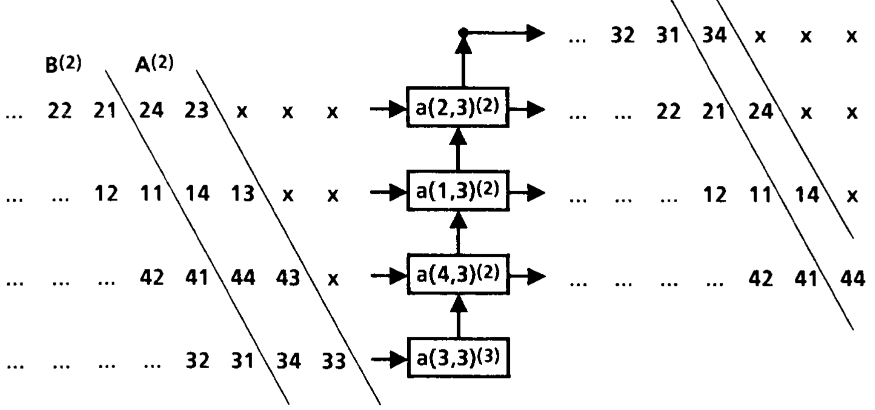


Figure 5. Operation of the third row when executing P1 for p = 4.

When the bottom PE is initialized, the remaining elements of the entering row are premultiplied by the stored value. This updated row is sent upwards through the vertical links. The other PEs premultiply this row by the values in their registers. The resulting rows are added to the rows received from the West ports, obtaining the updated rows sent through the East ports. In order to preserve the skew, the row traveling upwards is delayed one cycle (the dot means a unit delay) when it leaves the top PE. Note that the updated rows exit the stage ordered and skewed as required by the next stage. Primitive 1 requires m + 4p-2 cycles to be obtained by this SAP.

It must be mentioned that in [3] this array was also obtained by means of a dependence graph based method. There, the array was used for computing A*, a particular case of P1(A,B).

Primitive P2 can be obtained in p recurrences too. A sequence of p×m matrices $C^{(0)}, C^{(1)}, \ldots, C^{(p)}$ must be computed. Initially, $C^{(0)} := C$. The recurrence is: $C^{(k)} := C^{(k-1)} + A(\&,k) \times B(k,\&)$; $1 \leq k \leq p$. Now, the matrix notation refers to elements: A(&,k) and B(k,&) are the k-th column and the k-th row of matrices A and B respectively. As for P1, a p-stage pipeline with p PEs per stage solves the recurrence. Stage k computes $C^{(k)}$. Now, figure 6 will help us to explain the operation of a stage. Again, the third row is shown for a case with p=4. Matrices also enter by rows. Stage 3 stores A(&,3) and permits the remaining columns of A (column 4 in this case), to reach the next stage without modification. The skew and row reordering are the same as for P1. Row B(3,&) reaches first the bottom PE, being sent upwards from PE to PE. Once a PE is initialized and matrix A has passed, it computes a row by premultiplying B(3,&) by the value stored in its register. This row is added to the row of $C^{(2)}$ it receives, obtaining a row of $C^{(3)}$. Primitive P2 requires two links from PE to PE. One is used for passing B(k,&), and the other for reordering the bottom row of A and $C^{(k)}$. Primitive 2 is also executed in m + 4p-2 cycles.

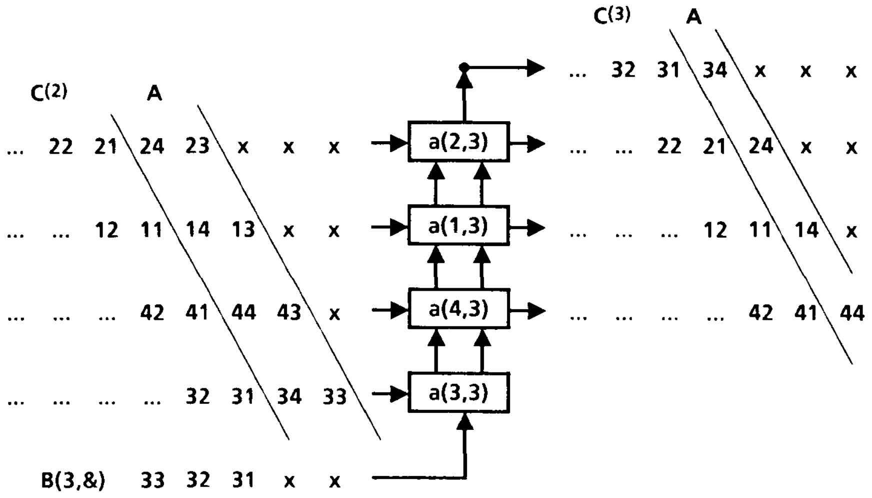


Figure 6. Operation of the third row when executing P2 for p = 4.

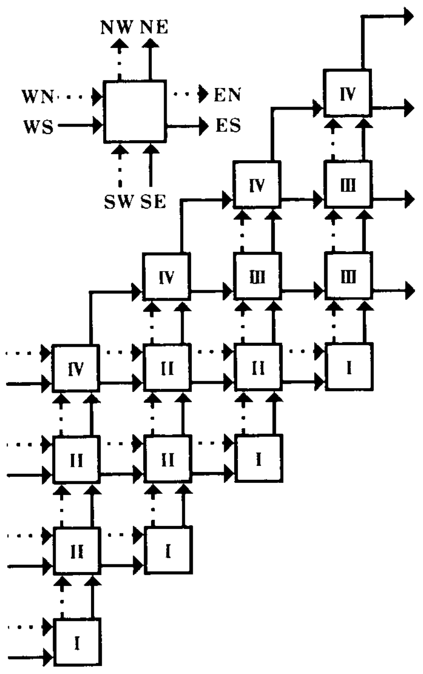
The VSAP resulting from the superposition of the arrays for P1(A,B) and P2(A,B,C) above described is depicted in figure 7. It has p×p orthogonally connected PEs. Dotted links are used only when executing P2. Through them matrix B arrives to the PEs. Solid links conduct matrices A and C. There are four different types of PEs. A PE with its ports named is also shown. It can be seen that a PE could not have all these ports. No data-counterflow is present.

Both primitives have two phases, an initialization phase where PE registers are loaded and a normal computation phase. In addition, P2 has an intermediate phase for passing the remaining columns of A. Henceforth, PEs must perform five different operations. The table in figure 7 indicates the actions carried out by each PE type when executing each operation. P1I and P1N denote the initialization and the normal phases for P1. P2I and P2N do the same for P2, while P2P corresponds to the intermediate phase. The internal register of all PEs is called r. Type IV PEs have an additional delay register d for attaining a proper output data skew. Parenthesized actions only apply to this type of PE. Two actions in the same line separated by a slash, indicate that only one is performed, depending on the PE type. Not specified output values do not matter. Some actions do not apply to all the PEs because they do not have all the ports.

In order to attain a simple plot of the VSAP, PEs have been grouped into four types. However, the table of operations show that they are very similar. All include a multiply-accumulate unit. Types II, III, and IV, only differ in the number of links and registers. Type I PEs must include an additional closure computation unit. As a result, PE complexity is kept relatively small.

A single control unit connected to the bottom-left PE is required. Control tokens are passed from PE to PE systolically. They travel upwards at unit speed, whereas three cycles are spent between type I PEs.

To conclude this section, a brief comment about the fact of forcing the same data formats for input and output is convenient. We have presented it as a design constraint. Another point of view is thinking that the VSAP purpose is two-fold: (a) to compute the desired result systolically, and (b) to perform succesive data reorderings



	I	II III (IV)
P1I	r:=WS*	r:=WS

P1N	NE:=r × WS	ES:=r × SE+WS
		NE:=SE/(NE:=d)
		(d:=SE)

P2I	r:=WS	r:=WS

P2P	NE:= WS	ES:=WS
		NE:=SE/(NE:=d)
		(d:=SE)

P2N	NE:=r × WN+WS
	NW:=WN
		ES:=r × SW+WS
		NE:=SE/(NE:=d)
		(d:=SE)
		NW:=SW
		EN:=WN

Figure 7. The VSAP, a PE with the name of the ports, and the table of operations.

for attaining the specified data output format. In some sense, the data rearranging network between the memory modules and the VSAP has been embedded into the array.

6. SUBPROBLEM EXECUTION, PERFORMANCE, AND OPTIMIZATIONS

Once described the VSAP low level operation, we return to the high level execution specified by Algorithm 2. Figure 8 outlines the blocks entering and exiting the array during the second iteration for a case with N/p=3. It is directly obtained from Algorithm 2. The PE input ports through which blocks come into the VSAP are indicated. The small, medium, and large blocks have p, N-p, and N columns respectively. All blocks have p rows. The up arrows point to the column arriving or leaving the array at the indicated cycle relative to the beginning of the second iteration.

Theorem 2: The described VSAP computes the APP of an N×N matrix in $N^3/p^2+N^2/p+3p-2$ cycles.

Observe that the bottom band has no holes. In general, the length of one of it sides is $N^3/p^2+N^2/p$. The skew adds p-1 cycles, and 2p-1 cycles are spent for crossing the VSAP once initialized. Hence, a total of $N^3/p^2+N^2/p+3p-2$ cycles are required.□

The address generator of each module is a counter with preloading. The central control unit sends control signals to its nearest address generator. This signals arrive to the remainig address generators in a systolic fashion. It can be easily verified that the sequence of matrices $A^{(k)}$ is computed in-place, only N^2 memory words are used.

For the sake of clarity, a suboptimal algorithm has been described. Note that there are holes between consecutive blocks leaving the array (see figure 8 again).

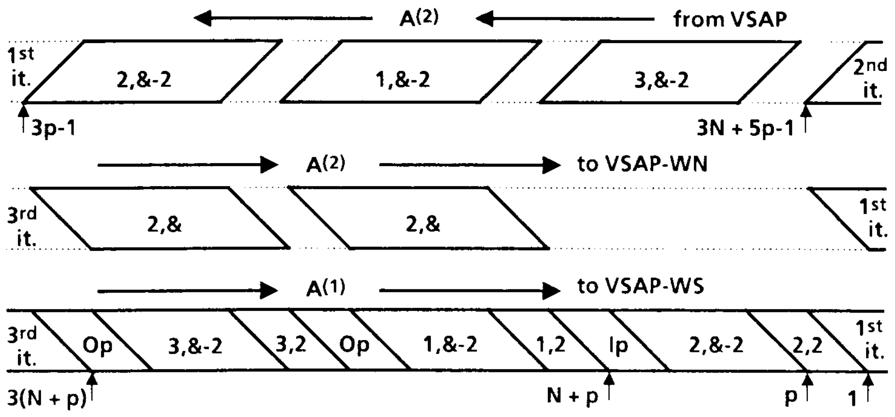


Figure 8. Block I/O sequencing of the second iteration for N/p = 3.

The purpose of Ip is to obtain $A(2,2)^{(2)}$ from the values stored in the array. This can be overlapped with the loading of $A(1,2)^{(1)}$ by simply modifying operation P2I. There is no conflict between the two actions. With this change, N cycles are saved. A much more interesting improvement is to avoid entering Op blocks, starting, instead, the loading of the following block. In this case, P2I and P2P must be changed. In addition, actions to store the entering block must be included (as in P2I). Again, there is no problem, all actions are orthogonal. The achieved saving amounts N^2/p-2N cycles. One additional optimization lasts to be done. The VSAP initialization when executing P1I (at the beginning of every iteration), can be performed when the last Op block of the previous iteration entered the array. Again, there is no contention for resources, and the result is a simple change in P1I, leading to reduce the execution time by N cycles. With this modifications in the operations, we reduce the total number of cycles to N^3/p^2+3p-2. Despite the transitory phases due to the skew, the VSAP outputs are delivering valid results in all the cycles. Thus, the optimal performance has been attained. No additional links neither additional functional units have been required. Only one more operation has appeared.

Let us define the efficiency of an array as the ratio the total number of operations performed to the number of cycles used by the array times the number of processors. For a case with $p=10$, an efficiency of 90 per cent is achieved with problem sizes of $N=100$ and $N=30$ for the suboptimal and optimal executions respectively.

7.DISCUSSION

In order to map the computations for solving an arbitrarily large problem into a fixed-size array two approaches can be taken. We can start with an array conceived to solve that problem (problem-size dependent) and perform PE groupings, pile pieces of the array, or folding it. On the other hand, it is possible to return to the original problem and decompose it into subproblems that fit the available SAP size. Even though the first approach is more methodic, the second gives us the opportunity to attain a very optimized design by carefully selecting design constraints to cope with a higher degree of freedom.

Henceforth, it has been developed an iterative block algorithm for solving the APP of an N-vertex graph supposing that we know how to obtain the APP of a graph with, at most, p vertices. The computations have been arranged in such a way that only two matrix operations suffice for obtaining the result. In other words, the APP has been decomposed into two primitive subproblems. Primitive 1 and 2 were defined as $P1(A,B)=A^*\times B$ and $P2(A,B,C)=A\times B+C$, were matrices A ,B , and C have $p\times p$, $p\times m$, and $p\times m$ elements respectively.

Two similar SAPs have been designed, one for each primitive. From their superposition a VSAP resulted. The VSAP has p×p orthogonally connected PEs. One of our prime concerns has been to design an array with good features for implementation. The I/O data skew is the same for all the blocks. This together with the lack of data counterflow permits to chain subproblems with no time penalties, keeping the PEs busy all the time. By conceiving the array to deliver matrix rows in the same order that they were received, can be folded to join their input and output ports.This allows to have only one memory bank composed of p memory modules placed along one the SAP polygon edges. In addition, there is no need for a data rearranging network, and all the matrices can be stored with the same scheme. Memory addressing is performed by simple counters. A centralized control unit generates all the control signals required by the PEs and address generators. Control signals are transmitted systolically. The resulting array has good implementation features by the observation of the indicated constraints.

With simple PEs able to perform five different operations the APP is solved in $N^3/p^2+N^2/p+3p-2$ cycles. Nevertheless, it was indicated how to obtain an optimal execution requiring N^3/p^2+3p-2 cycles by only slight changes in the PE operations.

REFERENCES

[1] G. Rote, "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)", Computing 34, 1985.
[2] Y. Robert, D. Trystam, "Systolic Solution of the Algebraic Path Problem", in SYSTOLIC ARRAYS, W. Moore, A. McCabe, R.Urquhart, eds., Adam-Hilger, 1987.
[3] P.S. Lewis, S.Y. Kung, "Dependence Graph Based Design of Systolic Arrays for the Algebraic Path Problem", Proc. Ann. Asilomar Conf. Sign. Syst. Comput., Nov. 1986.
[4] L.J. Guibas, H.T. Kung, C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms", Caltech Conf. on VLSI, Jan. 1979.
[5] S.Y. Kung, P.S. Lewis, S.C. Lo, "On Optimally Mapping to Systolic Arrays with Application to the Transitive Closure Problem", Proc. 1986 IEEE Int. Symp. on Circuits and Systems.
[6] S.Y. Kung, S.C. Lo, "A Spiral Systolic Architecture/Algorithm for Transitive Closure Problems", Proc. 1985 Conf. on Computer Design.
[7] S.Y. Kung, S.C. Lo, P.S. Lewis, "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems", IEEE Trans. on Computers, C-36, 5, May 1987.
[8] H.D. Cheng, K.S. Fu, "Algorithm Partition for a Fixed-Size VLSI Architecture Using Space-Time Domain Expansion", 7th Int. Sym. on Computer Arithmetic, 1985.
[9] F.J. Núñez, N. Torralba, "Transitive Closure Partitioning and Its Mapping to a Systolic Array", Proc. 1987 Int. Conf. on Parallel Processing, Penn State Univ., Aug. 1987.
[10] D.J. Lehmann, "Algebraic Structures for the Transitive Closure", Theoretical Computer Science, 4, 1977.1987.
[11] U. Zimmermann, "Linear and Combinatorial Optimization in Ordered Algebraic Structures", Ann. Discrete Math., 10, 1981.
[12] S. Warshall, "A Theorem on Boolean Matrices", J. ACM 9, 1, 1962.
[13] B. Roy, "Transitivité et Connexité", C. R. Acad. Sci. 249, 1959.
[14] R.W. Floyd, "Algorithm 97: Shortest Path", C. ACM 5, 6, 1962.
[15] M. Gondran, M. Minoux, S. Vajda, GRAPHS AND ALGORITHMS, Wiley, New York, 1984.
[16] K. Hwang, Y.H. Cheng, "Partitioned Matrix Algorithms for VLSI Arithmetic Systems", IEEE Trans. on Computers, C-31,12, Dec. 82.
[17] J.J. Navarro, J.M. Llabería, M. Valero, "Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors", IEEE Computer Mgz, July 1987.
[18] J.J. Navarro, J.M. Llabería, M. Valero, "Solving Matrix Problems with No Size-Restriction Using a One-Dimensional Systolic Array Processor", Proc. 1986 Int. Conf. on Parallel Processing.
[19] S.Y. Kung, "VLSI Array Processors", in SYSTOLIC ARRAYS, W. Moore, A. McCabe, R.Urquhart, eds., Adam-Hilger, 1987.