SICS/R-89/8910

**Computing Transitive Closure on
Systolic Arrays of Fixed Size**
by
Björn Lisper

Computing Transitive Closure on Systolic Arrays of Fixed Size*

Björn Lisper

Swedish Institute of Computer Science
Box 1263
S-164 28 Kista, SWEDEN

SICS Research Report SICS/R-89/8910, September 1989

Abstract

Forming the transitive closure of a binary relation (or directed graph) is an important part of many algorithms. When the relation is represented by a bit matrix, the transitive closure can be efficiently computed in parallel in a systolic array. Various such arrays for computing the transitive closure have been proposed. They all have in common, though, that the size of the array must be proportional to the number of nodes.

Here we propose two ways of computing the transitive closure of an arbitrarily big graph on a systolic array *of fixed size*. The first method is a simple partitioning of a well-known systolic algorithm for computing the transitive closure. The second is a block-structured algorithm for computing the transitive closure. This algorithm is suitable for execution on a systolic array, that can multiply fixed size bit matrices and compute transitive closure of graphs with a fixed number of nodes. The algorithm is, however, not limited to systolic array implementations; it works on *any* parallel architecture that can form the transitive closure of fixed-size graphs and the product of fixed-size bit matrices efficiently.

The shortest path problem, for directed graphs with weighted edges, can also be solved for arbitrarily large graphs on a fixed-size systolic array in the same manner, devised above, as the transitive closure is computed.

1 Introduction

How to compute the transitive closure of a binary relation (directed graph) is a much studied problem. It shows up in many areas, for instance when analyzing concurrent communicating finite-state processes [6]. Algebraically, computing the transitive closure can be seen as a special case of computing the closure of a matrix over a closed semiring, [1]. The classical Warshall's algorithm [9, 26] requires $O(n^3)$ operations when the relation is represented by a bit matrix and possibly less for sparse relations with a representation that utilizes the sparsity. Other authors have proposed more sophisticated serial algorithms which perform better for certain classes of sparse relations, [8, 22]. Parallel algorithms have also been proposed, in $O(\log n)$ time on $O(n^4)$ processors [4], or $O(\log^2 n)$ time on n^3 processors based on a fast parallel matrix multiplication algorithm [7].

During the last couple of years, some systolic array configurations have been proposed for computing transitive or transitive-reflexive closure, either mesh- or hexagonally connected square

*This work was presented at the Second Swedish Workshop on Computer Systems Architecture (DSA-89), Bålsta, Sweden, August 1989.

arrays, possibly with wraparound [5, 10, 24] or a linear array [11]. These arrays operate on an adjacency matrix representation of the relations involved and they are based on regular algorithms, like the aforementioned Warshall's algorithm, [5, 10, 11], or an algorithm that uses an extended adjacency matrix [24]. The square arrays use $O(n^2)$ processing elements to compute the transitive closure in time $O(n)$ and the linear array uses $O(n)$ processing elements in time $O(n^2)$. They all have in common, though, that the size of the array must match the size of the problem, i.e. it is not devised how to partition the computation so that arbitrarily big problems can be solved on an array of fixed size. This limitation seriously restricts the applicability of these arrays in practical situations. Some authors [20, 21] have considered how to partition arbitrarily big computations as to fit on a given systolic array, but to our knowledge nobody has yet shown how to do this for the computation of transitive closure.

In this paper, we devise two ways to compute the transitive closure of an arbitrarily big directed graph on a systolic array of fixed size. The first solution is a simple partitioning of the array given by Kung, Lo and Lewis [10], that allows the transitive closure to be successively computed, in several passes, on a fixed-size array. The second solution requires a systolic array that is capable of multiplying fixed-size bit matrices and computing the transitive closure of graphs of fixed size. We will give a block-matrix oriented algorithm that forms the transitive closure of a graph, given a partitioning of the graph into a number of fixed-size subgraphs, the transitive closure of the subgraphs and the adjacency matrices for the edges between elements in different subgraphs. This block-matrix oriented algorithm will need $O(n^3)$ fixed-size block matrix operations. Thus, the algorithm is, at least asymptotically, on par with Warshall's algorithm regarding the number of operations. The actual running time will, of course, depend on how fast these block matrix operations can be carried out. Preferably they are done in parallel, in order to make them fast. By no means, though, they have to be carried out systolically. The block-matrix oriented algorithm is perfectly general and it gives a way to partition the transitive closure for execution on *any* parallel machine that is too small to process the whole relation at once. The algorithm is, however, constructed with systolic implementations in mind and we will show how it can be very efficiently executed on a quadratic systolic array.

2 Preliminaries

2.1 Binary relations and transitive closure

In this paper we will consider binary relations. I , the identity relation, is defined by aIb iff $a = b$. The composition RR' of two relations R, R' is given by: $aRR'b$ iff there is a c such that aRc and $cR'b$. Powers R^i , $i \geq 0$, of a relation R are defined as follows: $R^0 = I$, and $R^i = RR^{i-1}$ for $i > 0$. The *transitive closure* of R , denoted R^+ , is the smallest transitive relation containing R . It is easily proved that $R^+ = \bigcup_{i=1}^{\infty} R^i$. When the underlying set has n elements, then $R^+ = \bigcup_{i=1}^n R^i$. The *transitive and reflexive closure* of R , R^* , is equal to $R^+ \cup I = R^0 \cup R^+$.

A binary relation R over a set can be seen as a directed graph, where the elements of the set are nodes and there is an edge from a to b iff aRb . In this setting, forming the transitive closure of R is the same as finding all (directed) paths in the graph, i.e. aR^+b iff there is a path from a to b .

A relation R over a set $\{a_1, \dots, a_n\}$ can be represented by an $n \times n$ bit matrix $m(R)$, defined by: $m(R)_{ij} = 1$ iff a_iRa_j . It is easily established that $m(RR') = m(R)m(R')$ for all relations R, R' .

A well-known and simple algorithm to compute the transitive closure of a given relation R is

Warshall's algorithm [26]. Below it is given in terms of bit matrix operations:

$$\begin{aligned} t_{ij0} &\leftarrow m(R)_{ij}, \quad 1 \leq i, j \leq n \\ t_{ijk} &\leftarrow t_{ijk-1} + t_{ikk-1} t_{kjk-1}, \quad 1 \leq i, j, k \leq n \end{aligned} \quad (1)$$

Here $m(R)$ is the bit matrix of the relation, “+” denotes OR and juxtaposition denotes AND. The results are $m(R^+)_{ij} = t_{ijn}$, $1 \leq i, j \leq n$. It is immediately seen from the recurrence that the algorithm uses $2n^3 + O(n^2)$ boolean operations to compute the transitive closure of R . With a different interpretation of $m(R)$ and the operation symbols, Warshall's algorithm computes other entities: if $m(R)$ consists of nonnegative distances between nodes, for instance (with $m(R)_{ij} = \infty$ if there is no edge from i to j), and if “+” denotes minimum and juxtaposition denotes addition, then the algorithm computes the shortest directed path between any two nodes.

2.2 Hardware design via space-time mappings

Fixed hardware for algorithms with fixed structure, like Warshall's algorithm above, can be derived by so-called *space-time mappings*. These are schedulings of the steps of the algorithm onto synchronous hardware. Formally this can be seen as a function from the set of steps to a discrete *space-time*, where every space-time coordinate is an event at a certain time in a certain location where something, for instance a step in the algorithm, may take place. The function must fulfil the following:

1. It must be injective, i.e. no two steps are scheduled at the same time and place.
2. It must be causal, that is: steps producing data must be scheduled at times less than steps using the data in question as inputs.

For algorithms with a large degree of repetitivity, such as Warshall's algorithm above, where the repetitivity is expressed by a multidimensional enumeration, it is often convenient to represent each step by a unique *index vector*. The index vectors can then be mapped to space-time rather than the steps in the algorithms that they represent. It has been found that *linear* (or affine) mappings are of great interest, since they tend to preserve the regularity of the given algorithm into the structure of the hardware. A multitude of literature about this synthesis technique has emerged during the last decade, see for instance [3, 5, 17, 15, 16, 18, 19, 23, 25].

3 The array of Kung, Lo and Lewis

Kung, Lo and Lewis [10] derived an efficient systolic array implementation of Warshall's algorithm. Their implementation uses n^2 cells to compute the transitive closure of a graph with n nodes in time $5n - 4$. By a critical path argument this can be shown to be a time-optimal systolic implementation. We will now describe their implementation.

Consider the recursion equation (1) defining Warshall's algorithm. Represent every assignment, producing an iterate t_{ijk} , by an index vector (i, j, k) . When $k > 0$, this assignment uses data produced by the assignments represented by $(i, j, k - 1)$, $(i, k, k - 1)$ and $(k, j, k - 1)$. These dependences give rise to the following data dependence vectors:

$$(0, 0, 1), (0, j - k, 1) \text{ and } (i - k, 0, 1).$$

Reindexing the assignments in order to regularize the communication pattern, according to¹

$$(i, j, k) \mapsto ((i - k - 1) \bmod n + 1, (j - k - 1) \bmod n + 1, k),$$

¹Not $(i, j, k) \mapsto ((i - k) \bmod n + 1, (j - k) \bmod n + 1, k)$ as stated in [10].

yields the following data dependence vectors instead:

$$\begin{aligned} & ((i - k - 1) \bmod n - (i - k) \bmod n, (j - k - 1) \bmod n - (j - k) \bmod n, 1) \\ & ((i - k - 1) \bmod n, (j - k - 1) \bmod n - (j - k) \bmod n, 1) \\ & ((i - k - 1) \bmod n - (i - k) \bmod n, (j - k - 1) \bmod n, 1). \end{aligned}$$

These data dependence vectors will, when $i = k$ or $j = k$, give rise to so-called *spiral connections* that are non-local in a planar topology. This non-locality is undesirable in a systolic implementation. In order to remove these connections we observe the following², from equation (1) for $i = k$ and $j = k$:

$$\begin{aligned} t_{ikk} &\leftarrow t_{ikk-1} + t_{ikk-1}t_{kkk-1} \\ &= t_{ikk-1}, \text{ and} \\ t_{kjk} &\leftarrow t_{kjk-1} + t_{kkk-1}t_{kjk-1} \\ &= t_{kjk-1}. \end{aligned}$$

Furthermore, for $0 \leq l < k$, we find that

$$\begin{aligned} t_{ikk-l} &\leftarrow t_{ikk-l-1} + t_{ikk-l-1}t_{kkk-l-1} \\ &= t_{ikk-l-1}, \text{ and} \\ t_{kjk-l} &\leftarrow t_{kjk-l-1} + t_{kkk-l-1}t_{kjk-l-1} \\ &= t_{kjk-l-1}. \end{aligned}$$

Finally, for $l = k$, $t_{ik0} = m(R)_{ik}$ and $t_{kj0} = m(R)_{kj}$. It follows that $t_{ikk} = m(R)_{ik}$ and $t_{kjk} = m(R)_{kj}$. Thus, the spiral connections can be replaced with direct inputs of the corresponding entries of the original bit matrix. The data transfers of these entries are easily localized.

Applying the following space-time mapping,

$$\begin{pmatrix} t \\ x \\ y \end{pmatrix} = \begin{pmatrix} 1 & 1 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix}$$

from transformed indices

$$(i', j', k') = (i - k + 1 \bmod n, j - k + 1 \bmod n, k)$$

will now give the design of Kung, Lo and Lewis. Figure 1 shows the structure of the array and the input and output patterns. Note that some control signals, directing the inputs and outputs of the operations to different lines, are omitted. For a detailed description, see [14].

4 Partitioning the array of Kung, Lo and Lewis

We will now show how to partition the array of Kung, Lo and Lewis in order to obtain a fixed-size array for computing transitive closure, i.e. we assume an array of the size of the problem and cut it into fixed-size parts, all of the same size and dimensions. The original computation can then be run in several passes on the fixed-size array. For clarity, we first skew the array of figure 1 spatially.

²Note that these identities hold regardless of whether or not, as assumed in [10], $t_{kkk-1} = 1$. The removal of spiral connections is thus correct also when computing the transitive closure of non-reflexive relations.

Figure 1: The array of Kung, Lo and Lewis.

Figure 2: The array of Kung, Lo and Lewis, spatially skewed.

This skewing in space is described by the following transformation of the space coordinates:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Partitioning of systolic array computations into several passes by the aid of linear mappings has been described by Moldovan and Fortes [20]. Here, we simply partition the computation by cutting the original array into pieces. Figure 3 shows how the skewed array can be cut into quadratic pieces of the same size. The action of each of these sub-arrays can be carried out by the array depicted in figure 4, where the appropriate input and output lines of the cells at different times are selected by control signals.

Partitioning the computation according to the above works because the quotient graph of the partitioned array is acyclic. Thus, the corresponding data dependence graph of subcomputations, where each node (i, j) is the subcomputation carried out in subarray (i, j) , is also acyclic (in fact, it is isomorphic with the quotient graph). It follows that this graph can be serially scheduled on a device that subsumes all the subarrays. The array shown in figure 4 fits this purpose.

Let us now consider the subcomputations in some detail. Assume that we partition an $n \times n$ Kung-Lo-Lewis-array into subarrays that are $N \times N$. Give every subarray (or the corresponding subcomputation) a two-dimensional index (u, v) . Each processor (x', y') in the skewed Kung-Lo-Lewis-array is mapped to the subarray $(\lceil x'/N \rceil, \lceil y'/N \rceil)$. Thus, any step in Warshall's algorithm, with transformed index vector (i', j', k') , will be carried out in subcomputation $(\lceil x'/N \rceil, \lceil y'/N \rceil)$, where

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} j' + k' - 1 \\ k' \end{pmatrix}.$$

Figure 3: Partitioning the array of Kung, Lo and Lewis.

Figure 4: Fixed-size array for computing transitive closure.

Figure 5: Data dependence graph of subcomputations.

A closer examination of the partitioning reveals the data dependence graph, when the subcomputations are considered as atomic. Define i_{max} according to the following:

$$i_{max} = \begin{cases} \lceil n/N \rceil, & N \text{ divides } n - 1 \\ \lceil n/N \rceil + 1, & \text{otherwise.} \end{cases}$$

Then each subcomputation will be uniquely indexed by a pair from the set

$$\{(u, v) \mid 1 \leq v \leq \lceil n/N \rceil, v \leq u \leq i_{max} + v - 1\}.$$

The following data dependencies will always be present:

$$(u, v) \prec (u + 1, v) \text{ and } (u, v) \prec (u, v + 1),$$

when both nodes belong to the set above. When N divides $n - 1$, there will also be a diagonal dependence between the computations to the far right:

$$(i_{max} + v - 1, v) \prec (i_{max} + v, v + 1)$$

when $1 \leq v \leq \lceil n/N \rceil - 1$.

4.1 Memory requirements

Every execution of a subcomputation on the fixed-size array can be seen as a single instruction being carried out on a serial device. Thus, scheduling techniques for serial computers can be applied to the dependence graph for the subcomputations. The analysis of the memory requirements for different schedules can also be carried out by standard methods for serial computations. In particular, we will play a *graph pebbling game* [12, 13] to analyze the memory requirements of the partitioned transitive closure algorithm.

Graph pebbling can be used to model storage allocation, when every edge in the data dependence graph represents data of the same size. Then we can assume that a pebble represents a storage location of that size. In our particular graph pebbling game, we put pebbles on edges of the dependence graph. A pebble being put on an edge means, that the datum of the edge is stored in the memory location of the pebble. When all the input and output edges of a node are pebbled, then

Figure 6: Data dependence graph of subcomputations, $i_{max} = 5$.

Figure 7: Graph pebbling. A node fires.

all its inputs are available, there is place for its results and it can then “fire”, i.e. the corresponding computation is carried out. Then the pebbles on the input edges can be removed and placed on other edges in the graph. The maximum number of pebbles ever needed gives the memory requirements.

We now apply the pebbling technique to the data dependence graph of the partitioned algorithm. Then we obtain the result below:

Lemma 1 *Exactly $O(n/N)$ pebbles are needed to pebble the dependence graph of the partitioned algorithm for transitive closure, regardless of execution order.*

Proof. Disregarding possible diagonal dependencies, there are three types of nodes in the dependence graph, as shown in figure 8. Nodes of the first type introduce one pebble; after their firing, one more pebble is needed than before. (Here we disregard the temporary need for pebbling both the inputs and outputs of the node to be executed: since only one node fires at a time and since the out-degree of the nodes is bounded, this gives only a constant overhead. Furthermore, we consider the input of the original bit matrix and the output of the transitive closure matrix to take place directly to the “outside world”: thus, no pebbles are needed for these.) Execution of nodes of type two conserves the number of pebbles. When a node of the third type fires, finally, a pebble is released.

We first prove that *at most* $O(n/N)$ pebbles are ever needed. Since there are i_{max} nodes of type one, at most i_{max} pebbles are ever introduced. i_{max} is either $\lceil n/N \rceil$ or $\lceil n/N \rceil + 1$. Thus, no more than $O(n/N)$ pebbles are ever introduced. At most two pebbles are temporarily needed to hold outputs during the firing of a node. Therefore, at most $O(n/N)$ pebbles are ever needed.

Now we prove that *at least* $O(n/N)$ pebbles are always needed. Consider an arbitrary node in column u but not in the first row. This node is, directly or indirectly, dependent on all nodes $(u', 1)$, $u' \leq u$, of type one. All these nodes introduce a pebble and they must all fire before the node in question can fire. The nodes removing pebbles are all situated in the last row, from column $i_{max} + 1$ and up. Thus, i_{max} pebbles must be introduced before any pebble can be removed. It follows that at least $O(n/N)$ pebbles are always needed.

Diagonal dependencies will not alter these figures with more than a constant factor. Thus, the order is the same even in presence of these. ■

A pebble in the analysis above represents the storage of the output of a subcomputation on an $N \times N$ -array. This output consists of $O(N)$ streams of $O(n)$ bits each, requiring a buffer of size $O(nN)$ for the storage. Since a total of $O(n/N)$ pebbles are needed, we obtain the following result.

Theorem 1 *The partitioned algorithm for transitive closure always requires $O(n^2)$ bits of storage for intermediate results.*

Theorem 1 is interesting from a practical point of view. One can imagine a device for computing transitive closure, equipped with a fixed-size systolic array and a fixed amount of buffer memory for storing intermediate results. This device may be connected to a general purpose computer through a communication link. When processing problems of more than a certain size, the device will have to swap intermediate results over this link. This is bound to degrade performance. If we know something about the average size of the problems to be processed, then, with the aid of theorem 1, the memory of the device can be dimensioned so that excessive swapping does not occur in most cases.

Figure 8: The three types of nodes on the dependence graph.

4.2 Scheduling the dependence graph

Since execution time and memory requirements are not affected by the scheduling of the dependence graph for the partitioned problem, the schedule could be freely chosen as to maximize the regularity of the execution pattern. One schedule is of immediate interest, namely “row-wise” scheduling. See figure 9. Other schedules are of course also possible and may have similar properties.

Figure 9: Scheduling the dependence graph “row-wise”.

This schedule allows for a simple rolling scheme for storing and retrieving intermediate results. This scheme can also be extended in a simple way to incorporate swapping to a host computer or an external mass storage device.

Let us finally note, that even since the scheduling here is described as strictly serial, the pipelining capability of the array can be used to overlap the execution of the subcomputations.

5 A block-matrix oriented algorithm

We will now demonstrate a block-matrix oriented algorithm for computing the transitive closure of a binary relation. It is based on the observation made in [2], that partitioning the bit matrix of a directed graph corresponds to partitioning the set of nodes of the graph. Let $m(R)$ be the bit matrix in question. Partition it according to the following:

$$m(R) = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

This corresponds to a partition of the nodes into two sets V_1 and V_2 . The matrix A represents edges between nodes in V_1 , D represents edges between nodes in V_2 , B represents edges from V_1 to V_2 and C , finally, represents edges from V_2 to V_1 .

As shown in [2], the transitive and reflexive closure $m(R)^*$ can be computed as follows, where $E = (A + BD^*C)^*$:

$$m(R)^* = \begin{pmatrix} E & EBD^* \\ D^*CE & D^* + D^*CEBD^* \end{pmatrix} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Figure 10: The partitioned graph and the block matrices, following [2].

This scheme is easily adapted to computing just the transitive closure $m(R)^+$, if now E denotes $(A + BD^*C)^+$, $E^* = E + I$ and $D^* = D^+ + I$, where I is the identity relation:

$$m(R)^+ = \begin{pmatrix} E & E^*BD^* \\ D^*CE^* & D^+ + D^*CEBD^* \end{pmatrix} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

The four block matrices E, F, G, H may now be computed according to the following sequence of steps:

$$\begin{aligned} T_1 &\leftarrow D^+ \\ T_2 &\leftarrow B(T_1 + I) \\ E &\leftarrow (A + T_2C)^+ \\ F &\leftarrow (E + I)T_2 \\ T_3 &\leftarrow (T_1 + I)C \\ G &\leftarrow T_3(E + I) \\ H &\leftarrow T_1 + GT_2 \end{aligned}$$

Let us now assume that we want to compute the transitive closure of an $nk \times nk$ -matrix, where $k > 0$, on a device that can operate on $n \times n$ -matrices efficiently. (If the dimensions of the matrix is not an exact multiple of n , then it can be padded with zeroes to the next higher multiple.) We thus want to divide the computation into a sequence of operations on $n \times n$ -matrices. This can be done, if we set $A_k = m(R)$ and then recursively, for $i = k - 1, \dots, 1$ set

$$A_{i+1} + B_{i+1}D_{i+1}^+C_{i+1} = \begin{pmatrix} A_i & B_i \\ C_i & D_i \end{pmatrix},$$

where A_i is $ni \times ni$, B_i is $n \times ni$, C_i is $ni \times n$ and D_i is $n \times n$.

Based on this recursive partitioning of the matrix and the sequence of steps for computing the transitive closure above, an algorithm can be formulated that only uses operations on $n \times n$ -matrices or operations that easily can be divided into such. The algorithm consists of a *downsweep* on successively smaller matrices and an *upsweep* on successively larger ones.

Figure 11: Recursive partitioning of A_i .

Block-matrix algorithm

Downsweep: $i = k - 1, \dots, 1$:

$$T_{1i} \leftarrow D_i^+ \tag{2}$$

$$T_{2i} \leftarrow B_i(T_{1i} + I) \tag{3}$$

$$T_{3i} \leftarrow (T_{1i} + I)C_i \tag{4}$$

$$T_{4i} \leftarrow A_i + T_{2i}C_i \tag{5}$$

Here, $T_{4i} = A_i + B_i D_i^+ C_i$.

$$E_1 \leftarrow T_{41}^+$$

Upsweep: $i = 1, \dots, k - 1$:

$$F_i \leftarrow (E_i + I)T_{2i} \tag{6}$$

$$G_i \leftarrow T_{3i}(E_i + I) \tag{7}$$

$$H_i \leftarrow T_{1i} + G_i T_{2i} \tag{8}$$

In the upsweep,

$$E_i = \begin{pmatrix} E_{i-1} & F_{i-1} \\ G_{i-1} & H_{i-1} \end{pmatrix}.$$

This algorithm requires a direct computation of the transitive closure of $n \times n$ -matrices only. Matrices of larger dimensions are multiplied and added, but these operations can easily be broken down into operations on $n \times n$ size blocks. Thus, given a device that can operate on $n \times n$ -matrices efficiently, the algorithm gives a scheme for computing the transitive closure of relations of arbitrary size in a number of passes on the device. The device can for instance be a systolic array that can act both as the Kung-Lo-Lewis array *and* perform matrix multiplication and addition, or it can be any parallel machine whose size is appropriate for performing these operations efficiently on $n \times n$ -matrices.

Figure 12: Formats of intermediate and result matrices in the block-matrix algorithm.

5.1 Performance analysis

We will now analyze the block-matrix oriented algorithm with respect to the number of operations. Assume that the logical operations AND, OR on bits have equal cost, which we can set to 1. Then, the costs for the matrix operations in step i of the downsweep are:

$$\begin{aligned}
 T_{1i} &\leftarrow D_i^+ & 2n^3 \\
 T_{2i} &\leftarrow B_i(T_{1i} + I) & n + 2in^3 \\
 T_{3i} &\leftarrow (T_{1i} + I)C_i & n + 2n^3 \\
 T_{4i} &\leftarrow A_i + T_{2i}C_i & n^2 + 2i^2n^3.
 \end{aligned}$$

Correspondingly, for the upsweep:

$$E_1 \leftarrow T_{41}^+ \quad 2n^3,$$

and

$$\begin{aligned}
 F_i &\leftarrow (E_i + I)T_{2i} & in + 2i^2n^3 \\
 G_i &\leftarrow T_{3i}(E_i + I) & in + 2i^2n^3 \\
 H_i &\leftarrow T_{1i} + G_iT_{2i} & n^2 + 2in^3.
 \end{aligned}$$

Summing over i , we obtain the total operation count

$$\begin{aligned}
 \sum_{i=1}^{k-1} (2n^3 + n + 2in^3 + n^2 + 2i^2n^3 + n + 2n^3) + 2n^3 + \sum_{i=1}^{k-1} (in + 2i^2n^3 + in + 2i^2n^3 + n^2 + 2in^3) &= \\
 2n^3 + (k-1)(4n^3 + 2n^2 + 2n) + (4n^3 + 2n) \sum_{i=1}^{k-1} i + 6n^3 \sum_{i=1}^{k-1} i^2 &= \\
 2n^3 + (k-1)(4n^3 + 2n^2 + 2n) + (4n^3 + 2n)k(k-1)/2 + 6n^3(k-1)k(2k-1)/6 &= \\
 2(nk)^3 - n(k-1)(n^2(k-2) - 2n - k - 2) &= \\
 2(nk)^3 + O(n^3k^2). &
 \end{aligned}$$

If $T_{1i} + I$ and $E_i + I$ are saved and reused, then $n(k-1) + nk(k-1)/2$ operations can be saved in exchange for some additional temporary storage requirements. It is interesting to compare the operation count for the block-matrix oriented algorithm with Warshall's original algorithm applied to the full $nk \times nk$ -matrix: Warshall's algorithm then needs $2(nk)^3$ operations. Thus, they have the same asymptotic complexity. Curiously though, the block matrix algorithm will have a somewhat lower operation count for some values of n and k when taking the lower-order terms into account.

In principle, one could minimize this count with respect to the block sizes under the restriction that nk is constant. This would lead to a potentially somewhat faster serial algorithm for dense bit matrices than Warshall's algorithm. In practice, though, the resulting serial speedup is likely to be wiped out by the added overhead of administrating the block operations.

5.2 Memory requirements

Consider again the block-matrix algorithm. Denote step i in the downsweep by $i.d$, the “bottom-step” $E_1 \leftarrow T_{41}^+$ by $0.b$ and step i in the upsweep by $i.u$. The steps are then totally ordered by the data dependencies, according to the following:

$$k.d \prec (k-1).d \prec \dots 1.d \prec 0.b \prec 1.u \prec \dots \prec (k-1).u \prec k.u.$$

Every step $i.d$ in the downsweep produces four intermediary results: T_{1i} , T_{2i} , T_{3i} and T_{4i} . The first three are used by $i.u$ and the fourth by $(i-1).d$. Each step $i.u$ in the upsweep in addition uses E_i produced by $(i-1).u$. Thus, between step $i.d$ and $(i-1).d$ the matrices T_{4i} and T_{1j} , T_{2j} , T_{3j} , $i \leq j < k$ have to be stored. Similarly, E_i and T_{1j} , T_{2j} , T_{3j} , $i \leq j < k$ must be stored between the steps $(i-1).u$ and $i.u$. Therefore exactly $(nk)^2$ bits of storage are needed between any two steps. This is the same as for the partitioned systolic Kung-Lo-Lewis' algorithm. Since the ordering given by the data dependencies is total, there is no way to rearrange the computations in order to lower the memory requirements.

Figure 13: Step i in the downsweep and the upsweep.

A simple way of storing the temporary and final results is to let them overwrite the corresponding parts of the original bit matrix, as indicated in figure 14. The analysis above shows that no crucial data will be overwritten. An out-of-core storage scheme can also be based on this; as long as the current submatrix of interest cannot be kept in primary memory, simply write the the temporary results T_{1i} , T_{2i} and T_{3i} to secondary storage as soon as they are created and keep them there until they are needed. Together with a smart memory handling scheme for the matrix operation $X + YZ$, where X is a large square matrix and Y and Z are “thin” matrices, swapping will be substantially reduced. This can be of use also when computing the transitive closure of large relations on a conventional, serial computer.

Figure 14: Storing intermediate and final results.

5.3 Systolic implementation

Let us now consider in more detail how to implement the block-matrix oriented algorithm in a systolic fashion. Since we already know how to compute the transitive closure of a $n \times n$ -matrix on a systolic array of matching size, the implementation issue amounts to how to multiply dense matrices of varying dimensions most efficiently on a systolic array. This was investigated, using formal methods, in [17, ch. 11.2]. Here we will use the resulting systolic algorithms.

Consider first step i in the downsweep. (2) and (3) are best computed blockwise according to figure 15, with $T_{1i} + I$ sitting fixed in the systolic array, one element per cell. Note that the computation of the different blocks can be pipelined. The product $T_{2i}C_i$ in (4) is also computed blockwise. If each block of T_{2i} is fixed in the systolic array during the computation of the corresponding “block-row” of the product, then the block multiplications for that row can be pipelined exactly as when computing (3).

With some extra logic the sums involved can be computed “on the fly”. By setting the proper inputs to “1” at the proper times, $T_{1i} + I$ can be formed while T_{1i} is loaded into the array. Similarly, an extra row of OR-gates can successively add A_i to $T_{2i}C_i$, block per block, while the blocks of this product are shifted out of the array.

The matrix multiplications (5), (6) and (7) during the upsweep can be performed in a similar fashion, but here it is advantageous to perform the matrix multiplication in a way where the operands flow through the array and the result is accumulated in place. See figure 16. The matrix sums can be computed similarly to the ones in the upsweep.

6 Conclusions

We have demonstrated two different methods to compute the transitive closure of a binary relation of arbitrary size on a systolic array of fixed size. The first method is based on a partitioning of the systolic array for computing transitive closure by Kung, Lo and Lewis, and it will implement the same version of Warshall’s algorithm as their array. The second method is based on a decomposition into blocks of the bit matrix of the relation. The resulting algorithm consists of a series of steps, where the transitive closure is formed for fixed-size block matrices and block matrices are multiplied with each other.

The two algorithms essentially have the same performance; for a problem of size n they both require $O(n^2)$ memory and they use $O(n^3)$ operations. The difference lies in simplicity and applicability: while the algorithm based on the Kung-Lo-Lewis-array is somewhat simpler to implement systolically, since the array only will have to perform one type of systolic computation, the block-oriented algorithm can easily be adapted for execution on any kind of parallel hardware that can perform operations on fixed-size bit matrices efficiently. Furthermore, the need in a systolic implementation of the block-oriented algorithm to perform both matrix multiplication and compute transitive closure in the same array does not pose any serious problems, since the cells in both cases will perform the same kind of accumulating operation – only the operand flow will differ. Thus, a simple capability to reconfigure the array is all that is needed, together with some additional gates for adding matrices on the fly during loading or unloading.

It should be pointed out that the algorithms here also, with a different interpretation of the function symbols, will compute the shortest-path problems for directed graphs with non-negative weights on the edges.

The ability to partition a systolic computation, so that it can be carried out on a given array, is crucial if systolic computing is ever to have any practical significance other than for a very limited range of applications. Therefore, the investigation made here is of importance if systolic

Figure 15: Systolic matrix multiplications during the downsweep.

Figure 16: Systolic matrix multiplications during the upsweep.

computation of transitive closure is to be implemented in practice. Another issue that has to be dealt with in practice is memory handling; when problems become large enough they will not fit into a limited memory and thus data has to be swapped in some fashion. Here we hint at some ways of doing this for the given systolic algorithms. An out-of-core memory handling strategy, that will reduce swapping, can be formulated for the block-matrix oriented algorithm. Memory size problems also arise on conventional computers: therefore, this algorithm may be of interest to implement also on serial machines.

A special-purpose device for computing transitive closure (or shortest paths) can be based on the algorithms given here. This device would consist of a systolic array, sufficient memory, arranged in banks, to host most problems that are anticipated, some logic for interfacing the memory banks and the array, and a standard communications interface so that the device easily can be hooked up to a host computer. Also needed is a protocol format for transferring bit matrices, or blocks thereof, over the connection and some logic to handle the swapping when a matrix is too large to be stored at once locally.

It should be noted, though, that such a device will *not* perform well for sparse relations. The algorithms given here have a data-independent structure: thus, they cannot utilize sparsity to reduce the amount of work done. The $O(n^3)$ operation count for the algorithms here will asymptotically give $O(n^3)$ time (albeit with a small constant), when problems become large and must be processed in several steps on the systolic array. There are serial algorithms, however, that will compute the transitive closure of a relation $\langle V, E \rangle$ in time $O(|V||E| + |V|^2)$ [8, 22]. If $|E| < O(|V|^2)$, then such an algorithm will perform asymptotically better than the fixed-size systolic algorithms. It is actually highly unlikely that *any* systolic algorithm will ever be devised that can utilize sparsity, since such algorithms typically depend on a careful analysis of data dependencies in advance; for sparse relations these dependencies are not known in advance and pre-scheduling cannot be used to obtain a systolic, purely synchronous and data-driven execution pattern with minimal run-time control.

For relations that are naturally represented and operated upon in matrix form, however, the systolic algorithms given here provide a fast, practical way for computing transitive closure and shortest paths.

7 Acknowledgements

I want to thank Bengt Jonsson for pointing out a serious error in a first version of the block-oriented algorithm, and Stefan Arnborg for providing a pointer to literature from which a correct algorithm could be derived.

References

- [1] S. K. Abdali and B. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40:257–274, 1985.
- [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [3] P. K. Cappello and K. Steiglitz. Unifying VLSI array design with linear transformations of space-time. Research Report TRCS83-03, Dept. Comput. Sci., UCSB, 1983.
- [4] L. Kučera. Parallel computations and conflicts in memory access. *Inf. Process. Lett.*, 14(2):93–96, April 1982.

- [5] M. C. Chen. Transformation of parallel programs in Crystal. In H.-J. Kugler, editor, *INFORMATION PROCESSING 86*, pages 455–462. Elsevier Publishers B.V. (North-Holland), 1986.
- [6] R. Cleveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In *Proc. 9th Int. Symp. on Protocol Specification, Testing and Verification*, 1989. To appear.
- [7] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10(4):657–675, November 1981.
- [8] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.
- [9] R. W. Floyd. Algorithm 96: Ancestor. *Comm. ACM*, 5:344–345, 1962.
- [10] S.-Y. Kung, S.-C. Lo, and P. S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Trans. Comput.*, C-36(5):603–614, May 1987.
- [11] P. Lee and Z. M. Kedem. Synthesizing linear array algorithms from nested for loop algorithms. *IEEE Trans. Comput.*, 37(12):1578–1598, December 1988.
- [12] T. Lengauer and R. Tarjan. The space complexity of pebble games on trees. *Inf. Process. Lett.*, 10:184–188, 1980.
- [13] T. Lengauer and R. Tarjan. Asymptotically tight bounds on the time-space tradeoff in a pebble game. *J.ACM*, 29:1087–1130, 1982.
- [14] P. S. Lewis and S. Y. Kung. Dependence graph design of systolic arrays for the algebraic path problem. In *Proc. 12th Ann. Asilomar Conf. Signals, Syst., Comput.*, November 1986.
- [15] B. Lisper. Time-optimal synthesis of systolic arrays with pipelined cells. Research Report YALEU/DCS/RR-560, Dept. Comput. Sci., Yale University, September 1987.
- [16] B. Lisper. Synthesis and equivalence of concurrent systems. *Theoretical Computer Science*, 58:183–199, 1988.
- [17] B. Lisper. *Synthesis of Synchronous Systems by Static Scheduling in Space-time*, volume 362 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, May 1989.
- [18] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32:93–114, 1984.
- [19] D. I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comput.*, C-31:1121–1126, October 1982.
- [20] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms in fixed size systolic arrays. *IEEE Trans. Comput.*, C-35:1–12, January 1986.
- [21] J. J. Navarro, J. M. Llaberia, and M. Valero. Partitioning: an essential step in mapping algorithms into systolic array processors. *Computer*, 20(7):77–89, July 1987.
- [22] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

- [23] P. Quinton. The systematic design of systolic arrays. Research Report RR 216, INRIA, Rennes, July 1983.
- [24] S. K. Rao, T. Citron, and T. Kailath. Mesh-connected processor arrays for the transitive closure problem. In *Proc. of 24th Conference on Decision and Control*, pages 1565–1570, Ft. Lauderdale, FL, December 1985. IEEE.
- [25] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proc. of the IEEE*, 76(3):259–269, March 1988.
- [26] S. Warshall. A theorem on boolean matrices. *J.ACM*, 9(1):11–12, January 1962.