Chapter 2

Mesh and Torus Networks

The *mesh* and the *torus* networks have processing units disposed as a grid. In a sense, they are the two dimensional versions of the linear array and ring, respectively. There are actually even more general mesh and torus architectures, defined for any number of dimensions, but we will focus on the basic 2-D versions.

A mesh (or torus) with P processing units can be drawn as a grid of side $\sqrt{P}$. Graphical representations of the mesh (Figure 5) and the torus (Figure 6) are given below. In the mesh each inner processor has 4 links, to the four cardinal directions, while processors along the edges have fewer links, either 3 or 2 depending on their position. On the other hand, in the torus, similarly to the ring, all processors have 4 links, those along the edges simply have connections looping back to the opposite edge of the network. In this kind of networks, it is often useful to use two indices, instead of just one, to identify processors, the first for the "row" and the second for the "column" of the network, as done for the elements of a matrix. Such indices will then range over the set $\{0, 1, \ldots, \sqrt{P}-1\}$. Also in this case, we will assume that each processing unit knows its pair of indices.

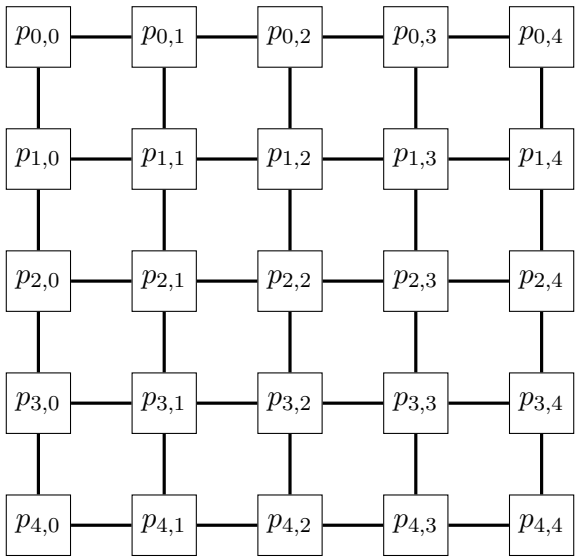


Figure 5: Mesh with 25 processing units.

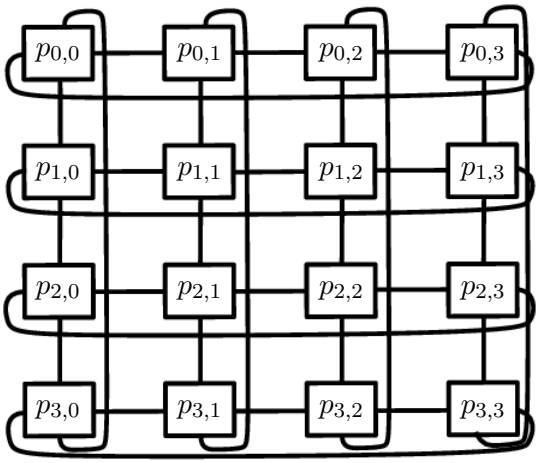


Figure 6: Torus with 16 processing units.

A mesh M_P with P processors and bidirectional links has the following properties.

- Diameter $diam(\mathrm{M}_P) = 2\sqrt{P} - 2$, i.e. the distance between, e.g., the processors at the top-left and bottom-right corners.
- Bisection bandwidth $b(\mathrm{M}_P) = \sqrt{P} + (\sqrt{P} \bmod 2)$.

A torus T_P with P processors and bidirectional links has the following properties.

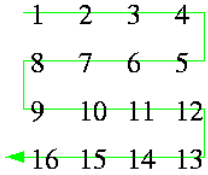
- Diameter $diam(\mathrm{T}_P) = 2\lfloor\sqrt{P}/2\rfloor$, i.e. the distance between any two opposite (horizontally and vertically) processors.
- Bisection bandwidth $b(\mathrm{T}_P) = 2\sqrt{P} + 2(\sqrt{P} \bmod 2)$.

2.1 Shearsort

Here we provide an algorithm to sort N values on a mesh of size $\sqrt{N} \times \sqrt{N}$, so with $P = N$ processing units, each with constant memory $M = 1$. Moreover, although not strictly required by the algorithm, for simplicity of the presentation assume that N is a power of 2 (and so also $\sqrt{N}$). The algorithm is called *shearsort* and is based on the use of any other parallel sorting algorithm for the linear array (e.g., odd-even transposition sort). The input is initially distributed one value per processing unit in the mesh. The algorithm alternates two phases until completion.

1. In the first phase each row $0 \leq i \leq \sqrt{N}-1$ of the mesh is sorted in parallel, as if it were a linear array, by employing some sorting algorithm for the linear array. However, even rows (i even) are sorted in regular order (smaller values on the left), while odd ones (i odd) in reverse order (smaller values on the right).
2. In the second phase each column of the mesh is sorted in parallel, similarly to before, by using some sorting algorithm on the corresponding linear array. This time all columns are sorted in regular order (smaller values in the upper part).

These two phases are alternated for a total of $\log N + 1$ phases, the first and the last being the same phase 1. So phase 1 is repeated $\log \sqrt{N} + 1$ times, while phase 2 just $\log \sqrt{N}$ times. After all $\log N + 1$ phases have been performed, the algorithm terminates, and the output can be found, as usual, divided among all processing units of the mesh. The output will be the sorted sequence of the input values in *snakelike order*. Snakelike order means that the output, contained in the processing units of the mesh, must be read one row at a time from the top, from left to right along even rows $0, 2, 4, \ldots, \sqrt{N}-2$ and from right to left along odd rows $1, 3, 5, \ldots, \sqrt{N}-1$, as shown below.



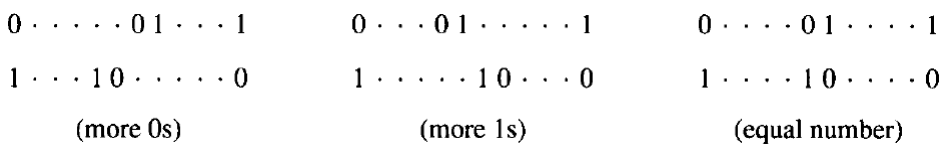
To prove the correctness of the algorithm let us assume that it uses odd-even transposition sort for sorting the rows and the columns during the computation. This makes shearsort an oblivious comparison-exchange algorithm. Indeed, the only actual sorting is performed by the odd-even transposition sort along the corresponding linear arrays of the mesh, which is oblivious comparison-exchange. Therefore, by 0–1 sorting lemma (§ 1.1), it is enough to show that the algorithm is capable of sorting all input sets consisting solely of 0s and 1s. Given such an input set, we can divide the rows of the mesh in three regions depending on their content.

- An upper region with rows containing only 0s.
- A middle region with rows containing both 0s and 1s.
- A lower region with rows containing only 1s.

Some of these regions may be empty. Moreover, the middle region may have some row that contains only 0s or only 1s, which still belongs to the middle region because mixed rows occur between it and the corresponding upper or lower region. An example of the described division is reported below.

0	0	0	0	0	0	upper region
0	0	1	1	1	0	
0	0	0	0	0	0	middle region
1	0	0	1	0	1	
1	1	1	1	1	1	lower region
1	1	1	1	1	1	

Clearly, the upper and lower regions are already sorted. Indeed, none of the phases described above would change the content of those processing units. So, the only region of the mesh which requires to and will be changed is the middle one. Let $S \leq \sqrt{N}$ be the number of rows currently in the middle mixed region. Consider consecutive pairs of those rows, of which there are $\lfloor S/2 \rfloor$, and examine the effect of phase 1 (sorting rows) on them. Depending on whether together the two rows in each pair contain more 0s, more 1s, or an equal number of each, we have the following possible outcomes



assuming that the top row of the pair is even indexed, otherwise we would obtain mirrored outcomes. It is easy to see that: in the first (and third) case every column of the pair contains at least one 0, while in the second (and third) case every column contains at least one 1. Thus, applying phase 2 (sorting columns) to any of those cases, we obtain, for each pair of the middle region, at least one newly sorted row of only 0s or 1s, depending on the case, which will be moved to the upper or lower region, respectively. So, after both phases, we will have in total at least $\lfloor S/2 \rfloor$ new fully sorted rows, divided among the upper and lower regions, and the number of rows in the middle unsorted region will decrease by that amount. This means that, iterating phases 1 and 2, S will continue to decrease until eventually becoming $S \leq 1$. At that point, either the whole input has been sorted, or only one remaining row is not yet. Then, applying one last time phase 1 (sorting rows) will complete the sorting process. To count how many iterations are actually needed, we just need to check the initial situation of the input. In the worst case, initially, the middle region extends to the whole mesh, i.e. $S = \sqrt{N}$, and the other regions are empty. Then, recalling that $\sqrt{N}$ is a power of 2, after one iteration at least $S/2$ rows have been sorted and moved to the upper or lower region, so the middle region will now have at most $S/2$ rows. Since this process continues to cut at least half of the size of the middle region at every iteration, after at most $\log \sqrt{N}$ iterations the size must have become $S \leq 1$. Then, as already mentioned, we just need one more phase 1 to terminate. So, in total the algorithm requires at most $2 \log \sqrt{N} + 1 = \log N + 1$ phases, given that it performs two phases per iteration plus one final phase.

It is worth noting that the choice of sorting rows in alternating directions, leading to the snakelike order in the end, is not arbitrary. In fact, suppose, for example, to sort all rows in the same regular direction (smaller values on the left). The resulting algorithm would not be correct any more, independently of the number of iterations. This is because in such a way smaller values can only move towards the top-left corner of the mesh, which can lead to wrong placements. A simple counterexample with 9 elements is the following.

$$\begin{matrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{matrix}$$

Note that every row and every column is already sorted, w.r.t. to the corresponding regular orders, so nothing would change after any iteration. However, the mesh is not sorted if read by rows from left to right (*row major*), nor by columns from top to bottom (*column major*), nor even by diagonals.

Performances

- The parallel execution time is $\mathcal{T}_N(N) = \Theta(\sqrt{N} \log N)$ since the algorithm performs $1 + \log N$ phases, each taking $\Theta(\sqrt{N})$ parallel time, then the work is $\mathcal{W} = \Theta(N\sqrt{N} \log N)$.
- The speedup is $\mathcal{S} = \Theta(N \log N)/\Theta(\sqrt{N} \log N) = \Theta(\sqrt{N})$ w.r.t. the fastest sequential sorting algorithm, hence the efficiency is $\mathcal{E} = \Theta(\sqrt{N}/N) = \Theta(1/\sqrt{N})$.

2.2 Matrix Multiplication

Matrix multiplication is a classical problem for which a number algorithms have been proposed, both sequential and parallel. For simplicity we will restrict to the product of square matrices. Given two square matrices A and B of size $N \times N$, we want to compute their product $C = A \cdot B$, where each element is defined by $C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \cdot B_{k,j}$, for every $0 \leq i, j \leq N-1$. Moreover, we want to do this on a torus of size $N \times N$, so with $P = N^2$ processing units with constant memory $M = 3$.

Since each processor $p_{i,j}$ of the network has only constant memory $M = 3$, the two input matrices must be both distributed one element per processing unit. So, initially processor $p_{i,j}$ will contain elements $A_{i,j}$ and $B_{i,j}$ of the input matrices. The third memory cell will instead be used to store the corresponding element $C_{i,j}$ of the product. Initially, every $C_{i,j} = 0$. Observe that some processors cannot use their initial input values right away. For instance, processor $p_{0,1}$ cannot immediately compute any component of the sum $C_{0,1} = A_{0,0} \cdot B_{0,1} + A_{0,1} \cdot B_{1,1} + \ldots + A_{0,N-1} \cdot B_{N-1,1}$, since its input values are $A_{0,1}$ and $B_{0,1}$. Therefore, to avoid idle waiting, the algorithm must first move the data to a configuration where each processing unit contains a pair of input values that can immediately use for computing one addend of the corresponding element of C. To understand what such a configuration could be, we first give an example with $N = 3$.

$$
\begin{aligned}
C_{0,0} &= A_{0,0} \cdot B_{0,0} + A_{0,1} \cdot B_{1,0} + A_{0,2} \cdot B_{2,0} \\
C_{0,1} &= A_{0,0} \cdot B_{0,1} + A_{0,1} \cdot B_{1,1} + A_{0,2} \cdot B_{2,1} \\
C_{0,2} &= A_{0,0} \cdot B_{0,2} + A_{0,1} \cdot B_{1,2} + A_{0,2} \cdot B_{2,2} \\
C_{1,0} &= A_{1,0} \cdot B_{0,0} + A_{1,1} \cdot B_{1,0} + A_{1,2} \cdot B_{2,0} \\
C_{1,1} &= A_{1,0} \cdot B_{0,1} + A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{1,2} &= A_{1,0} \cdot B_{0,2} + A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\
C_{2,0} &= A_{2,0} \cdot B_{0,0} + A_{2,1} \cdot B_{1,0} + A_{2,2} \cdot B_{2,0} \\
C_{2,1} &= A_{2,0} \cdot B_{0,1} + A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\
C_{2,2} &= A_{2,0} \cdot B_{0,2} + A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}
\end{aligned}
$$

Now, for each $C_{i,j}$, we need to find a pair of values $A_{i,k}$ and $B_{k,j}$ such that they will not be chosen for any other $C_{i',j'}$. This is due to the fact that all inputs must still be uniquely distributed, a different pair to each processing unit. There are many configurations that satisfy such conditions, even just by symmetry. For instance, consider the one depicted in Figure 7 below. Recalling that initially each processing unit $p_{i,j}$ received values $A_{i,j}$ and $B_{i,j}$, we can trace back how the data were moved to reach such a configuration. Indeed, it is not hard to see that each row i of A has been rotated leftwards by i positions, and each column j of B has been rotated upwards by j positions. Moreover, the same works also for a general N. In fact, after the rotations, each processor $p_{i,j}$ will contain elements $A_{i,k}$ and $B_{k,j}$, where $k = i+j \bmod N$, since the row (resp. column) was rotated leftwards (resp. upwards) by i (resp. j) positions. So, the algorithm, before starting the actual computation, needs $N-1$ steps to move the data to the described configuration, since the last row (resp. column) of A (resp. B) needs to be moved by $N-1$ positions. This phase can only be avoided if data are input already in such a configuration, which we assume not to.

In the next phase of the procedure the processing units perform the actual computation of the product. To do this, they alternate arithmetical operations, using the pair of values

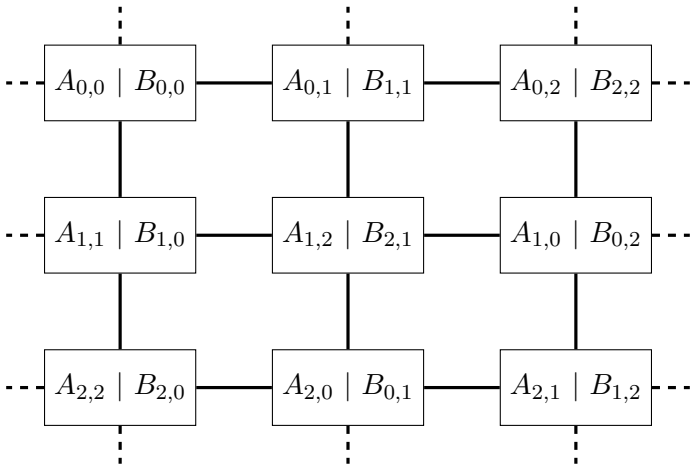


Figure 7: A distribution of the input allowing each processor to perform some computation.

currently stored, and communications to obtain the next pair of input values. In particular, during each step, processor $p_{i,j}$ updates its own $C_{i,j} = C_{i,j} + A_{i,k} \cdot B_{k,j}$, whatever $A_{i,k}$ and $B_{k,j}$ it currently contains. Then, it forwards the value $A_{i,k}$ to its left neighbour, and $B_{k,j}$ to the neighbour above, while receiving values $A_{i,k'}$ and $B_{k',j}$, where $k' = k+1 \bmod N$, from the right and below links, respectively. So, after each step the rows of A will be rotated leftwards by one position, and the columns of B upwards by one position. Since rows/columns always rotate in the same direction, after N steps each processing unit must have seen every pair of values it needed to compute the corresponding element of the product. Therefore, the two phases will take a total of $N - 1 + N = 2N - 1$ steps, at the end of which the torus will contain the product C of matrices A and B.

Interestingly, the presented algorithm works also when the torus has fewer processors $P < N^2$, hence size $\sqrt{P} \times \sqrt{P}$, with N multiple of $\sqrt{P}$. In such a case, we use *block distribution* on both rows and columns for the input of matrices A and B. So, processing unit $p_{i,j}$ will initially receive blocks $A(i,j)$ and $B(i,j)$ of size $n \times n$ where $n = N/\sqrt{P}$, i.e.

$$A(i,j) = \begin{bmatrix} A_{in,\,jn} & A_{in,\,jn+1} & \cdots & A_{in,\,(j+1)n-1} \\ A_{in+1,\,jn} & A_{in+1,\,jn+1} & & \\ \vdots & & \ddots & \\ A_{(i+1)n-1,\,jn} & A_{(i+1)n-1,\,jn+1} & \cdots & A_{(i+1)n-1,\,(j+1)n-1} \end{bmatrix}$$

and similarly for $B(i,j)$. Moreover, each processor $p_{i,j}$ will aim at computing the corresponding block $C(i,j)$ of the product C. Therefore, this time each processing unit will need more memory $M = 3N^2/P$, which is not constant any more. The algorithm will then perform the same steps described before, for both phases, but using blocks instead of single elements.

1. Each row i of the torus will rotate leftwards the blocks $A(i,\_)$ therein contained by i positions, while the processors in each column j will rotate upwards their blocks $B(\_,j)$ by j positions.

2. During each step of the second phase, every processor $p_{i,j}$ will update its block $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$, using its current $A(i,k)$ and $B(k,j)$, and then

forward them leftwards and upwards, respectively.

Note that now the steps are not constant any more, not even the communications, since they involve blocks of N^2/P elements. Furthermore, the arithmetical operations require to compute a matrix product, since the blocks $A(i,k)$ and $B(k,j)$ are actually matrices. On the other hand, the two phases take just $\sqrt{P}-1$ and $\sqrt{P}$ steps, respectively.

Performances

- The parallel execution time is $\mathcal{T}_{N^2}(N) = \Theta(N)$ since the procedure takes $2N-1$ constant steps, then the work is $\mathcal{W} = \Theta(N^3)$.
- The speedup is $\mathcal{S} = O(N^{2.37\sim})/\Theta(N) = O(N^{1.37\sim})$ w.r.t. the fastest sequential algorithm (although, as already mentioned, such an algorithm is actually impractical due to the huge constants), then the efficiency is $\mathcal{E} = O(N^{1.37\sim}/N^2) = O(1/N^{0.63\sim})$.

2.3 Transitive Closure of a Graph

The *transitive closure* of a (directed) graph $G = (V, E)$ is denoted by $G^* = (V, E^*)$, where $E^* = \{(u,v) \in V \times V \mid \text{there is a (directed) path from } u \text{ to } v\}$. This is a common operation that arises, for example, in situations where a graph is used to represent a relation between objects. The nodes V of a graph can be assumed to be enumerated by natural numbers, so that $V = \{0, 1, \ldots, |V|-1\}$. Then, the *adjacency matrix* A of a graph $G = (V, E)$ is a $|V| \times |V|$ matrix whose elements $A_{i,j}$, for $i, j \in V$, are 1 if $(i,j) \in E$, and 0 otherwise. Similarly to before, we can define the transitive closure of an adjacency matrix A of a graph G, denoted by A^*, as the adjacency matrix of G^*. An example of graph, its adjacency matrix, and the corresponding transitive closures is reported in Figure 8.

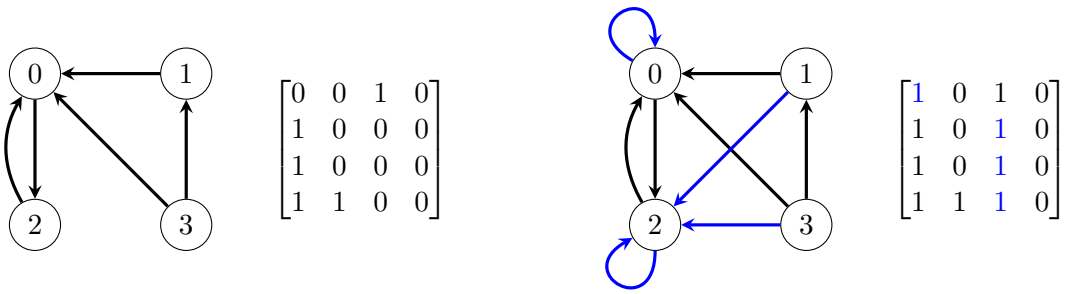


Figure 8: A graph and its adjacency matrix (left) and their transitive closures (right).

Before giving the parallel algorithm, we describe a general procedure for computing the transitive closure consisting of $|V|$ phases. Each phase may add new edges, depending on those currently in the graph. Let $G^k = (V, E^k)$ and $G^{k+1} = (V, E^{k+1})$ be the graph at the beginning and at the end of phase k, respectively. Then, $G^0 = G$ is the initial graph and $G^{|V|}$ is the result of the procedure, which, as we will prove next, is the transitive closure G^*. At each phase $0 \leq k < |V|$, the graph is updated by $E^{k+1} = E^k \cup \{(i,j) \in V \times V \mid (i,k), (k,j) \in E^k\}$. In plain words, during phase k, for all $i, j \in V$, if in G^k there exist an edge from i to k and one from k to j, we add the edge (i,j) corresponding to their transitive

closure. The proof that after phase $|V|-1$, the graph $G^{|V|}$ is the transitive closure G^*, is based on one crucial observation: for all $0 \leq k \leq |V|$, the graph G^k contains an edge (i,j) if and only if in the original graph G there is a path from i to j which, excluding i and j, contains only nodes in $\{0,1,\ldots,k-1\}$. After phase $|V|-1$, so for $k=|V|$, the latter set is the whole set of nodes V, and so $G^{|V|}$ must be the transitive closure. We can prove the previous observation by induction on k. Initially, for $k=0$, the graph $G^0=G$ contains only edges (i,j) already in G, which are the only possible paths from i to j without other nodes, since the set $\{0,\ldots,k-1\}=\emptyset$. Then, assuming the property holds for $k \geq 0$, we show that it holds also for $k+1$. G^{k+1} is the result of phase k, which added every edge (i,j) such that there were edges (i,k) and (k,j) in G^k. By inductive hypothesis, this means that in G there are paths from i to k and from k to j going only through nodes in $\{0,\ldots,k-1\}$. Thus, concatenating such paths, in G there is also a path from i to j going though nodes in $\{0,\ldots,k\}$, which proves the property for $k+1$.

We can now explain the parallel algorithm to compute the transitive closure of a directed graph $G=(V,E)$ with $|V|=N$ nodes, on a mesh with the same size of the corresponding adjacency matrix A, that is, with $P=N^2$ processing units with constant memory. The algorithm follows the N phases described above, performing them in a systolic way. However, this time, each phase takes more than a single step, but some steps belonging to different phases can be parallelized. The algorithm is applied directly to the adjacency matrix A, which is input one row at a time from the top of the mesh: each element of the row to the corresponding processing unit of the first row of the mesh. Another difference, w.r.t. the systolic algorithms that we saw for the linear array network, is that data will not get forwarded to the next row of the mesh at every step. Instead, a row A_i input to the mesh will actually stop at the first row of the mesh that did not received a row already. It is easy to see that such a row of the mesh is actually row i, since the matrix A is input starting from the first row A_0, which will stop immediately in row 0 of the mesh, and so on for the following rows of A. This also means that row A_i will reach row i of the mesh during step $2i+1$, since it must wait i steps before entering the mesh and then another $i+1$ steps to reach the corresponding row. The actual computation done by the algorithm, which will be explained later, happens only when a row i of the mesh receives another row of the matrix other than A_i. Finally, when a row i of processors does not receive any more inputs, it will forward to the next row of processors its own stored row A_i, which stayed there for N steps since initially arriving, while every other row of the matrix passed through (+1 more idle step at the end). So, counting from the start, the row A_i will start moving again after step $N+2i+1$. After starting moving again, a row A_i will flow downwards through the mesh for $N-1-i$ steps, up to the last row $N-1$ of processors, which will also send away such values, during the next step, as an output. So, since the beginning of the computation, row A_i will be output from the last row of the mesh during step $2N+i+1$. These outputs will be the actual rows of the transitive closure A^*. Moreover, the algorithm will take $3N$ steps to complete its execution, since the last row A_{N-1} will be output during step $2N+(N-1)+1=3N$. The data flow for an adjacency matrix and a mesh with 4 rows (and 4 columns) is depicted in Figure 9 below. The notation $A_i^{\sim}$ in the figure indicates that A_i went already through some computation, but is not yet the transitive closure, which instead is denoted by A_i^*.

Now, recall the $|V|=N$ phases of the general procedure described above. Let $E_i = \{(u,v) \in E \mid u=i\}$ be the subset of edges whose source is the node i. Observe that, for every $0 \leq i \leq N-1$, the edges E_i^{k+1} resulting after phase k depend only on the edges in E_i^k and E_k^k, since $E_i^{k+1} = E_i^k \cup \{(i,j) \mid j \in V \wedge (i,k) \in E_i^k \wedge (k,j) \in E_k^k\}$. Initially the edges E_i^0

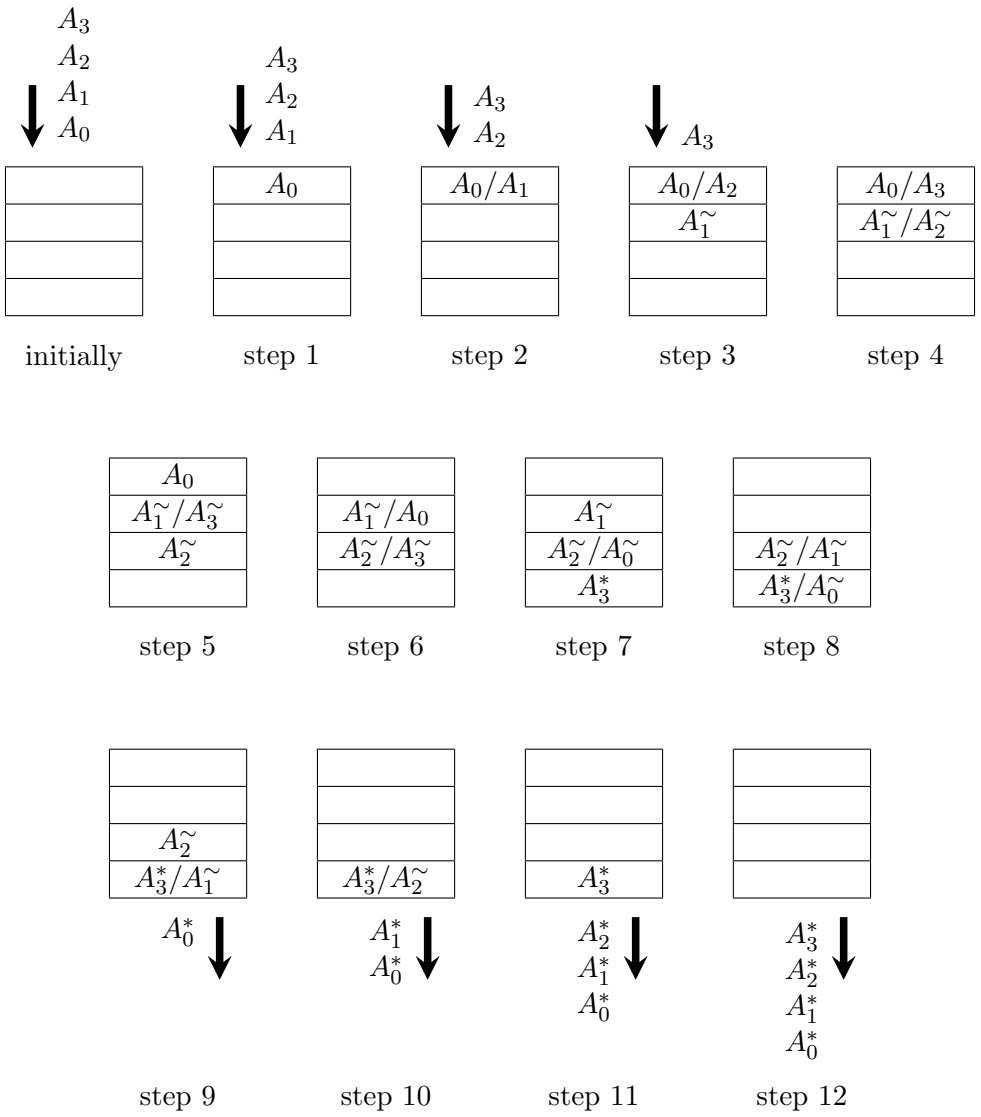


Figure 9: Flow of the rows of the adjacency matrix of a graph with 4 nodes for computing the transitive closure on a 4×4 mesh.

are those given by the row A_i of the adjacency matrix. The algorithm aims at computing the subsets E_i^k, starting from $k = 1$, separately, possibly in parallel for different is. For instance, the first to be computed is E_1^1, when row A_1, currently corresponding to E_1^0, is received by the first row of the mesh, which has already stored row A_0, corresponding to E_0^0. In this case, processor $p_{0,0}$ "broadcasts" the received value $A_{1,0}$ to all other processors of that row. Here broadcast means to send the value to all (horizontal) neighbours, which in turn will forward it along the same direction, until it reaches the leftmost and rightmost processors of that row. When the value $A_{1,0}$ reaches a processor $p_{0,j}$ of that row, the latter updates its own received value $A_{1,j} = A_{1,j} \vee (A_{1,0} \wedge A_{0,j})$. This is exactly what is done in the first step of the procedure above for the edge $(1, j)$, since $A_{1,0} \wedge A_{0,j}$ corresponds to the existence of edges $(1, 0)$ in E_1^0 and $(0, j)$ in E_0^0. So, after all processing units in the first row of the mesh have received the value $A_{1,0}$ and possibly updated their own $A_{1,j}$,

the row A_1 will correspond to the subset of edges E_1^1. This process is repeated similarly for every row of the matrix passing over another one already stored in a row of the mesh. In general, whenever row A_i is received by row k of the mesh, containing row A_k of the matrix, processor $p_{k,k}$ broadcasts the received value $A_{i,k}$ and every other processor $p_{k,j}$ that receives it updates its own $A_{i,j} = A_{i,j} \vee (A_{i,k} \wedge A_{k,j})$.

By the time row A_i reaches row i of the mesh, where it stops and is stored for awhile, it has been updated up to correspond to E_i^i. Indeed, while such a row A_i, initially corresponding to E_i^0, traverses the rows $k = 0, 1, \ldots, i-1$ before, which, by the same reasoning, already contain the corresponding E_k^k, it is updated to E_i^{k+1}, since this requires only to know E_i^k and E_k^k. Then, the last of those updates, for $k = i-1$, will compute E_i^i. In order to complete the computation of each E_i^N, the row A_i, currently corresponding to E_i^i, must pass over the remaining $k = i+1, i+2, \ldots, N-1$ rows of the matrix, once they have all been updated to the corresponding E_k^k, which is guaranteed once they have also reached their matching row k of the mesh. To perform this second part of the computation row A_i starts moving downwards again and traverses the remaining rows of the mesh below it, up to the last one. Note, however, that this must happen after that all previous rows A_h, for $h < i$, have first passed over row i itself, otherwise such rows would not have the opportunity to pass over A_i any more. For this reason, the row i of the mesh waits until step $N + 2i + 1$ before forwarding its stored row A_i. Finally, once row A_i passes over all rows of the mesh below and is output, it will correspond to the edges in E_i^N, as desired.

As already mentioned, the algorithm takes $3N$ steps. However, many of those steps require to broadcast some values along rows of the mesh. If we were to do that, we would need to add an upper bound of $N - 1$ steps for each broadcast, that is, the steps required to move a value from one opposite to the other of the row of N processors. This would lead to a quadratic upper bound $O(N^2)$ of constant steps. This may be true for a naive implementation, where the data move downwards one row and then wait for the broadcasting phase to complete before continuing. But it turns out that we can do much better, if we take into account the actual dependencies between the operations. Indeed, the need to wait for broadcasting can be eliminated through a technique known as *retiming*. This allows to insert delays on some communications, while anticipating some others, in such a way that each processor still receives its inputs in the same order/combination as before. Doing so we are able to actually perform all the computation in $3N + 2N - 2 = 5N - 2$ constant steps, by inserting a total delay of $2N - 2$ steps in order to ensure that data dependencies are still satisfied while eliminating useless waiting for broadcasts.

Performances

- The parallel execution time is $\mathcal{T}_{N^2}(N) = \Theta(N)$ since with retiming the procedure takes $5N - 2$ constant steps, then the work is $\mathcal{W} = \Theta(N^3)$.
- The speedup is $\mathcal{S} = \Theta(N^3)/\Theta(N) = \Theta(N^2)$ w.r.t. a naive sequential algorithm (faster sequential algorithms exist, but they are based on matrix multiplication, and so impractical due to the huge constants), then the efficiency is $\mathcal{E} = \Theta(N^2/N^2) = \Theta(1)$.

2.4 Least-Weight Paths of a Graph

Most interestingly, the previous parallel algorithm for computing the transitive closure of a directed graph can be easily adapted to solve a number of graphs problems. For example,

the computation of the *least-weight (directed) path* between each pair of nodes i, j. In this setting, every edge $(i, j) \in E$ is labelled by a weight $w_{i,j}$ and the adjacency matrix A becomes the weights matrix W, containing the corresponding weights instead of values in $\{0, 1\}$. Moreover, every possible edge will be assumed to be present in the graph, i.e. $E = V \times V$, but possibly with infinite weight. In order to ensure that a least-weight path exists for every pair of nodes, we will also assume that the graph does not contain negative-weight cycles.

The algorithm is as the one for computing the transitive closure, with one difference: the update performed by each processor $p_{k,j}$, after storing the weight $W_{k,j}$ and receiving $W_{i,j}$ and $W_{i,k}$, changes to $W_{i,j} = \min(W_{i,j}, W_{i,k}+W_{k,j})$. In the end, each element $W_{i,j}$ of the output matrix will correspond to the total weight of the least-weight path from node i to node j. To compute also the actual path connecting such nodes, it is enough to record (and keep updated) also the next node in the path, for each pair of nodes. For instance, suppose such a node $x_{i,j}$ is recorded in another matrix X, for all $0 \leq i, j \leq |V| - 1$. Rows of such a matrix will flow along with those of W, and similarly the elements that are broadcasted. Initially $x_{i,j} = j$, i.e., the successor of i for the single edge (i, j). Then, at every update $W_{i,j} = \min(W_{i,j}, W_{i,k}+W_{k,j})$, also $x_{i,j}$ is updated with either $x_{i,j}$ or $x_{i,k}$, depending on which value is used as the minimum for $W_{i,j}$. At the end of the computation the actual least-weight path from i to j is given by the sequence $i, x_{i,j}, x_{x_{i,j},j}, \ldots$ until j is reached.

So, the same network and algorithm that we used to compute the transitive closure of a graph can be used to compute shortest (least-weight) paths between all pairs of nodes. Thus, the performances of the algorithm are still those given in the previous section.

2.5 Connected Components of a Graph

Another graph problem that can take advantage of the algorithm for the transitive closure is that of finding the connected components of an undirected graph. A *connected component* is a maximal subset $S \subseteq V$ of nodes such that for every pair of nodes $i, j \in S$, there is a path from i to j. To be maximal means that no other node $k \in V \setminus S$ in the graph is connected by a path to some node of S.

The computation of all connected components of a graph can be performed as follows. First, we compute the transitive closure A^* of the adjacency matrix. Then, $A^*_{i,j} = 1$ if and only if nodes i and j are in the same connected component. So, it is enough to associate each node $i \in V$ with the index $k = \min\{j \in V \mid A^*_{i,j} = 1\}$, that is, the index k of leftmost 1 in the row i of the matrix. Nodes associated with the same index k form a connected component, one for each different k.

Again, performances are similar to those of the transitive closure algorithm since the added steps, which take place after computing the transitive closure, can still be performed on the same mesh network in linear time (in parallel). However, note that this time the speedup and efficiency are not as good, because the connected components can be computed in just $O(|E|)$ steps sequentially.