Course Name: Analysis and Design of Algorithms

Topics to be covered

- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

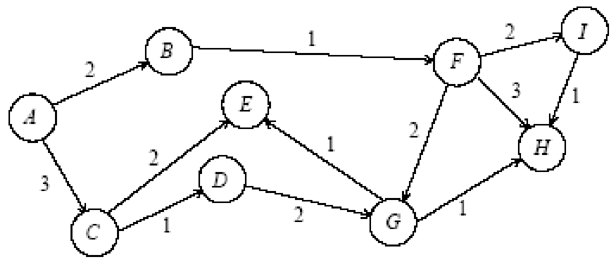
All-Pairs Shortest Paths

- Given a weighted graph *G(V,E,w)*, the *all-pairs shortest paths* problem is to find the shortest paths between all pairs of vertices $v_i, v_j \in V$.
- A number of algorithms are known for solving this problem.

All-Pairs Shortest Paths: Matrix-Multiplication Based Algorithm

- Consider the multiplication of the weighted adjacency matrix with itself - except, in this case, we replace the multiplication operation in matrix multiplication by addition, and the addition operation by minimization.
- Notice that the product of weighted adjacency matrix with itself returns a matrix that contains shortest paths of length 2 between any pair of nodes.
- It follows from this argument that A^n contains all shortest paths.

Matrix-Multiplication Based Algorithm



$$A^1 = \begin{pmatrix} 0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^8 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Matrix-Multiplication Based Algorithm

- A^n is computed by doubling powers - i.e., as A, A^2, A^4, A^8, and so on.
- We need *log n* matrix multiplications, each taking time $O(n^3)$.
- The serial complexity of this procedure is $O(n^3 \log n)$.
- This algorithm is not optimal, since the best known algorithms have complexity $O(n^3)$.

Matrix-Multiplication Based Algorithm: Parallel Formulation

- Each of the *log n* matrix multiplications can be performed in parallel.
- We can use $n^3/\log n$ processors to compute each matrix-matrix product in time *log n*.
- The entire process takes $O(\log^2 n)$ time.

Dijkstra's Algorithm

- Execute n instances of the single-source shortest path problem, one for each of the n source vertices.
- Complexity is $O(n^3)$.

Dijkstra's Algorithm: Parallel Formulation

- Two parallelization strategies - execute each of the *n* shortest path problems on a different processor (source partitioned), or use a parallel formulation of the shortest path problem to increase concurrency (source parallel).

Dijkstra's Algorithm: Source Partitioned Formulation

- Use n processors, each processor P_i finds the shortest paths from vertex v_i to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.
- It requires no interprocess communication (provided that the adjacency matrix is replicated at all processes).
- The parallel run time of this formulation is: $\Theta(n^2)$.
- While the algorithm is cost optimal, it can only use n processors. Therefore, the isoefficiency due to concurrency is p^3.

Dijkstra's Algorithm: Source Parallel Formulation

- In this case, each of the shortest path problems is further executed in parallel. We can therefore use up to n^2 processors.
- Given p processors ($p > n$), each single source shortest path problem is executed by p/n processors.
- Using previous results, this takes time:

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p)}^{\text{communication}}.$$

- For cost optimality, we have $p = O(n^2/\log n)$ and the isoefficiency is $\Theta((p \log p)^{1.5})$.

Floyd's Algorithm

- For any pair of vertices $v_i, v_j \in V$, consider all paths from v_i to v_j whose intermediate vertices belong to the set $\{v_1, v_2, \ldots, v_k\}$. Let $p_{i,j}^{(k)}$ (of weight $d_{i,j}^{(k)}$ be the minimum-weight path among them.
- If vertex v_k is not in the shortest path from v_i to v_j, then $p_{i,j}^{(k)}$ is the same as $p_{i,j}^{(k-1)}$.
- If f v_k is in $p_{i,j}^{(k)}$, then we can break $p_{i,j}^{(k)}$ into two paths - one from v_i to v_k and one from v_k to v_j . Each of these paths uses vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$.

Floyd's Algorithm

From our observations, the following recurrence relation follows:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

This equation must be computed for each pair of nodes and for *k = 1, n*. The serial complexity is $O(n^3)$.

Floyd's Algorithm

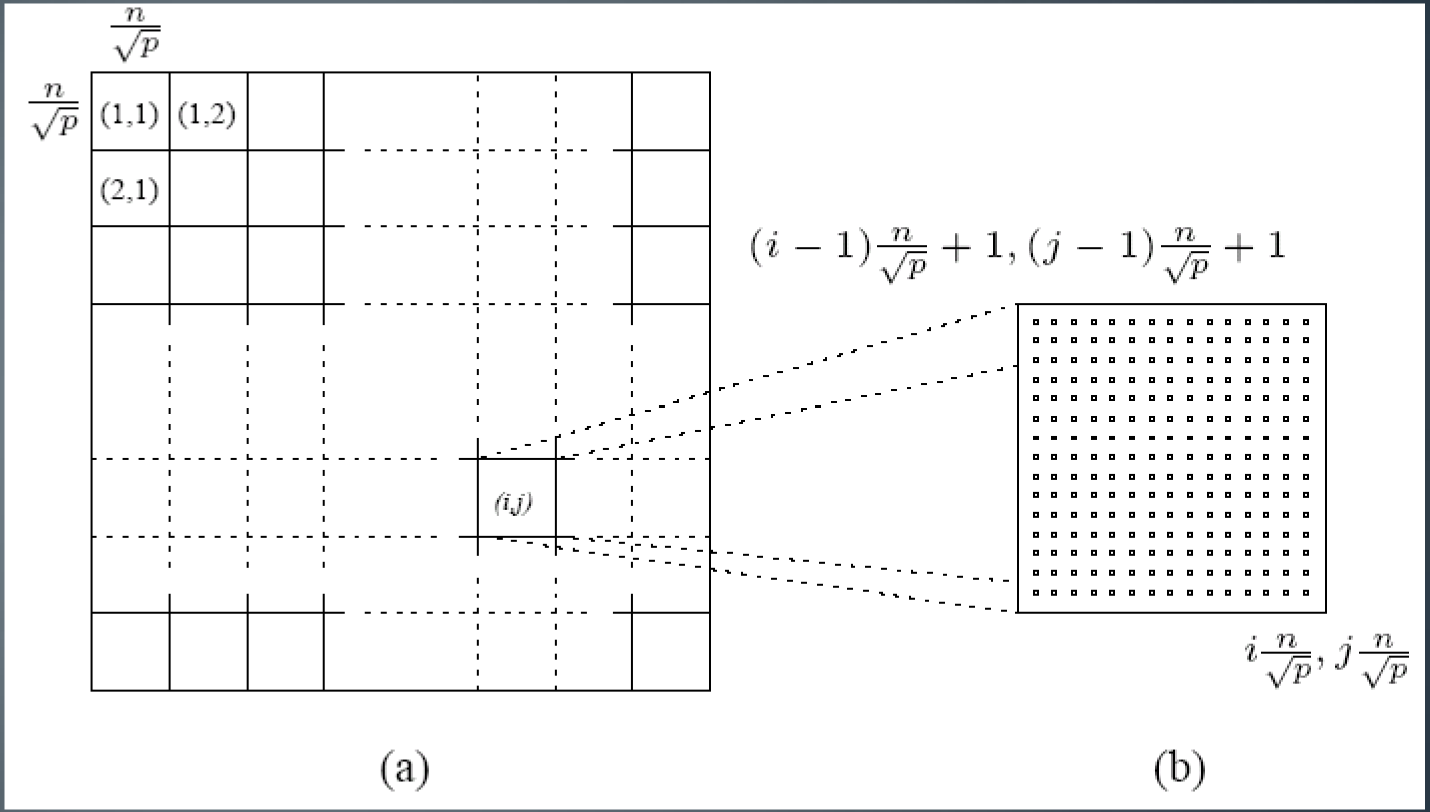
procedure FLOYD_ALL_PAIRS_SP(A)
begin
    $D^{(0)} = A$;
    for $k := 1$ **to** n **do**
        for $i := 1$ **to** n **do**
            for $j := 1$ **to** n **do**
                $d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$;
end FLOYD_ALL_PAIRS_SP

Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph $G = (V,E)$ with adjacency matrix A.

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

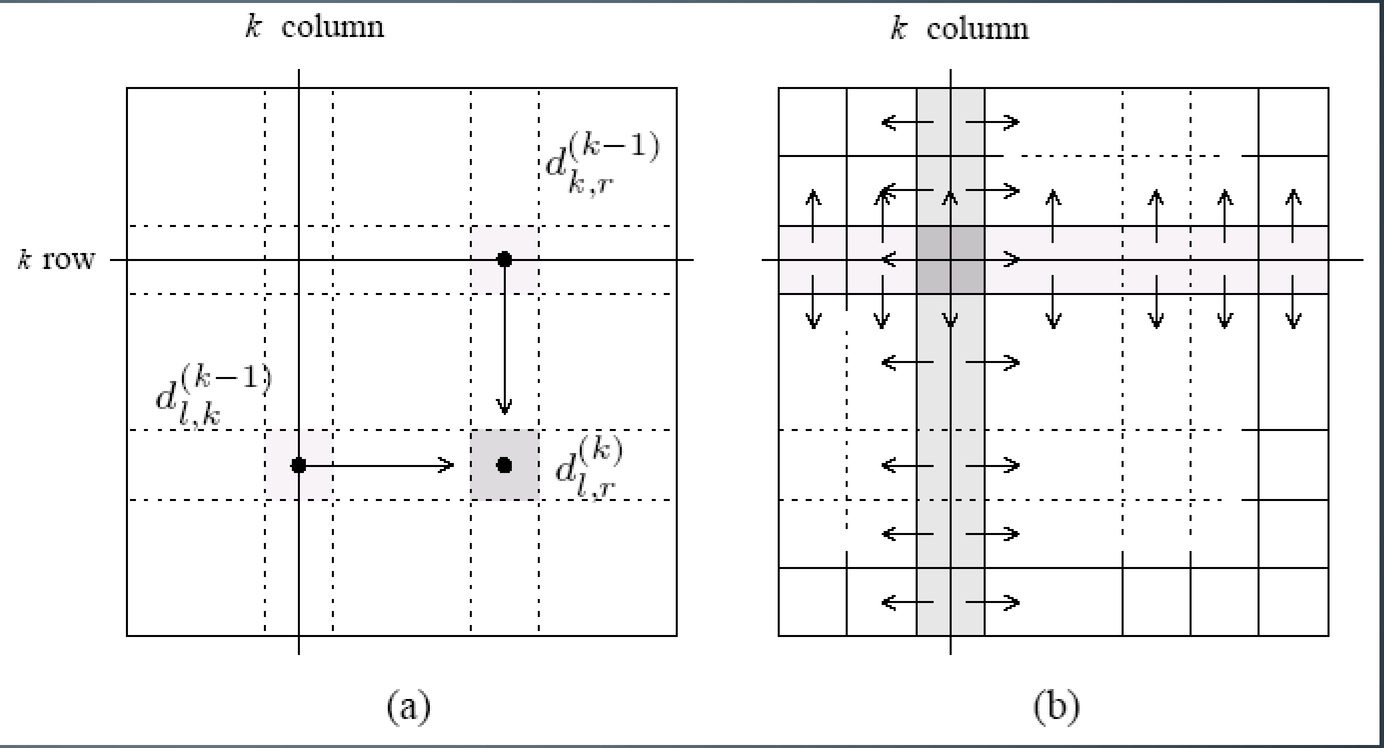
- Matrix $D^{(k)}$ is divided into p blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$.
- Each processor updates its part of the matrix during each iteration.
- To compute $d_{l,k}^{(k-1)}$ processor $P_{i,j}$ must get $d_{l,k}^{(k-1)}$ and $d_{k,r}^{(k-1)}$.
- In general, during the k^{th} iteration, each of the $\sqrt{p}$ processes containing part of the k^{th} row send it to the $\sqrt{p}$ - 1 processes in the same column.
- Similarly, each of the $\sqrt{p}$ processes containing part of the k^{th} column sends it to the $\sqrt{p} - 1$ processes in the same row.

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping



(a) Matrix $D^{(k)}$ distributed by 2-D block mapping into $\sqrt{p} \times \sqrt{p}$ subblocks, and (b) the subblock of $D^{(k)}$ assigned to process $P_{i,j}$.

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping



(a) Communication patterns used in the 2-D block mapping. When computing $d_{i,j}^{(k)}$, information must be sent to the highlighted process from two other processes along the same row and column. (b) The row and column of $\sqrt{p}$ processes that contain the k^{th} row and column send them along process columns and rows.

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

procedure FLOYD_2DBLOCK($D^{(0)}$)
begin
    for $k := 1$ **to** n **do**
    begin
        each process $P_{i,j}$ that has a segment of the k^{th} row of $D^{(k-1)}$;
            broadcasts it to the $P_{*,j}$ processes;
        each process $P_{i,j}$ that has a segment of the k^{th} column of $D^{(k-1)}$;
            broadcasts it to the $P_{i,*}$ processes;
        each process waits to receive the needed segments;
        each process $P_{i,j}$ computes its part of the $D^{(k)}$ matrix;
    end
end FLOYD_2DBLOCK

Floyd's parallel formulation using the 2-D block mapping. $P_{*,j}$ denotes all the processes in the j^{th} column, and $P_{i,*}$ denotes all the processes in the i^{th} row. The matrix $D^{(0)}$ is the adjacency matrix.

Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- During each iteration of the algorithm, the k^{th} row and k^{th} column of processors perform a one-to-all broadcast along their rows/columns.
- The size of this broadcast is $n/\sqrt{p}$ elements, taking time $\Theta((n \log p)/\sqrt{p})$.
- The synchronization step takes time $\Theta(\log p)$.
- The computation time is $\Theta(n^2/p)$.
- The parallel run time of the 2-D block mapping formulation of Floyd's algorithm is

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}} \log p\right)}^{\text{communication}}.$$

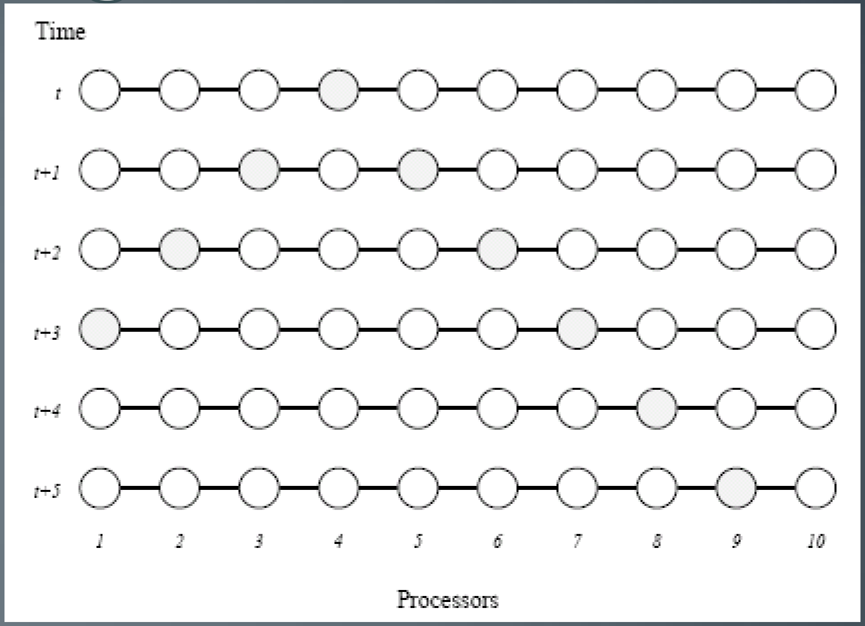
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- The above formulation can use $O(n^2 / \log^2 n)$ processors cost-optimally.
- The isoefficiency of this formulation is $\Theta(p^{1.5} \log^3 p)$.
- This algorithm can be further improved by relaxing the strict synchronization after each iteration.

Floyd's Algorithm: Speeding Things Up by Pipelining

- The synchronization step in parallel Floyd's algorithm can be removed without affecting the correctness of the algorithm.
- A process starts working on the k^{th} iteration as soon as it has computed the $(k-1)^{th}$ iteration and has the relevant parts of the $D^{(k-1)}$ matrix.

Floyd's Algorithm: Speeding Things Up by Pipelining



Communication protocol followed in the pipelined 2-D block mapping formulation of Floyd's algorithm. Assume that process 4 at time *t* has just computed a segment of the k^{th} column of the $D^{(k-1)}$ matrix. It sends the segment to processes 3 and 5. These processes receive the segment at time $t + 1$ (where the time unit is the time it takes for a matrix segment to travel over the communication link between adjacent processes). Similarly, processes farther away from process 4 receive the segment later. Process 1 (at the boundary) does not forward the segment after receiving it.

Floyd's Algorithm: Speeding Things Up by Pipelining

- In each step, $n/\sqrt{p}$ elements of the first row are sent from process $P_{i,j}$ to $P_{i+1,j}$.
- Similarly, elements of the first column are sent from process $P_{i,j}$ to process $P_{i,j+1}$.
- Each such step takes time $\Theta(n/\sqrt{p})$.
- After $\Theta(\sqrt{p})$ steps, process $P_{\sqrt{p},\sqrt{p}}$ gets the relevant elements of the first row and first column in time $\Theta(n)$.
- The values of successive rows and columns follow after time $\Theta(n^2/p)$ in a pipelined mode.
- Process $P_{\sqrt{p},\sqrt{p}}$ finishes its share of the shortest path computation in time $\Theta(n^3/p) + \Theta(n)$.
- When process $P_{\sqrt{p},\sqrt{p}}$ has finished the $(n-1)^{th}$ iteration, it sends the relevant values of the n^{th} row and column to the other processes.

Floyd's Algorithm: Speeding Things Up by Pipelining

- The overall parallel run time of this formulation is

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

- The pipelined formulation of Floyd's algorithm uses up to *$O(n^2)$* processes efficiently.
- The corresponding isoefficiency is *$\Theta(p^{1.5})$*.

All-pairs Shortest Path: Comparison

- The performance and scalability of the all-pairs shortest paths algorithms on various architectures with bisection bandwidth. Similar run times apply to all cube architectures, provided that processes are properly mapped to the underlying processors.

	Maximum Number of Processes for $E = \Theta(1)$	Corresponding Parallel Run Time	Isoefficiency Function
Dijkstra source-partitioned	$\Theta(n)$	$\Theta(n^2)$	$\Theta(p^3)$
Dijkstra source-parallel	$\Theta(n^2/\log n)$	$\Theta(n \log n)$	$\Theta((p \log p)^{1.5})$
Floyd 1-D block	$\Theta(n/\log n)$	$\Theta(n^2 \log n)$	$\Theta((p \log p)^3)$
Floyd 2-D block	$\Theta(n^2/\log^2 n)$	$\Theta(n \log^2 n)$	$\Theta(p^{1.5} \log^3 p)$
Floyd pipelined 2-D block	$\Theta(n^2)$	$\Theta(n)$	$\Theta(p^{1.5})$