

**University of Dhaka**  
**Computer Science and Engineering**  
**4th Year 2nd Semester B.Sc.: 2022**  
**CSE4269– Parallel and Distributed Systems**

**Assignment Code: A4**

**Assignment Title: Floyd’s all-pairs shortest path algorithm .**

---

- **Objectives**

The objective of this assignment is to solve an embarrassingly parallel problem, with a certain class of MPI routines, called ‘collective’ .

- **Programming Assignment**

Given a weighed graph  $G = (V, E)$  with  $n$  nodes, the cost of an edge from node  $i$  to  $j$  is  $c_{i,j}$ . Floyd’s algorithm calculates the cost  $d_{i,j}$  of the shortest path between each pair of nodes  $(i, j)$  in  $V$ . A recurrence formulation for calculating  $d_{i,j}$  using  $k$  intermediate nodes is provided in formula 12.6 of the book [Introduction to Parallel Computing: Ananth Grama]. You are required to calculate  $d_{i,j}$  using the formula for each pair of nodes in the given graph, using a parallel dynamic programming solution strategy. In your parallel solution, the cost matrix at level  $k$  which computes  $d_{i,j}^k$ , i.e., the  $d_{i,j}$  values using  $k$  intermediate nodes, is calculated from the cost matrix at level  $k - 1$ . Given a graph with  $n$  nodes, the final goal is to calculate the cost matrix at level  $n$ , which includes all  $n$  nodes. The possible solution strategies:

1. Strategy : Row- and column-wise one-to-all broadcasts are used while computing the level  $k$  matrix. The algorithm is discussed in the textbook (Chapter 12.4.1 and 10.4.2) .
2. Strategy 2: Pipelining replaces the one-to-all broadcasts.

Here is what you are required to do:

1. In this assignment, you are required to implement the parallel programs for each of the above two solution strategies and test on a graph of  $n$  nodes using  $P$  processors.
2. Write a sequential version of the algorithm (Chapter 12.4.1) and execute it on a single node of the cluster. Measure the execution time.
3. You are also required to measure the speed-up with different values of  $n$  and  $P$  for both the parallel versions.
4. Explain your experimental findings, in terms of which of the two versions is better (if any) and why

- **Your Task**

- Complete the programming part and run the experiment in your own machine. See the performance and plot that accordingly.
- Repeat the same in the cluster.

## • Submission

- Name of Submitted file:
  - \* Serial Version: serialFloyd.c/ serialFloyd.cpp
  - \* Parallel Version: parallelFloyd.c/ parallelFloyd.cpp
  - \* Pipeline Version: pipelineFloyd.c/ pipelineFloyd.cpp
- Create "makefile" for assignment4.
- Put proper comments on the program where you incorporate or modify the code. Please use standard comment style.
- Submit all the source file (serial, parallel, pipelined) and the makefile.
- Input File: input.txt contains the input matrix for the graph. It needs to be large enough to split, so that multiple processes/processors have sufficient number of elements to work with concurrently.
- The generated Output File: output.txt, contains the shortest path among all the nodes of the graph
- The report contain:
  1. Speed-up plot and explanation in two different configuration (i) own machine, (ii) cluster.

## • Submission Deadline

- Programming assignment: 05.11.2023
- Marks distribution will be announced later

# Thank You

### 12.4.1 Floyd's All-Pairs Shortest-Paths Algorithm

Consider a weighted graph  $G$ , which consists of a set of nodes  $V$  and a set of edges  $E$ . An edge from node  $i$  to node  $j$  in  $E$  has a weight  $c_{i,j}$ . Floyd's algorithm determines the cost  $d_{i,j}$  of the shortest path between each pair of nodes  $(i, j)$  in  $V$  ([Section 10.4.2](#)). The cost of a path is the sum of the weights of the edges in the path.

Let  $d_{i,j}^k$  be the minimum cost of a path from node  $i$  to node  $j$ , using only nodes  $v_0, v_1, \dots, v_{k-1}$ . The functional equation of the DP formulation for this problem is

Equation 12.6

$$d_{i,j}^k = \begin{cases} c_{i,j} & k = 0 \\ \min \{d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})\} & 0 \leq k \leq n-1 \end{cases}$$

Since  $d_{i,j}^n$  is the shortest path from node  $i$  to node  $j$  using all  $n$  nodes, it is also the cost of the overall shortest path between nodes  $i$  and  $j$ . The sequential formulation of this algorithm requires  $n$  iterations, and each iteration requires time  $\Theta(n^2)$ . Thus, the overall run time of the sequential algorithm is  $\Theta(n^3)$ .

[Equation 12.6](#) is a serial polyadic formulation. Nodes  $d_{i,j}^k$  can be partitioned into  $n$  levels, one for each value of  $k$ . Elements at level  $k+1$  depend only on elements at level  $k$ . Hence, the formulation is serial. The formulation is polyadic since one of the solutions to  $d_{i,j}^k$  requires a composition of solutions to two subproblems  $d_{i,k}^{k-1}$  and  $d_{k,j}^{k-1}$  from the previous level. Furthermore, the dependencies between levels are sparse because the computation of each element in  $d_{i,j}^{k+1}$  requires only three results from the preceding level (out of  $n^2$ ).

A simple CREW PRAM formulation of this algorithm uses  $n^2$  processing elements. Processing elements are organized into a logical two-dimensional array in which processing element  $P_{i,j}$  computes the value of  $d_{i,j}^k$  for  $k = 1, 2, \dots, n$ . In each iteration  $k$ , processing element  $P_{i,j}$  requires the values  $d_{i,j}^{k-1}$ ,  $d_{i,k}^{k-1}$ , and  $d_{k,j}^{k-1}$ . Given these values, it computes the value of  $d_{i,j}^k$  in constant time. Therefore, the PRAM formulation has a parallel run time of  $\Theta(n)$ . This

Figure 1:

formulation is cost-optimal because its processor-time product is the same as the sequential run time of  $\Theta(n^3)$ . This algorithm can be adapted to various practical architectures to yield efficient parallel formulations ([Section 10.4.2](#)).

As with serial monadic formulations, data locality is of prime importance in serial polyadic formulations since many such formulations have sparse connectivity between levels.

Figure 2:

### 10.4.2 Floyd's Algorithm

Floyd's algorithm for solving the all-pairs shortest paths problem is based on the following observation. Let  $G = (V, E, w)$  be the weighted graph, and let  $V = \{v_1, v_2, \dots, v_n\}$  be the

vertices of  $G$ . Consider a subset  $\{v_1, v_2, \dots, v_k\}$  of vertices for some  $k$  where  $k \leq n$ . 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  $\{v_1, v_2, \dots, v_k\}$ . Let  $p_{i,j}^{(k)}$  be the minimum-weight path among them, and let  $d_{i,j}^{(k)}$  be

the weight of  $p_{i,j}^{(k)}$ . 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  $p_{i,j}^{(k-1)}$ . However, if  $v_k$  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  $\{v_1, v_2, \dots, v_{k-1}\}$ . Thus,

$d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$ . These observations are expressed in the following recurrence equation:

Equation 10.5

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

Figure 3:

The length of the shortest path from  $v_i$  to  $v_j$  is given by  $d_{i,j}^{(n)}$ . In general, the solution is a matrix  $D^{(n)} = (d_{i,j}^{(n)})$ .

Floyd's algorithm solves [Equation 10.5](#) bottom-up in the order of increasing values of  $k$ . [Algorithm 10.3](#) shows Floyd's all-pairs algorithm. The run time of Floyd's algorithm is determined by the triple-nested for loops in lines 4–7. Each execution of line 7 takes time  $\Theta(1)$ ; thus, the complexity of the algorithm is  $\Theta(n^3)$ . [Algorithm 10.3](#) seems to imply that we must store  $n$  matrices of size  $n \times n$ . However, when computing matrix  $D^{(k)}$ , only matrix  $D^{(k-1)}$  is needed. Consequently, at most two  $n \times n$  matrices must be stored. Therefore, the overall space complexity is  $\Theta(n^2)$ . Furthermore, the algorithm works correctly even when only one copy of  $D$  is used (Problem 10.6).

**Algorithm 10.3** Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph  $G = (V, E)$  with adjacency matrix  $A$ .

```

1.  procedure FLOYD_ALL_PAIRS_SP (  $A$  )
2.    begin
3.       $D^{(0)} = A$ ;
4.      for  $k := 1$  to  $n$  do
5.        for  $i := 1$  to  $n$  do
6.          for  $j := 1$  to  $n$  do
               $d_{i,j}^{(k)} := \min (d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$ ;
7.
8.    end FLOYD_ALL_PAIRS_SP

```

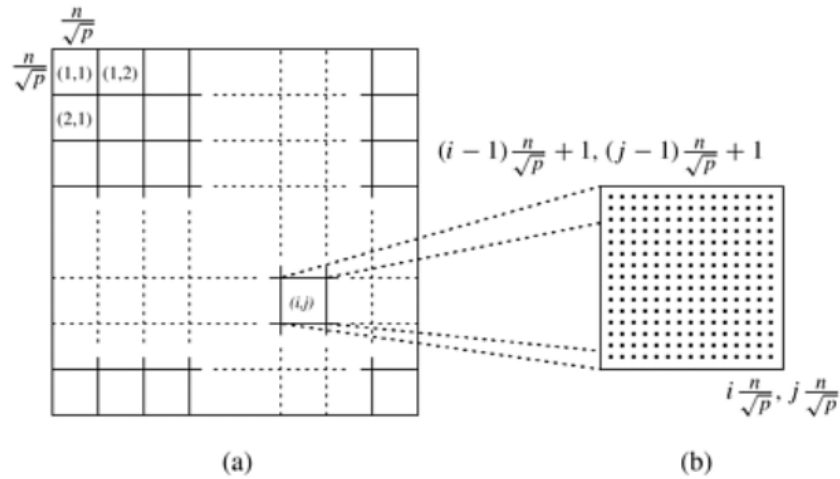
#### Parallel Formulation

A generic parallel formulation of Floyd's algorithm assigns the task of computing matrix  $D^{(k)}$  for each value of  $k$  to a set of processes. Let  $p$  be the number of processes available. Matrix  $D^{(k)}$  is partitioned into  $p$  parts, and each part is assigned to a process. Each process computes the  $D^{(k)}$  values of its partition. To accomplish this, a process must access the corresponding segments of the  $k^{\text{th}}$  row and column of matrix  $D^{(k-1)}$ . The following section describes one technique for partitioning matrix  $D^{(k)}$ . Another technique is considered in Problem 10.8.

**2-D Block Mapping** One way to partition matrix  $D^{(k)}$  is to use the 2-D block mapping ([Section 3.4.1](#)). Specifically, matrix  $D^{(k)}$  is divided into  $p$  blocks of size  $(n/\sqrt{p}) \times (n/\sqrt{p})$ , and each block is assigned to one of the  $p$  processes. It is helpful to think of the  $p$  processes as arranged in a logical grid of size  $\sqrt{p} \times \sqrt{p}$ . Note that this is only a conceptual layout and does not necessarily reflect the actual interconnection network. We refer to the process on the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column as  $P_{i,j}$ . Process  $P_{i,j}$  is assigned a subblock of  $D^{(k)}$  whose upper-left corner is  $((i-1)n/\sqrt{p} + 1, (j-1)n/\sqrt{p} + 1)$  and whose lower-right corner is  $(in/\sqrt{p}, jn/\sqrt{p})$ . Each process updates its part of the matrix during each iteration. [Figure 10.7\(a\)](#) illustrates the 2-D block mapping technique.

Figure 4:

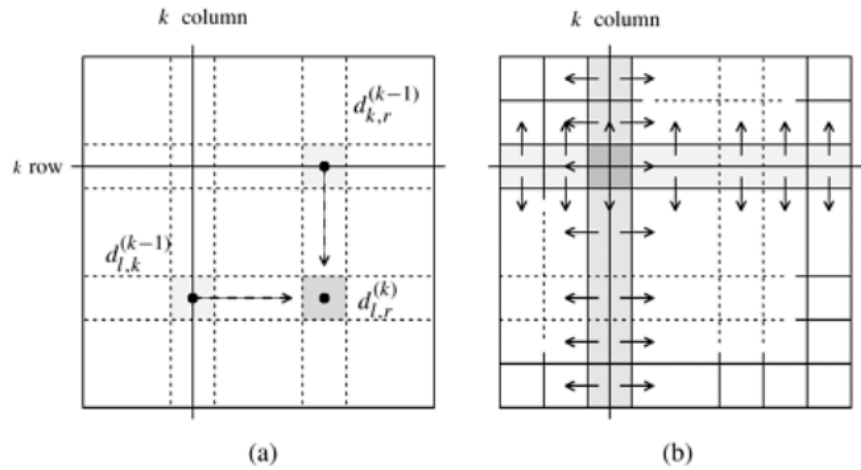
Figure 10.7. (a) Matrix  $D^{(k)}$  distributed by 2-D block mapping into  $\sqrt{p} \times \sqrt{p}$  subblocks, and (b) the subblock of  $D^{(k)}$  assigned to process  $P_{i,j}$ .



During the  $k^{\text{th}}$  iteration of the algorithm, each process  $P_{i,j}$  needs certain segments of the  $k^{\text{th}}$  row and  $k^{\text{th}}$  column of the  $D^{(k-1)}$  matrix. For example, to compute  $d_{l,r}^{(k)}$  it must get  $d_{l,k}^{(k-1)}$  and  $d_{k,r}^{(k-1)}$ . As Figure 10.8 illustrates,  $d_{l,k}^{(k-1)}$  resides on a process along the same row, and element  $d_{k,r}^{(k-1)}$  resides on a process along the same column as  $P_{i,j}$ . Segments are transferred as follows. During the  $k^{\text{th}}$  iteration of the algorithm, each of the  $\sqrt{p}$  processes containing part of the  $k^{\text{th}}$  row sends it to the  $\sqrt{p} - 1$  processes in the same column. Similarly, each of the  $\sqrt{p}$  processes containing part of the  $k^{\text{th}}$  column sends it to the  $\sqrt{p} - 1$  processes in the same row.

Figure 10.8. (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  $k^{\text{th}}$  row and column send them along process columns and rows.

Figure 5:



Algorithm 10.4 shows the parallel formulation of Floyd's algorithm using the 2-D block mapping. We analyze the performance of this algorithm on a  $p$ -process message-passing computer with a cross-bisection bandwidth of  $\Theta(p)$ . During each iteration of the algorithm, the  $k^{\text{th}}$  row and  $k^{\text{th}}$  column of processes perform a one-to-all broadcast along a row or a column of  $\sqrt{p}$  processes. Each such process has  $n/\sqrt{p}$  elements of the  $k^{\text{th}}$  row or column, so it sends  $n/\sqrt{p}$  elements. This broadcast requires time  $\Theta((n \log p)/\sqrt{p})$ . The synchronization step on line 7 requires time  $\Theta(\log p)$ . Since each process is assigned  $n^2/p$  elements of the  $D^{(k)}$  matrix, the time to compute corresponding  $D^{(k)}$  values is  $\Theta(n^2/p)$ . Therefore, the parallel run time of the 2-D block mapping formulation of Floyd's algorithm is

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

Since the sequential run time is  $W = \Theta(n^3)$ , the speedup and efficiency are as follows:

Equation 10.6

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta((n^2 \log p)/\sqrt{p})}$$

$$E = \frac{1}{1 + \Theta((\sqrt{p} \log p)/n)}$$

Figure 6:

From [Equation 10.6](#) we see that for a cost-optimal formulation  $(\sqrt{p} \log p)/n = O(1)$ ; thus, 2-D block mapping can efficiently use up to  $O(n^2/\log^2 n)$  processes. [Equation 10.6](#) can also be used to derive the isoefficiency function due to communication, which is  $\Theta(p^{1.5} \log^3 p)$ . The isoefficiency function due to concurrency is  $\Theta(p^{1.5})$ . Thus, the overall isoefficiency function is  $\Theta(p^{1.5} \log^3 p)$ .

**Speeding Things Up** In the 2-D block mapping formulation of Floyd's algorithm, a synchronization step ensures that all processes have the appropriate segments of matrix  $D^{(k-1)}$  before computing elements of matrix  $D^{(k)}$  (line 7 in [Algorithm 10.4](#)). In other words, the  $k^{\text{th}}$  iteration starts only when the  $(k-1)^{\text{th}}$  iteration has completed and the relevant parts of matrix  $D^{(k-1)}$  have been transmitted to all processes. The synchronization step can be removed without affecting the correctness of the algorithm. To accomplish this, a process starts working on the  $k^{\text{th}}$  iteration as soon as it has computed the  $(k-1)^{\text{th}}$  iteration and has the relevant parts of the  $D^{(k-1)}$  matrix. This formulation is called *pipelined 2-D block mapping*. A similar technique is used in [Section 8.3](#) to improve the performance of Gaussian elimination.

**Algorithm 10.4** Floyd's parallel formulation using the 2-D block mapping.  $P_{*,j}$  denotes all the processes in the  $j^{\text{th}}$  column, and  $P_{i,*}$  denotes all the processes in the  $i^{\text{th}}$  row. The matrix  $D^{(0)}$  is the adjacency matrix.

```

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

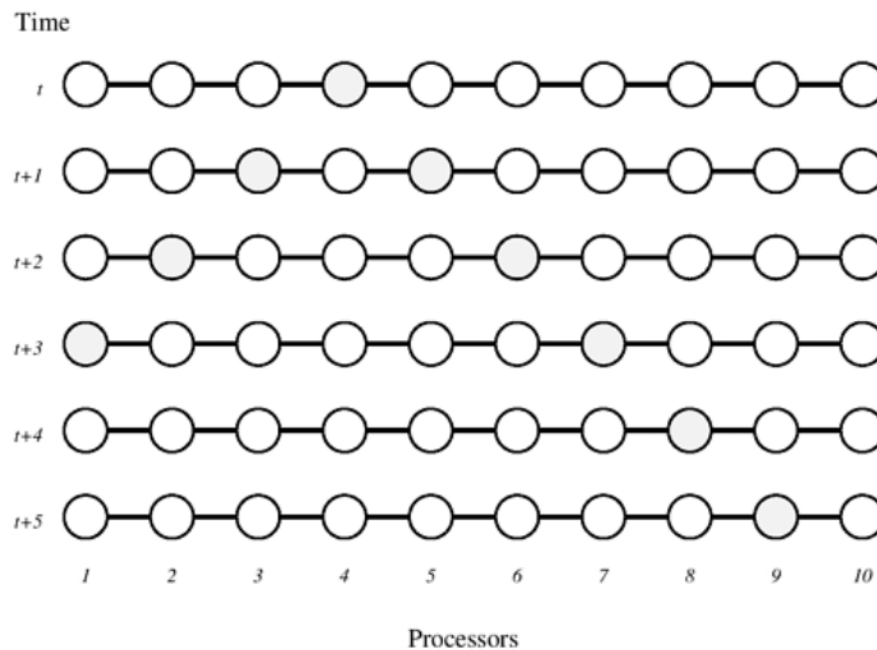
```

Consider a  $p$ -process system arranged in a two-dimensional topology. Assume that process  $P_{i,j}$  starts working on the  $k^{\text{th}}$  iteration as soon as it has finished the  $(k-1)^{\text{th}}$  iteration and has received the relevant parts of the  $D^{(k-1)}$  matrix. When process  $P_{i,j}$  has elements of the  $k^{\text{th}}$  row and has finished the  $(k-1)^{\text{th}}$  iteration, it sends the part of matrix  $D^{(k-1)}$  stored locally to processes  $P_{i,j-1}$  and  $P_{i,j+1}$ . It does this because that part of the  $D^{(k-1)}$  matrix is used to compute the  $D^{(k)}$  matrix. Similarly, when process  $P_{i,j}$  has elements of the  $k^{\text{th}}$  column and has finished the  $(k-1)^{\text{th}}$  iteration, it sends the part of matrix  $D^{(k-1)}$  stored locally to processes  $P_{i-1,j}$  and  $P_{i+1,j}$ . When process  $P_{i,j}$  receives elements of matrix  $D^{(k)}$  from a process along its row in the logical mesh, it stores them locally and forwards them to the process on the side opposite from where it received them. The columns follow a similar communication protocol. Elements of matrix  $D^{(k)}$  are not forwarded when they reach a mesh boundary. [Figure 10.9](#) illustrates this communication and termination protocol for processes within a row (or a column).

Figure 7:



Figure 10.9. 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  $k^{\text{th}}$  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.



Consider the movement of values in the first iteration. 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. Hence, 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  $(n-1)^{\text{th}}$  iteration, it sends the relevant values of the  $n^{\text{th}}$  row and column to the other processes. These values reach process  $P_{1,1}$  in time  $\Theta(n)$ . The overall parallel run time of this formulation is

Figure 8:

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

Since the sequential run time is  $W = \Theta(n^3)$ , the speedup and efficiency are as follows:

Equation 10.7

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta(n)}$$

$$E = \frac{1}{1 + \Theta(p/n^2)}$$

Table 10.1. The performance and scalability of the all-pairs shortest paths algorithms on various architectures with  $O(p)$  bisection bandwidth. Similar run times apply to all  $k$ - $d$  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	$\Theta(n^2)$	$\Theta(n)$	$\Theta(p^{1.5})$

From [Equation 10.7](#) we see that for a cost-optimal formulation  $p/n^2 = O(1)$ . Thus, the pipelined formulation of Floyd's algorithm uses up to  $O(n^2)$  processes efficiently. Also from [Equation 10.7](#), we can derive the isoefficiency function due to communication, which is  $\Theta(p^{1.5})$ . This is the overall isoefficiency function as well. Comparing the pipelined formulation to the

Figure 9: