FLOYD's Algorithm

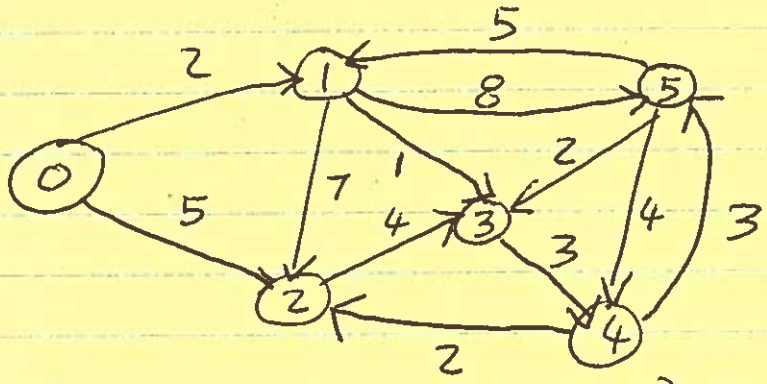
tables showing driving distances btw pairs of cities.

intersection of city A, city B (rows, cols) contains the length of the shortest path of roads btw A, B.
(route likely passes thru other cities)

Floyd's algorithm is used to generate such tables

all-pairs shortest-path problem

graph → V finite set of vertices
graph → E finite set of edges btw pairs of vertices



(numerical value associated with each edge)

weighted directed graph (edges are oriented)

(length of the shortest path btw every pair of vertices)

all-pairs shortest-path pb:
given a weighted directed graph, find the

((determined by weights of edges not #edges)) <-7

(ex) length of shortest path btw vertex 0 and vertex 5 is eq to 9

0 → 1, 1 → 3, 3 → 4, 4 → 5

[2], [1], [3], [3]

data structure to represent a weighted directed graph: adjacency matrix

advantages:

(1) constant time access to all edges

(2) does not require more memory than for storing the solution

nxn matrix for a n-vertices graph
(i,j) element is the weight of the edge i → j

for non existent edges → we use ∞ (extremely high value)

	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0

(IN-PLACE)

upon termination (of the alg), the matrix contains the lengths of the shortest paths btw every pair of vertices

Floyd's Algorithm $\Theta(n^3)$ time complexity

INPUT n, #vertices
$a[n,n]$ adjacency matrix | numbering $0,\dots,n-1$

OUTPUT transformed matrix a, containing the lengths of the shortest paths

```
for k in 0,...,n-1
    for i in 0,...,n-1
        for j in 0,...,n-1
            a[i,j] = min(a[i,j], a[i,k]+a[k,j])
        end for
    end for
end for
```

OUTPUT

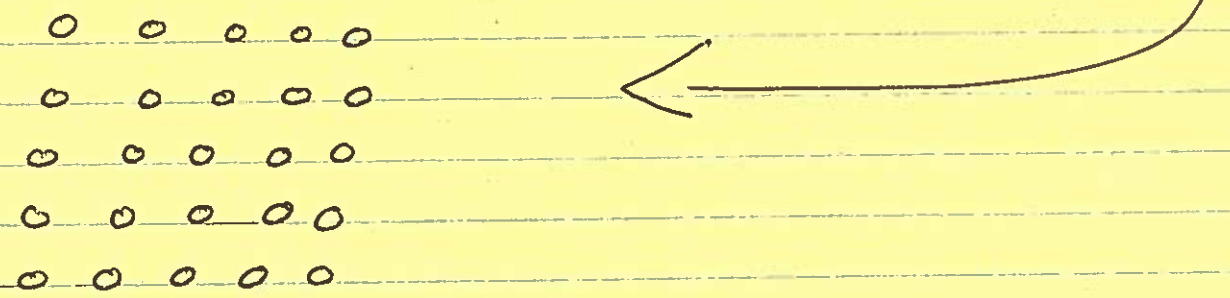
	0	1	2	3	4	5
0	0	2	5	3	6	9
1	∞	0	6	1	4	7
2	∞	15	0	4	7	10
3	∞	11	5	0	3	6
4	∞	8	2	5	0	3
5	∞	5	6	2	4	0

Design of parallel Floyd's alg.

PARTITIONING

the alg. executes the same assignment statement n^3 times

opportunity for parallelism:
decompose the domain of the computation, i.e. divide matrix A in n^2 elements
associate a primitive task with each element:

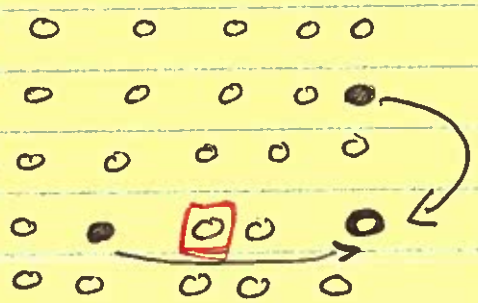


COMMUNICATION

each update of element $a[i,j]$ requires access to elements $a[i,k]$ and $a[k,j]$ for example:

$k=1$, $a[3,4]$ needs $a[3,1]$ & $a[1,4]$

$k=2$, $a[3,4]$ needs $a[3,2]$ & $a[3,4]$



for each value of k:

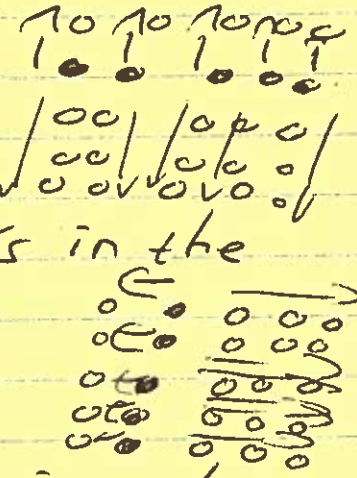
(1) element a[k m] is needed by every task associated with elements in column m.

(2) element a[m k] is needed by every task' associated with elements in row m.

~>

CONSEQ

~> during iteration k:



(1) each elm in row k of a gets broadcast to the tasks in the same column

(2) each elm in col k of a gets broadcast to the tasks in the same row

---

Q can every elm of a be updated simultaneously?
(updating a[i,j] requires a[i,k] & a[k,j])

A  YES because the values of a[i,k] & a[k,j] do not change during iteration k.

this is because: (during iter k:)

$$a[i,k] = \min(a[i,k],\ a[i,k] + a[k,k])$$

($a[k,k]$: positive)

$$a[k,j] = \min(a[k,j],\ a[k,k] + a[k,j])$$

($a[k,k]$: positive)

so none of them can decrease.

there is no dependency b/w the update of $a[i,j]$ and the updates of $a[i,k]$ & $a[k,j]$

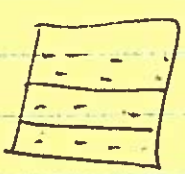
~> for each iteration k of the outer loop, we can perform the broadcasts and then update every elm of a in parallel.

TASK // AGGLOMERATION

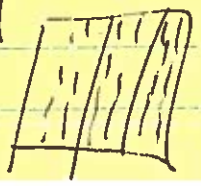
to minimize communication we can agglomerate the n^2 primitive tasks into p tasks, for p procs.

two natural agglomerations:
group tasks in same row or col.

results in
two data decompositions



(1) row-wise block-striped decomp.



(2) column-wise block-striped decomp.

→ see next page

if we agglomerate tasks in the same row, the broadcast that occurs among primitive tasks in the same row is eliminated.

if we agglomerate tasks in the same column, the broadcast that occurs among prim. tasks in the same col. is eliminated.

choice b+w row/col agglom.

we choose row-wise agglom. because it makes it simpler to work with the row-major order of matrix storage in C. in primary memory
↳ 1st row then 2nd row ...

(1) each proc is responsible for a group of contiguous rows

(2) each proc is responsible for a group of contiguous cols

```
/*
 *   Floyd's all-pairs shortest path
 *
 *   Given an NxN matrix of distances between pairs of
 *   vertices, this MPI program computes the shortest path
 *   between every pair of vertices.
 *
 *   This program shows:
 *      how to dynamically allocate multidimensional arrays
 *      how one process can handle I/O for the others
 *      broadcasting of a vector of elements
 *      messages with different tags
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 4 September 2002
 */

#include <stdio.h>
#include <mpi.h>
#include "../MyMPI.h"

typedef int dtype;
#define MPI_TYPE MPI_INT

int main (int argc, char *argv[]) {
   dtype** a;           /* Doubly-subscripted array */
   dtype*  storage;     /* Local portion of array elements */
   int     i, j, k;
   int     id;          /* Process rank */
   int     m;           /* Rows in matrix */
   int     n;           /* Columns in matrix */
   int     p;           /* Number of processes */
   double  time, max_time;

   void compute_shortest_paths (int, int, int**, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_row_striped_matrix (argv[1], (void *) &a,
      (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

   if (m != n) terminate (id, "Matrix must be square\n");

/*
   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
*/
   MPI_Barrier (MPI_COMM_WORLD);
   time = -MPI_Wtime();
   compute_shortest_paths (id, p, (dtype **) a, n);
   time += MPI_Wtime();
   MPI_Reduce (&time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
      MPI_COMM_WORLD);
   if (!id) printf ("Floyd, matrix size %d, %d processes: %6.2f seconds\n",
      n, p, max_time);
/*
```

double → _DOUBLE

indicate the type of matrix we are manipulating

e.g. change int → double

MACRO

// reading original matrix

//check distance that matrix. matrix is square.

```
   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
*/
   MPI_Finalize();
}

void compute_shortest_paths (int id, int p, dtype **a, int n)
{
   int  i, j, k;
   int  offset;   /* Local index of broadcast row */
   int  root;     /* Process controlling row to be bcast */
   int* tmp;      /* Holds the broadcast row */

   tmp = (dtype *) malloc (n * sizeof(dtype));
   for (k = 0; k < n; k++) {
      root = BLOCK_OWNER(k,p,n);
      if (root == id) {
         offset = k - BLOCK_LOW(id,p,n);
         for (j = 0; j < n; j++)
            tmp[j] = a[offset][j];
      }
      MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
      for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
         for (j = 0; j < n; j++)
            a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
   }
   free (tmp);
}
```

// prints transformed distance matrix

Floyd's alg.

-> implements

4 parameters

process rank

#procs

sizeof matrix

pointer to the process' portion of the distance matrix

-> n iterations

-> determine which process controls row k

every process allocate an array of n integers, called tmp, to store row k

this process is the root of the BCAST tree

During each iteration k of the algorithm, row k must be made available to every process.

-> after the call to MPI_Bcast, each process has a copy of row k in its array tmp.