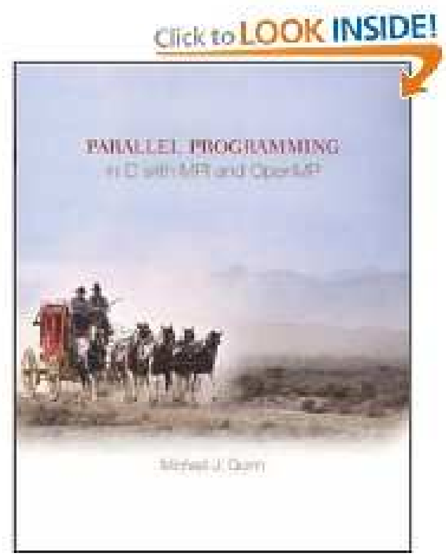
Floyd's algorithm

Overview

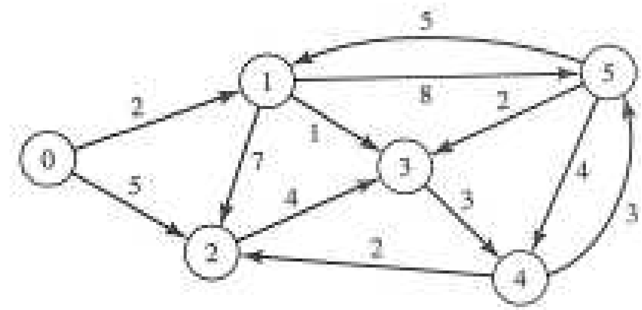
- Chapter 6 from *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**



  - Floyd's algorithm: solving *the all-pairs shortest-path problem*

Finding shortest paths

- Starting point: a graph of vertices and weighted edges



- Each edge is of a direction and has a length
  - if there's path from vertex i to j, there may not be path from vertex j to i
  - path length from vertex i to j may be different than path length from vertex j to i
- Objective: finding the shortest path between every pair of vertices $(i \to j)$
- Application: table of driving distances between city pairs

Adjacency matrix

- There are n vertices
- The direct path length from vertex i to vertex j is stored as $a[i, j]$
- An $n \times n$ adjacency matrix a keeps the entire connectivity info

	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

- If $a[i, j]$ is ∞, it means there is no direct path from vertex i to vertex j

Example of all-pairs shortest path

For the adjacency matrix given on the previous slide, the solution of the all-pairs shortest path is as follows:

	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

Table of shortest path lengths

Floyd's algorithm

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for $k \leftarrow$ 0 to $n - 1$
    for $i \leftarrow$ 0 to $n - 1$
        for $j \leftarrow$ 0 to $n - 1$
            $a[i, j] \leftarrow \min(a[i, j],\ a[i, k] + a[k, j])$
        endfor
    endfor
endfor

Some observations

- Floyd's algorithm is an exhaustive and incremental approach
- The entries of the a-matrix are updated n rounds
- $a[i, j]$ is compared with all n possibilities,
  that is, against $a[i, k] + a[k, j]$, for $0 \leq k \leq n - 1$
- n^3 of comparisons in total

Source of parallelism

- During the k'th iteration, the work is (in C syntax)

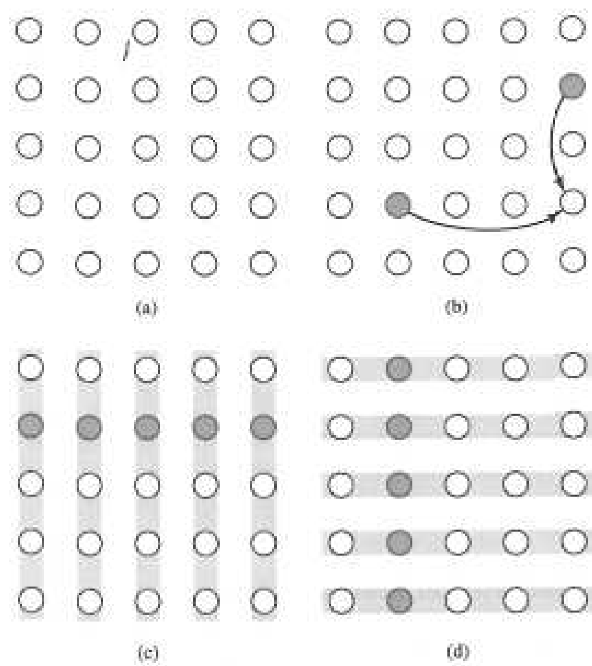
```
for (i=0; i<n; i++)
  for (j=0; j<n, j++)
    a[i][j] = MIN( a[i][j], a[i][k]+a[k][j] );
```

- Can all the entries in a be updated concurrently?
- Yes, because the k'th column and the k'th row remain the same during the k'th iteration!
  - Note that `a[i][k]=MIN(a[i][k],a[i][k]+a[k][j])` will be the same as `a[i][k]`
  - Note that `a[k][j]=MIN(a[k][j],a[k][k]+a[k][j])` will be the same as `a[k][j]`

Design of a parallel algorithm

Using Foster's design methodology:

- Partitioning — each $a[i, j]$ is a primitive task
- Communication — during the k'th iteration, updating $a[i, j]$ needs values of $a[i, k]$ and $a[k, j]$
  - broadcast $a[k, j]$ to $a[0, j], a[1, j], \ldots, a[n-1, j]$
  - broadcast $a[i, k]$ to $a[i, 0], a[i, 1], \ldots, a[i, n-1]$



Agglomeration and mapping

- Let one MPI process be responsible for a piece of the a matrix
- Memory storage of a is accordingly divided
- The division can in principle be arbitrary, as long as the number of all $a[i, j]$ entries is divided evenly
- However, a row-wise block data division is very convenient
  - 2D arrays in C are row-major
  - easy to send/receive an entire row of a
- We therefore choose to assign one MPI process with a number of consecutive rows of a

Communication pattern

- Recall that in the k'th iteration:

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

- Since entries of a are divided into rowwise blocks, so $a[i,k]$ is also in the memory of the MPI process that owns $a[i,j]$
- However, $a[k,j]$ is probably in another MPI process's memory
- Communication is therefore needed!
  - Before the k'th iteration, the MPI process that owns the k'th row of the a matrix should broadcast this row to everyone else

Recap: creating 2D arrays in C

To create a 2D array with m rows and n columns:

```
int **B, *Bstorage, i;
...
Bstorage=(int*)malloc(m*n*sizeof(int));
B=(int**)malloc(m*sizeof(int*));
for (i=0; i<m; i++)
  B[i] = &Bstorage[i*n];
```

The underlying storage is contiguous, making it possible to send and receive an entire 2D array.

Global index vs. local index

- Suppose a matrix (2D array) is divided into row-wise blocks and distributed among p MPI processes
- Process i only allocates storage for its assigned row block
  - from row $\lfloor (i \cdot n)/p \rfloor$ of matrix a until row $\lfloor ((i+1) \cdot n)/p \rfloor - 1$
- We need to know: Which global row does a local row correspond to?
- Mapping: local index $\rightarrow$ global index
- On process number `proc_id`
  `global_index=BLOCK_LOW(proc_id, p, n)+local_index`

Main work of parallel Floyd's algorithm

```
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);
}
```

Matrix input

- Recall that each MPI process only stores a part of the a matrix
- When reading a from a file, we can
  - let only process $p-1$ do the input
  - once the number of rows needed by process i are read in, they are sent from process $p-1$ to process i using `MPI_Send`
  - process i must issue a matching `MPI_Recv`
- The above simple strategy is not parallel
- Parallel I/O can be done using MPI-2 commands

Matrix output

- For example, we let only process 0 do the output
- Each process needs to send its part of a to process 0
- To avoid many processes sending its entire subdata to process 0 at the same time
  - Process 0 communicates with the other processes in turn
  - Each process waits for a “hint” (a short message) from process 0 before sending its data (a large message)

Deadlock

- Typical deadlock example 1

```
if (rank==0) {
  MPI_Recv(&b,1,MPI_INT,1,tag_b,MPI_COMM_WORLD,&status)
  MPI_Send(&a,1,MPI_INT,1,tag_a,MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&a,1,MPI_INT,0,tag_a,MPI_COMM_WORLD,&status)
  MPI_Send(&b,1,MPI_INT,0,tag_b,MPI_COMM_WORLD);
}
```

- Typical deadlock example 2

```
if (rank==0) {
  MPI_Send(&a,1,MPI_INT,1,1,MPI_COMM_WORLD);
  MPI_Recv(&b,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
} else if (rank==1) {
  MPI_Send(&b,1,MPI_INT,0,0,MPI_COMM_WORLD);
  MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
}
```

Analysis

- Serial algorithm time usage: $n^3\chi$
- Parallel algorithm
  - non-communication time usage: $n^2\lceil n/p\rceil\chi$
  - communication (broadcast) time usage: $n\lceil\log_2 p\rceil(\lambda + 4n/\beta)$
    - assuming each entry of matrix a needs 4 bytes
    - assuming λ as communication latency
    - assuming β as communication bandwidth (# bytes per second)
- Read Section 6.7 for a more detailed analysis that allows overlap between computation and communication

Exercises

- Write an MPI program that uses p processes to produce a JPEG picture of $n \times n$ pixels. The picture should have white background and a black circle (of radius $n/4$) in the middle. (The existing C code collection
  `http://heim.ifi.uio.no/xingca/inf-verk3830/simple-jpeg.tar.gz`
  can be used.)
- Implement the complete Floyd's algorithm and try it on a large enough adjacency matrix.