Floyd's Algorithm

A method to find the shortest distance between two points when \textit{multiple} paths are possible.

Can be represented as a directed graph which must be traversed in a particular direction.

The weights of the edges represent the “distance” between the vertices.
The graph can be represented as a numerical adjacency matrix.

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>9</td>
<td>5</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Resulting Adjacency Matrix Containing Distances
Advantages to using a matrix

• The adjacency matrix $a[i,j]$ initially holds the lengths between each pair of vertices

• Computationally, there is constant time access to each element

• When the analysis is complete, the shortest path distance can be stored in the matrix – this keeps memory usage the same
The sequential algorithm

n is the number of vertices

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

$n^3$ algorithm
Pictorial representation of algorithm

Shortest path from $i$ to $k$ through 0, 1, ..., $k-1$

Computed in previous iterations

Shortest path from $i$ to $j$ through 0, 1, ..., $k-1$

Shortest path from $k$ to $j$ through 0, 1, ..., $k-1$
Designing the parallel algorithm

Domain or functional decomposition?

• Look at pseudocode
• Same assignment statement executed $n^3$ times
• No functional parallelism

Domain decomposition: divide matrix $a$ into its $n^2$ elements

• Each $a[i,j]$ represents a task – need to find a shortest distance.
• However, to find the distance need to look at $a[i,k]$ and $a[k,j]$ for all $k$
Communication

- Every task in column \( m \) needs the value of \( a[k,m] \)
- Every task in row \( m \) needs the value of \( a[m,k] \)
- Can this value be broadcast?

Let \( k \) control the outer loop of the algorithm:

\[
\begin{align*}
    a[i,k] &= \min(a[i,k], a[i,k]+a[k,k]) \leftarrow 0 \\
    a[k,j] &= \min(a[k,j], a[k,k]+a[k,j]) \\
    a[k,j] &\uparrow \\
    &\quad 0
\end{align*}
\]

A broadcast is possible for specific values of \( k \)
Communication (cont.)

Primitive tasks

Iteration $k$: every task in row $k$ broadcasts its value w/in task column

Updating $a[3,4]$ when $k = 1$

Iteration $k$: every task in column $k$ broadcasts its value w/in task row
Agglomeration and Mapping

Number of tasks: static, depends on value of n
Communication among tasks: structured
Computation time per task: constant

Strategy:
• Agglomerate tasks to minimize communication
• Create one combined task per MPI process
Consider two data decompositions

Rowwise block striped

Columnwise block stripe

(a) (b)
Comparing the two decompositions

Columnwise block striped
  Broadcast within columns eliminated
Rowwise block striped
  Broadcast within rows eliminated
  Reading matrix from file simpler because elements in C/C++ matrices stored in row major order
Choose rowwise block striped decomposition
How to input a large adjacency matrix

Assume that the matrix is stored row by row in a file.

1) Each process reads its own row (or rows) of initial data. The process must seek the correct location in the shared file.

2) A master process reads all the rows and sends the data to the appropriate process.

Method 2) minimizes memory usage because only one process needs to read and send the data. Eliminates the seek in method 1).
File Input – reading row by row
How are rows distributed?

\[ n = \text{size of the matrix}, \quad p = \text{number of processes} \]

Let process \( i \) have rows:
\[ \text{floor}(i \times n/p) \text{ to } \text{floor}((i+1) \times n/p)-1 \]

If \( i = p-1 \), number of rows = \( \text{ceil}(n/p) \)

\( \text{ceil}(n/p) \) is the maximum number of rows any process will have, so
- process \( p-1 \) can read the rows and distribute them.
- It reads its own rows last.
Point-to-point Communication is required to send and receive the elements of a

- Involves a pair of processes
- One process sends a message
- Other process receives the message
Function MPI_Send() – a blocking MPI function

```c
int MPI_Send (  
    void *message //memory location,  
    int count //# of items to send,  
    MPI_Datatype  datatype,  
    int dest //rank of receiving process,  
    int tag //message ID,  
    MPI_Comm comm
)
```
Return from MPI_Send()

- Function blocks until message buffer free
- Message buffer is free when
  - Message copied to system buffer, or
  - Message transmitted
- Typical scenario
  - Message copied to system buffer
  - Transmission overlaps computation
Function MPI_Recv() - returns when the expected message is available in the local buffer

```c
int MPI_Recv (  
    void *message // data stored here,  
    int count // maximum amount of memory,  
    MPI_Datatype datatype,  
    int source // ranking of sender,  
    int tag // message ID,  
    MPI_Comm comm,  
    MPI_Status *status // was it successful  
)
```
Return from MPI_Recv()

- Function blocks until message in buffer
- If message never arrives, function never returns
- MPI_Status is a structure guaranteed to have a field MPI_ERROR
- If the message size is larger than allocated memory, an overflow error occurs
- If the message size is less than count, it is stored at the beginning of the allocated memory.
Relationship of Send/Receive in the code

...  
if (ID == j) {
    ...
    Receive from i
    ...
}
...

...  
if (ID == i) {
    ...
    Send to j
    ...
}
...

Receive is before Send. Why does this work?
Inside MPI_Send() and MPI_Recv()
Deadlock is possible in MPI

Deadlock: process waiting for a condition that will never become true

Easy to write send/receive code that deadlocks
- Two processes: both receive before send
- Send tag doesn’t match receive tag
- Process sends message to wrong destination process
Dynamic 1-D Array Creation
Using malloc() is straightforward

Run-time Stack

A

Heap
Dynamic 2-D Array Creation – matrices are stored in row major order

\[ B(n,m) \] – B is a pointer to a pointer
Programming 2-D array allocation

```c
int **B, *Bstorage;
Bstorage = (int*)malloc(m*n*sizeof(int));
B = (int**)malloc(m*sizeof(int*));

for (i = 0; i < m; i++)
    B[i] = &Bstorage[i*n]
```
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);
}
Computation/communication overlap

Key:
- Compute
- Set up message
- Wait
Computational Complexity

- Innermost loop has complexity $\Theta(n)$
  
  \[ \text{for } (j = 0; j < n; j++) \]

- Middle loop executed at most $\text{ceil}(n/p)$ times
  
  \[ \text{for } (i = 0; i < \text{BLOCK\_SIZE}(id, p, n); i++) \]

- Outer loop executed $n$ times
  
  \[ \text{for } (k = 0; k < n; k++) \]

- Overall complexity $\Theta(n^3/p)$
Communication complexity

- No communication in inner loop
- No communication in middle loop
- Broadcast in outer loop — complexity is $\Theta(n \log p)$
- Overall complexity $\Theta(n^2 \log p)$
Summary

- Two matrix decompositions
  - Rowwise block striped
  - Columnwise block striped
- Blocking send/receive functions
  - MPI_Send()
  - MPI_Recv()
- Overlapping communications with computations