All-Pairs Shortest Paths - Floyd’s Algorithm

Parallel and Distributed Computing

Department of Computer Science and Engineering (DEI)
Instituto Superior Técnico

November 2, 2010
All-Pairs Shortest Paths, Floyd’s Algorithm

- Partitioning
- Input / Output
- Implementation and Analysis
- Benchmarking
All Pairs Shortest Paths

Given a weighted, directed graph $G(V, E)$, determine the shortest path between any two nodes in the graph.
All Pairs Shortest Paths

Given a weighted, directed graph $G(V, E)$, determine the shortest path between any two nodes in the graph.
All Pairs Shortest Paths

Given a weighted, directed graph \( G(V, E) \), determine the shortest path between any two nodes in the graph.

\[
\begin{bmatrix}
0 & -2 & -5 & 4 \\
\infty & 0 & 9 & \infty \\
7 & \infty & 0 & -3 \\
8 & 0 & 6 & 0
\end{bmatrix}
\]

Adjacency Matrix
The Floyd-Warshall Algorithm

Recursive solution based on *intermediate* vertices.

Let $p_{ij}$ be the minimum-weight path from node $i$ to node $j$ among paths that use a subset of intermediate vertices $\{0, \ldots, k - 1\}$. 
The Floyd-Warshall Algorithm

Recursive solution based on *intermediate* vertices.

Let $p_{ij}$ be the minimum-weight path from node $i$ to node $j$ among paths that use a subset of intermediate vertices $\{0, \ldots, k - 1\}$.

Consider an additional node $k$:

$k \not\in p_{ij}$

then $p_{ij}$ is shortest path considering the subset of intermediate vertices $\{0, \ldots, k\}$.

$k \in p_{ij}$

then we can decompose $p_{ij}$ as $i \xrightarrow{p_{ik}} k \xrightarrow{p_{kj}} j$, where subpaths $p_{ik}$ and $p_{kj}$ have intermediate vertices in the set $\{0, \ldots, k - 1\}$.
The Floyd-Warshall Algorithm

Recursive solution based on *intermediate* vertices.

Let $p_{ij}$ be the minimum-weight path from node $i$ to node $j$ among paths that use a subset of intermediate vertices $\{0, \ldots, k-1\}$.

Consider an additional node $k$:

$k \notin p_{ij}$

then $p_{ij}$ is shortest path considering the subset of intermediate vertices $\{0, \ldots, k\}$.

$k \in p_{ij}$

then we can decompose $p_{ij}$ as $i \xrightarrow{p_{ik}} k \xrightarrow{p_{kj}} j$, where subpaths $p_{ik}$ and $p_{kj}$ have intermediate vertices in the set $\{0, \ldots, k-1\}$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = -1 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 0 \end{cases}$$
The Floyd-Warshall Algorithm

1. for \( k \leftarrow 0 \) to \( |V| - 1 \)
2. for \( i \leftarrow 0 \) to \( |V| - 1 \)
3. for \( j \leftarrow 0 \) to \( |V| - 1 \)
4. \[ d[i, j] \leftarrow \min(d[i, j], d[i, k] + d[k, j]) \]
The Floyd-Warshall Algorithm

1. for $k \leftarrow 0$ to $|V| - 1$
2. for $i \leftarrow 0$ to $|V| - 1$
3. for $j \leftarrow 0$ to $|V| - 1$
4. $d[i, j] \leftarrow \min(d[i, j], d[i, k] + d[k, j])$

Complexity?
The Floyd-Warshall Algorithm

1. for k ← 0 to |V| − 1
2. for i ← 0 to |V| − 1
3. for j ← 0 to |V| − 1
4. \[ d[i, j] \leftarrow \min(d[i, j], d[i, k] + d[k, j]) \]

Complexity: \( \Theta(|V|^3) \)
Partitioning:

Domain decomposition: divide adjacency matrix into its $V^2$ elements (computation in the inner loop is primitive task).
Partitioning:

Domain decomposition: divide adjacency matrix into its $|V|^2$ elements (computation in the inner loop is primitive task).
Communication:

Let $k = 1$. Row sweep, $i = 2$. 
Communication:

Let $k = 1$. Row sweep, $i = 2$. 

\[ \begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  & & & \\
  & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array} \]
Communication:

Let $k = 1$. Row sweep, $i = 2$. 
Let $k = 1$. Row sweep, $i = 2$. 

Communication:
Communication:

Let $k = 1$. Row sweep, $i = 2$. 
Communication:

Let $k = 1$. Column sweep, $j = 3$. 
Communication:

Let $k = 1$. Column sweep, $j = 3$. 
Communication:

Let $k = 1$. Column sweep, $j = 3$. 
Communication:

Let $k = 1$. Column sweep, $j = 3$. 

\[
\begin{array}{cccc}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{array}
\]
Communication:

In iteration $k$, every task in row/column $k$ broadcasts its value within task row/column.
Agglomeration and Mapping

Agglomeration and Mapping:
Agglomeration and Mapping:

- create one task per MPI process
- agglomerate tasks to minimize communication
Agglomeration and Mapping:

- create one task per MPI process
- agglomerate tasks to minimize communication

Possible decompositions: row-wise vs column-wise block striped ($n = 11, p = 3$).

Relative merit?
Agglomeration and Mapping

Agglomeration and Mapping:

- create one task per MPI process
- agglomerate tasks to minimize communication

Possible decompositions: row-wise vs column-wise block striped \((n = 11, p = 3)\).

Relative merit?

- Column-wise block striped
  - Broadcast within columns eliminated
- Row-wise block striped
  - Broadcast within rows eliminated
  - Reading, writing and printing matrix simpler
Comparing Decompositions

Choose row-wise block striped decomposition.

What’s the best way to distribute $n$ rows over $p$ tasks?

Some tasks get $\lceil \frac{n}{p} \rceil$ rows, other get $\lfloor \frac{n}{p} \rfloor$. Which task gets which size?

Distributed approach: distribute larger blocks evenly.

First element of task $i$: $\lfloor \frac{i}{n} \rfloor$

Last element of task $i$: $\lfloor \left( \frac{i+1}{p} \right) \rfloor - 1$

Task owner of element $j$: $\lfloor \frac{p(\frac{j+1}{n}) - 1}{n} \rfloor$
Comparing Decompositions

Choose row-wise block striped decomposition.

What’s the best way to distribute $n$ rows over $p$ tasks?

Some tasks get $\left\lfloor \frac{n}{p} \right\rfloor$ rows, other get $\left\lceil \frac{n}{p} \right\rceil$.

Which task gets which size?
Comparing Decompositions

Choose row-wise block striped decomposition.

What’s the best way to distribute $n$ rows over $p$ tasks?

Some tasks get $\left\lceil \frac{n}{p} \right\rceil$ rows, other get $\left\lfloor \frac{n}{p} \right\rfloor$.

Which task gets which size?

**Distributed approach:** distribute larger blocks evenly.

First element of task $i$: $\left\lfloor \frac{i \cdot n}{p} \right\rfloor$

Last element of task $i$: $\left\lfloor (i + 1) \frac{n}{p} \right\rfloor - 1$

Task owner of element $j$: $\left\lfloor \frac{(p(j + 1) - 1)}{n} \right\rfloor$
Dynamic Matrix Allocation

Array allocation:

Matrix allocation:
Dynamic Matrix Allocation

Array allocation:

Matrix allocation:
Dynamic Matrix Allocation

Array allocation:

Matrix allocation:
Why don't we read the whole file and then execute a MPI Scatter?
Why don't we read the whole file and then execute a MPI Scatter?
Why don't we read the whole file and then execute a MPI Scatter?
Why don't we read the whole file and then execute a MPI Scatter?
Why don't we read the whole file and then execute a MPI Scatter?
Why don’t we read the whole file and then execute a MPI Scatter?
Why don’t we read the whole file and then execute a MPI Scatter?
Why don’t we read the whole file and then execute a MPI_Scatter?
Point-to-point Communication

- involves a pair of processes
  - one process sends a message
  - other process receives the message

Diagram:

- Task h: Compute
- Task i: Compute, Send to j
- Task j: Wait, Receive from i

Time line:

José Monteiro (DEI / IST)
MPI_Send

int MPI_Send (  
    void *message,  
    int count,  
    MPI_Datatype datatype,  
    int dest,  
    int tag,  
    MPI_Comm comm
)

int MPI_Recv (  
    void *message,  
    int count,  
    MPI_Datatype datatype,  
    int source,  
    int tag,  
    MPI_Comm comm,  
    MPI_Status *status  
)
... if (id == j) {
    ...
    Receive from i
    ...
}
...

... if (id == i) {
    ...
    Send to j
    ...
}
...
Receive is before Send! Why does this work?
Internals of Send and Receive

Sending Process:
- Program Memory
- System Buffer

Receiving Process:
- System Buffer
- Program Memory

MPI_Send

MPI_Recv
Internals of Send and Receive

Sending Process

MPI_Send

Receiving Process

MPI_Recv

José Monteiro (DEI / IST)
Internals of Send and Receive

Sending Process
- Program Memory
- System Buffer

Receiving Process
- System Buffer
- Program Memory

MPI_Send
Internals of Send and Receive

Sending Process
- Program Memory
- System Buffer

Receiving Process
- Program Memory
- System Buffer

MPI_Send
MPI_Recv
Return from MPI_Send

- function blocks until message buffer free
Return from MPI_Send

- function blocks until message buffer free

- message buffer is free when
  - message copied to system buffer, or
  - message transmitted
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
Return from MPIRecv

- function blocks until message in buffer
function blocks until message in buffer

if message never arrives, function never returns!
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
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
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
```c
void compute_shortest_paths (int id, int p, double **a, int n)
{
    int  i, j, k;
    int  offset; /* Local index of broadcast row */
    int  root;  /* Process controlling row to be bcast */
    double* tmp; /* Holds the broadcast row */

    tmp = (double *) malloc (n * sizeof(double));
    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_DOUBLE, 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);
}
```
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time?
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time: $\alpha n^3$
Computation time of parallel program?
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time: $\alpha n^3$

Computation time of parallel program: $\alpha n \left\lceil \frac{n}{p} \right\rceil n$
- innermost loop executed $n$ times
- middle loop executed at most $\left\lceil \frac{n}{p} \right\rceil$ times
- outer loop executed $n$ times

Number of broadcasts?
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time: $\alpha n^3$

Computation time of parallel program: $\alpha n \left\lceil \frac{n}{p} \right\rceil n$

- innermost loop executed $n$ times
- middle loop executed at most $\left\lceil \frac{n}{p} \right\rceil$ times
- outer loop executed $n$ times

Number of broadcasts: $n$

- one per outer loop iteration

Broadcast time?
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time: $\alpha n^3$

Computation time of parallel program: $\alpha n \left\lceil \frac{n}{p} \right\rceil n$

- innermost loop executed $n$ times
- middle loop executed at most $\left\lceil \frac{n}{p} \right\rceil$ times
- outer loop executed $n$ times

Number of broadcasts: $n$
  - one per outer loop iteration

Broadcast time: $\left\lceil \log p \right\rceil \left( \lambda + \frac{4n}{\beta} \right)$
  - each broadcast has $\left\lceil \log p \right\rceil$ steps
  - $\lambda$ is the message latency
  - $\beta$ is the bandwidth
  - each broadcast sends $4n$ bytes
Analysis of the Parallel Algorithm

Let $\alpha$ be the time to compute an iteration.
Sequential execution time: $\alpha n^3$

Computation time of parallel program: $\alpha n \left\lceil \frac{n}{p} \right\rceil n$

- innermost loop executed $n$ times
- middle loop executed at most $\left\lceil \frac{n}{p} \right\rceil$ times
- outer loop executed $n$ times

Number of broadcasts: $n$
- one per outer loop iteration

Broadcast time: $\left\lceil \log p \right\rceil \left( \lambda + \frac{4n}{\beta} \right)$
- each broadcast has $\left\lceil \log p \right\rceil$ steps
- $\lambda$ is the message latency
- $\beta$ is the bandwidth
- each broadcast sends $4n$ bytes

Expected parallel execution time:

$$\alpha n^2 \left\lceil \frac{n}{p} \right\rceil + n \left\lceil \log p \right\rceil \left( \lambda + \frac{4n}{\beta} \right)$$
Previous expression will overestimate parallel execution time: after the first iteration, broadcast transmission time overlaps with computation of next row.

Expected parallel execution time:

$$\alpha n^2 \left\lceil \frac{n}{p} \right\rceil + n \lceil \log p \rceil \lambda + \lceil \log p \rceil \frac{4n}{\beta}$$

Experimental measurements:

$$\alpha = 25, 5 \text{ ns}$$
$$\lambda = 250 \text{ } \mu\text{s}$$
$$\beta = 10^7 \text{ bytes/s}$$
## Experimental Results

<table>
<thead>
<tr>
<th>Procs</th>
<th>Ideal</th>
<th>Predict 1</th>
<th>Predict 2</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.5</td>
<td>25.5</td>
<td>25.5</td>
<td>25.5</td>
</tr>
<tr>
<td>2</td>
<td>12.8</td>
<td>13.4</td>
<td>13.0</td>
<td>13.9</td>
</tr>
<tr>
<td>3</td>
<td>8.5</td>
<td>9.5</td>
<td>8.9</td>
<td>9.6</td>
</tr>
<tr>
<td>4</td>
<td>6.4</td>
<td>7.7</td>
<td>6.9</td>
<td>7.3</td>
</tr>
<tr>
<td>5</td>
<td>5.1</td>
<td>6.6</td>
<td>5.7</td>
<td>6.0</td>
</tr>
<tr>
<td>6</td>
<td>4.3</td>
<td>5.9</td>
<td>4.9</td>
<td>5.2</td>
</tr>
<tr>
<td>7</td>
<td>3.6</td>
<td>5.5</td>
<td>4.3</td>
<td>4.5</td>
</tr>
<tr>
<td>8</td>
<td>3.2</td>
<td>5.1</td>
<td>3.9</td>
<td>4.0</td>
</tr>
</tbody>
</table>
All-Pairs Shortest Paths, Floyd’s Algorithm

- Partitioning
- Input / Output
- Implementation and Analysis
- Benchmarking
Next Class

Performance metrics