

Floyd's Algorithm

Introduction

- Used to find shortest paths in a weighted graph
- Travel maps containing driving distance from one point to another
 - Represented by tables
 - Shortest distance from point A to point B given by intersection of row and column
 - Route may pass through other cities represented in the table
- Navigation systems

All-pairs shortest-path problem

- Graph $G = (V, E)$
- Weighted and directed graph
- Problem: Find the length of the shortest path between every pair of vertices
 - Length of the path is strictly determined by the weight of its edges
 - It is not based on the number of edges traversed
- Representation of weighted directed graph by adjacency matrix
 - $n \times n$ matrix for a graph with n vertices – adjacency matrix
 - * Chosen for constant time access to every edge
 - Nonexistent edges may be assigned a 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

- Algorithm

```
Input n: Number of vertices
      a[0..n-1][0..n-1] -- Adjacency matrix
```

```
Output: Transformed a that contains shortest path lengths
```

```
for ( k = 0; k < n; k++ )
  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] );
```

- Easy to see that the algorithm is $\Theta(n^3)$

- Solution

	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

Creating arrays at run time

- Allocate arrays on heap at run-time to make the program more useful, such as

```
int * arr;
arr = ( int * ) malloc ( n * sizeof ( int ) );
```

- 2D arrays in C are represented by a 1D array of pointers, using malloc
- Problem in transmission as memory should be contiguous
- Allocate them by first allocating space, and then, putting pointers in place

```
int ** arr;                // The main array
int * storage;             // Contiguous storage for array
storage = ( int * ) malloc ( m * n * sizeof ( int ) );
arr = ( int ** ) malloc ( m * sizeof ( int * ) );
for ( i = 0; i < m; i++ )
    arr[i] = storage + ( i * n );
```

- Initialize the array elements either by using `a[i][j]` notation, or by using `storage` if initialized *en masse*

Designing parallel algorithm

- Partitioning
 - Choose either domain decomposition or functional decomposition
 - Same assignment statement executed n^3 times
 - * No functional parallelism
 - Easy to do domain decomposition
 - * Divide matrix A into 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 a given value of k
 - * Element `a[k, m]` is needed by every task associated with elements in column m
 - * Element `a[m, k]` is needed by every task associated with elements in row m
 - In iteration k , each element in row k gets broadcast to the tasks in the same column
 - Each element in column k gets broadcast to tasks in the same row
 - Do we need to update every element of matrix concurrently?
 - * Values of `a[i][k]` and `a[k][j]` do not change during iteration k

- * Update to $a[i][k]$ is

$$a[i][k] = \min (a[i,k], a[i,k] + a[k,k]);$$
- * Update to $a[k][j]$ is

$$a[k][j] = \min (a[k,j], a[k,k] + a[k,j]);$$
- * In both the above updates, $a[i][k]$ and $a[k][j]$ cannot decrease (all numbers are positive)
- * Hence, no dependence between updates of $a[i][j]$ and the updates of $a[i][k]$ and $a[k][j]$
- * For each iteration k of the outer loop, perform broadcasts and update every element of matrix in parallel

Agglomeration and mapping

- Use the decision tree to determine agglomeration and mapping strategy
 - Number of tasks: static
 - Communication among tasks: structured
 - Computation time per task: constant
- Agglomerate tasks to minimize communication
 - One task per MPI process
- Agglomerate n^2 primitive tasks into p tasks using either of the following two methods
 - Row-wise block striped
 - * Agglomerate tasks in the same row
 - * Broadcast within rows eliminated; data values local to task
 - * During every iteration of outer loop, one task broadcasts n elements to all other tasks
 - * Time for each broadcast: $\lceil \log p \rceil (\lambda + n/\beta)$
 - Column-wise block striped
 - * Agglomerate tasks within same column
 - * Broadcast within columns eliminated
 - * Time for each broadcast: $\lceil \log p \rceil (\lambda + n/\beta)$
 - Both the above methods result in same performance, look outside the computational kernel
 - Final choice: Simpler to read matrix from file with row-wise block striped if the file stores data as row-major order
 - * Internally, C stores the matrices as row major as well

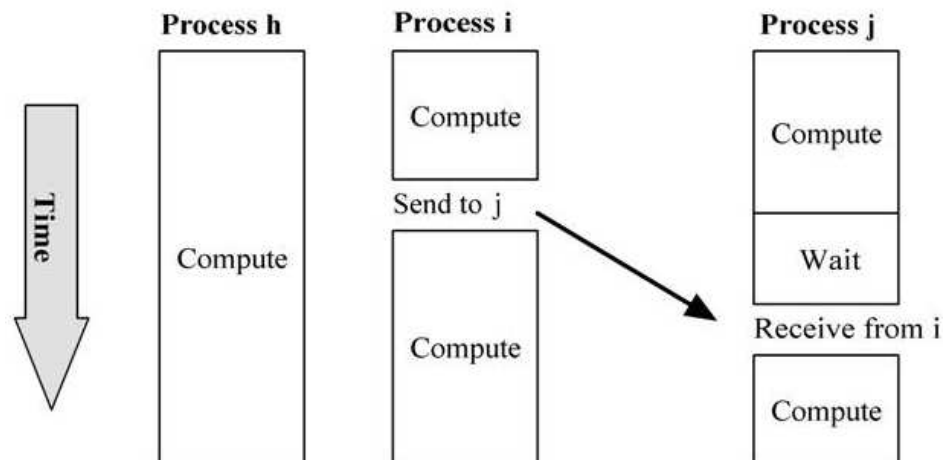
Matrix I/O

- Choice to let each process open the file, access the appropriate row (seek to it), and read the row
- Another method to read the matrix rows in last process and send them to appropriate process
 - Process i responsible for rows $\lfloor in/p \rfloor$ through $\lfloor (i+1)n/p \rfloor - 1$
 - Last process responsible for $\lceil n/p \rceil$ rows (largest buffer)
 - * No extra space needed for file input buffering
 - Last process reads the rows for each process in a loop and sends the data to appropriate task
- All the printing done by process 0, by getting data from other processes
 - Print all the output in correct order
 - Processes $1, 2, \dots, p-1$ simply wait for a message from process 0, then send process 0 their portion of the matrix

* Process 0 never receives more than one submatrix at a time

Point-to-point communication

- Function to read matrix from file
 - Executed by process $p - 1$
 - Reads a contiguous group of matrix rows
 - Sends a message containing these rows directly to the process responsible to manage them
- Function to print the matrix
 - Each process sends the group of matrix rows to process 0
 - Process 0 receives the message and prints the rows to standard output
- Communication involves a pair of processes
 - One process sends a message
 - Other process receives the message



- Process h is not involved in communication
- Both *Send* and *receive* are blocking
- Both send and receive need to be conditionally executed by process rank

```

...
if ( id == i )
{
    ...
    /* Send message to process j */
    ...
}
else if ( id == j )
{
    ...
    /* Receive message from process i */
    ...
}

```

- Communication calls must be in conditionally executed code

- Function MPI_Send

- Perform a blocking send

```
int MPI_Send ( void * buffer, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm );
```

buffer Starting address of the array of data items to send

count Number of data items in array (nonnegative integer)

datatype Data type of each item (uniform since it is an array); defined by an MPI constant

dest Rank of destination (integer)

tag Message tag, or integer label; allows identification of message purpose

comm Communicator; group of processes participating in this communication function

- Function blocks until the message buffer is again available
- Message buffer is free when
 - * Message copied to system buffer, or
 - * Message transmitted (may overlap computation)

- Function MPI_Recv

- Blocking receive for a message

```
int MPI_Recv ( void * buffer, int count, MPI_Datatype datatype, int src,
               int tag, MPI_Comm comm, MPI_Status * status );
```

buffer Starting address of receive buffer

count Maximum number of data items in receive buffer

datatype Data type of each item (uniform since it is an array); defined by an MPI constant

src Rank of source (integer)

- * Can be specified as MPI_ANY_SOURCE to receive the message from any source in the communicator
- * Process rank in this case can be determined through status

tag Desired tag value (integer)

- * Can be specified as MPI_ANY_TAG
- * Received tag can be determined through status

comm Communicator; group of processes participating in this communication function

status Status objects; must be allocated before call to MPI_Recv

- Blocks until the message has been received, or until an error condition causes the function to return
- count contains the maximum length of message
 - * Actual length of received message can be determined with MPI_Get_count
- status contains information about the just-completed function
 - status->MPI_source** Rank of the process sending message
 - status->MPI_tag** Message's tag value
 - status->MPI_ERROR** Error condition
- Function blocks until message arrives in buffer
- If message never arrives, function never returns

- Deadlock

- Process blocked on a condition that will never become true
- Easy to write send/receive code that deadlocks
 - * Two processes with rank 0 and 1, each wanting to compute the average of two numbers in an array a[2]

- Process i has the updated value of $a[i]$
- * Both receive before send


```
float      a[2], avg;
int        me;                // Process rank
MPI_Status status;
...
int other = 1 - me;
MPI_Recv ( a + other, 1, MPI_FLOAT, other, 0, MPI_COMM_WORLD, &status );
MPI_Send ( a + me, 1, MPI_FLOAT, other, 0, MPI_COMM_WORLD );
avg = ( a[0] + a[1] ) / 2.0;
```

 - Process 0 blocks inside MPI_Recv waiting for message from process 1 to arrive
 - Process 1 blocks inside MPI_Recv waiting for message from process 0 to arrive
 - Both are deadlocked
- * Send tag does not match receive tag


```
float      a[2], avg;
int        me;                // Process rank
MPI_Status status;
...
int other = 1 - me;
MPI_Isend ( a + me, 1, MPI_FLOAT, other, me, MPI_COMM_WORLD, &request );
MPI_Recv ( a + other, 1, MPI_FLOAT, other, me, MPI_COMM_WORLD, &status );
avg = ( a[0] + a[1] ) / 2.0;
```

 - Processes block in MPI_Recv because the tag is not correct
- * Other problems could be wrong destination in send or wrong source in receive

Documenting the parallel program

- Housekeeping, starting in `floyd.c`
 - Use a typedef and a #define macro to indicate matrix data types
- `floyd.c:main()` responsible to read and print original matrix
 - It verifies that the matrix is square
 - All work done by `compute_shortest_paths` with four parameters

Analysis and benchmarking

- Sequential version performance: $\Theta(n^3)$
- Analysis of parallel algorithm
 - Innermost loop has complexity $\Theta(n)$
 - Middle loop executed at most $\lceil n/p \rceil$ times due to rowwise block-striped decomposition of matrix
 - Overall computational complexity: $\Theta(n^2/p)$
- Communication complexity
 - No communication in inner loop
 - No communication in middle loop
 - Broadcast in outer loop

- * Passing a single message of length n from one PE to another has time complexity $\Theta(n)$
- * Broadcasting to p PEs requires $\lceil \log p \rceil$ message-passing steps
- * Complexity of broadcasting: $\Theta(n \log p)$

- Outermost loop

- For every iteration of outermost loop, parallel algorithm must compute the root PE taking constant time
- Root PE copies the correct row of A to array `tmp`, taking $\Theta(n)$ time
- The loop itself executes n times

- Overall time complexity

$$\Theta(n(1 + n + n \log p + n^2/p)) = \Theta(n^3/p + n^2 \log p)$$

- Prediction for the execution time on commodity cluster

- n broadcasts, with $\lceil \log p \rceil$ steps each
- Each step passes messages of $4n$ bytes
- Expected communication time of parallel program: $n \lceil \log p \rceil (\lambda + 4n/\beta)$
- Average time to update a single cell: χ
- Expected computation time for parallel program: $n^2 \lceil n/p \rceil \chi$
- Execution time

$$n^2 \lceil n/p \rceil \chi + n \lceil \log p \rceil (\lambda + 4n/\beta)$$

- This expression overestimates the parallel execution time because it ignores the fact that there can be considerable overlap between computation and communication