Floyd's Algorithm

Introduction

- Travel maps from one point to another
- Navigation systems

All-pairs shortest-path problem

- Graph $G = (V, E)$
- Weighted graph
- Directed graph
- Problem: Find the length of the shortest path between every pair of vertices
- Representation of weighted directed graph by adjacency matrix
  - $n \times n$ matrix for a graph with n vertices
  - 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

```
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 $\Theta(n^3)$

Creating arrays at run time

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

```
int ** a;                        // The main array
int  * storage;                  // Contiguous storage for array
storage = ( int * ) malloc ( m * n * sizeof ( int ) );
a = ( int ** ) malloc ( m * sizeof ( int * ) );
for ( i = 0; i < m; i++ )
    a[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
  - Domain or functional decomposition
  - Same assignment statement executed n^3 times
  - No functional parallelism
  - Domain decomposition
    * Divide matrix A into n^2 elements
    * Associate a primitive task with each element
- Communication
  - 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] );
      ```
  - For each iteration k of the outer loop, perform broadcasts and update every element of matrix in parallel

Agglomeration and mapping

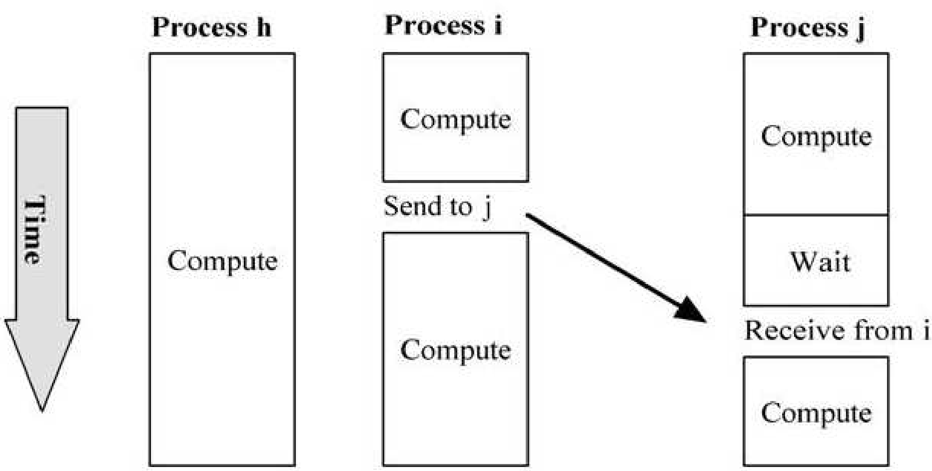
- Number of tasks: static
- Communication among tasks: structured
- Computation time per task: constant
- Agglomerate tasks to minimize communication
  - One task per MPI process
- Agglomerate n^2 primitive tasks into p tasks
  - Row-wise block striped
    * Broadcast within rows eliminated
    * During 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
    * Broadcast within columns eliminated
    * Time for each broadcast: $\lceil \log p \rceil (\lambda + n/\beta)$
  - Simpler to read matrix from file with row-wise block striped

Matrix I/O

- 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)
- All the printing done by process 0, by getting data from other processes

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



  - Both *Send* and *receive* are blocking
  - Both send and receive need to be conditionally executed by process rank
- 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

statis->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
    * Both receive before send

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

* Send tag does not match receive tag

```
float      a, b, c;
int        id;             // Process rank
MPI_Status status;
...
switch ( id )
{
case 0:
    MPI_Send ( &a, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD );
    MPI_Recv ( &b, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &status );
    break;

case 1:
    MPI_Send ( &b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD );
    MPI_Recv ( &a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status );
}
c = ( a + b ) / 2.0;
```

    * Process sends message to wrong destination

Documenting the parallel program

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
  - Outer loop executed n times
  - Overall complexity: $\Theta(n^3/p)$
- Communication complexity
  - No communication in inner loop
  - No communication in middle loop
  - Broadcast in outer loop
    * Complexity: $\Theta(n \log p)$
  - Overall complexity: $\Theta(n^2 \log p)$
- Overall time complexity
$$\Theta(n^3/p + n^2 \log p)$$
- 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
$$n^2\lceil n/p \rceil\chi + n\lceil \log p \rceil(\lambda + 4n/\beta)$$