All-Pairs Shortest Paths - Floyd's Algorithm

Parallel and Distributed Computing

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

November 6, 2012

Outline

- All-Pairs Shortest Paths, Floyd's Algorithm
    - Partitioning
    - Input / Output
    - Implementation and Analysis
    - Benchmarking

Shortest Paths

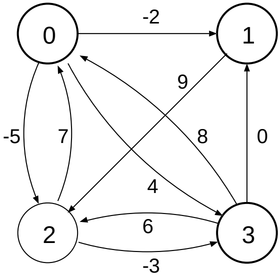
All Pairs Shortest Paths

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

Shortest Paths

All Pairs Shortest Paths

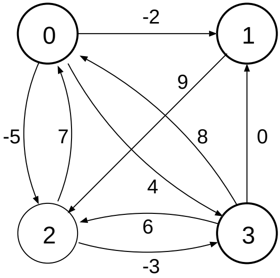
Given a weighted, directed graph $G(V, E)$, determine the shortest path between any two nodes in the graph.



Shortest Paths

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:

$k \notin p_{ij}$

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 $i \overset{p_{ik}}{\rightsquigarrow} k \overset{p_{kj}}{\rightsquigarrow} j$, 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 $i \overset{p_{ik}}{\leadsto} k \overset{p_{kj}}{\leadsto} j$, 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 $k \leftarrow 0$ 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: $\Theta(|V|^3)$

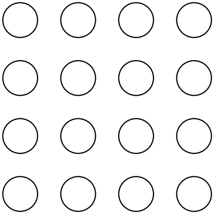
Partitioning

Partitioning:

Partitioning

Partitioning:

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



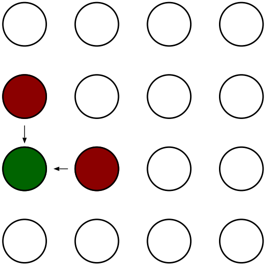
Communication

Communication:

Communication

Communication:

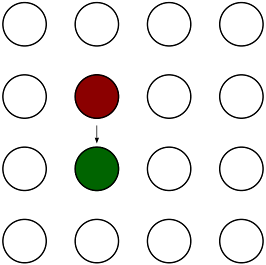
Let $k = 1$. Row sweep, $i = 2$.



Communication

Communication:

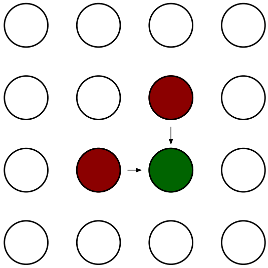
Let $k = 1$. Row sweep, $i = 2$.



Communication

Communication:

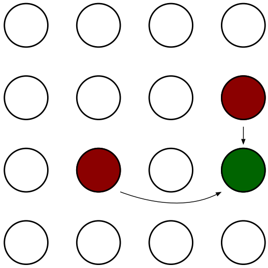
Let $k = 1$. Row sweep, $i = 2$.



Communication

Communication:

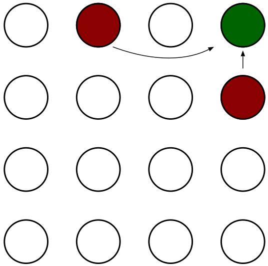
Let $k = 1$. Row sweep, $i = 2$.



Communication

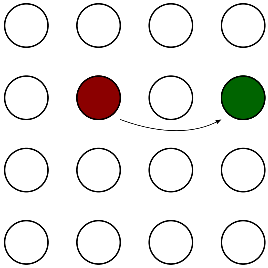
Communication:

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



Communication:

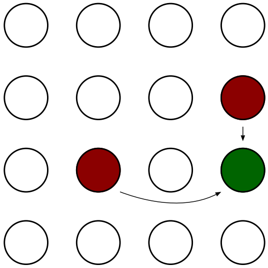
Let $k = 1$. Column sweep, $j = 3$.



Communication

Communication:

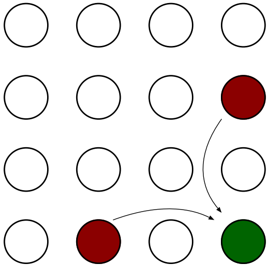
Let $k = 1$. Column sweep, $j = 3$.



Communication

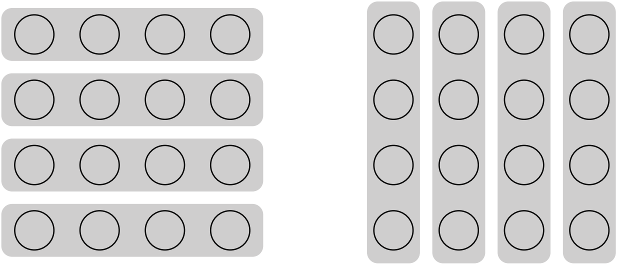
Communication:

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



Communication:

In iteration k, every task in row/column k broadcasts its value within task row/column.



Agglomeration and Mapping

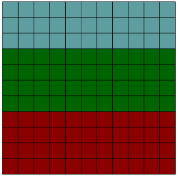
Agglomeration and Mapping:

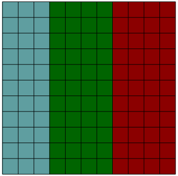
Agglomeration and Mapping

Agglomeration and Mapping:

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

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





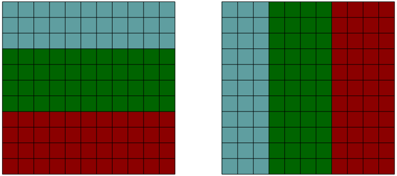
Relative merit?

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.

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?

Comparing Decompositions

Choose row-wise block striped decomposition.

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.

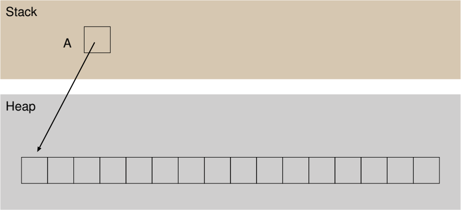
First element of task i: $\left\lfloor i\frac{n}{p} \right\rfloor$

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

Task owner of element j: $\lfloor (p(j+1)-1)/n \rfloor$

Dynamic Matrix Allocation

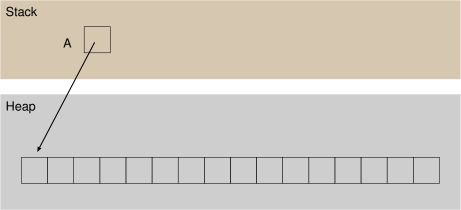
Array allocation:



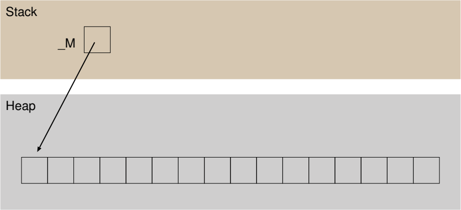
Matrix allocation:

Dynamic Matrix Allocation

Array allocation:

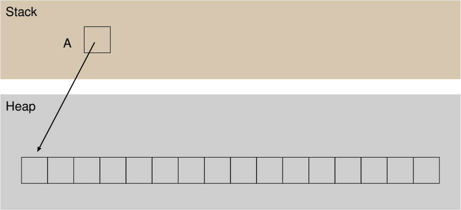


Matrix allocation:

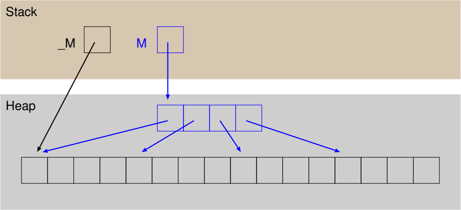


Dynamic Matrix Allocation

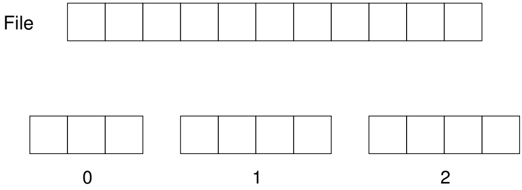
Array allocation:



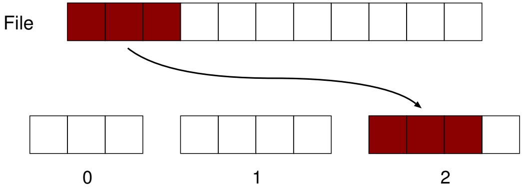
Matrix allocation:



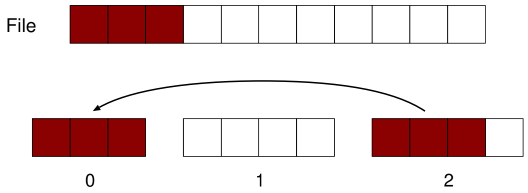
Reading the Graph Matrix



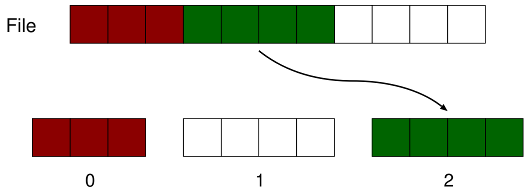
Reading the Graph Matrix



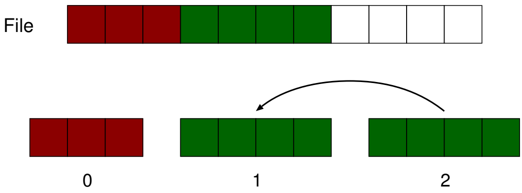
Reading the Graph Matrix



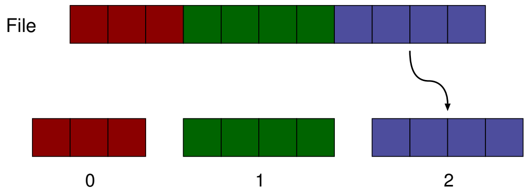
Reading the Graph Matrix



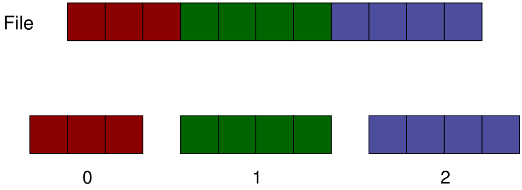
Reading the Graph Matrix



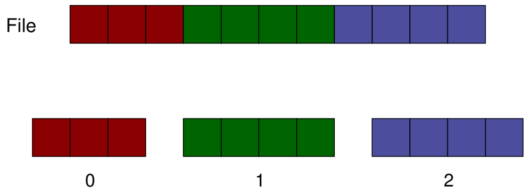
Reading the Graph Matrix



Reading the Graph Matrix



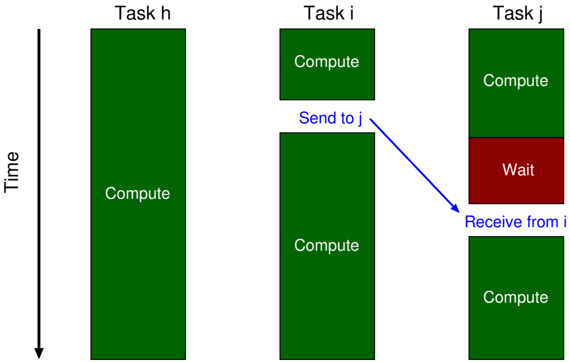
Reading the Graph Matrix



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



MPI_Send

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

MPI_Recv

```
int MPI_Recv (
      void          *message,
      int            count,
      MPI_Datatype   datatype,
      int            source,
      int            tag,
      MPI_Comm       comm,
      MPI_Status    *status
)
```

Coding Send / Receive

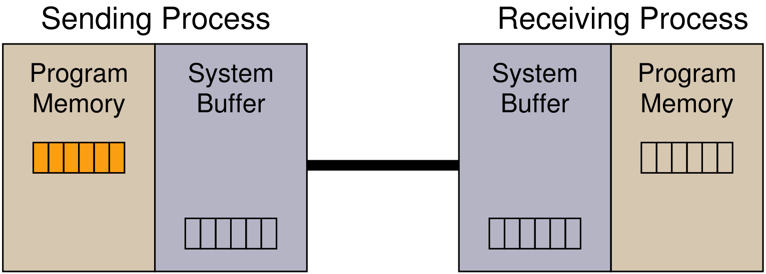
```
...
if (id == j) {
   ...
   Receive from i
   ...
}

...

if (id == i) {
   ...
   Send to j
   ...
}
...
```

Coding Send / Receive

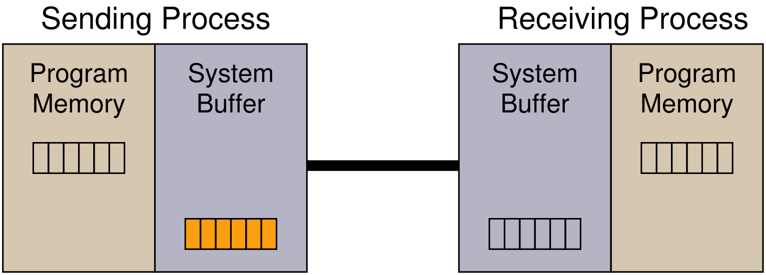
```
...
if (id == j) {
   ...
   Receive from i
   ...
}

...

if (id == i) {
   ...
   Send to j
   ...
}
...
```

Receive is before Send! Why does this work?

Internals of Send and Receive

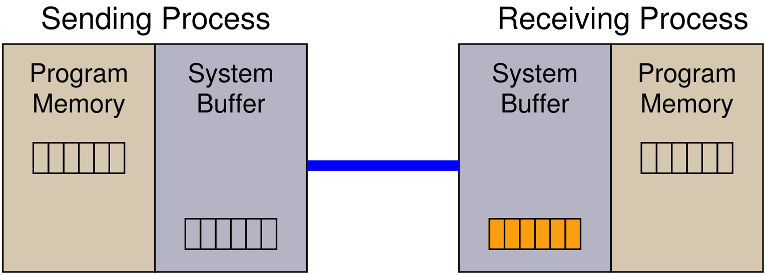


Internals of Send and Receive



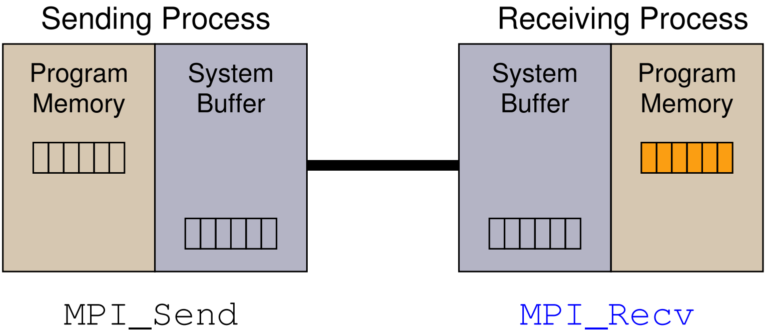
`MPI_Send`

Internals of Send and Receive



`MPI_Send`

Internals of Send and Receive



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 MPI_Recv

- function blocks until message in buffer

Return from MPI_Recv

- function blocks until message in buffer

- if message never arrives, function never returns!

Deadlock

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

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

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 Code

```
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 α be the time to compute an iteration.
Sequential execution time?

Analysis of the Parallel Algorithm

Let α be the time to compute an iteration.
Sequential execution time: αn^3

Computation time of parallel program?

Analysis of the Parallel Algorithm

Let α be the time to compute an iteration.

Sequential execution time: αn^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 α be the time to compute an iteration.
Sequential execution time: αn^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 α be the time to compute an iteration.
Sequential execution time: αn^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: $\lceil \log p \rceil \left(\lambda + \frac{4n}{\beta} \right)$

- each broadcast has $\lceil \log p \rceil$ steps
- λ is the message latency
- β is the bandwidth
- each broadcast sends $4n$ bytes

Analysis of the Parallel Algorithm

Let α be the time to compute an iteration.
Sequential execution time: αn^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: $\lceil \log p \rceil \left(\lambda + \frac{4n}{\beta} \right)$

- each broadcast has $\lceil \log p \rceil$ steps
- λ is the message latency
- β is the bandwidth
- each broadcast sends $4n$ bytes

Expected parallel execution time: $\alpha n^2 \left\lceil \frac{n}{p} \right\rceil + n \lceil \log p \rceil \left(\lambda + \frac{4n}{\beta} \right)$

Analysis of the Parallel Algorithm

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:

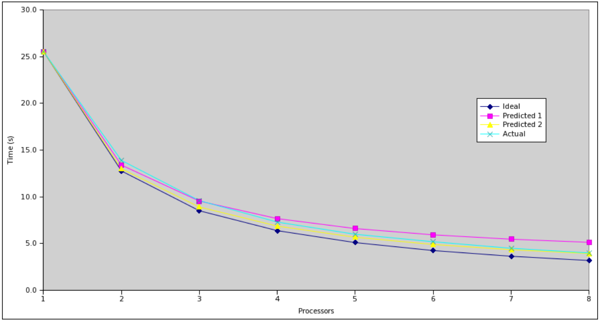
$\alpha = 25, 5$ ns

$\lambda = 250$ μs

$\beta = 10^7$ bytes/s

Experimental Results

Procs	Ideal	Predict 1	Predict 2	Actual
1	25,5	25,5	25,5	25,5
2	12,8	13,4	13,0	13,9
3	8,5	9,5	8,9	9,6
4	6,4	7,7	6,9	7,3
5	5,1	6,6	5,7	6,0
6	4,3	5,9	4,9	5,2
7	3,6	5,5	4,3	4,5
8	3,2	5,1	3,9	4,0



Review

- All-Pairs Shortest Paths, Floyd's Algorithm
  - Partitioning
  - Input / Output
  - Implementation and Analysis
  - Benchmarking

Next Class

- Performance metrics