CIS 631
Parallel Processing

Lecture 10: Parallel Algorithms

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

- Portions of the lectures slides were adopted from:
  - Chapters 8, 9, and 10
Outline

- Dense matrix algorithms
- Sorting algorithms
- Graph algorithms
Dense Matrix Algorithms

- Great deal of activity in algorithms and software for solving linear algebra problems
  - Solution of linear systems ($Ax = b$)
  - Least-squares solution of over- or under-determined systems ($\min ||Ax-b||$)
  - Computation of eigenvalues and eigenvectors ($Ax=\lambda x$)
  - Driven by numerical problem solving in scientific computation

- Solutions involves various forms of matrix computations
- Focus on high-performance matrix algorithms
  - Key insight is to maximize computation to communication
Solving a System of Linear Equations

- $Ax=b$
  
  $a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,n-1}x_{n-1} = b_0$

  $a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,n-1}x_{n-1} = b_1$

  ...

  $A_{n-1,0}x_0 + a_{n-1,1}x_1 + ... + a_{n-1,n-1}x_{n-1} = b_{n-1}$

- Gaussian elimination
  
  - Forward elimination to $Ux=y$ ($U$ is upper triangular)
    - Without or with partial pivoting
  
  - Back substitution to solve for $x$

  - Parallel algorithms based on $A$ partitioning
Sequential Gaussian Elimination

1. procedure GAUSSIAN ELIMINATION (A, b, y)
2. Begin
3. for k := 0 to n - 1 do /* Outer loop */
4. begin
5. for j := k + 1 to n - 1 do
6. \[ A[k, j] := A[k, j]/A[k, k]; /* Division step */ \]
7. \[ y[k] := b[k]/A[k, k]; \]
8. \[ A[k, k] := 1; \]
9. for i := k + 1 to n - 1 do
10. begin
11. for j := k + 1 to n - 1 do
13. \[ b[i] := b[i] - A[i, k] \times y[k]; \]
14. \[ A[i, k] := 0; \]
15. endfor;       /*Line9*/
16. endfor;       /*Line3*/
17. end GAUSSIAN ELIMINATION
**Computation Step in Gaussian Elimination**

\[
\begin{align*}
5x + 3y &= 22 \\
8x + 2y &= 13
\end{align*}
\]

\[
\begin{align*}
x &= (22 - 3y) / 5 \\
x &= (22 - 3y) / 5 \\
y &= (13 - 176/5) / (24/5 + 2)
\end{align*}
\]
# Rowwise Partitioning on Eight Processes

<table>
<thead>
<tr>
<th>P_0</th>
<th>1</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>0</td>
<td>1</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
</tr>
<tr>
<td>P_2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
</tr>
<tr>
<td>P_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
</tr>
<tr>
<td>P_4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
</tr>
<tr>
<td>P_5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
</tr>
<tr>
<td>P_6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
</tr>
<tr>
<td>P_7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
</tr>
</tbody>
</table>

(a) Computation:

(i) \( A[k,j] := A[k,j] / A[k,k] \) for \( k < j < n \)

(ii) \( A[k,k] := 1 \)

(b) Communication:

One-to-all broadcast of row \( A[k,*] \)
Rowwise Partitioning on Eight Processes

<table>
<thead>
<tr>
<th></th>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_5</th>
<th>P_6</th>
<th>P_7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7)</td>
<td>0 1 (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)</td>
<td>0 0 1 (2,3) (2,4) (2,5) (2,6) (2,7)</td>
<td>0 0 0 1 (3,4) (3,5) (3,6) (3,7)</td>
<td>0 0 0 (4,3) (4,4) (4,5) (4,6) (4,7)</td>
<td>0 0 0 (5,3) (5,4) (5,5) (5,6) (5,7)</td>
<td>0 0 0 (6,3) (6,4) (6,5) (6,6) (6,7)</td>
<td>0 0 0 (7,3) (7,4) (7,5) (7,6) (7,7)</td>
</tr>
</tbody>
</table>

(c) Computation:


(ii) $A[i,k] := 0$ for $k < i < n$
## 2D Mesh Partitioning on 64 Processes

(a) Rowwise broadcast of \( A[i,k] \) for \( (k - 1) < i < n \)

(b) \( A[k,j] := A[k,j]/A[k,k] \) for \( k < j < n \)

(c) Columnwise broadcast of \( A[k,j] \) for \( k < j < n \)

(d) \( A[i,j] := A[i,j]-A[i,k] \times A[k,j] \) for \( k < i < n \) and \( k < j < n \)
Back Substitution to Find Solution

1. **procedure** BACK SUBSTITUTION \((U, x, y)\)
2. begin
3. for \( k := n - 1 \) downto 0 do /* Main loop */
4. begin
5. \( x[k] := y[k]; \)
6. for \( i := k - 1 \) downto 0 do
7. \( y[i] := y[i] - x[k] \times U[i, k]; \)
8. endfor;
9. end BACK SUBSTITUTION
Dense Linear Algebra (www.netlib.gov)

- Basic Linear Algebra Subroutines (BLAS)
  - Level 1 (vector-vector): vectorization
  - Level 2 (matrix-vector): vectorization, parallelization
  - Level 3 (matrix-matrix): parallelization
- LINPACK (Fortran)
  - Linear equations and linear least-squares
- EISPACK (Fortran)
  - Eigenvalues and eigenvectors for matrix classes
- LAPACK (Fortran, C) (LINPACK + EISPACK)
  - Use BLAS internally
- ScaLAPACK (Fortran, C, MPI) (scalable LAPACK)
Sorting Algorithms

- Task of arranging unordered collection into order
- Permutation of a sequence of elements
- Internal versus external sorting
  - External sorting uses auxiliary storage
- Comparison-based
  - Compare pairs of elements and exchange
  - $O(n \log n)$
- Noncomparison-based
  - Use known properties of elements
  - $O(n)$
Sorting on Parallel Computers

- Where are the elements stored?
  - Need to be distributed across processes
  - Sorted order will be with respect to process order

- How are comparisons performed?
  - One element per process
    - compare-exchange
    - interprocess communication will dominate execution time
  - More than one element per process
    - compare-split

- Sorting networks
  - Based on comparison network model

- Contrast with shared memory sorting algorithms
Single vs. Multi Element Comparision

- One element per processor

\[ a_i \rightarrow a_j \quad a_i, a_j \quad a_j, a_i \quad \min\{a_i, a_j\} \quad \max\{a_i, a_j\} \]

\[ P_i \rightarrow P_j \quad P_i \rightarrow P_j \quad P_i \rightarrow P_j \quad P_i \rightarrow P_j \]

- Multiple elements per processor

\[ 1, 6, 8, 1, 13 \rightarrow 2, 7, 9, 10, 12 \]

\[ P_i \rightarrow P_j \quad P_i \rightarrow P_j \]

\[ 1, 2, 6, 7, 8, 9, 10, 11, 12, 13 \rightarrow 1, 2, 6, 7, 8, 9, 10, 11, 12, 13 \]

\[ P_i \rightarrow P_j \quad P_i \rightarrow P_j \]

\[ 1, 2, 6, 7, 8 \rightarrow 9, 10, 11, 12, 13 \]

\[ P_i \rightarrow P_j \quad P_i \rightarrow P_j \]
**Sorting Networks**

- Networks to sort $n$ elements in less than $O(n \log n)$
- Key component in network is a comparator
  - Increasing or decreasing comparator

  ![Comparator Diagram](image)

- Comparators connected in parallel and permute elements
## Sorting Network Design

- Multiple comparator stages (# stages, # comparators)
- Connected together by interconnection network
- Output of last stage is the sorted list
- $O(\log^2 n)$ sorting time
- Convert any sorting network to sequential algorithm
Bitonic Sort

- Create a *bitonic sequence* then sort the sequence

- Bitonic sequence
  - sequence of elements \(<a_0, a_1, \ldots, a_{n-1}>\)
  - \(<a_0, a_1, \ldots, a_i>\) is monotonically increasing
  - \(<a_i, a_{i+1}, \ldots, a_{n-1}>\) is monotonically decreasing

- Sorting using *bitonic splits* is called *bitonic merge*

- *Bitonic merge network* is a network of comparators
  - Implement bitonic merge

- Bitonic sequence is formed from unordered sequence
  - Bitonic sort creates a bitonic sequence
  - Start with sequence of size two (default bitonic)
Bitonic Sort Network

Unordered sequence

<table>
<thead>
<tr>
<th>Wires</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>14</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>20</td>
<td>20</td>
<td>9</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>8</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0010</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>20</td>
<td>12</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0011</td>
<td>9</td>
<td>5</td>
<td>20</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0100</td>
<td>3</td>
<td>3</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0101</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0110</td>
<td>12</td>
<td>14</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>0111</td>
<td>14</td>
<td>12</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>90</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
</tr>
<tr>
<td>1000</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>95</td>
<td>95</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>40</td>
<td>23</td>
<td>35</td>
<td>95</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1010</td>
<td>60</td>
<td>60</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1011</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1100</td>
<td>23</td>
<td>23</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1101</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1110</td>
<td>95</td>
<td>95</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1111</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>

Bitonic sequence

- **decrease**
- **increase**
Bitonic Merge Network

Bitonic sequence

Sorted sequence

Wires

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

3 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0

5 5 5 5 3 3 3 3 3 3 3 3 3 3 3 3

8 8 8 8 5 5 5 5 5 5 5 5 5 5 5 5

9 9 9 9 8 8 8 8 8 8 8 8 8 8 8 8

10 10 10 10 9 9 9 9 9 9 9 9 9 9 9 9

12 12 12 12 10 10 10 10 10 10 10 10 10 10 10 10

14 14 14 14 12 12 12 12 12 12 12 12 12 12 12 12

20 20 20 20 14 14 14 14 14 14 14 14 14 14 14 14

95 95 95 95 18 18 18 18 18 18 18 18 18 18 18 18

90 90 90 90 20 20 20 20 20 20 20 20 20 20 20 20

60 60 60 60 35 35 35 35 35 35 35 35 35 35 35 35

40 40 40 40 23 23 23 23 23 23 23 23 23 23 23 23

35 35 35 35 60 60 60 60 60 60 60 60 60 60 60 60

23 23 23 23 90 90 90 90 90 90 90 90 90 90 90 90

18 18 18 18 95 95 95 95 95 95 95 95 95 95 95 95

0 0 0 0 90 90 90 90 90 90 90 90 90 90 90 90

20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

CIS 631 - Parallel Processing

Lecture 9
Parallel Bitonic Sort on a Hypercube

1. procedure BITONIC SORT(label, d)
2. begin
3. for i := 0 to d - 1 do
4.   for j := i downto 0 do
5.     if (i + 1)st bit of label = j th bit of label then
6.       comp exchange max(j);
7.     else
8.       comp exchange min(j);
9. end BITONIC SORT
Parallel Bitonic Sort on a Hypercube (Last stage)

Step 1

Step 2

Step 3

Step 4
Bubble Sort and Variants

- Can easily parallelize sorting algorithms of $O(n^2)$
- **Bubble sort** compares and exchanges adjacent elements
  - $O(n)$ each pass
  - $O(n)$ passes
  - Available parallelism?
- **Odd-even transposition sort**
  - Compares and exchanges odd and even pairs
  - After $n$ phases, elements are sorted
  - Available parallelism?
# Odd-Even Transposition Sort

<table>
<thead>
<tr>
<th>Unsorted</th>
<th>Phase 1 (odd)</th>
<th>Phase 2 (even)</th>
<th>Phase 3 (odd)</th>
<th>Phase 4 (even)</th>
<th>Phase 5 (odd)</th>
<th>Phase 6 (even)</th>
<th>Phase 7 (odd)</th>
<th>Phase 8 (even)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 3 8 5 6 4 1</td>
<td>2 3 3 8 5 6 1 4</td>
<td>2 3 3 5 8 1 6 4</td>
<td>2 3 3 5 1 8 4 6</td>
<td>2 3 3 1 5 4 8 6</td>
<td>2 3 1 3 4 5 6 8</td>
<td>2 1 3 3 4 5 6 8</td>
<td>1 2 3 3 4 5 6 8</td>
<td>1 2 3 3 4 5 6 8</td>
</tr>
</tbody>
</table>

*Sorted*
Parallel Odd-Even Transposition Sort on Ring

1. procedure ODD-EVEN PAR\((n)\)
2. begin
3. \(id := \) process’s label
4. for \(i := 1\) to \(n\) do
5. begin
6. if \(i\) is odd then
7. if \(id\) is odd then
8. compare-exchange \(\text{min}(id + 1)\);
9. else
10. compare-exchange \(\text{max}(id - 1)\);
11. if \(i\) is even then
12. if \(id\) is even then
13. compare-exchange \(\text{min}(id + 1)\);
14. else
15. compare-exchange \(\text{max}(id - 1)\);
16. end for
17. end ODD-EVEN PAR
Quicksort

- Quicksort has average complexity of $O(n \log n)$
- Divide-and-conquer algorithm
  - Divide into subsequences where every element in first is less than or equal to every element in the second
  - Pivot is used to split the sequence
  - Conquer step recursively applies quicksort algorithm
- Available parallelism?
Sequential Quicksort

1. procedure QUICKSORT (A, q, r )
2. begin
3. if q < r then
4. begin
5. x := A[q];
6. s := q;
7. for i := q + 1 to r do
8. if A[i] ≤ x then
9. begin
10. s := s + 1;
11. swap(A[s], A[i]);
12. end if
13. swap(A[q], A[s]);
14. QUICKSORT (A, q, s);
15. QUICKSORT (A, s + 1, r);
16. end if
17. end QUICKSORT
Parallel Shared Address Space Quicksort

First Step

pivot = 7

Second Step

pivot = 5
pivot = 17

Lecture 9
Parallel Shared Address Space Quicksort

**Lecture 9**

---

**Third Step**

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>9</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>18</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

pivot selection
pivot = 11

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

after local rearrangement

<table>
<thead>
<tr>
<th>10</th>
<th>9</th>
<th>8</th>
<th>12</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
</table>

after global rearrangement

---

**Fourth Step**

<table>
<thead>
<tr>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

after local rearrangement

---

**Solution**

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>
Bucket Sort and Sample Sort

- **Bucket sort** is popular when elements (values) are uniformly distributed over an interval
  - Create $m$ buckets and place elements in appropriate bucket
  - $O(n \log(n/m))$
  - If $m=n$, can use value as index to achieve $O(n)$ time

- **Sample sort** is used when uniformly distributed assumption is not true
  - Distributed to $m$ buckets and sort each with quicksort
  - Draw sample of size $s$
  - Sort samples and choose $m-1$ elements to be *splitters*
  - Split into $m$ buckets and proceed with bucket sort
Parallel Sample Sort

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment
Graph Algorithms

- Graph theory important in computer science
- Many complex problems are graph problems
- $G = (V, E)$
  - $V$ finite set of points called vertices
  - $E$ finite set of edges
  - $e \in E$ is an pair $(u, v)$, where $u, v \in V$
  - Unordered and ordered graphs
Graph Terminology

- Vertex adjacency if \((u,v)\) is an edge
- *Path* from \(u\) to \(v\) if there is an edge sequence starting at \(u\) and ending at \(v\)
- If there exists a path, \(v\) is *reachable* from \(u\)
- A graph is *connected* if all pairs of vertices are connected by a path
- A *weighted* graph associates weights with each edge
- *Adjacency matrix* is an \(n \times n\) array \(A\) such that
  - \(A_{i,j} = 1\) if \((v_i,v_j) \in E\); \(0\) otherwise
  - Can be modified for weighted graphs (\(\infty\) is no edge)
  - Can represent as *adjacency lists*
Graph Representations

- Adjacency matrix

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

- Adjacency list
Minimum Spanning Tree

- A spanning tree of an undirected graph $G$ is a subgraph of $G$ that is a tree containing all the vertices of $G$
- The minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight
- Prim’s algorithm can be used
  - Greedy algorithm
  - Selects an arbitrary starting vertex
  - Chooses new vertex guaranteed to be in MST
  - $O(n^2)$
  - Prim’s algorithm is iterative
Prim’s Minimum Spanning Tree Algorithm

1. procedure PRIM MST( V, E, w, r )
2. begin
3.   VT := {r};
4.   d[r] := 0;
5.   for all v ∈ (V - VT ) do
6.     if edge (r, v) exists set d[v] := w(r, v);
7.     else set d[v] :=∞;
8.   while VT ≠ V do
9.     begin
10.    find a vertex u such that d[u] := min{d[v]|v ∈ (V - VT )};
11.    VT := VT ∪ {u};
12.    for all v ∈ (V - VT ) do
13.       d[v] := min{d[v], w(u, v)};  
14.   endwhile
15. end PRIM MST
Example: Prim’s MST Algorithm

(a) Original graph

(b) After the first edge has been selected

\[
\begin{array}{cccccccc}
& a & b & c & d & e & f \\
\hline
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & \infty & 5 & 0 \\
\end{array}
\]
Example: Prim’s MST Algorithm

(c) After the second edge has been selected

(d) Final minimum spanning tree

\[
\begin{array}{ccccccc}
  & a & b & c & d & e & f \\
 a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & 5 & 0 \\
\end{array}
\]
Parallel Formulation of Prim’s Algorithm

- Difficult to perform different iterations of the **while** loop in parallel because $d[v]$ may change each time
- Can parallelize each iteration though
- Partition vertices into $p$ subsets $V_i$, $i=0,...,p-1$
- Each process $P_i$ computes
  \[ d_i[u]=\min\{d_i[v] \mid v \in (V-V_T) \cap V_i\} \]
- Global minimum is obtained using all-to-one reduction
- New vertex is added to $V_T$ and broadcast to all processes
- New values of $d[v]$ are computed for local vertex
- $O(n^2/p) + O(n \log p)$ (computation + communication)
Partitioning in Prim’s Algorithm

\[ d[1..n] \]

\[
\begin{array}{|c|c|c|}
\hline
| \frac{n}{p} | \\
\hline
\end{array}
\]

(a)

\[ A \]

\[
\begin{array}{|c|c|c|}
\hline
\text{Processors} & 0 & 1 & i & p-1 \\
\hline
\end{array}
\]

(b)
Single-Source Shortest Paths

- Find *shortest path* from a vertex \( v \) to all other vertices
- The shortest path in a weighted graph is the edge with the minimum weight
- Weights may represent time, cost, loss, or any other quantity that accumulates additively along a path
- Dijkstra’s algorithm finds shortest paths from a vertex \( s \)
  - Similar to Prim’s MST algorithm
    - MST with vertex \( v \) as starting vertex
  - Incrementally finds shortest paths in greedy manner
  - Keep track of minimum cost to reach a vertex from \( s \)
  - \( O(n^2) \)
Dijkstra’s Single-Source Shortest Paths Algorithm

1. **procedure** DIJKSTRA SINGLE SOURCE SP\((V, E, w, s)\)
2. **begin**
3. \(V_T := \{s\}\);
4. **for** all \(v \in (V - V_T)\) **do**
5. \(\text{if } (s, v) \text{ exists set } l[v] := w(s, v);\)
6. \(\text{else set } l[v] := \infty;\)
7. **while** \(V_T \neq V\) **do**
8. **begin**
9. \(\text{find a vertex } u \text{ such that } l[u] := \min\{l[v] | v \in (V - V_T)\};\)
10. \(V_T := V_T \cup \{u\};\)
11. **for** all \(v \in (V - V_T)\) **do**
12. \(l[v] := \min\{l[v], l[u] + w(u, v)\};\)
13. **endwhile**
14. **end** DIJKSTRA SINGLE SOURCE SP
Parallel Formulation of Dijkstra’s Algorithm

- Very similar to Prim’s MST parallel formulation
- Use 1D block mapping as before
- All processes perform computation and communication similar to that performed in Prim’s algorithm
- Parallel performance is the same
  - \( O(n^2/p) + O(n \log p) \)
  - Scalability
    - \( O(n^2) \) is the sequential time
    - \( O(n^2) / [O(n^2/p) + O(n \log p)] \)
All Pairs Shortest Path

- Find the shortest path between all pairs of vertices
- Outcome is a $n \times n$ matrix $D = \{d_{i,j}\}$ such that $d_{i,j}$ is the cost of the shortest path from vertex $v_i$ to vertex $v_j$

- Dijsktra’s algorithm
  - Execute single-source algorithm on each process
  - $O(n^3)$
  - Source-partitioned formulation (use sequential algorithm)
  - Source-parallel formulation (use parallel algorithm)

- Floyd’s algorithm
  - Builds up distance matrix from the bottom up
Floyd’s All-Pairs Shortest Paths Algorithm

1. procedure FLOYD ALL PAIRS SP(A)
2. begin
3. \( D^{(0)} = A; \)
4. for \( k := 1 \) to \( n \) do
5. for \( i := 1 \) to \( n \) do
6. for \( j := 1 \) to \( n \) do
7. \( d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j} ; \)
8. end FLOYD ALL PAIRS SP
Parallel Floyd’s Algorithm

1. procedure FLOYD ALL PAIRS PARALLEL (A)
2. begin
3. \( D^{(0)} = A; \)
4. for \( k := 1 \) to \( n \) do
  5. forall \( P_{i,j} \), where \( i, j \leq n \), do in parallel
  6. \( d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j} ; \)
7. end FLOYD ALL PAIRS PARALLEL
Next Class

- Algorithms for simulation
- Analytical modeling of parallel programs