CIS 631
Parallel Processing

Lecture 11: 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 (classic algorithm)
  
  - Forward elimination to $Ux = y$ ($U$ is upper triangular)
    
    - Without or with partial pivoting
  
  - Back substitution to solve for $x$
  
  - Parallel algorithms based on partitioning of $A$
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**
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**
15.            **endfor**; /*Line9*/
16.        **endfor**; /*Line3*/
17.     **end** GAUSSIAN ELIMINATION
Computation Step in Gaussian Elimination

\[5x + 3y = 22\]
\[8x + 2y = 13\]
\[x = \frac{22 - 3y}{5}\]
\[8(22 - 3y)/5 + 2y = 13\]
\[y = \frac{13 - 176/5}{24/5 + 2}\]
### 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</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(0.1)</td>
<td>(0.2)</td>
<td>(0.3)</td>
<td>(0.4)</td>
<td>(0.5)</td>
<td>(0.6)</td>
<td>(0.7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.2)</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>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.3)</td>
<td>(2.3)</td>
<td>(2.4)</td>
<td>(2.5)</td>
<td>(2.6)</td>
<td>(2.7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.4)</td>
<td>(3.3)</td>
<td>(3.4)</td>
<td>(3.5)</td>
<td>(3.6)</td>
<td>(3.7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.5)</td>
<td>(4.3)</td>
<td>(4.4)</td>
<td>(4.5)</td>
<td>(4.6)</td>
<td>(4.7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.6)</td>
<td>(5.3)</td>
<td>(5.4)</td>
<td>(5.5)</td>
<td>(5.6)</td>
<td>(5.7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.7)</td>
<td>(6.3)</td>
<td>(6.4)</td>
<td>(6.5)</td>
<td>(6.6)</td>
<td>(6.7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(7.3)</td>
<td>(7.4)</td>
<td>(7.5)</td>
<td>(7.6)</td>
<td>(7.7)</td>
<td></td>
<td></td>
<td></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>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
<th>P₆</th>
<th>P₇</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(0,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(4,3)</td>
<td>(5,3)</td>
<td>(6,3)</td>
<td>(7,3)</td>
</tr>
<tr>
<td></td>
<td>(0,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(2,4)</td>
<td>(4,4)</td>
<td>(5,4)</td>
<td>(6,4)</td>
<td>(7,4)</td>
</tr>
<tr>
<td></td>
<td>(0,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(2,5)</td>
<td>(4,5)</td>
<td>(5,5)</td>
<td>(6,5)</td>
<td>(7,5)</td>
</tr>
<tr>
<td></td>
<td>(0,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(2,6)</td>
<td>(4,6)</td>
<td>(5,6)</td>
<td>(6,6)</td>
<td>(7,6)</td>
</tr>
<tr>
<td></td>
<td>(0,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td>(2,7)</td>
<td>(4,7)</td>
<td>(5,7)</td>
<td>(6,7)</td>
<td>(7,7)</td>
</tr>
<tr>
<td></td>
<td>(0,6)</td>
<td>(1,7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0,7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Computation:

for \( k < i < n \) and \( k < j < n \)

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

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

<table>
<thead>
<tr>
<th></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>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>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>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>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>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>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>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>

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

<table>
<thead>
<tr>
<th></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>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>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>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>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>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>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>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>

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

<table>
<thead>
<tr>
<th></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>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>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>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>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>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>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>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>

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] xU[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)
Numerical Libraries

- PETSc (http://www.mcs.anl.gov/petsc/petsc-as)
  - data structures / routines for partial differential equations
  - MPI based
- SuperLU (http://crd.lbl.gov/~xiaoye/SuperLU/)
  - Large sparse nonsymmetric linear systems
- Hypre (http://www.llnl.gov/CASC/hypre)
  - Large sparse linear systems
- TAO (http://www.mcs.anl.gov/research/projects/tao/)
  - Toolkit for Advanced Optimization
- DOE ACTS (http://acts.nersc.gov/)
  - Advanced CompuTational Software
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**

  - $a_i \rightarrow a_j \rightarrow a_i, a_j \rightarrow a_j, a_i \rightarrow \min\{a_i, a_j\} \rightarrow \max\{a_i, a_j\}$

- **Multiple elements per processor**

  - $1, 6, 8, 13 \rightarrow 2, 7, 9, 10, 12 \rightarrow 1, 6, 8, 11, 13, 2, 7, 9, 10, 12$

---

Lecture 9
**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

\[\begin{align*}
x' &= \min\{x, y\} \\
y' &= \max\{x, y\}
\end{align*}\]

- 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

Bitonic sequence

decrease

increase
Bitonic Merge Network

Bitonic sequence

Sorted sequence

Wires

0000 3

0001 5

0010 8

0011 9

0100 10

0101 12

0110 14

0111 20

1000 95

1001 90

1010 60

1011 40

1100 35

1101 23

1110 18

1111 0

3

5

8

9

10

12

14

20

95

90

60

40

35

23

18

0

5

8

10

12

14

18

20

23

35

60

20

40

90

95

9

9

3

5

8
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)
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

Unsorted

3  2  3  8  5  6  4  1

2  3  3  8  5  6  1  4

2  3  3  5  8  1  6  4

2  3  3  5  1  8  4  6

2  3  3  1  5  4  8  6

2  3  1  3  4  5  6  8

2  1  3  3  4  5  6  8

1  2  3  3  4  5  6  8

Sorted

1  2  3  3  4  5  6  8
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 min(id + 1);
9.              else
10.             compare-exchange max(id - 1);
11.          if i is even then
12.             if id is even then
13.                compare-exchange min(id + 1);
14.             else
15.                compare-exchange 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 \( A[i] \leq x \) 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

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>pivot selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>13</td>
<td>18</td>
<td>2</td>
<td>17</td>
<td>14</td>
</tr>
</tbody>
</table>
| pivot=7

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>after local rearrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>18</td>
<td>13</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>after global rearrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Second Step

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>pivot selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
| pivot=5

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>pivot=17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>after local rearrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>after global rearrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Parallel Shared Address Space Quicksort

P_0 | P_1 | P_2 | P_3 | P_4
---|---|---|---|---
1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19

pivot = 11

Third Step

P_0 | P_1 | P_2 | P_3 | P_4
---|---|---|---|---
1 2 3 4 5 6 7 10 13 17 14 15 9 8 12 11 16 18 19 20

after local rearrangement

Fourth Step

P_2 | P_3
---|---
10 9 8 12 11 13 17 14 15 16

after local rearrangement

P_0 | P_1 | P_2 | P_3 | P_4
---|---|---|---|---
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Solution
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
- \textit{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 \textit{reachable} from \(u\)
- A graph is \textit{connected} if all pairs of vertices are connected by a path
- A \textit{weighted} graph associates weights with each edge
- \textit{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 \textit{adjacency lists}
Graph Representations

- Adjacency matrix

```
A =
[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]
```

- 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** \(v \in (V - VT)\) **do**
6. \(\text{if edge } (r, v) \text{ exists set } d[v] := w(r, v);\)
7. \(\text{else set } d[v] := \infty;\)
8. **while** \(VT \neq V\) **do**
9. **begin**
10. \(\text{find a vertex } u \text{ such that } d[u] := \min\{d[v] | v \in (V - VT)\};\)
11. \(VT := VT \cup \{u\};\)
12. **for all** \(v \in (V - VT)\) **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

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

(c) After the second edge has been selected

(d) Final minimum spanning tree
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 $V_i$, $i=0,\ldots,p-1$
- 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{align*}
\text{Processors} & \quad 0 & \quad 1 & \quad i & \quad p-1 \\
\end{align*}

(a)
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. \hspace{1cm} **if** $(s, v)$ exists set $l[v] := w(s, v)$;
6. \hspace{1cm} **else** set $l[v] := \infty$;
7. **while** $V_T \neq V$ **do**
8. **begin**
9. \hspace{1cm} find a vertex $u$ 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. \hspace{1cm} $l[v] := \min\{l[v], l[u] + w(u, v)\}$;
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 $v_j$
- Dijkstra’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. \hspace{1em} for \( i := 1 \) to \( n \) do
6. \hspace{2em} for \( j := 1 \) to \( n \) do
7. \hspace{3em} \[ d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}; \]
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. \hspace{1em} forall \( P_{i,j} \), where \( i, j \leq n \), do in parallel
6. \hspace{2em} \( d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}; \)
7. end FLOYD ALL PAIRS PARALLEL
Parallel Graph Algorithm Library – Boost

- Parallel Boost Graph Library (Indiana University)
  - Generic C++ library for high-performance parallel and distributed graph computation
  - Builds on the Boost Graph Library (BGL)
    - offers similar data structures, algorithms, and syntax
  - Research platform for parallel graph algorithms
  - Provide solid implementations for solving large-scale graph problems
  - Boost Software License (BSD-like)

Original BGL: Algorithms

- Searches (breadth-first, depth-first, A*)
- Single-source shortest paths (Dijkstra, Bellman-Ford, DAG)
- All-pairs shortest paths (Johnson, Floyd-Warshall)
- Minimum spanning tree (Kruskal, Prim)
- Components (connected, strongly connected, biconnected)
- Maximum cardinality matching
- Max-flow (Edmonds-Karp, push-relabel)
- Sparse matrix ordering (Cuthill-McKee, King, Sloan, minimum degree)
- Layout (Kamada-Kawai, Fruchterman-Reingold, Gursoy-Atun)
- Betweenness centrality
- PageRank
- Isomorphism
- Vertex coloring
- Transitive closure
- Dominator tree
The original BGL is large, stable, efficient
- Lots of algorithms, graph types
- Peer-reviewed code with many users, nightly regression testing, and so on
- Performance comparable to FORTRAN.

Who should use the BGL?
- Programmers comfortable with C++
- Users with graph sizes from tens of vertices to millions of vertices
**BGL-Python**

- Python is ideal for rapid prototyping:
  - It’s a scripting language (no compiler)
  - Dynamically typed means less typing for you
  - Easy to use: you already know Python…

- BGL-Python provides access to the BGL from within Python
  - Similar interfaces to C++ BGL
  - Easier to learn than C++
  - Great for scripting, GUI applications
  - `help(bgl.dijkstra_shortest_paths)`
The Parallel BGL

- A version of the C++ BGL for computational clusters
  - Distributed memory for huge graphs
  - Parallel processing for improved performance
- An active research project
- Closely related to the original BGL
  - Parallelizing BGL programs should be “easy”

A simple, directed graph… distributed across 3 processors
## Parallel Graph Algorithms

- Breadth-first search
- Eager Dijkstra’s single-source shortest paths
- Crauser et al. single-source shortest paths
- Depth-first search
- Minimum spanning tree (Boruvka, Dehne & Götz)
- Connected components
- Strongly connected components
- Biconnected components
- PageRank
- Graph coloring
- Fruchterman-Reingold layout
- Max-flow (Dinic’s)
Parallel BGL in Python

- Preliminary support for the Parallel BGL in Python
  - Just `import boost.graph.distributed`
  - Similar interface to sequential BGL-Python

- Several options for usage with MPI:
  - Straight MPI: `mpirun -np 2 python script.py`
  - pyMPI: allows interactive use of the interpreter

- Initially used to prototype our distributed Fruchterman-Reingold implementation.
Parallel BGL Summary

- The Parallel BGL is built for huge graphs
  - Millions to hundreds of millions of nodes
  - Distributed-memory parallel processing on clusters
  - Future work will permit larger graphs…
- Parallel programming has a learning curve
  - Parallel graph algorithms much harder to write
  - Distributed graph manipulation can be tricky
- Parallel BGL is an active research library

http://osl.iu.edu/research/pbgl/
Next Class

- Algorithms for simulation
- Analytical modeling of parallel programs