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:
  - A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing," 2003.
  - 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

- $Ax=b$

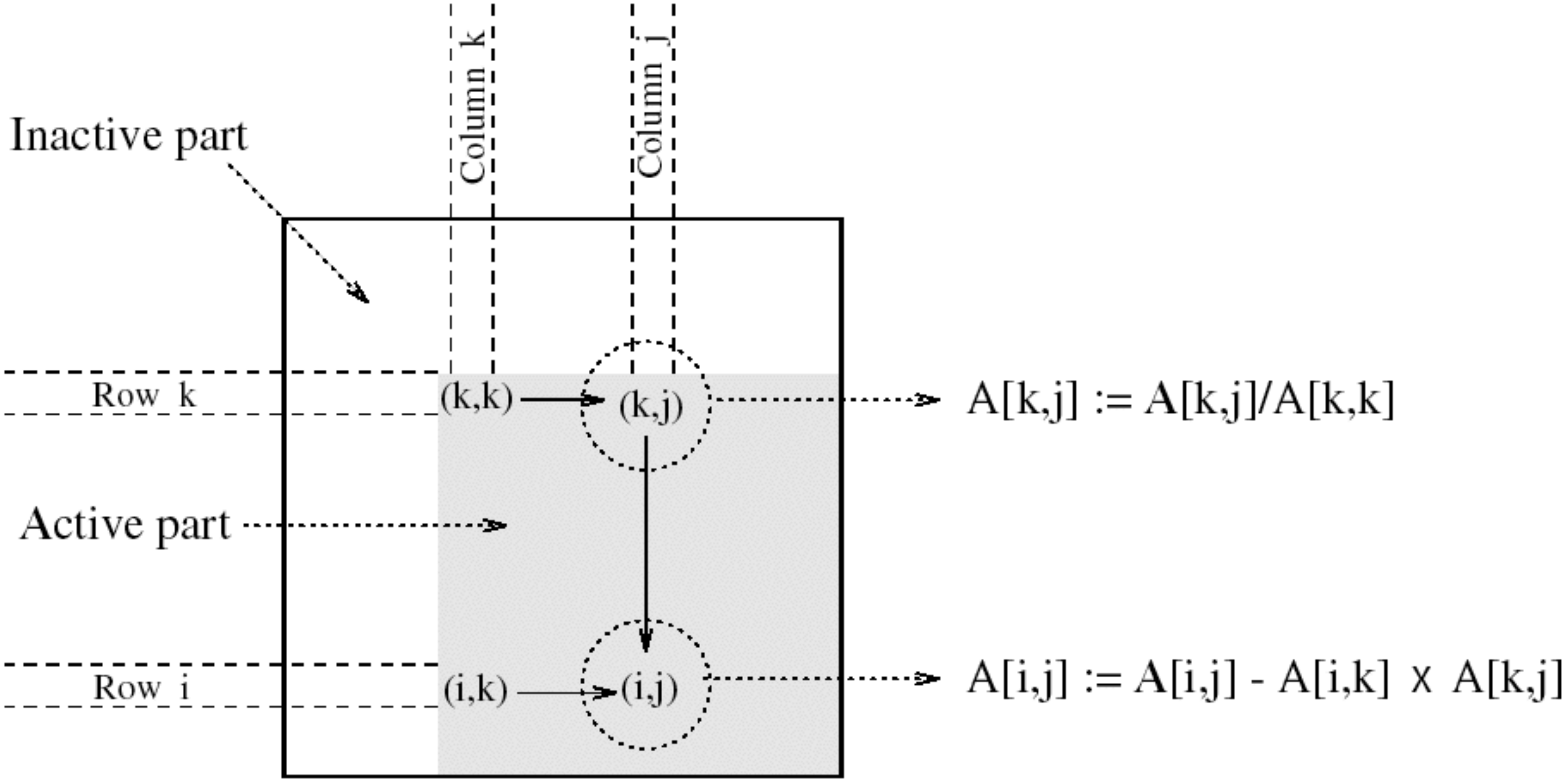
$$\begin{array}{lllll}
a_{0,0}x_0 & + a_{0,1}x_1 & + \ldots + & a_{0,n-1}x_{n-1} & = b_0 \\
a_{1,0}x_0 & + a_{1,1}x_1 & + \ldots + & a_{1,n-1}x_{n-1} & = b_1 \\
\ldots & & & & \\
A_{n-1,0}x_0 & + a_{n-1,1}x_1 & + \ldots + & a_{n-1,n-1}x_{n-1} & = b_{n-1}
\end{array}$$

- 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

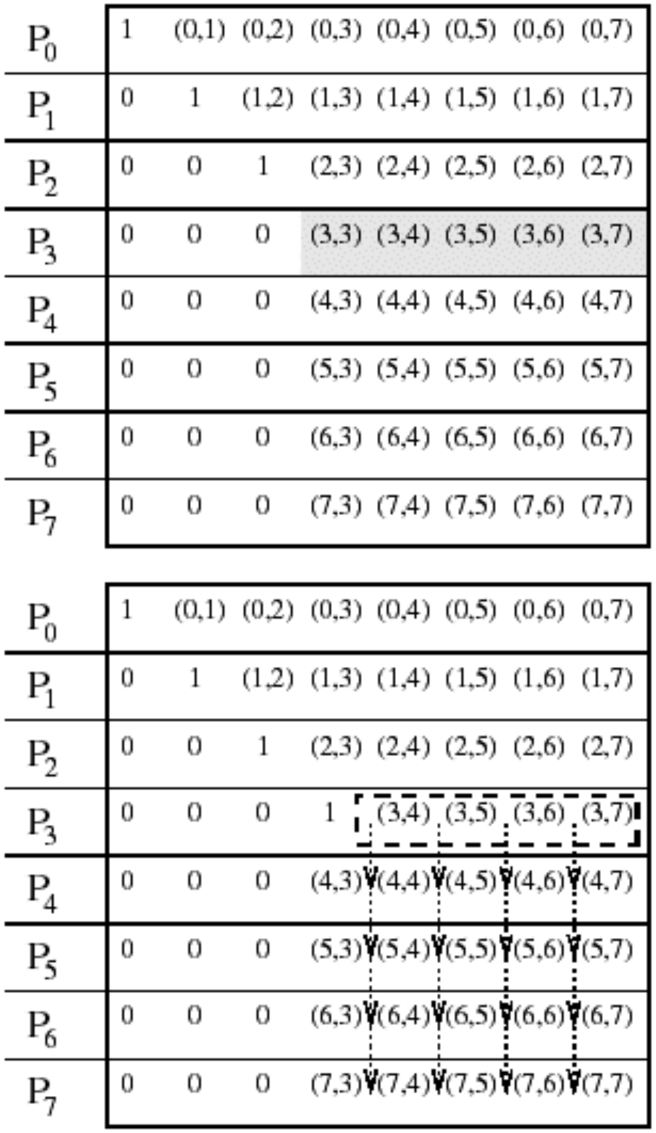
```
1.  procedure GAUSSIAN ELIMINATION ($A$, $b$, $y$)
2.  Begin
3.      for $k$ := 0 to $n$ - 1 do /* Outer loop */
4.      begin
5.          for $j$ := $k$ + 1 to $n$ - 1 do
6.              $A[k, j] := A[k, j]/A[k, k]$; /* Division step */
7.          $y[k] := b[k]/A[k, k]$;
8.          $A[k, k] :=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

Computation Step in Gaussian Elimination



$$\begin{array}{l} 5x + 3y = 22 \\ 8x + 2y = 13 \end{array} \Rightarrow \begin{array}{l} x = (22 - 3y) / 5 \\ 8(22 - 3y)/5 + 2y = 13 \end{array} \Rightarrow \begin{array}{l} x = (22 - 3y) / 5 \\ y = (13 - 176/5) / (24/5 + 2) \end{array}$$

Rowwise Partitioning on Eight Processes



P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Computation:

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

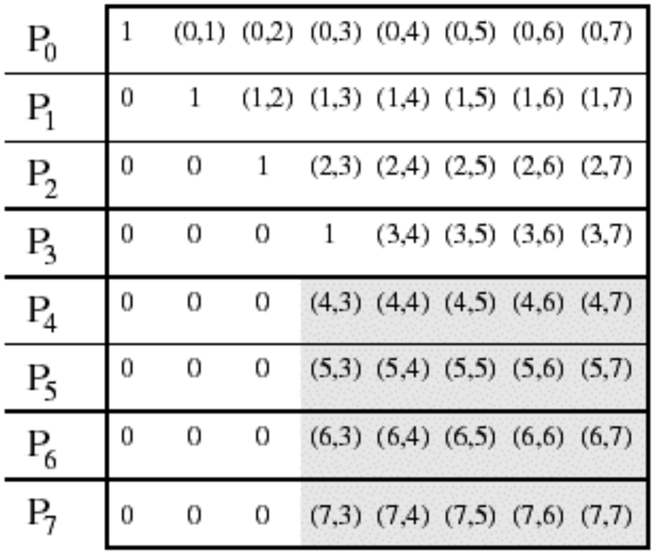
(ii) A[k,k] := 1

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(b) Communication:

One−to−all broadcast of row A[k,*]

Rowwise Partitioning on Eight Processes



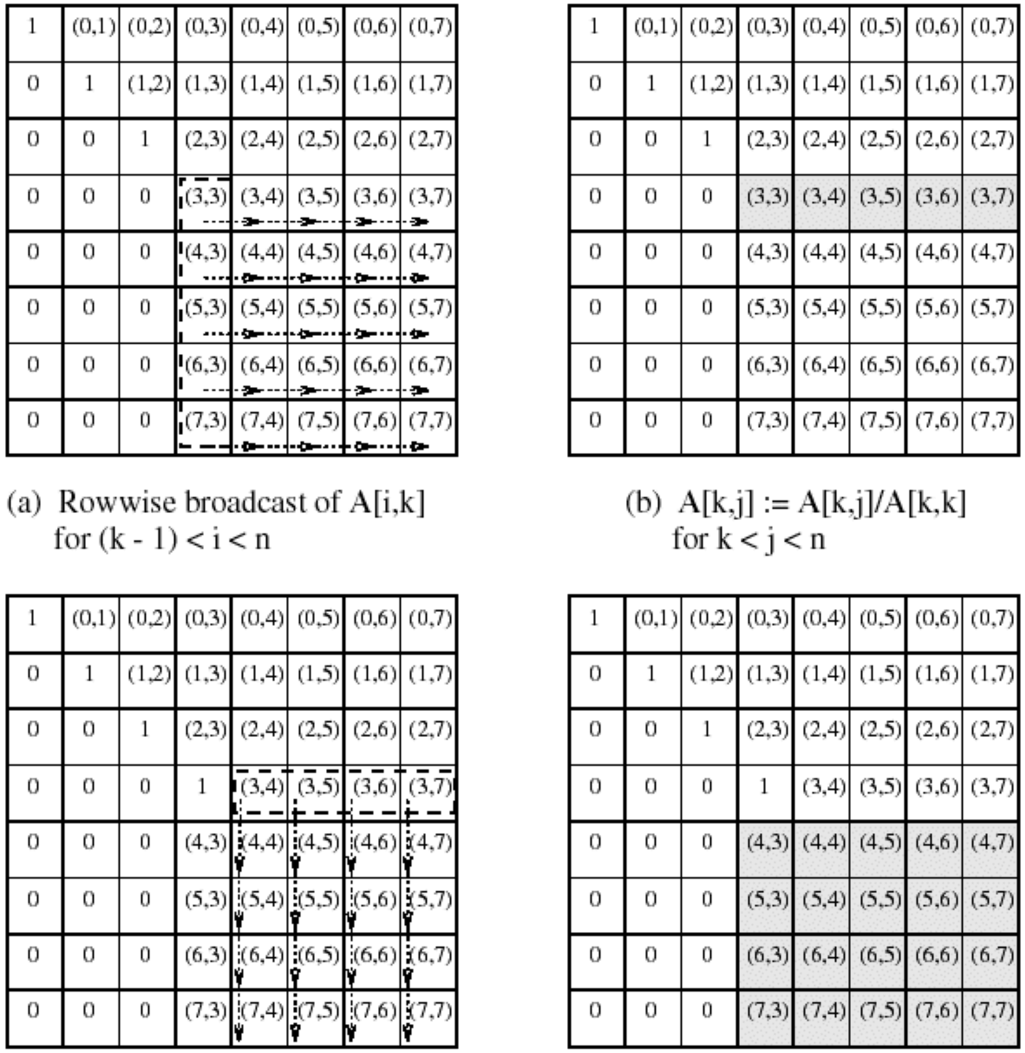
P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(c) Computation:

(i) $A[i,j] := A[i,j] - A[i,k] \times A[k,j]$
for $k < i < n$ and $k < j < n$

(ii) $A[i,k] := 0$ for $k < i < n$

2D Mesh Partitioning on 64 Processes



1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

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

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

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

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

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

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(d) A[i,j] := A[i,j]-A[i,k] x A[k,j]
for k < i < n and k < j < n

Back Substitution to Find Solution

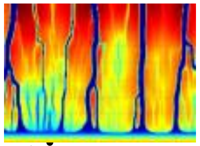
```
procedure BACK SUBSTITUTION ($U, x, y$)
begin
    for $k := n - 1$ downto 0 do /* Main loop */
    begin
        $x[k] := y[k]$;
        for $i := k - 1$ downto 0 do
            $y[i] := y[i] - x[k] \times U[i, k]$;
    endfor;
end BACK SUBSTITUTION
```

Dense Linear Algebra (www.netlib.gov)

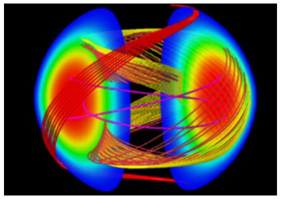
- Basic Linear Algebra Subroutines (BLAS)
  - Level 1 (*vector-vector*): vectorization
  - Level 2 (*matrix-vector*): vectorization, parallelization
  - Level 3 (*matrix-matrix*): parallelization
- LINPACK (Fortran)
  - Linear equations and linear least-squares
- EISPACK (Fortran)
  - Eigenvalues and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

Numerical Libraries

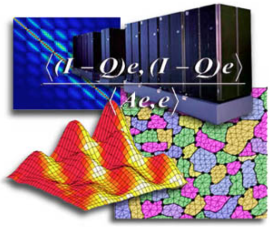
- PETSc (http://www.mcs.anl.gov/petsc/petsc-as)
  - data structures / routines for partial differential equations
  - MPI based



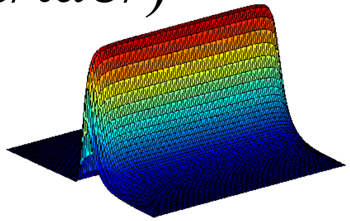
- SuperLU (http://crd.lbl.gov/~xiaoye/SuperLU/)
  - Large sparse nonsymmetric linear systems



- 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

ACTS COLLECTION

Sorting Algorithms

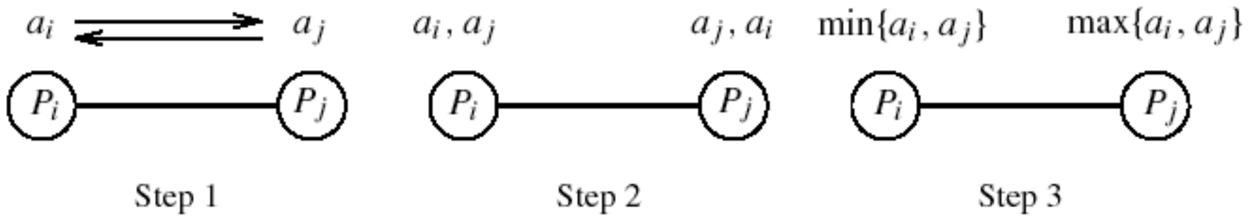
- Task of arranging unordered collection into order
- Permutation of a sequence of elements
- Internal versus external sorting
  - External sorting uses auxiliary storage
- Comparison-based
  - Compare pairs of elements and exchange
  - *O(n log n)*
- Noncomparison-based
  - Use known properties of elements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

Sorting on Parallel Computers

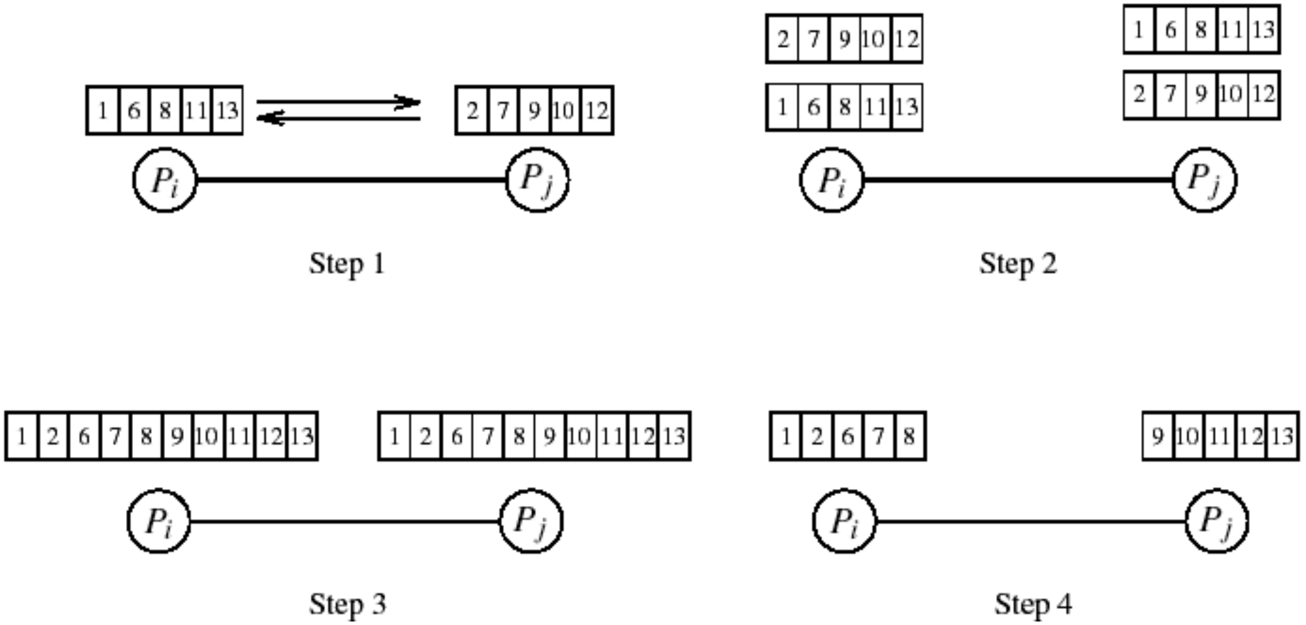
- Where are the elements stored?
  - Need to be distributed across processes
  - Sorted order will be with respect to process order
- How are comparisons performed?
  - One element per process
    - compare-exchange
    - interprocess communication will dominate execution time
  - More than one element 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

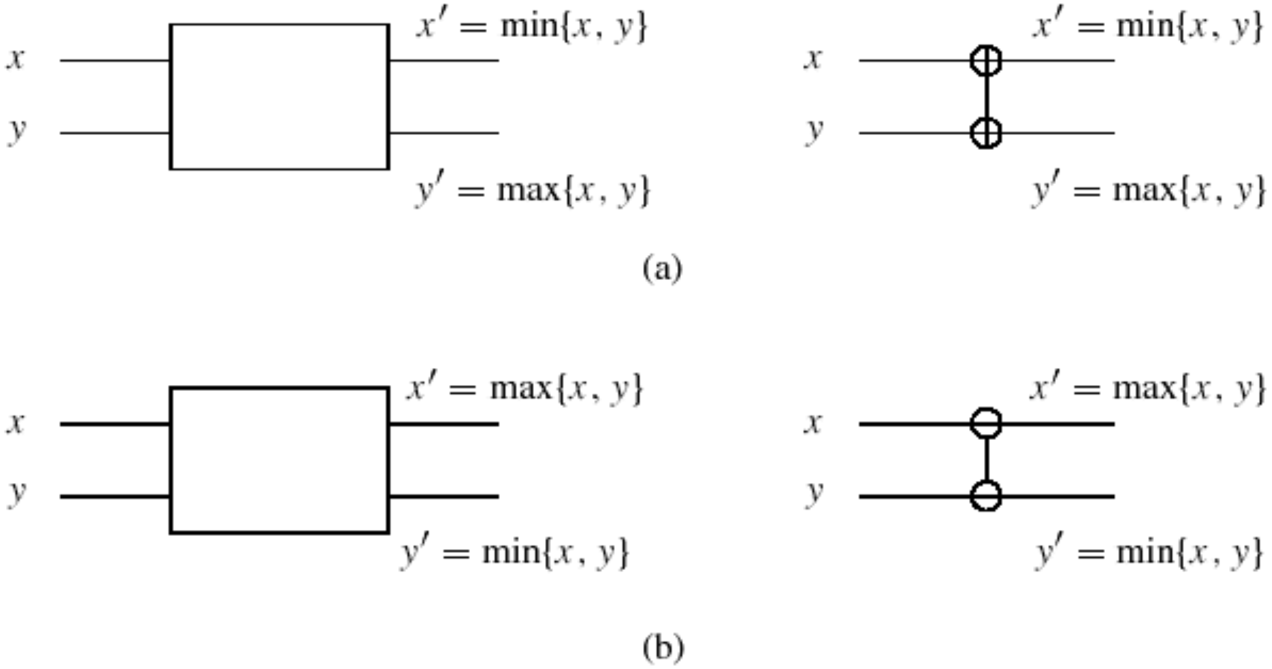


- Multiple elements per processor



Sorting Networks

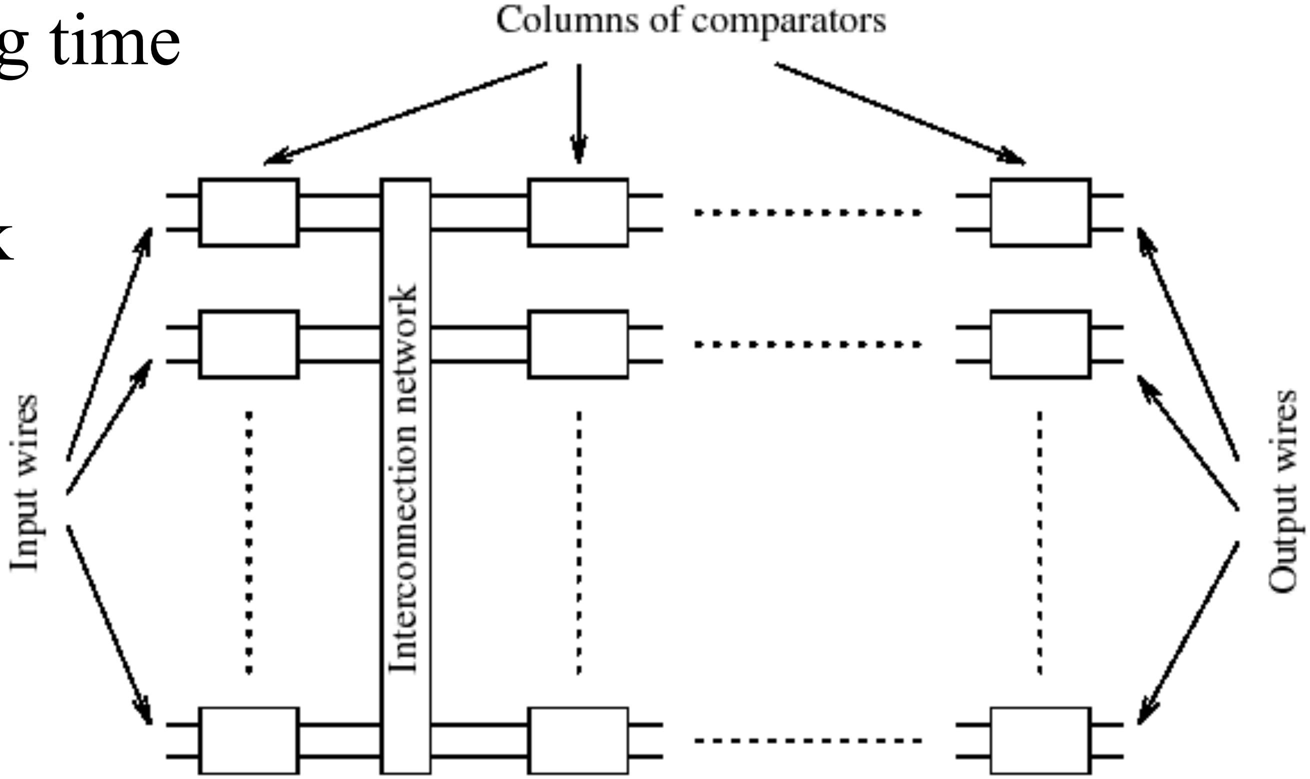
- Networks to sort *n* elements in less than *O(n log n)*
- Key component in network is a comparator
    - Increasing or decreasing comparator



- 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

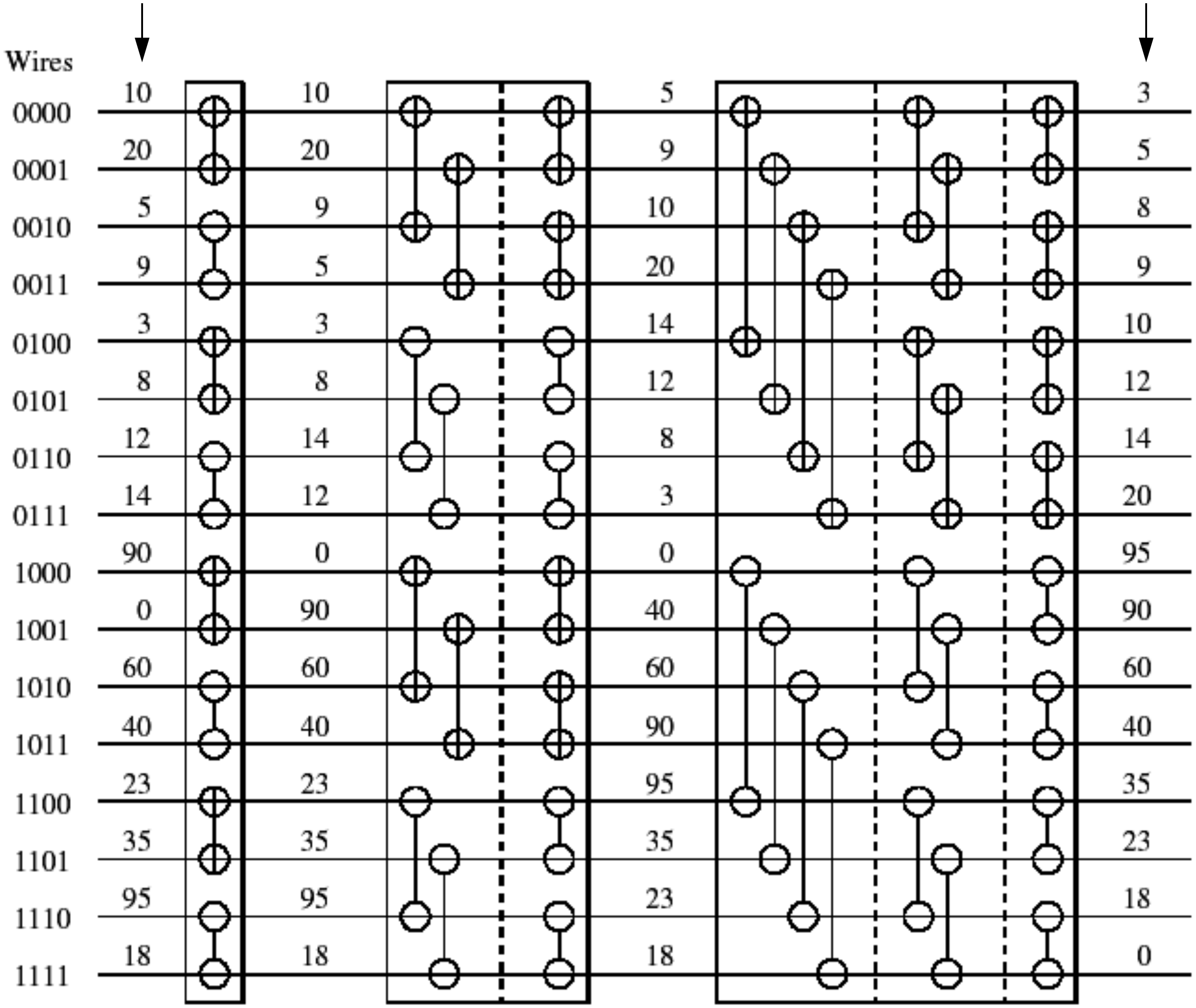


Bitonic Sort

- 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

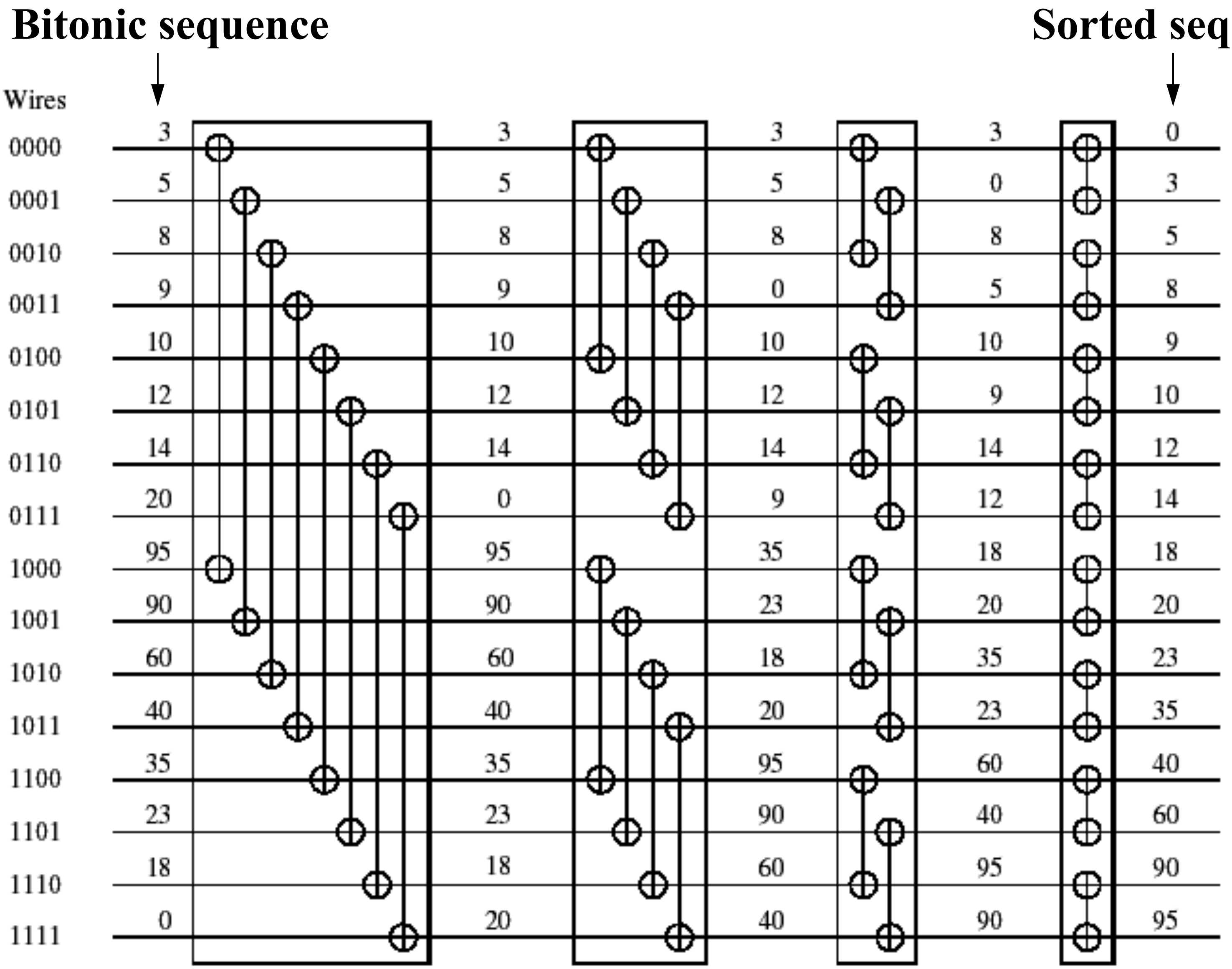
Unordered sequence **Bitonic sequence**



⊕ **decrease**

⊖ **increase**

Bitonic Merge Network

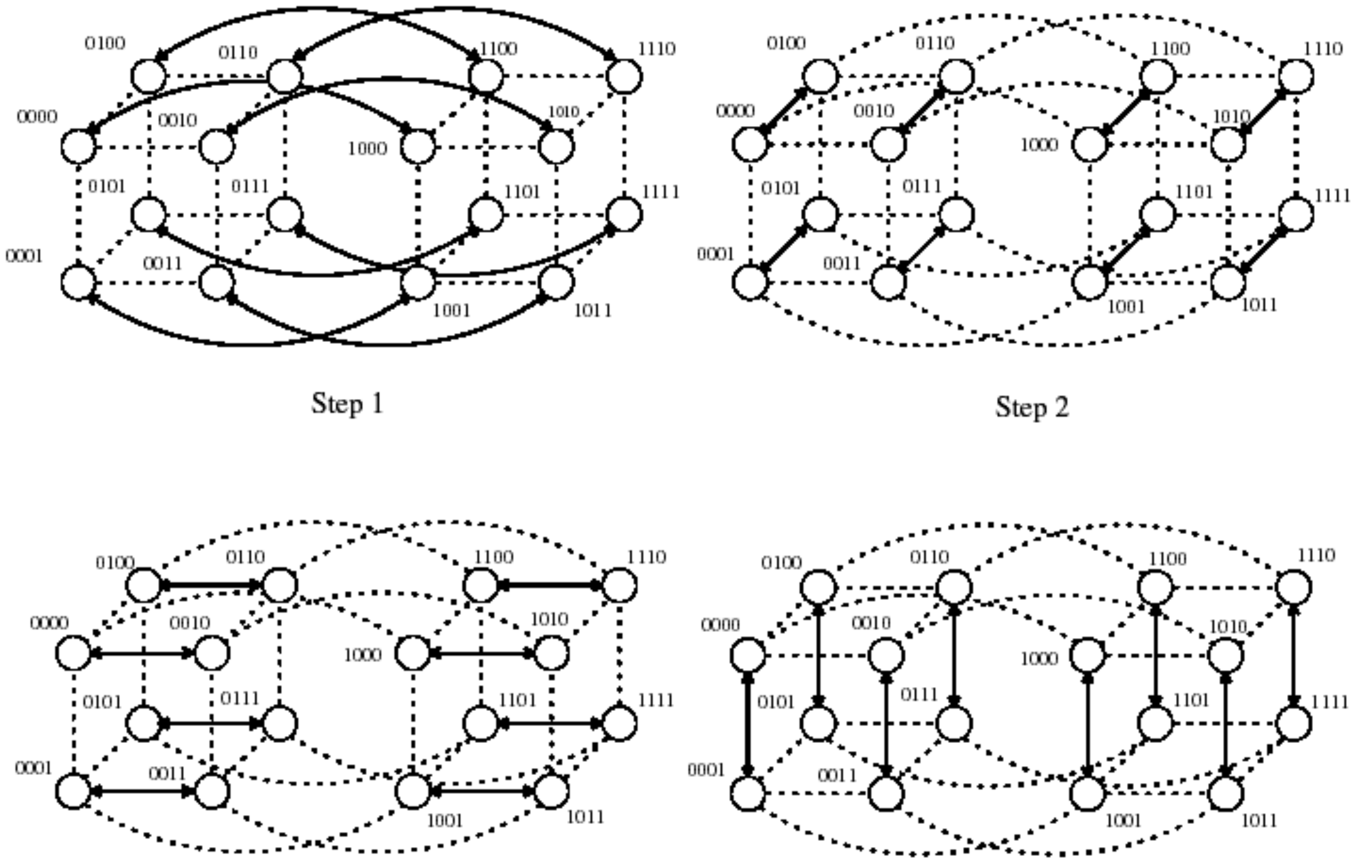


Wires					
0000	3	3	3	3	0
0001	5	5	5	0	3
0010	8	8	8	8	5
0011	9	9	0	5	8
0100	10	10	10	10	9
0101	12	12	12	9	10
0110	14	14	14	14	12
0111	20	0	9	12	14
1000	95	95	35	18	18
1001	90	90	23	20	20
1010	60	60	18	35	23
1011	40	40	20	23	35
1100	35	35	95	60	40
1101	23	23	90	40	60
1110	18	18	60	95	90
1111	0	20	40	90	95

Parallel Bitonic Sort on a Hypercube

```
procedure BITONIC SORT(label, d)
begin
  for $i := 0$ to $d - 1$ do
    for $j := i$ downto 0 do
      if $(i + 1)$st bit of label = $j$ th bit of label then
        comp exchange max(j);
      else
        comp exchange min(j);
end BITONIC SORT
```

Parallel Bitonic Sort on a Hypercube (Last stage)



Step 1

Step 2

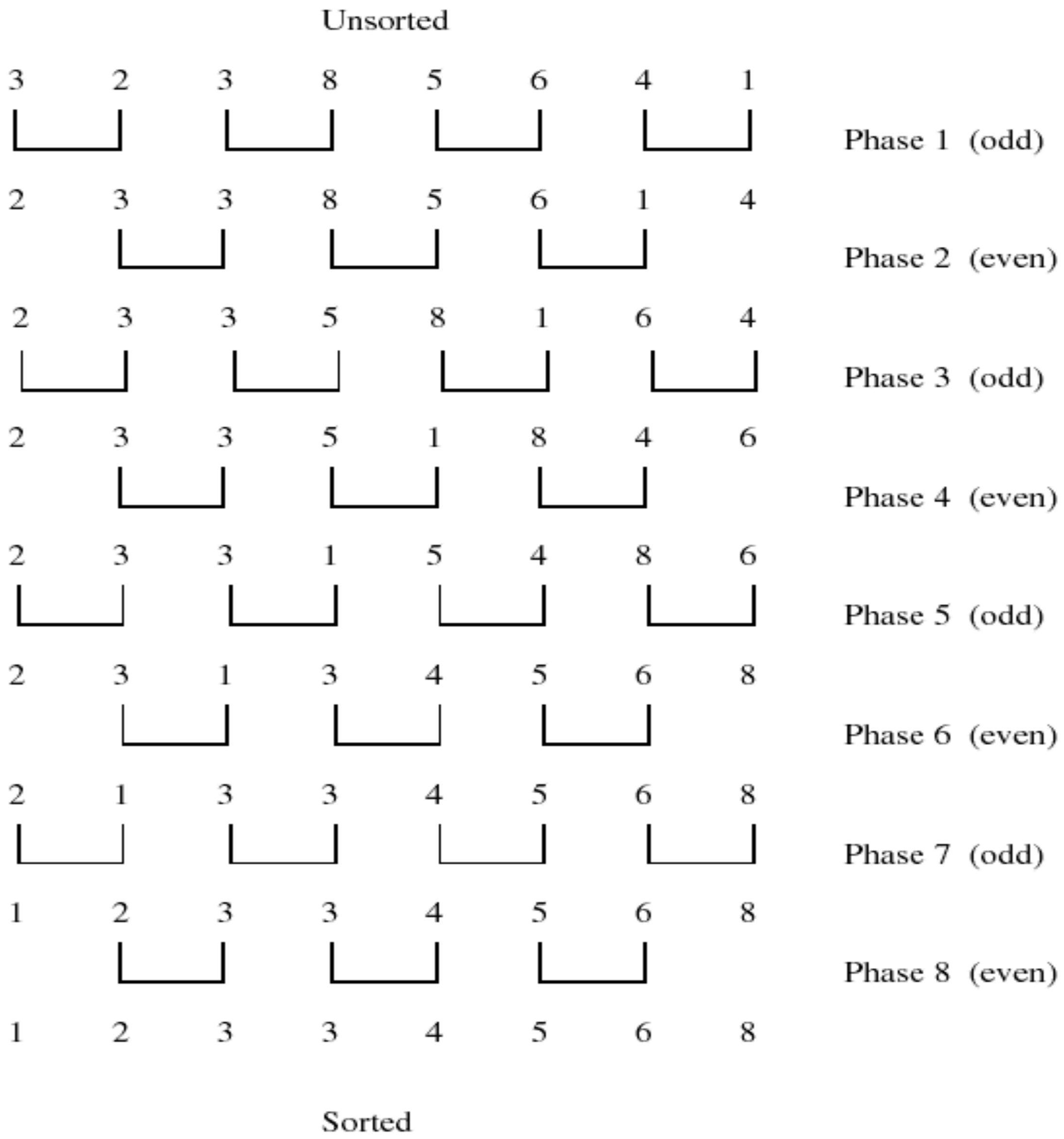
Step 3

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



Unsorted

3 2 3 8 5 6 4 1

Phase 1 (odd)

2 3 3 8 5 6 1 4

Phase 2 (even)

2 3 3 5 8 1 6 4

Phase 3 (odd)

2 3 3 5 1 8 4 6

Phase 4 (even)

2 3 3 1 5 4 8 6

Phase 5 (odd)

2 3 1 3 4 5 6 8

Phase 6 (even)

2 1 3 3 4 5 6 8

Phase 7 (odd)

1 2 3 3 4 5 6 8

Phase 8 (even)

1 2 3 3 4 5 6 8

Sorted

Parallel Odd-Even Transposition Sort on Ring

```
procedure ODD-EVEN PAR(n)
begin
    id := process's label
    for i := 1 to n do
    begin
        if i is odd then
            if id is odd then
                compare-exchange min(id + 1);
            else
                compare-exchange max(id - 1);
        if i is even then
            if id is even then
                compare-exchange min(id + 1);
            else
                compare-exchange max(id - 1);
    end for
end ODD-EVEN PAR
```

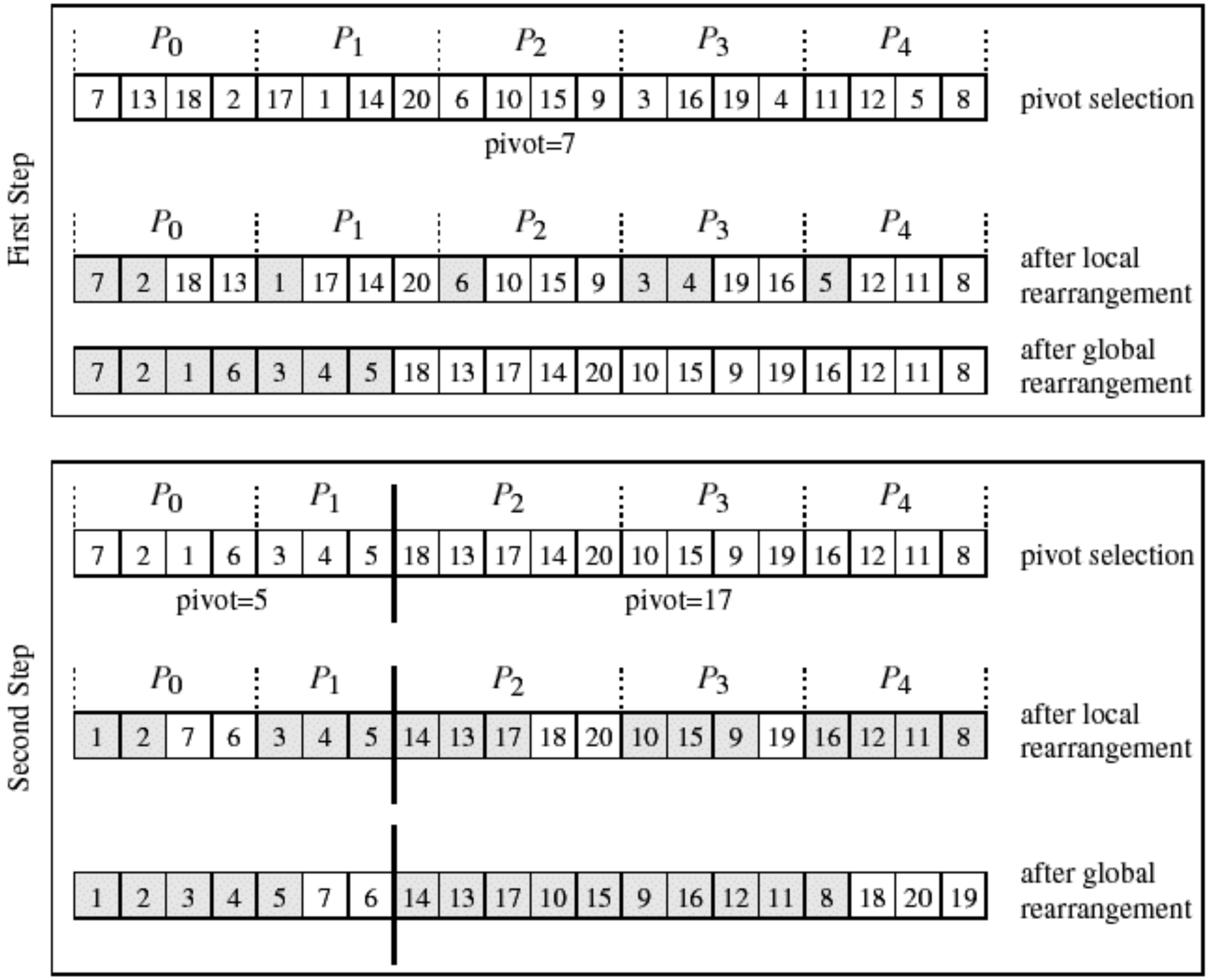
Quicksort

- *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

```
procedure QUICKSORT ($A, q, r$)
begin
    if $q < r$ then
    begin
        $x := A[q]$;
        $s := q$;
        for $i := q + 1$ to $r$ do
            if $A[i] \leq x$ then
            begin
                $s := s + 1$;
                swap($A[s], A[i]$);
            end if
        swap($A[q], A[s]$);
        QUICKSORT ($A, q, s$);
        QUICKSORT ($A, s + 1, r$);
    end if
end QUICKSORT
```

Parallel Shared Address Space Quicksort



First Step

pivot selection: 7 13 18 2 | 17 1 14 20 | 6 10 15 9 | 3 16 19 4 | 11 12 5 8

after local rearrangement: 7 2 18 13 | 1 17 14 20 | 6 10 15 9 | 3 4 19 16 | 5 12 11 8

after global rearrangement: 7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8

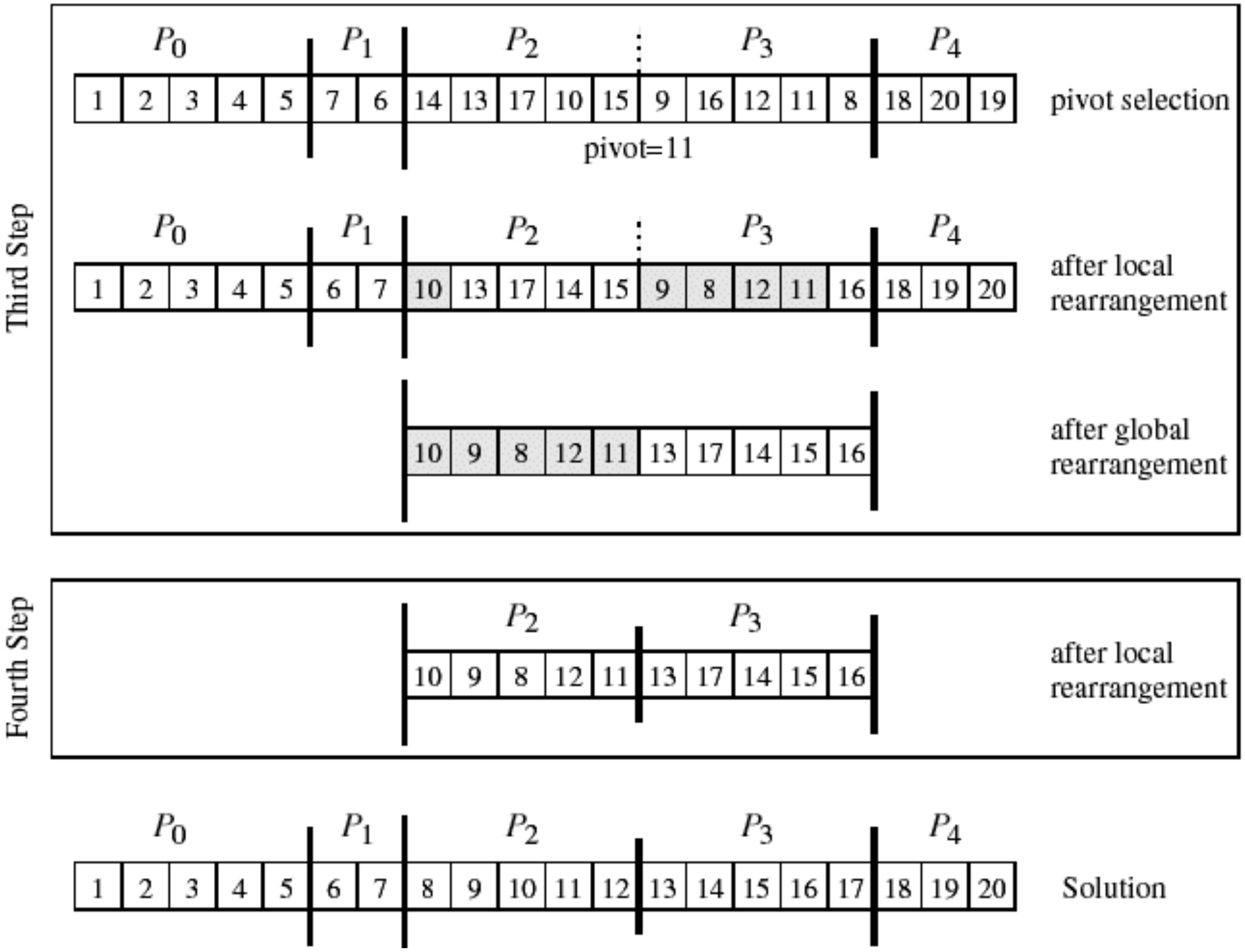
Second Step

pivot selection: 7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8

after local rearrangement: 1 2 7 6 3 4 5 14 13 17 18 20 10 15 9 19 16 12 11 8

after global rearrangement: 1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19

Parallel Shared Address Space Quicksort



Third Step

pivot selection: 1 2 3 4 5 | 7 6 | 14 13 17 10 15 ⋮ 9 16 12 11 8 | 18 20 19

after local rearrangement: 1 2 3 4 5 | 6 7 | 10 13 17 14 15 ⋮ 9 8 12 11 16 | 18 19 20

after global rearrangement: | 10 9 8 12 11 13 17 14 15 16 |

Fourth Step

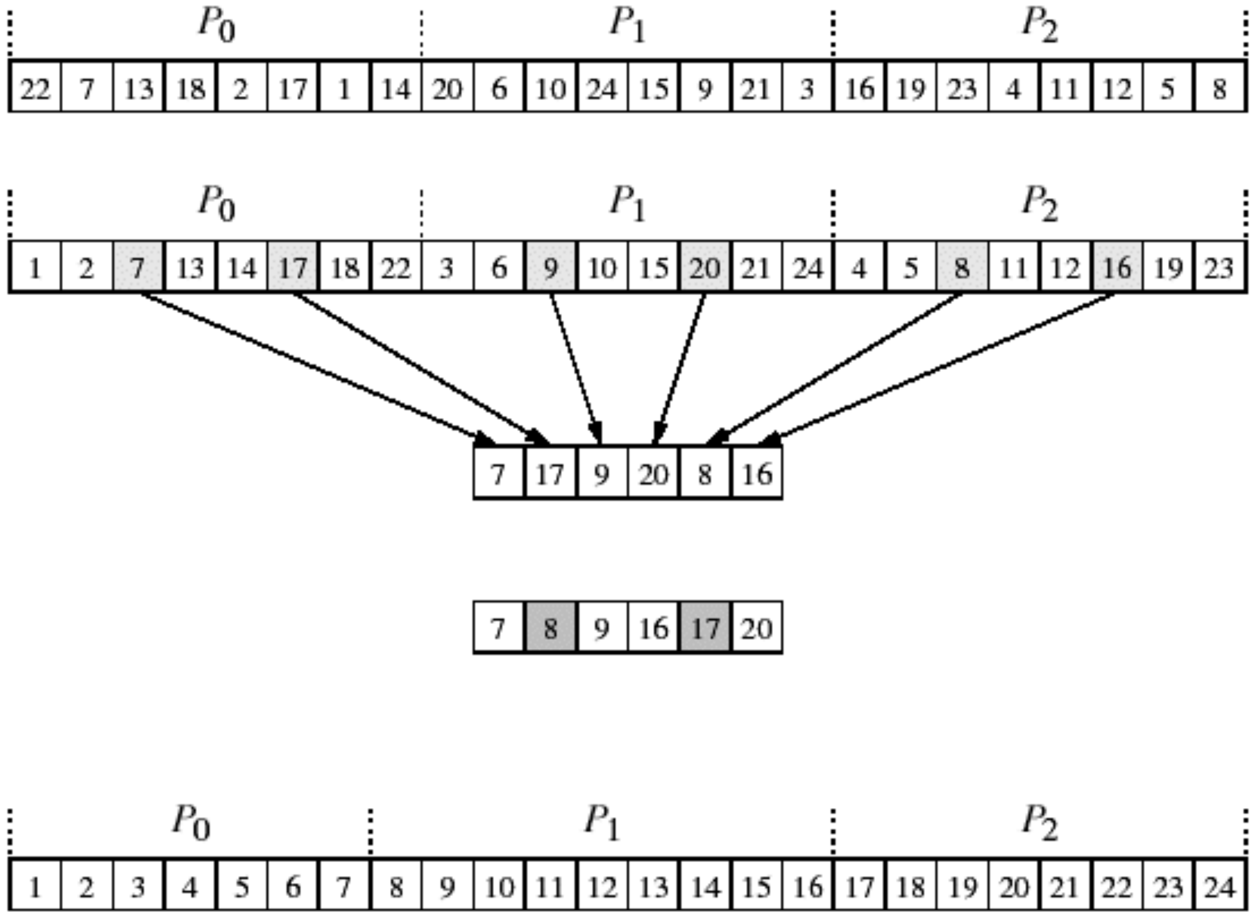
after local rearrangement: | 10 9 8 12 11 | 13 17 14 15 16 |

Solution: 1 2 3 4 5 | 6 7 | 8 9 10 11 12 | 13 14 15 16 17 | 18 19 20

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

P_0	22	7	13	18	2	17	1	14
P_1	20	6	10	24	15	9	21	3
P_2	16	19	23	4	11	12	5	8

Local sort & sample selection

P_0	1	2	**7**	13	14	**17**	18	22
P_1	3	6	**9**	10	15	**20**	21	24
P_2	4	5	**8**	11	12	**16**	19	23

Sample combining

7	17	9	20	8	16

Global splitter selection

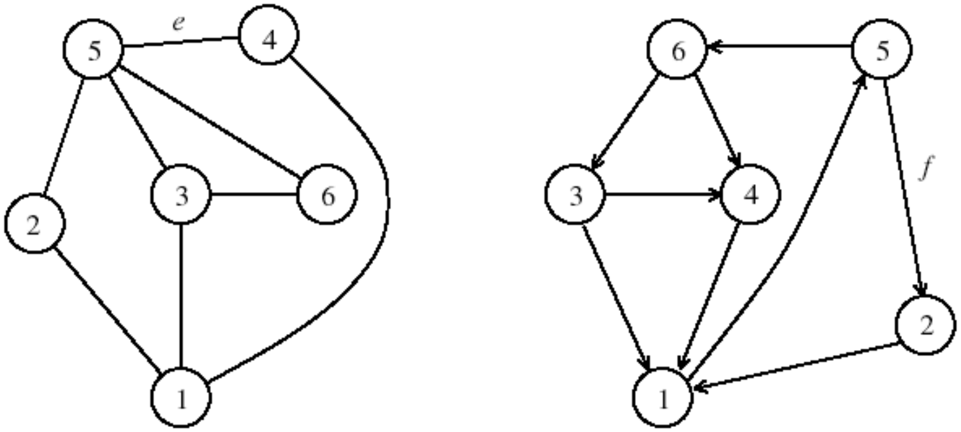
7	**8**	9	16	**17**	20

Final element assignment

P_0	1	2	3	4	5	6	7		
P_1	8	9	10	11	12	13	14	15	16
P_2	17	18	19	20	21	22	23	24	

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

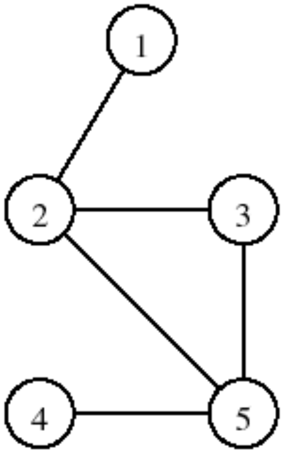


Graph Terminology

- 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 x n* array *A* such that
  - $A_{i,j} = 1$ if $(v_i,v_j) \in E$; 0 otherwise
  - Can be modified for weighted graphs (∞ is no edge)
  - Can represent as *adjacency lists*

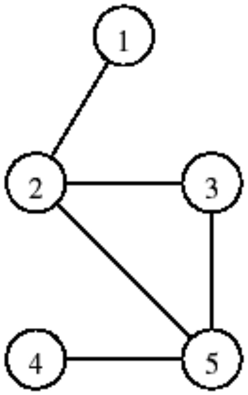
Graph Representations

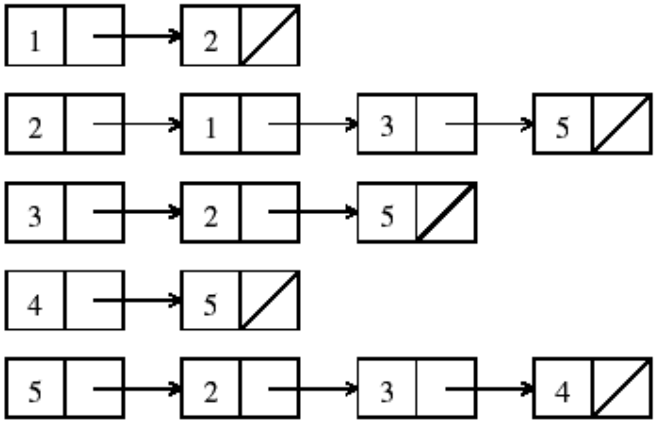
- Adjacency matrix



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

- Adjacency list





Minimum Spanning Tree

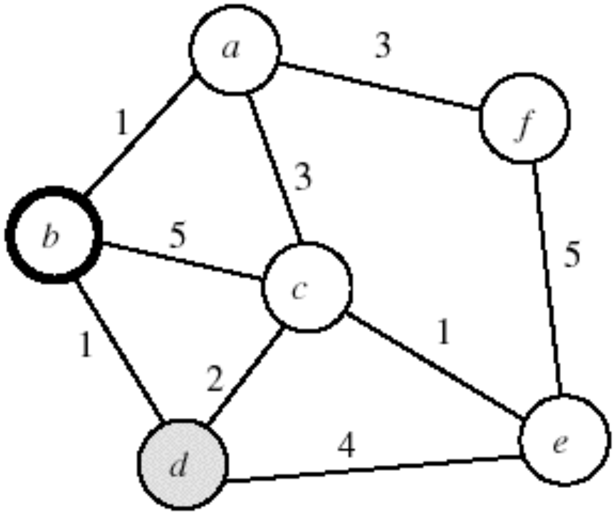
- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*
- The *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight
- Prim's algorithm can be used
  - Greedy algorithm
  - Selects an arbitrary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

Prim's Minimum Spanning Tree Algorithm

```
procedure PRIM MST($V, E, w, r$)
begin
    $VT := \{r\}$;
    $d[r] := 0$;
    for all $v \in (V - VT)$ do
        if edge $(r, v)$ exists set $d[v] := w(r, v)$;
        else set $d[v] := \infty$;
    while $VT \neq V$ do
    begin
        find a vertex $u$ such that $d[u] := \min\{d[v] | v \in (V - VT)\}$;
        $VT := VT \cup \{u\}$;
        for all $v \in (V - VT)$ do                                  *
            $d[v] := \min\{d[v], w(u, v)\}$;
    endwhile
end PRIM MST
```

Example: Prim's MST Algorithm

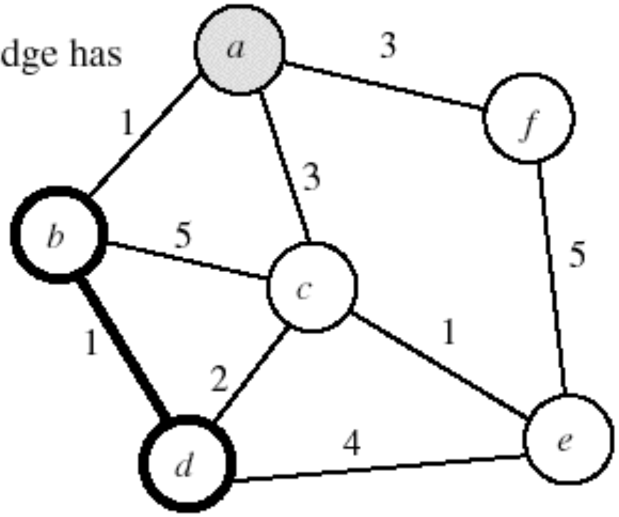
(a) Original graph



	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \left[\begin{array}{cccccc} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{array}\right]$$

(b) After the first edge has been selected

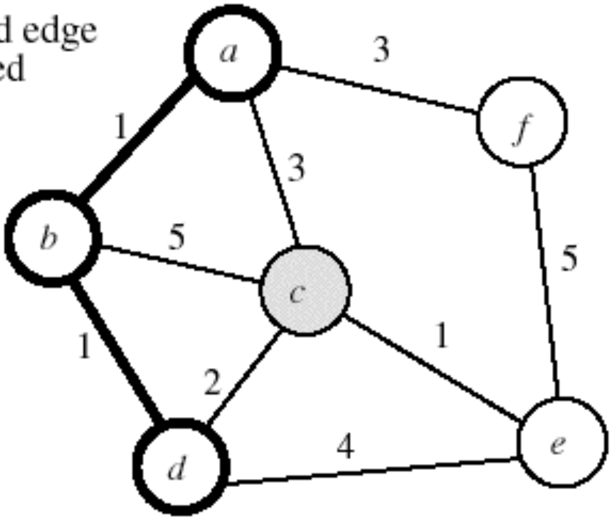


	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \left[\begin{array}{cccccc} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{array}\right]$$

Example: Primʼs MST Algorithm

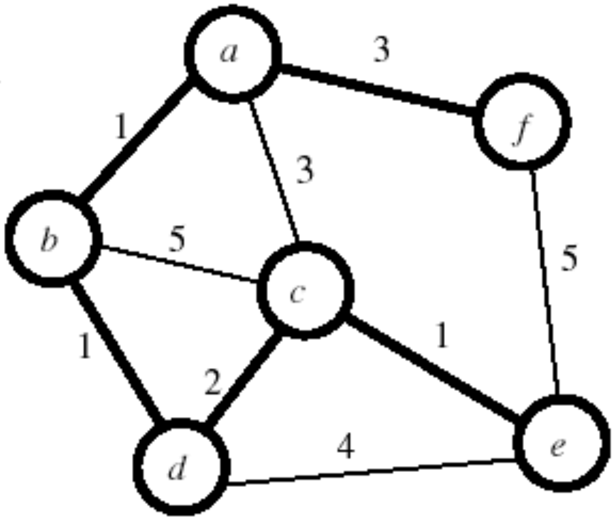
(c) After the second edge has been selected



	a	b	c	d	e	f
d[]	1	0	2	1	4	3

$$\begin{array}{c|cccccc|} a & 0 & 1 & 3 & \infty & \infty & 3 \\ b & 1 & 0 & 5 & 1 & \infty & \infty \\ c & 3 & 5 & 0 & 2 & 1 & \infty \\ d & \infty & 1 & 2 & 0 & 4 & \infty \\ e & \infty & \infty & 1 & 4 & 0 & 5 \\ f & 2 & \infty & \infty & \infty & 5 & 0 \end{array}$$

(d) Final minimum spanning tree



	a	b	c	d	e	f
d[]	1	0	2	1	1	3

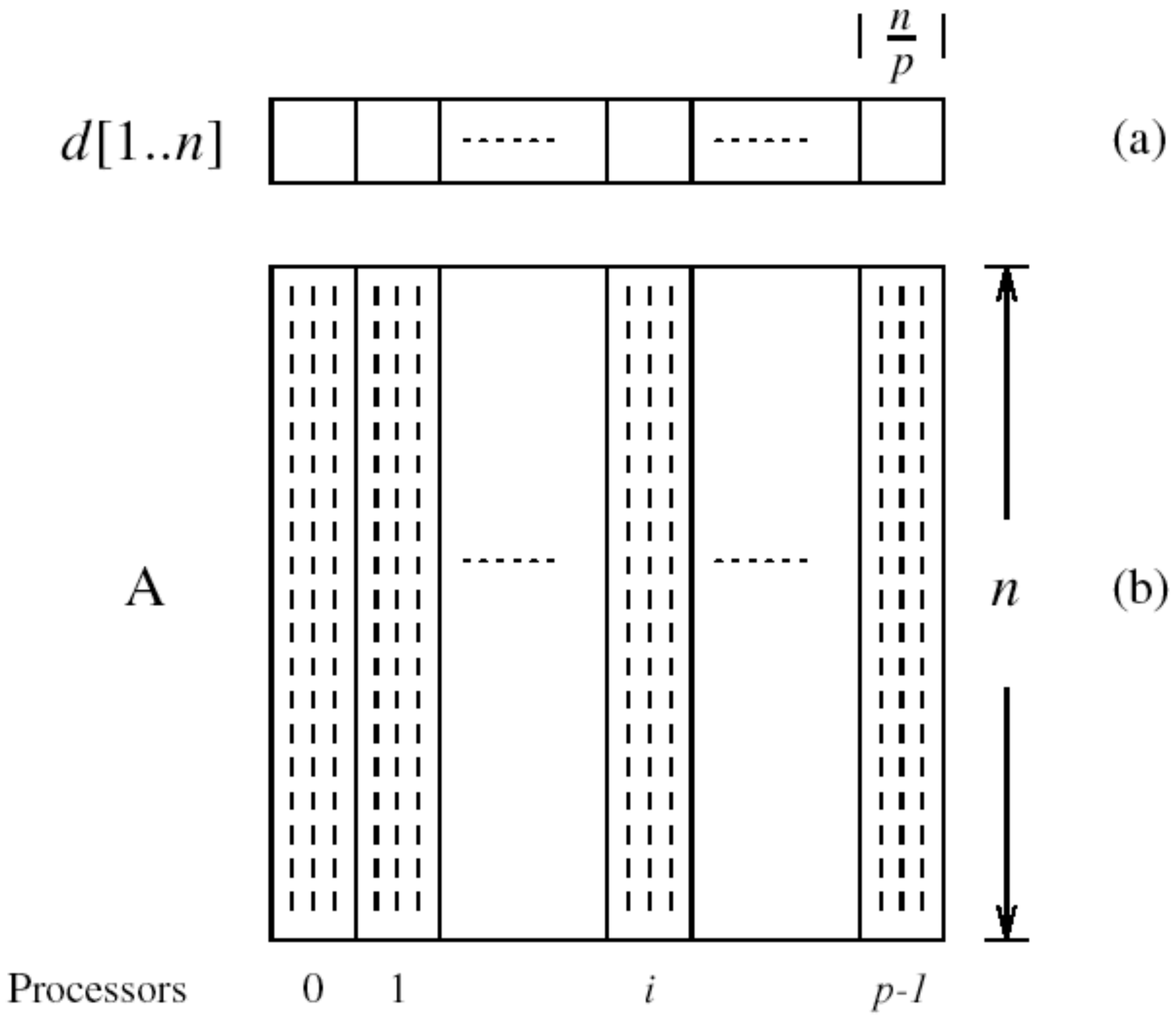
$$\begin{array}{c|cccccc|} a & 0 & 1 & 3 & \infty & \infty & 3 \\ b & 1 & 0 & 5 & 1 & \infty & \infty \\ c & 3 & 5 & 0 & 2 & 1 & \infty \\ d & \infty & 1 & 2 & 0 & 4 & \infty \\ e & \infty & \infty & 1 & 4 & 0 & 5 \\ f & 2 & \infty & \infty & \infty & 5 & 0 \end{array}$$

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, $i=0,\ldots,p-1$
- 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



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

```
procedure DIJKSTRA SINGLE SOURCE SP($V, E, w, s$)
begin
    $V_T := \{s\}$;
    for all $v \in (V - V_T)$ do
        if $(s, v)$ exists set $l[v] := w(s, v)$;
        else set $l[v] := \infty$;
    while $V_T \neq V$ do
    begin
        find a vertex $u$ such that $l[u] := \min\{l[v] | v \in (V - V_T)\}$;
    $VT := V_T \cup \{u\}$;
    for all $v \in (V - V_T)$ do
        $l[v] := \min\{l[v], l[u] + w(u, v)\}$;
    endwhile
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 x 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

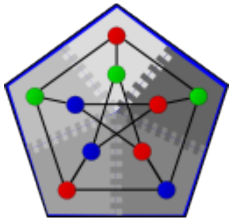
Floyd’s All-Pairs Shortest Paths Algorithm

```
procedure FLOYD ALL PAIRS SP($A$)
begin
    $D^{(0)} = A$;
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j} , d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}$ ;
end FLOYD ALL PAIRS SP
```

Parallel Floyd's Algorithm

```
procedure FLOYD ALL PAIRS PARALLEL ($A$)
begin
    $D^{(0)} = A$;
    for $k := 1$ to $n$ do
        forall $P_{i,j}$, where $i, j \leq n$, do in parallel
            $d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}$;
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)

D. Gregor and A. Lumsdaine, "The Parallel BGL: A Generic Library for Distributed Graph Computations, Parallel Object-Oriented Scientific Computing (POOSC), July 2005.

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

Original BGL Summary

- 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

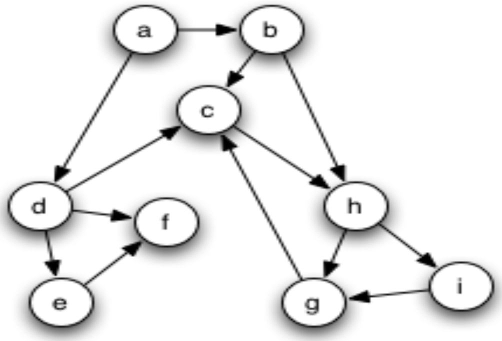
BGL-Python

- 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)`

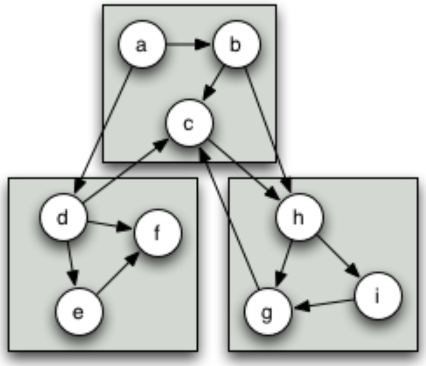
The Parallel BGL

- 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