AMS 250: An Introduction to High Performance Computing

Parallel Algorithms

Shawfeng Dong

shaw@ucsc.edu

(831) 459-2725

Astronomy & Astrophysics

University of California, Santa Cruz

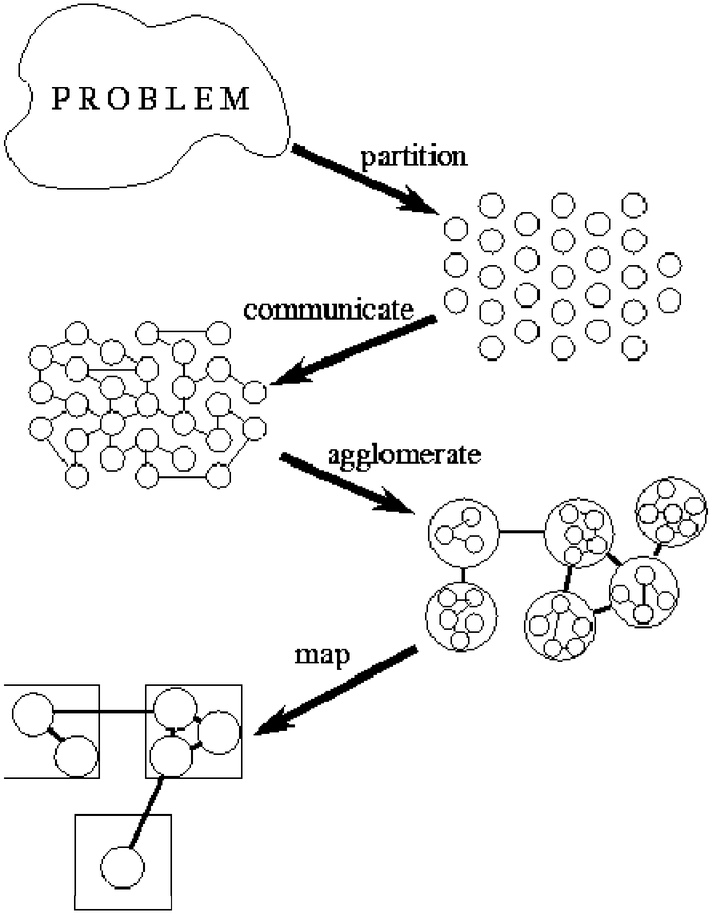
Outline

- Methodical Design – PCAM
- Types of Parallel Programs
- Dense Matrix Algorithms
- Sorting Algorithms
- Graph Algorithms

Methodological Design

PCAM

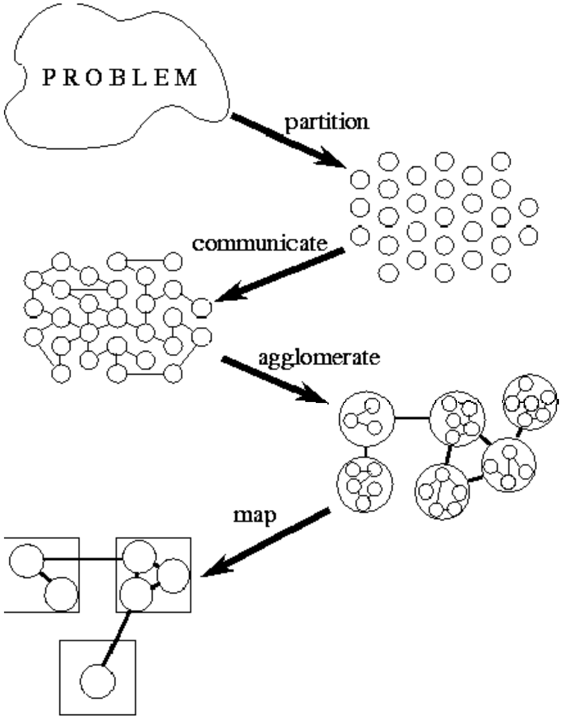
- Partition
  - – Task/data decomposition
- Communication
  - – Task execution coordination
- Agglomeration
  - – Evaluation of the structure
- Mapping
  - – Resource assignment



Designing and Building Parallel Programs, by *Ian Foster*: http://www.mcs.anl.gov/~itf/dbpp/text/book.html

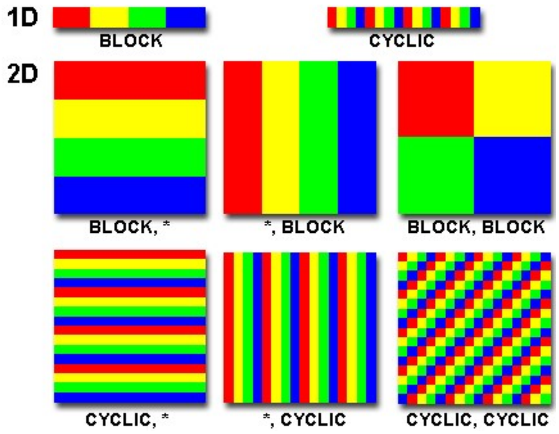
Partition

- Partition stage is intended to expose opportunities for parallel execution
- The focus is on defining large number of small task to yield a *fine-grained* decomposition of the problem
- A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which the computation operates
- *Domain decomposition* focuses on data
- *Functional decomposition* focuses on computation
- Mixing of domain/functional decomposition is possible

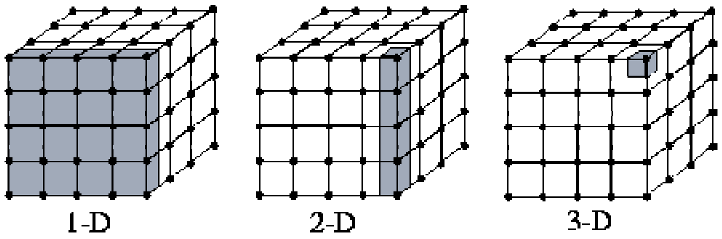


Domain Decomposition

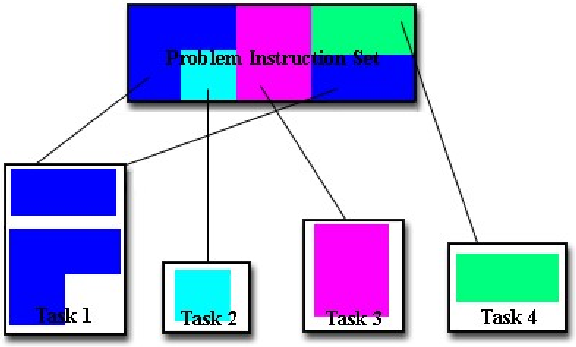
- 1D grid
- 2D grid



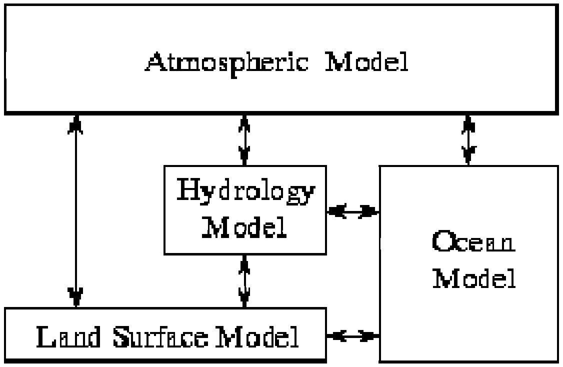
- 3D grid



Functional Decomposition



- Functional decomposition of a climate model

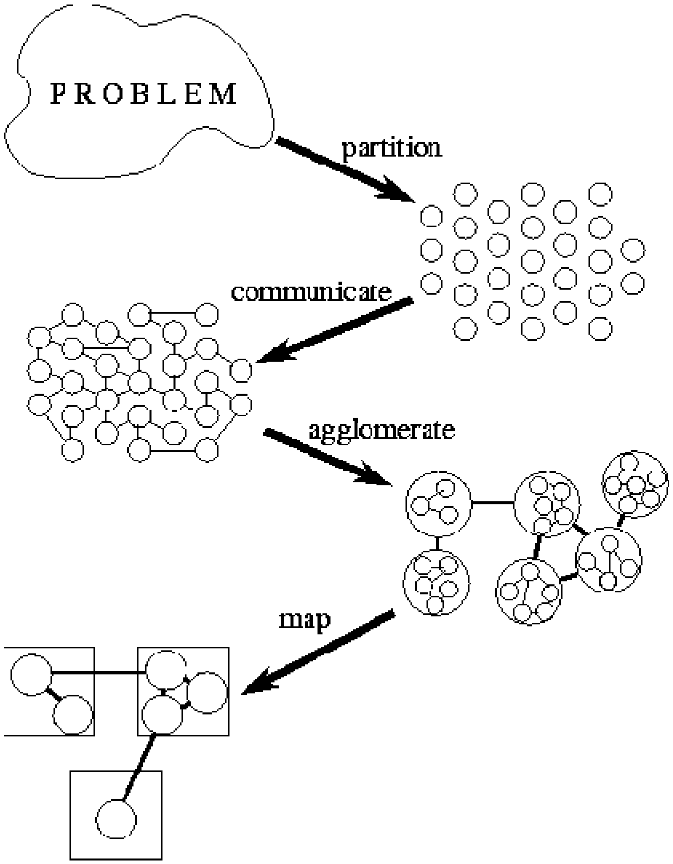


Partitioning Checklist

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. Does the number of tasks scale with problem size? If not, your parallel algorithm may not be able to solve larger problems with more processors
5. Have you identified several alternative partitions?

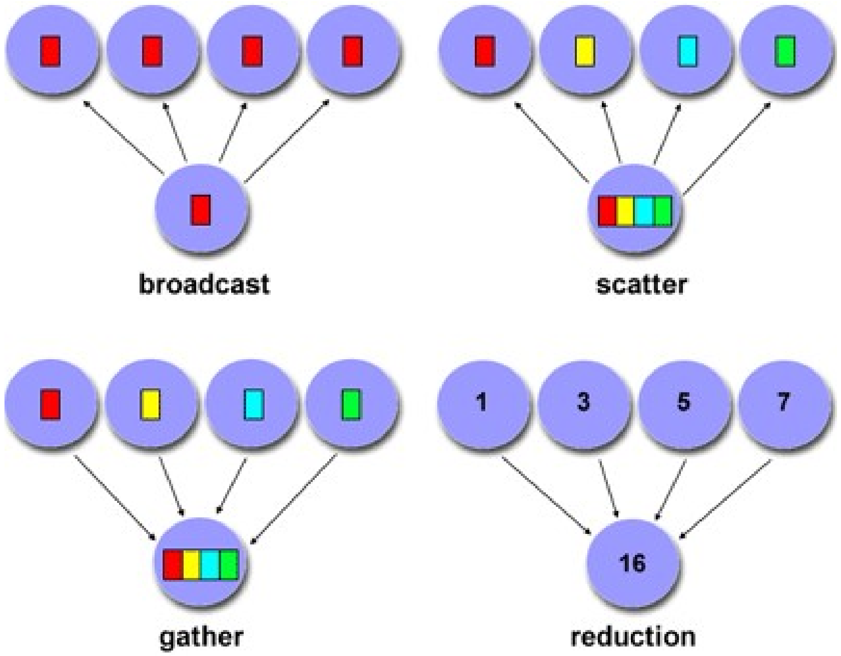
Communication

- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control
- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: degree of coordination
- Granularity and frequency of communication
  - Size of data exchange
- Think of communication as interaction and control
  - Applicable to both shared and distributed memory parallelism



Types of Communication

- Point-to-point
- Group-based
- Hierarchical
- Collective

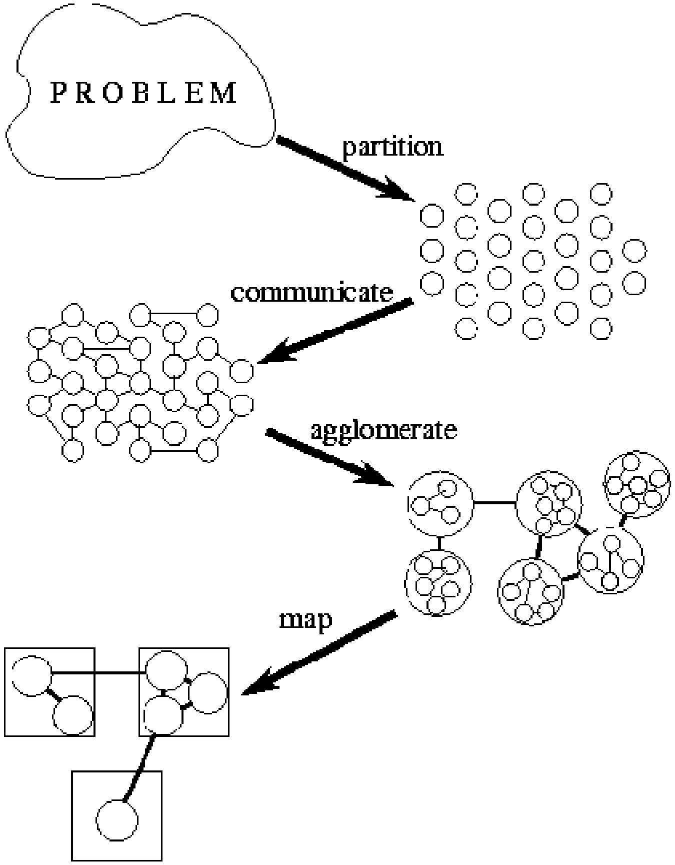


Communication Design Checklist

1. Do all tasks perform about the same number of communication operations?
   - If not, revisit your design to see whether communication operations can be distributed more equitably.
2. Does each task communicate only with a small number of neighbors?
   - If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure.
3. Are communication operations able to proceed concurrently?
   - If not, try to use divide-and-conquer techniques to uncover concurrency.
4. Is computation associated with different tasks able to proceed concurrently?
   - If not, try to reorder computation and communication operations.

Agglomeration

- In the 3rd stage, we move from the abstract towards concrete implementation
- Revisit partitioning and communication, with a view to obtaining an efficient algorithm
- Is it useful to combine, or *agglomerate* tasks?
- Is it useful to *replicate* data and/or computation?
- Three goals guiding decisions concerning agglomeration and replication:
  1. Reducing communication cost by increasing computation and communication *granularity*
  2. Retaining *flexibility* with respect to scalability and mapping decisions
  3. Reducing *software engineering* costs



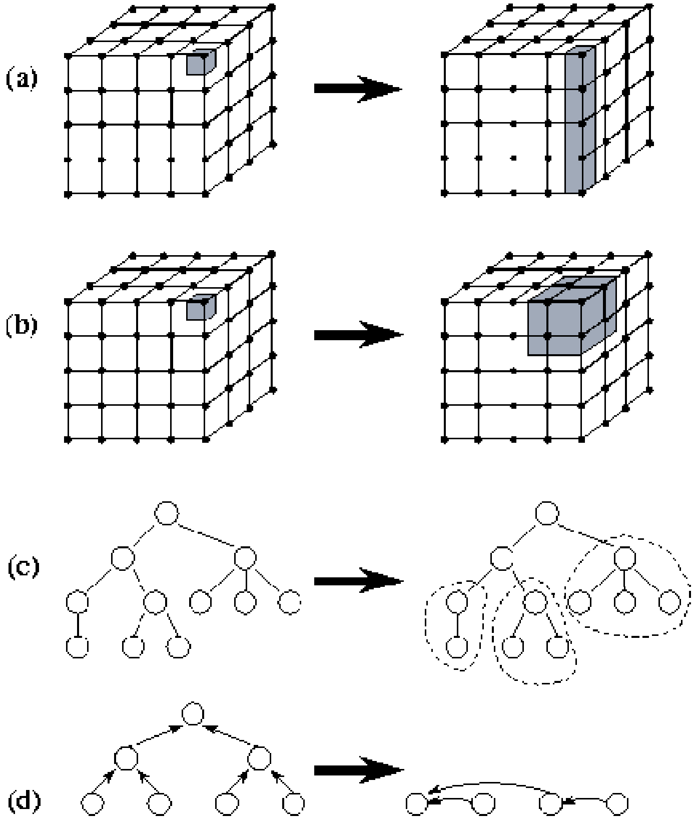
Examples of Agglomeration

a. Reducing the dimension of the decomposition from 3 to 2

b. Combing adjacent tasks to yield a 3D decomposition of higher granularity

c. Coalescing a subtrees in a divide-and-conquer structure

d. Combining nodes in a tree algorithm

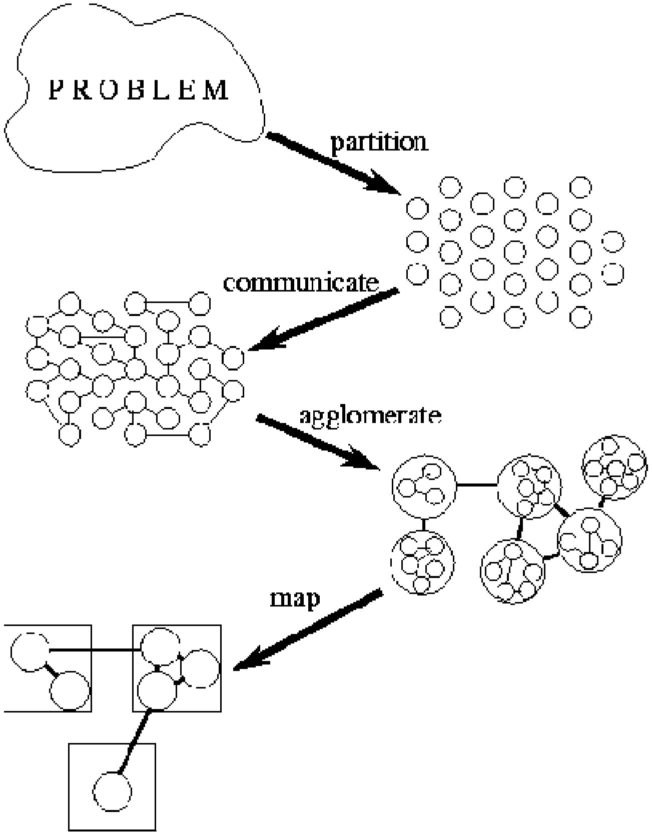


Agglomeration Design Checklist

1. Has agglomeration reduced communication costs by increasing locality?
2. If agglomeration has replicated computation, do the benefits of this replication outweigh its costs?
3. If agglomeration replicates data, does the replication compromise scalability?
4. Has agglomeration yielded tasks with similar computation and communication costs?
5. Does the number of tasks still scale with problem size?
6. If agglomeration eliminated opportunities for concurrent execution, is there still sufficient concurrency?
7. Is there room for more agglomeration?
8. If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code?

Mapping

- Specify where each task is to execute
  - Less of a concern on shared-memory systems
- Two sometimes-conflicting strategies to minimize execution time:
  1. Place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency
  2. Place tasks that communicate frequently on the *same* processor, so as to increase locality
- The mapping problem is *NP-complete*
  - Use specialized strategies, heuristics and problem classifications



Mapping Algorithms

- Load-balancing algorithms
- Data-based algorithms
  - Think of computational load with respect to amount of data being operated on
  - Assign data (i.e., work) in some known manner to balance
  - Take into account data interactions
- Task-based (task-scheduling) algorithms
  - Used when functional decomposition yields many tasks with weak locality requirements
  - Use task assignment to keep processors busy computing
  - Consider centralized and decentralize schemes

Mapping Design Checklist

1. If considering an SPMD design for a complex problem, have you also considered an algorithm based on dynamic task creation and deletion?
2. If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm?
3. If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck?
4. If using a dynamic load-balancing scheme, have you evaluated the relative costs of different strategies?
5. If using probabilistic or cyclic methods, do you have a large enough number of tasks to ensure reasonable load balance?

Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things
- Parallel execution models
  - Data parallel
  - Pipelining (Producer-Consumer)
  - Task graph
  - Work pool
  - Master-Worker

Data Parallel

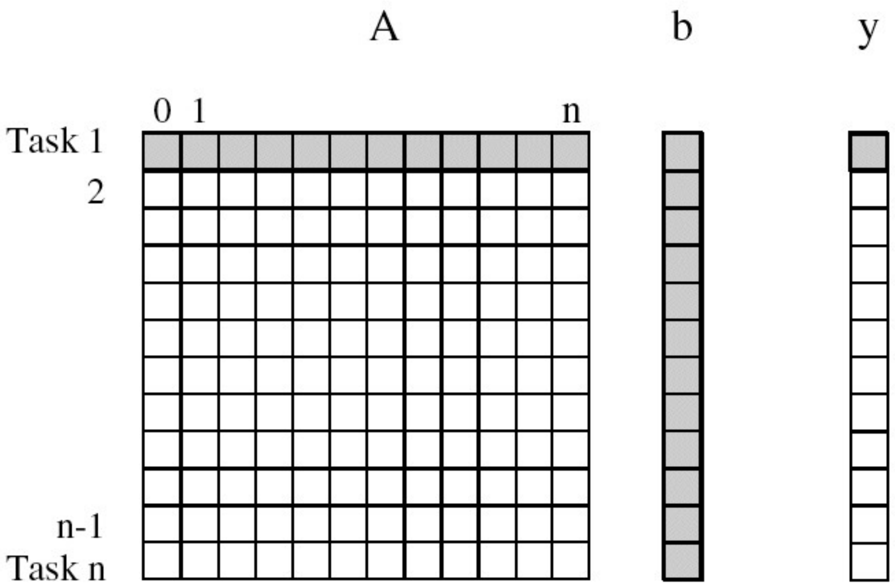
- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on different data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- There may be interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
- Single Program Multiple Data (SPMD)
  - Convenient way to implement data parallel computation
  - More associated with distributed memory parallel execution

Matrix - Vector Multiplication

- A x b = y
- Allocate tasks to rows of A

  $y[i] = \sum_{j} A[i,j]*b[j]$

- Dependencies?
- Speedup?
- Computing each element of y can be done independently



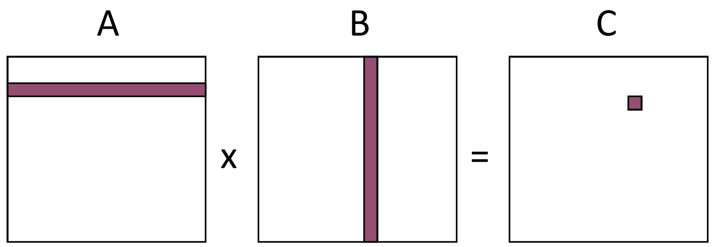
Matrix Multiplication

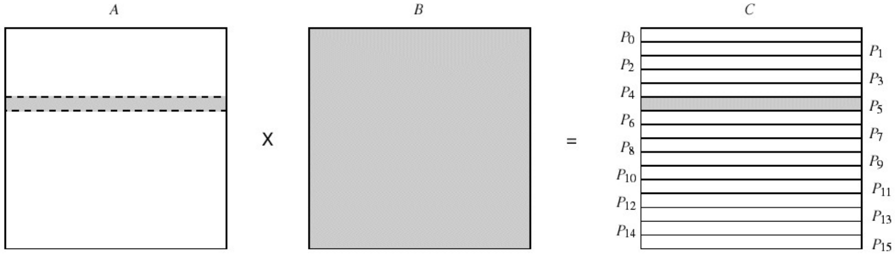
- A x B = C
- A[i,:] • B[:,j] = C[i,j]

- Row partitioning
  - *N* tasks

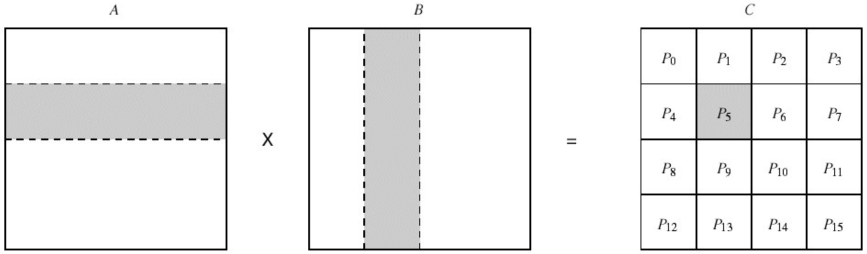
- Block partitioning
  - N^2/m^2 tasks

- Shading shows data sharing in matrix *B*





(a)



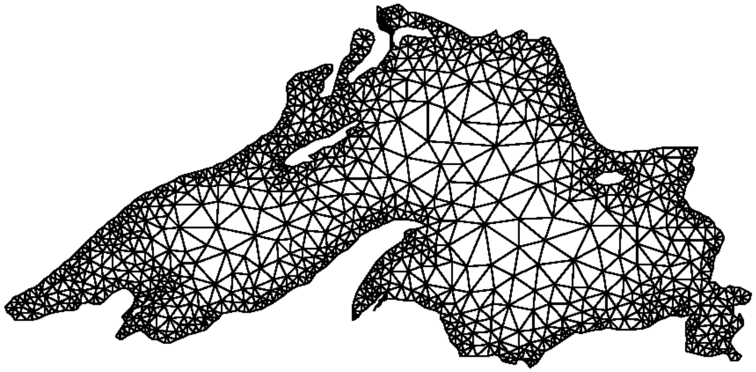
(b)

Granularity of Task and Data Decompositions

- Granularity can be with respect to tasks and data
- Task granularity
  - Equivalent to choosing the number of tasks
  - Fine-grained decomposition results in large number of tasks
  - Coarse-grained decomposition has smaller number of tasks
  - Translates to data granularity after number of tasks chosen
    - consider matrix multiplication
- Data granularity
  - Think of computational load with respect to amount of data being operated on
  - Relative to data as a whole
  - Decomposition decisions based on input, output, input-output, or intermediate data

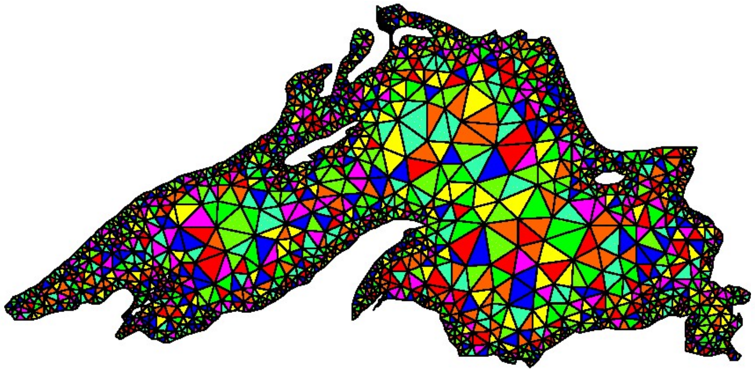
Mesh Allocation to Processors

- Mesh model of Lake Superior
- How to assign mesh elements to processors

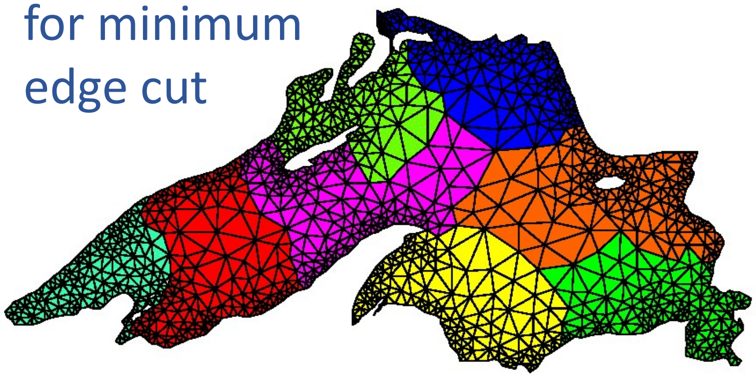


- Distribution onto 8 processors:

randomly



graph partitioning for minimum edge cut

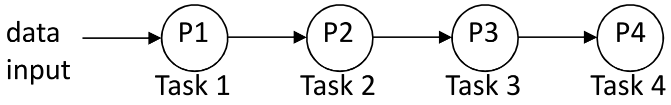


Pipeline Model

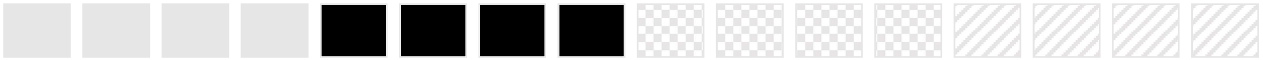
- Stream of data operated on by succession of tasks

  Task 1 ◻ Task 2 ■ Task 3 ▦ Task 4 ▨

  – Tasks are assigned to processors

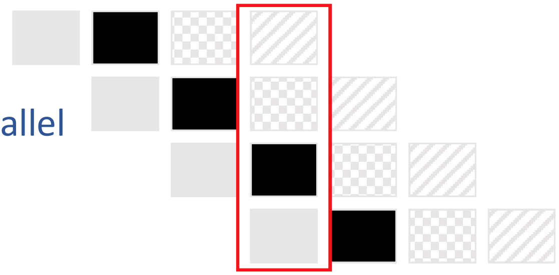


- Consider *N* data units
  - Sequential



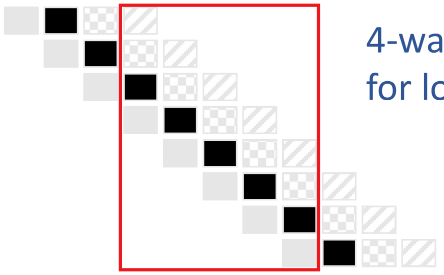
  - Parallel (each task assigned to a processor)

4 data units



4-way parallel

8 data units

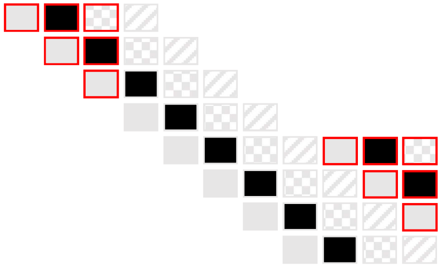


4-way parallel, but for longer time

Pipeline Performance

- *N* data and *T* tasks
- Each task takes unit time *t*
- Sequential time = *N*T*t*
- Parallel pipeline time = *start + finish + (N-T+1)* t*

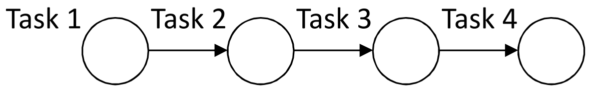
  = O(N)   (for N>>T)



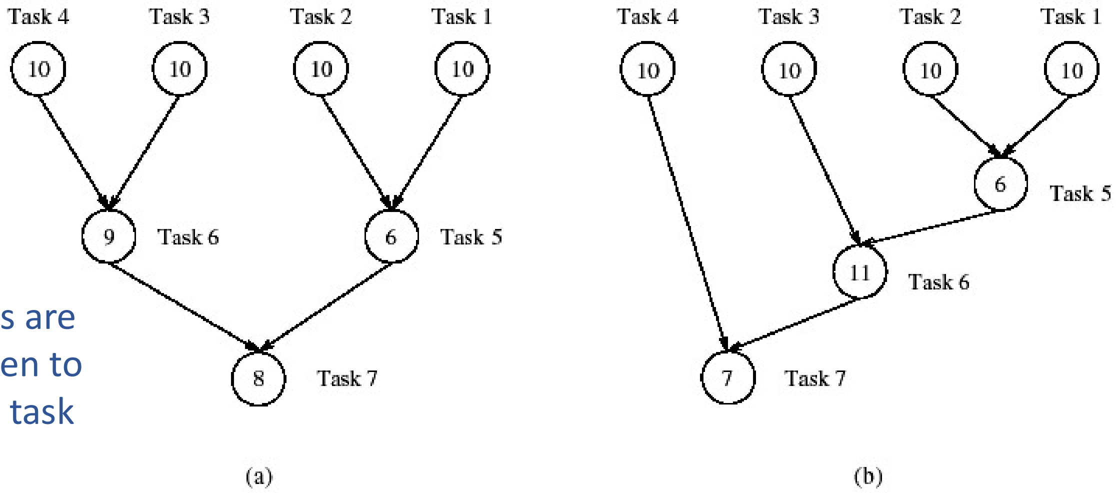
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Shorter tasks to do
- Pipeline computation is a special form of *producer-consumer* parallelism
  - output of producer tasks = input of consumer tasks

Tasks Graphs

- Computations in any parallel algorithms can be viewed as a task dependency graph
- Task dependency graphs can be non-trivial
  - Pipeline



  - Arbitrary (represents the algorithm dependencies)



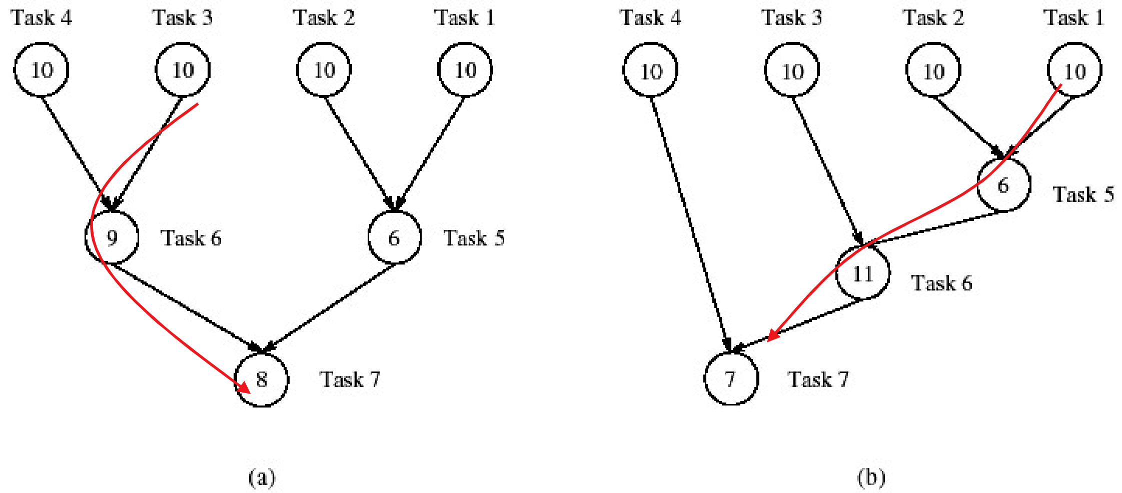
(a)

(b)

Numbers are time taken to perform task

Task Graph Performance

- Determined by the *critical path* (*span*)
    - Sequence of dependent tasks that takes the longest time



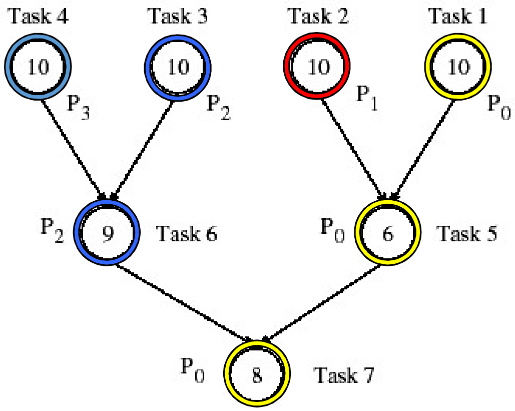
Min time = 27

Min time = 34

- *Critical path length* bounds parallel execution time

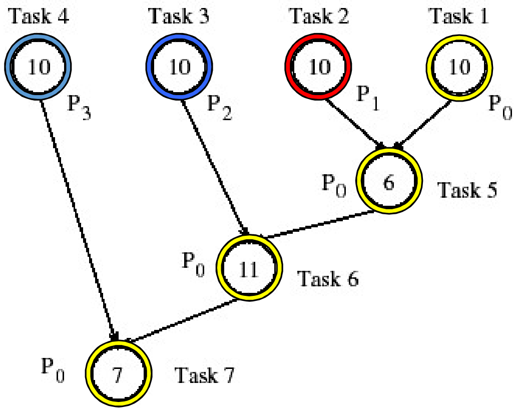
Task Assignment to Processors

- Given a set of tasks and number of processors
- How to assign (*map*) tasks to processors?
- Should take dependencies into account
- Task mapping will determine execution time



(a)

Total time = 27

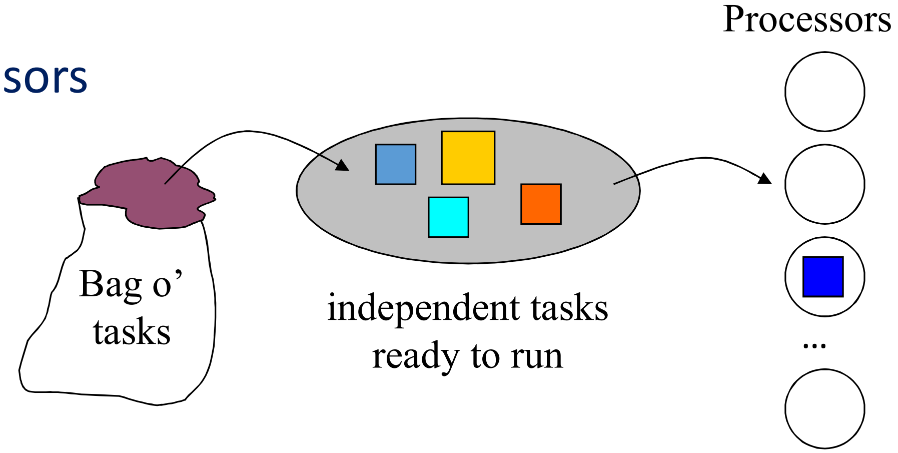


(b)

Total time = 33

Bag o’ Tasks Model and Worker Pool

- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o’ Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisfied, it is moved to a ready-to-run queue
  - Scheduler assigns a task to a free processor
- Dynamic approach that is effective for load balancing



Master-Worker Parallelism

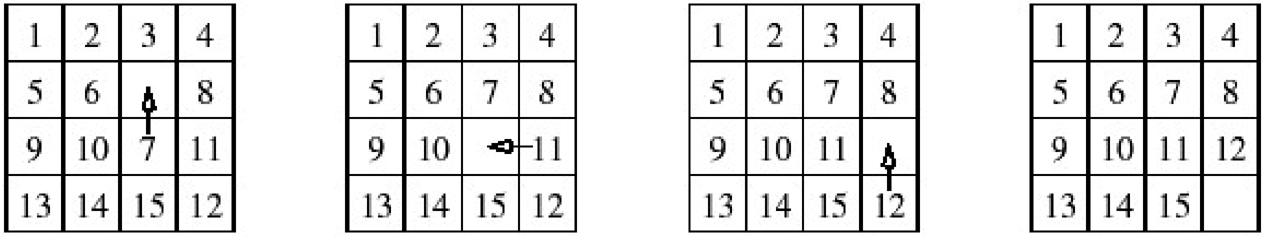
- One or more master processes generate work
- Masters allocate work to worker processes
- Workers are idle if they have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but must be told to do so
- Master may become the bottleneck if not careful

Search-Based (Exploratory) Decomposition

- 15-puzzle problem:

  https://en.wikipedia.org/wiki/15_puzzle

- 15 tiles numbered 1 through 15 placed in 4x4 grid
  - Blank tile located somewhere in grid
  - Initial configuration is out of order
  - Find shortest sequence of moves to put in order

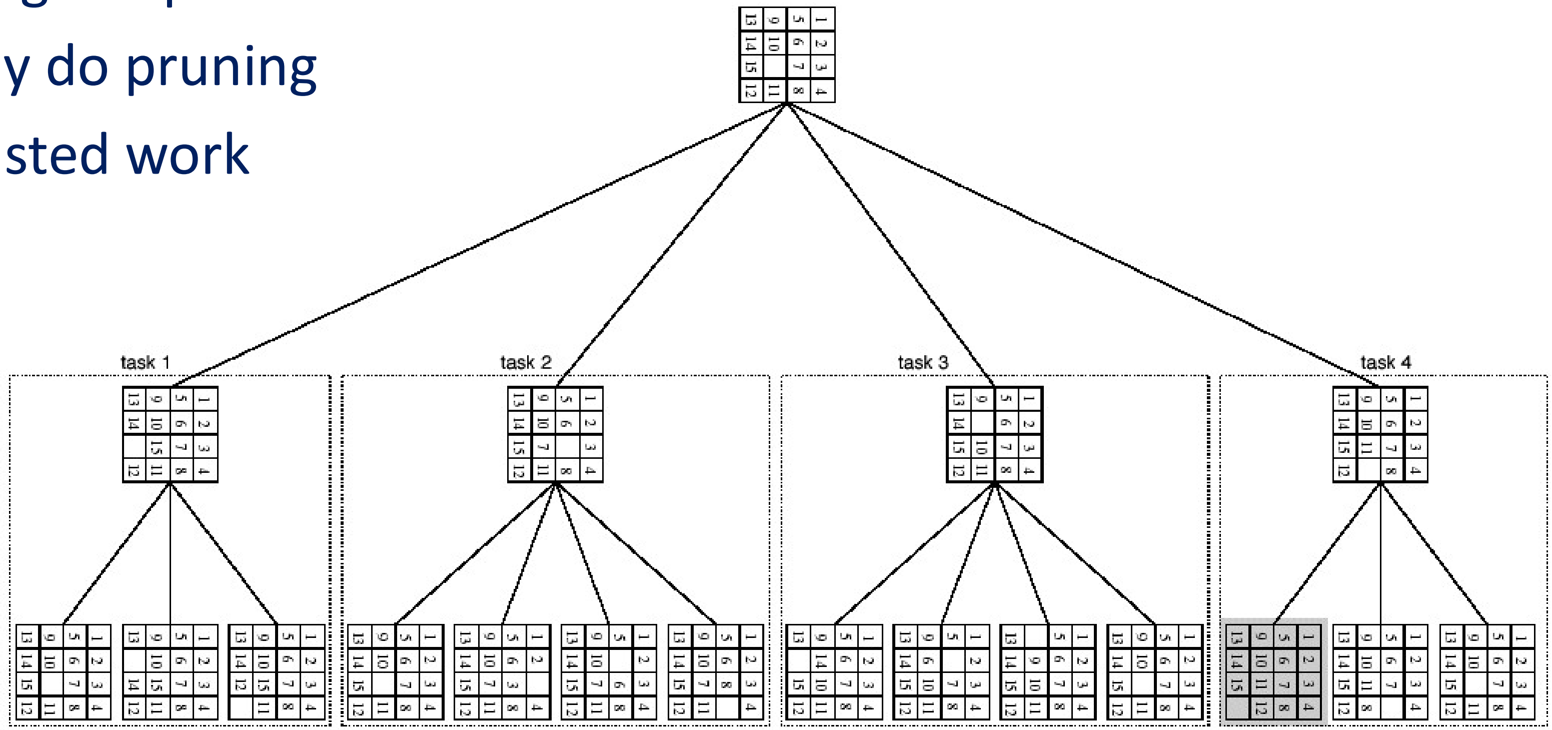


(a) (b) (c) (d)

- Sequential search across space of solutions
  - May involve some heuristics

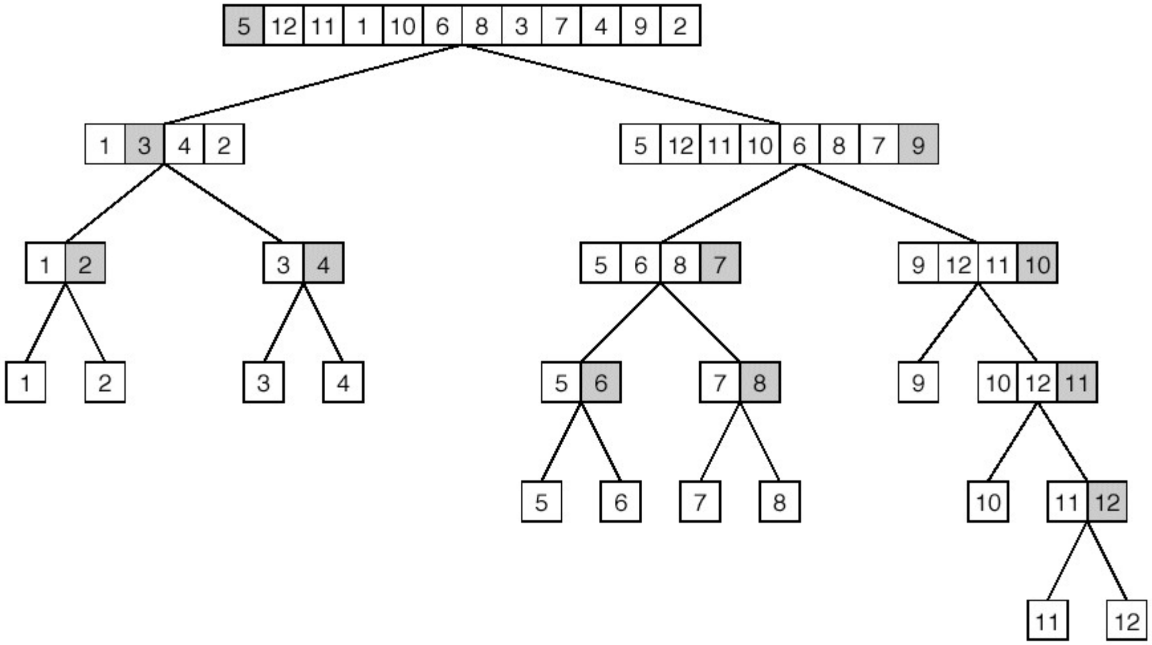
Parallelizing the 15-Puzzle Problem

- Enumerate move choices at each stage
- Assign to processors
- May do pruning
- Wasted work



Divide-and-Conquer Parallelism

- Break problem up in orderly manner into smaller, more manageable chunks and solve
- Quicksort example



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

- $Ax=b$:

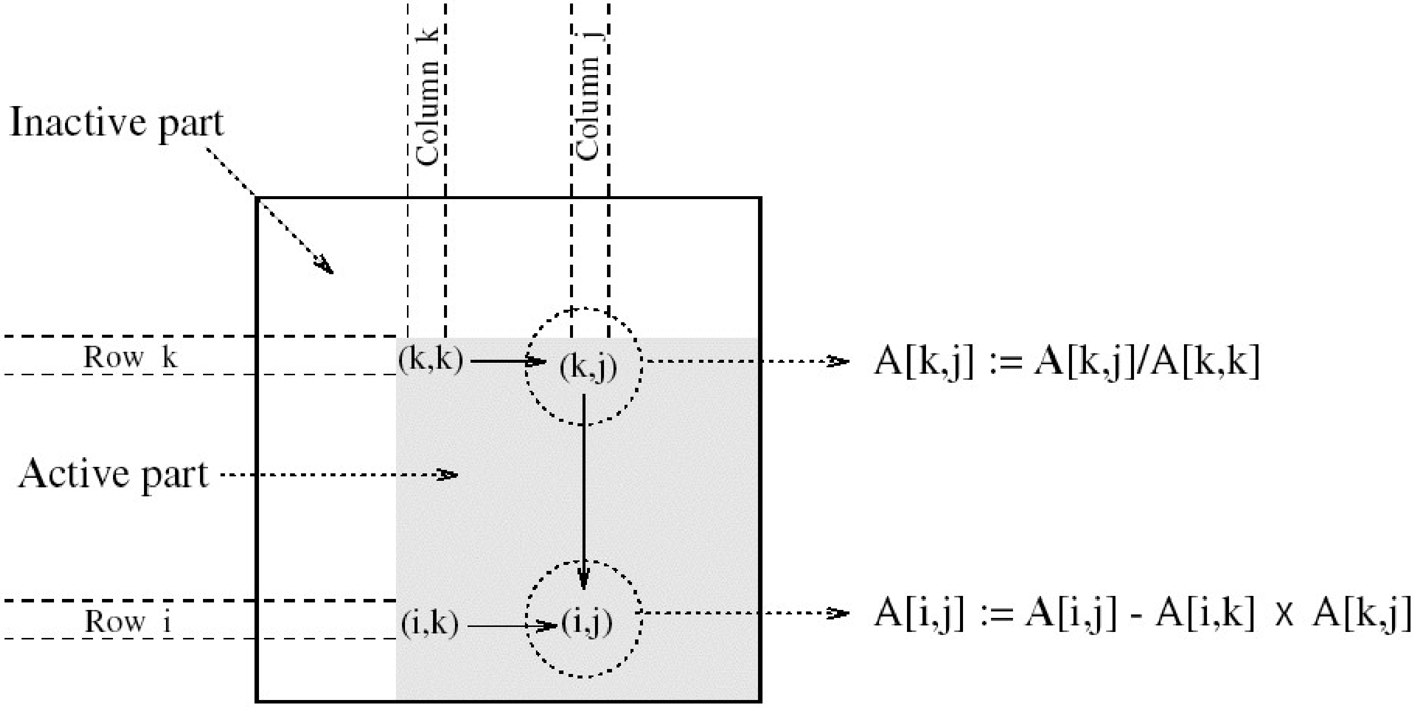
$$
\begin{array}{llll}
a_{0,0}x_0 & + a_{0,1}x_1 & + \ldots + & a_{0,n-1}x_{n-1} & = b_0 \\
a_{1,0}x_0 & + a_{1,1}x_1 & + \ldots + & a_{1,n-1}x_{n-1} & = b_1 \\
\ldots & & & & \\
A_{n-1,0}x_0 & + a_{n-1,1}x_1 & + \ldots + & a_{n-1,n-1}x_{n-1} & = b_{n-1}
\end{array}
$$

- 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.                 $A[i, j]$ := $A[i, j] - A[i, k] \times A[k, j]$; /* Elimination step */
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



Inactive part

Column k

Column j

Row k

(k,k)

(k,j)

$A[k,j] := A[k,j]/A[k,k]$

Active part

Row i

(i,k)

(i,j)

$A[i,j] := A[i,j] - A[i,k] \times A[k,j]$

$5x + 3y = 22$
$8x + 2y = 13$

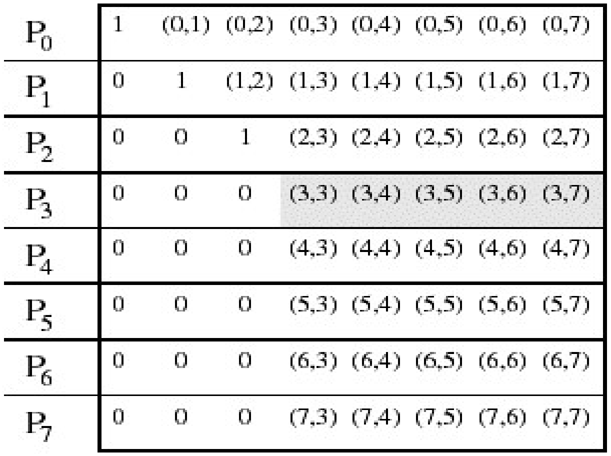
⇨

$x = (22 - 3y) / 5$
$8(22 - 3y)/5 + 2y = 13$

⇨

$x = (22 - 3y) / 5$
$y = (13 - 176/5) / (24/5 + 2)$

Rowwise Partitioning on Eight Processes

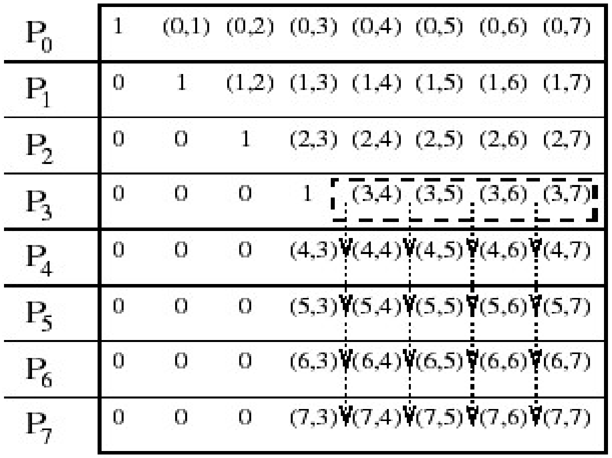


P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0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

(a) Computation:

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

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

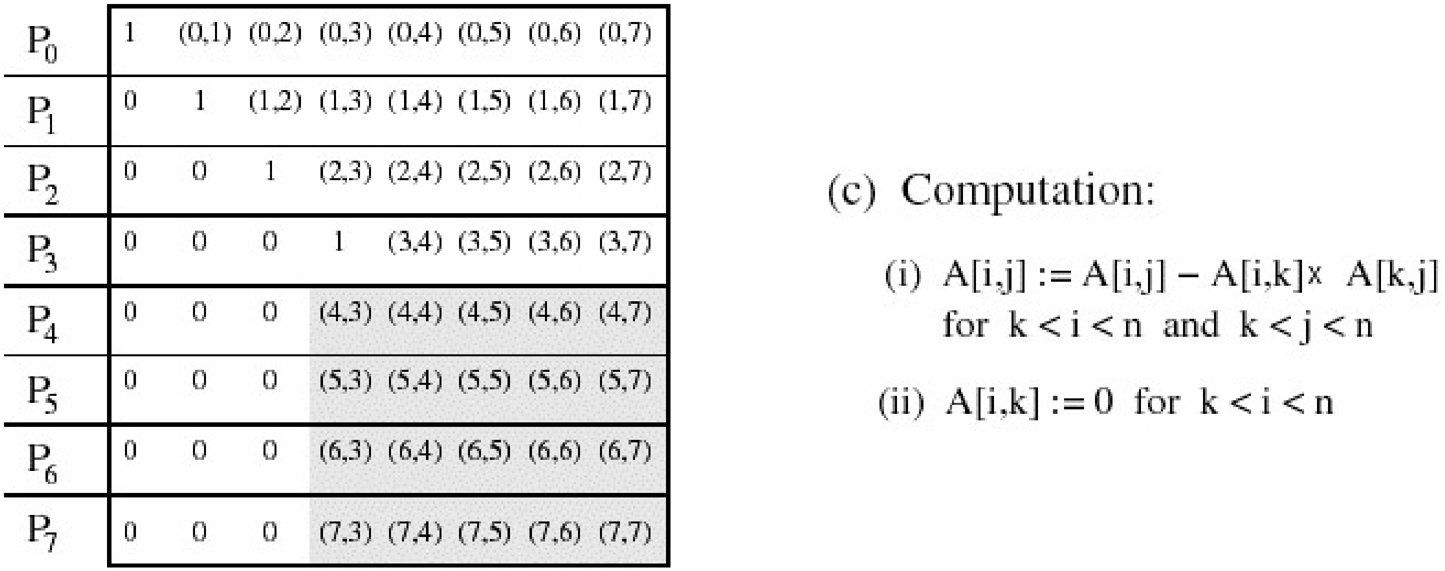


P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)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

(b) Communication:

One–to–all broadcast of row A[k,*]

Rowwise Partitioning on Eight Processes (cont’d)



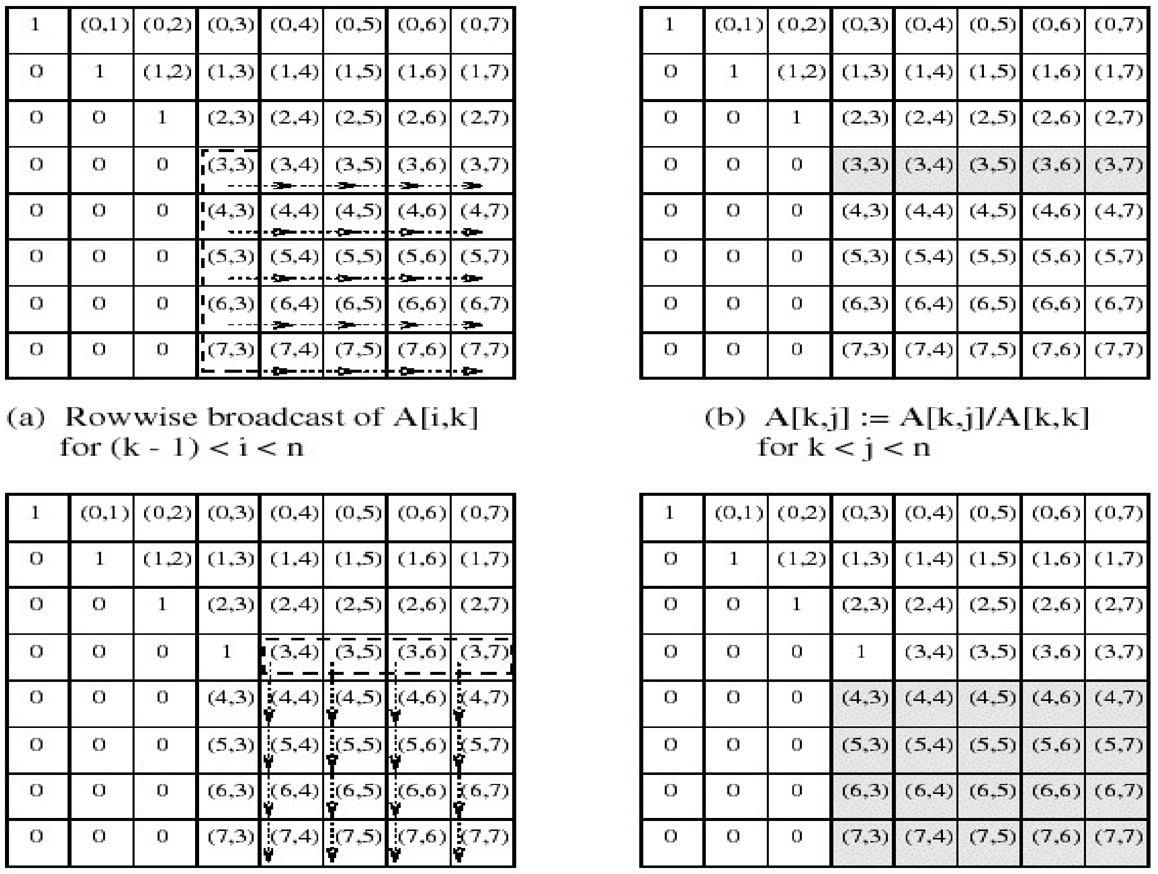
P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)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

- 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)

http://www.netlib.org/

Sorting Algorithms

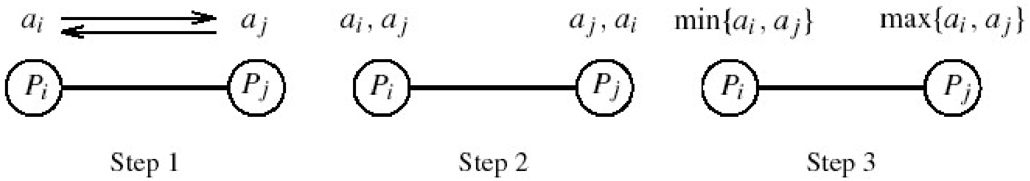
- Sorting is any process of arranging unordered collection into order
  - Permutation of a sequence of elements
- Internal versus external sorting
  - External sorting uses auxiliary storage
- Comparison-based
  - Compare pairs of elements and exchange
  - *O(n log n)*
- Noncomparison-based
  - Use known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

Sorting on Parallel Computers

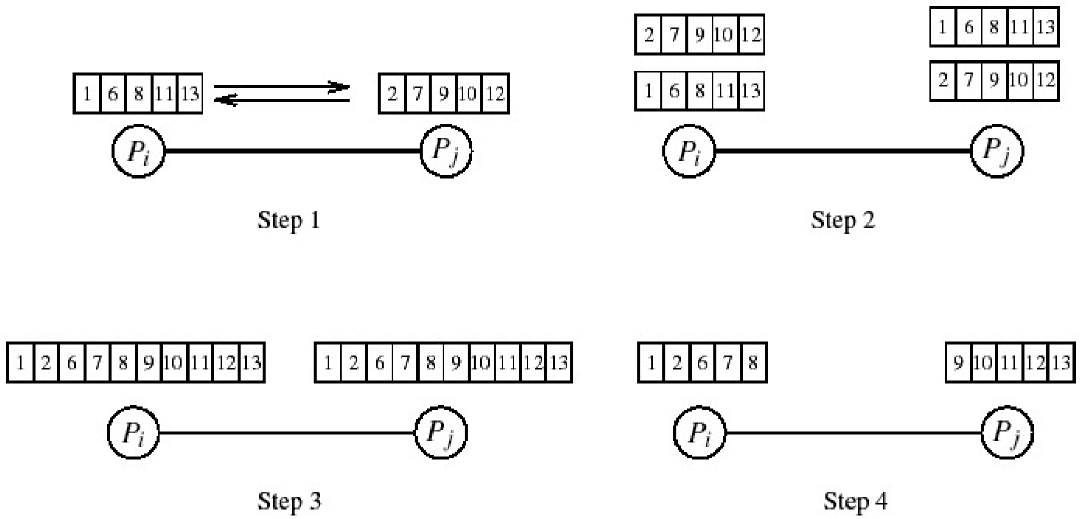
- Where are the elements stored?
  - Need to be distributed across processes
  - Sorted order will be with respect to process order
- How are comparisons performed?
  - One element per process
    - compare-exchange
    - interprocess communication will dominate execution time
  - More than one element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

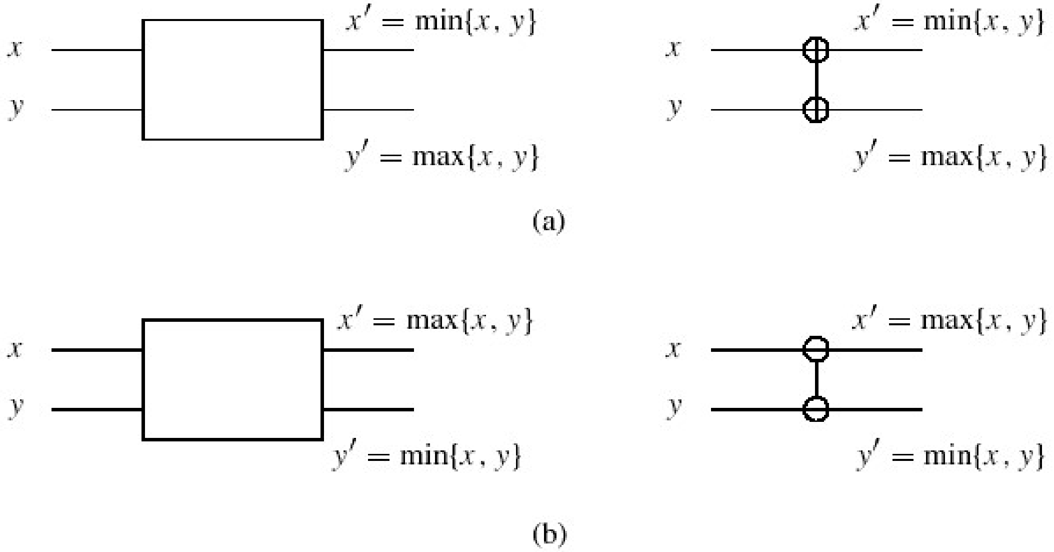


- Multiple elements per processor



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



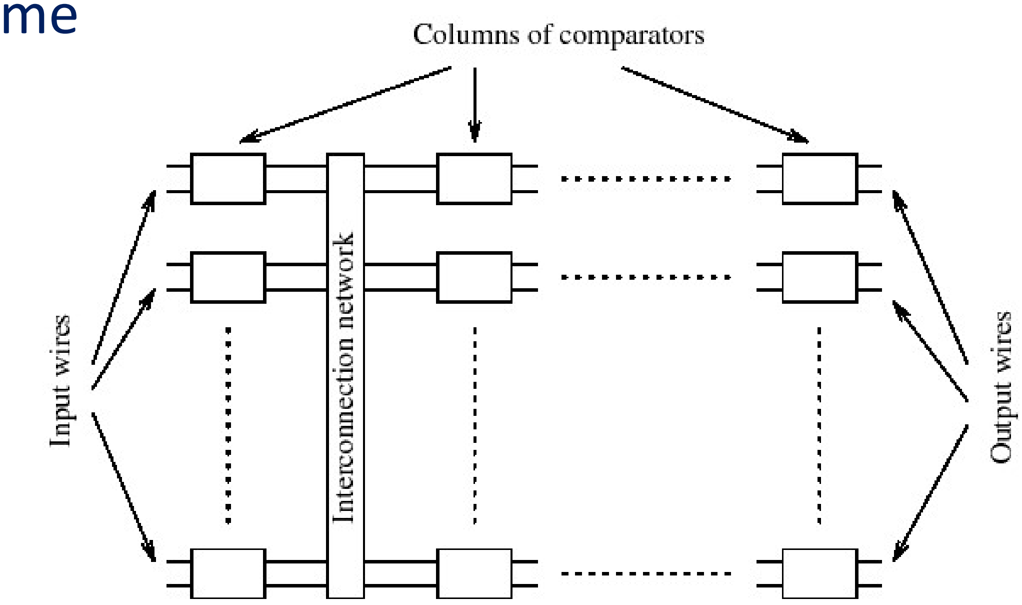
(a)

(b)

- Comparators are connected in parallel and permute elements

Sorting Network Design

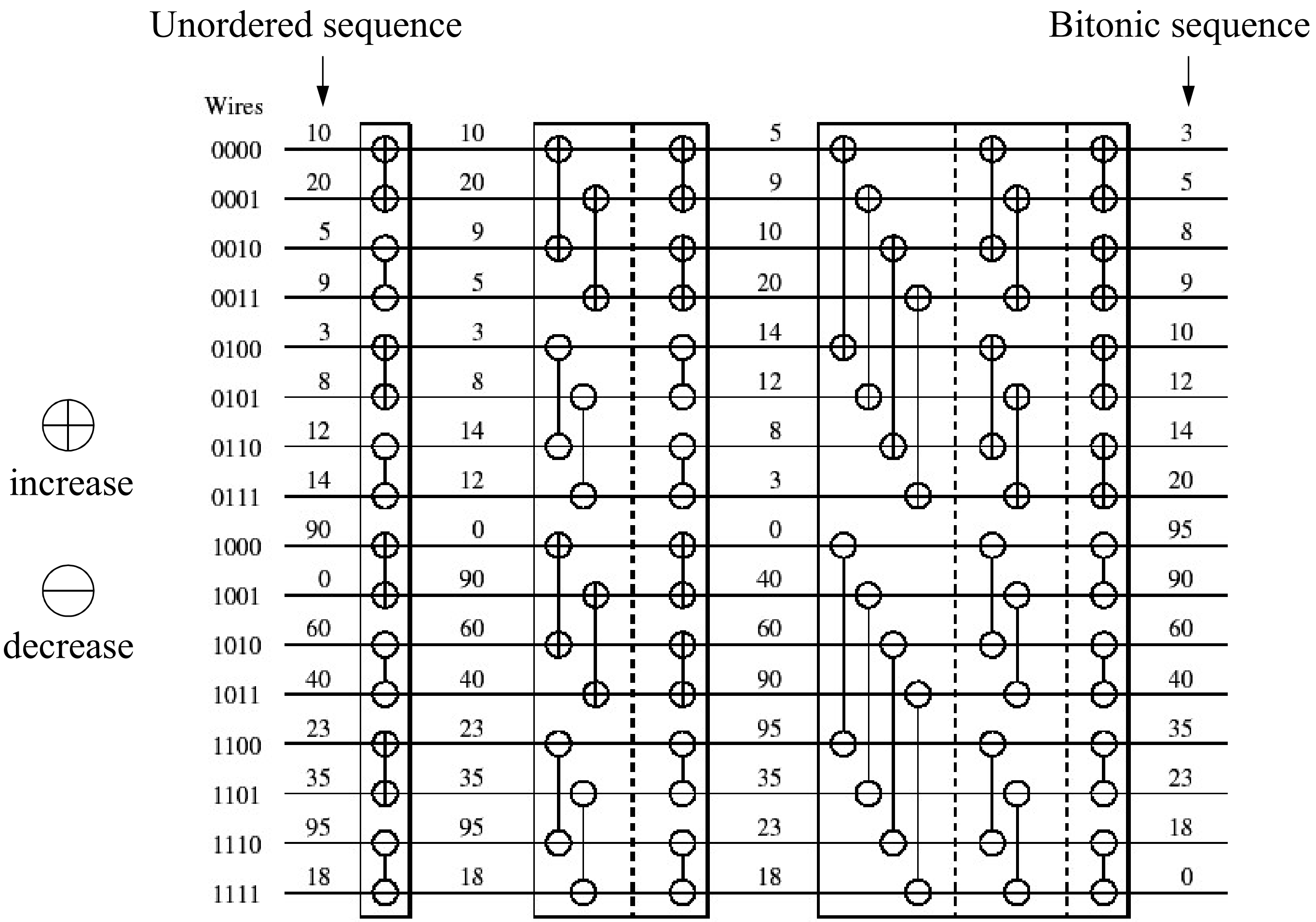
- Multiple comparator stages (# stages, # comparators)
- Connected together by interconnection network
- Output of last stage is the sorted list
- O(log^2(n)) sorting time
- Convert any sorting network to sequential algorithm



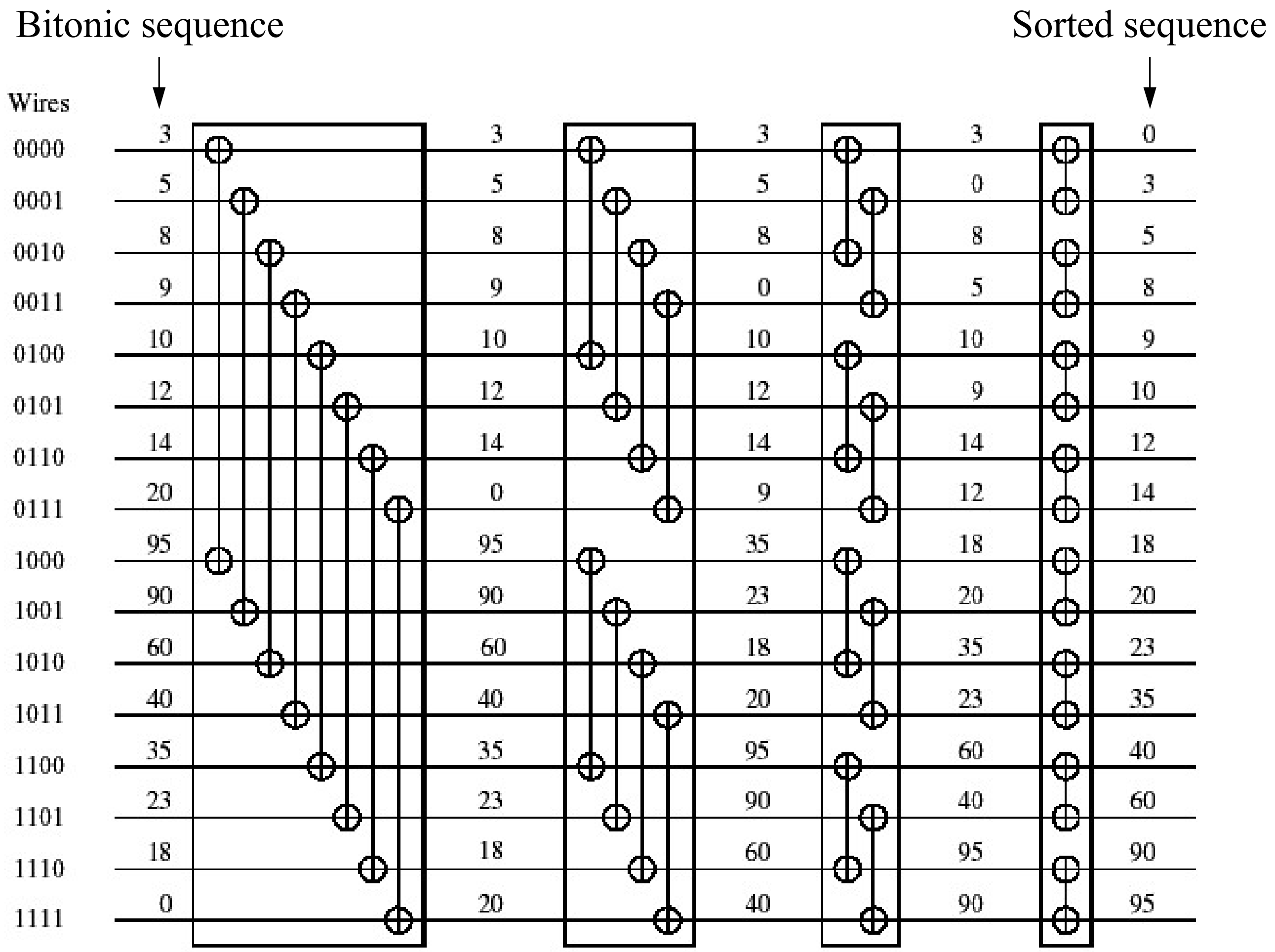
Bitonic Sort

- 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

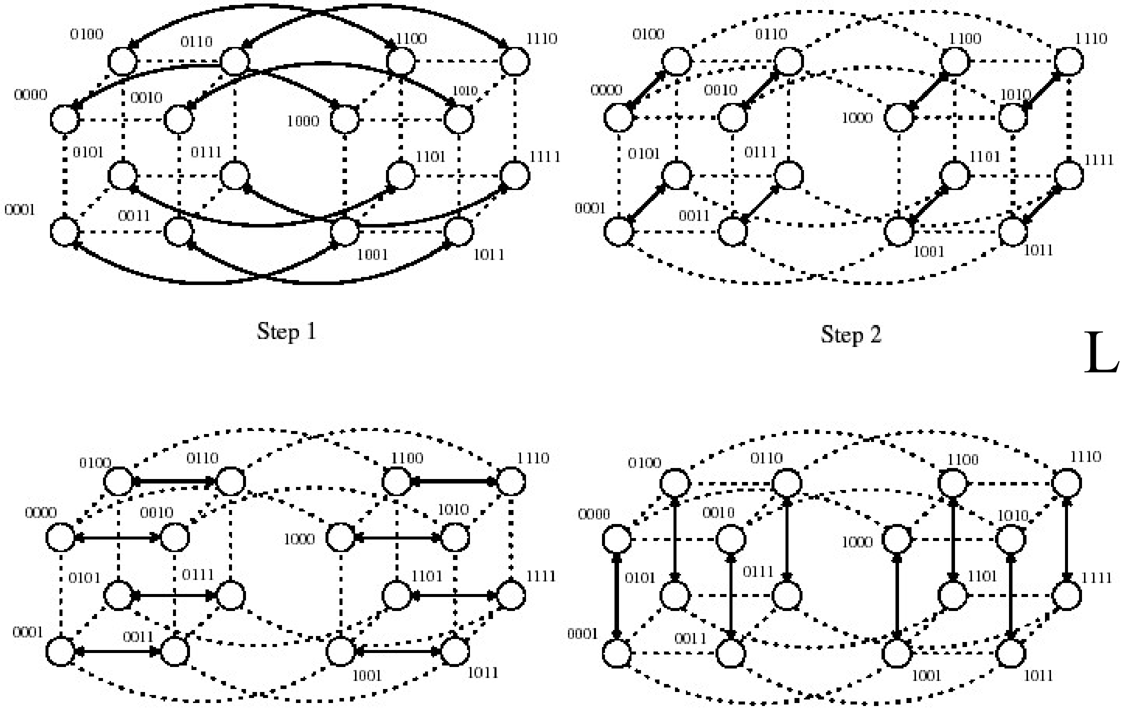


Bitonic Merging Network



Wires	Bitonic sequence				Sorted sequence
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



Step 1

Step 2

Step 3

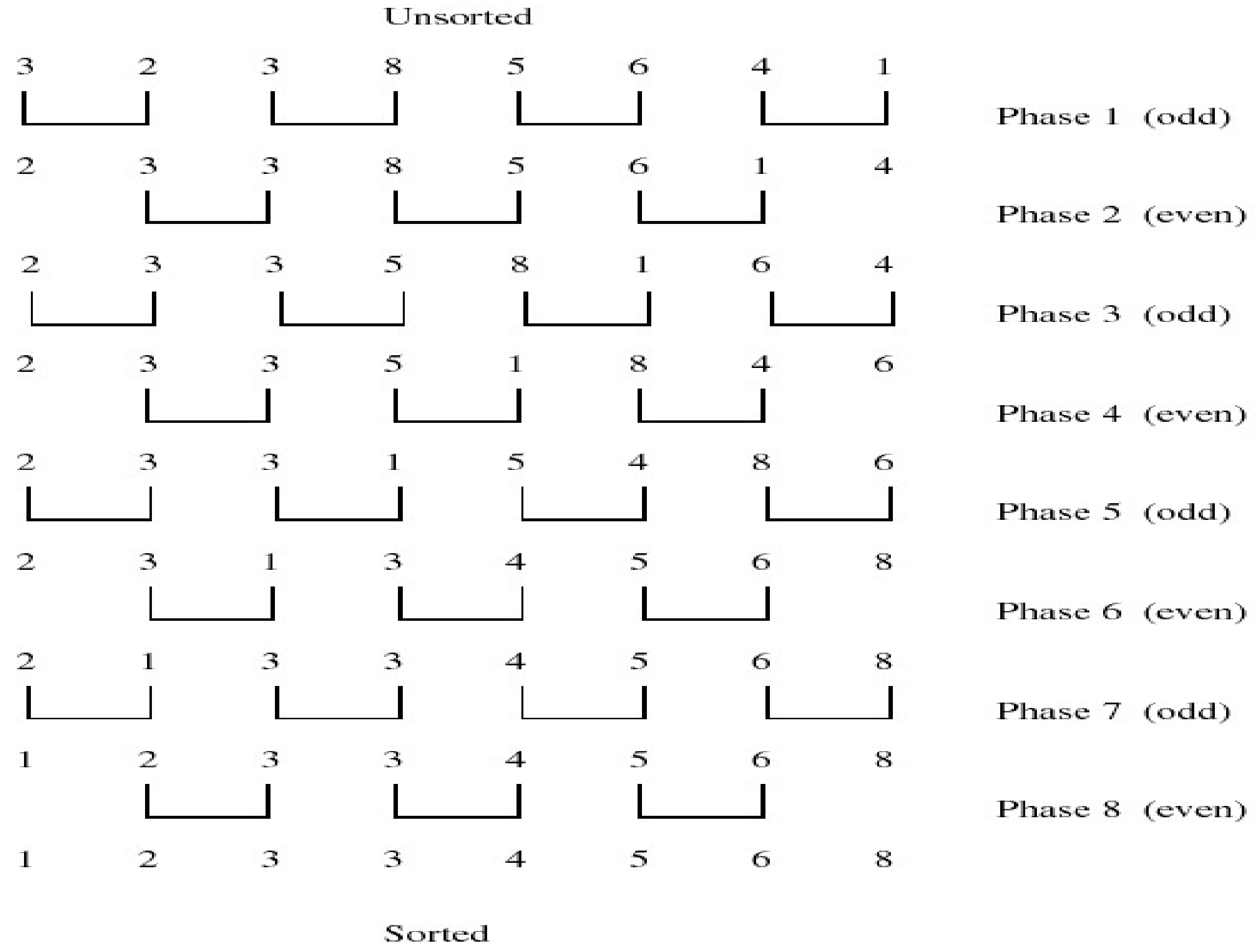
Step 4

Last stage

Bubble Sort and Variants

- We 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

```
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
```

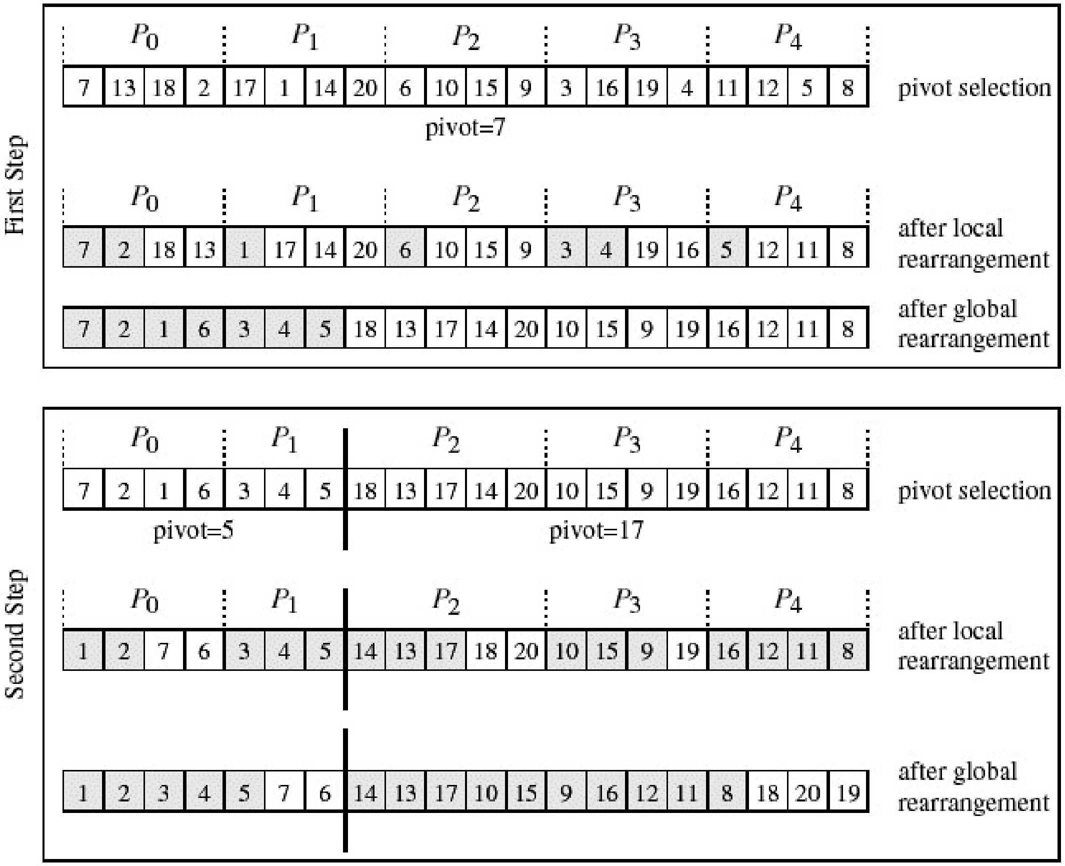
Quicksort

- *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
  - Recursively apply quicksort algorithm to subsequences
- 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

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

pivot=7

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

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

Second Step

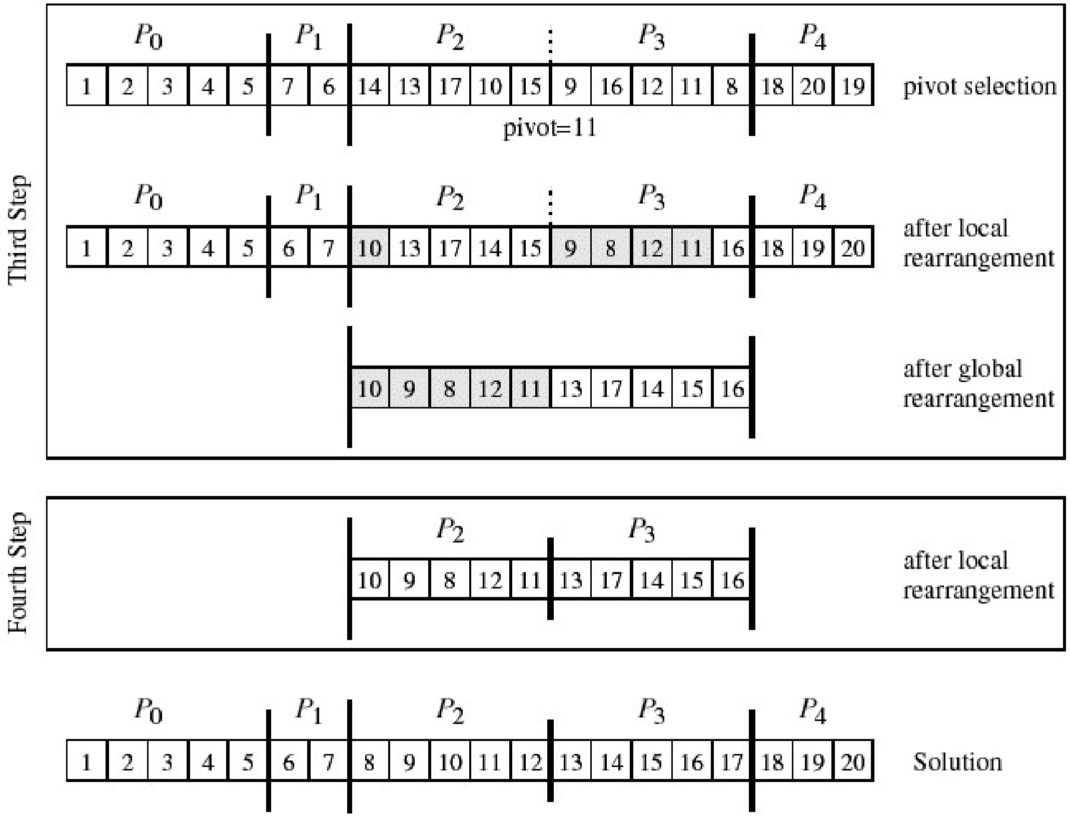
7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8 — pivot selection

pivot=5 pivot=17

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

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

Parallel Shared Address Space Quicksort (cont'd)



Third Step

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

pivot=11

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

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

Fourth Step

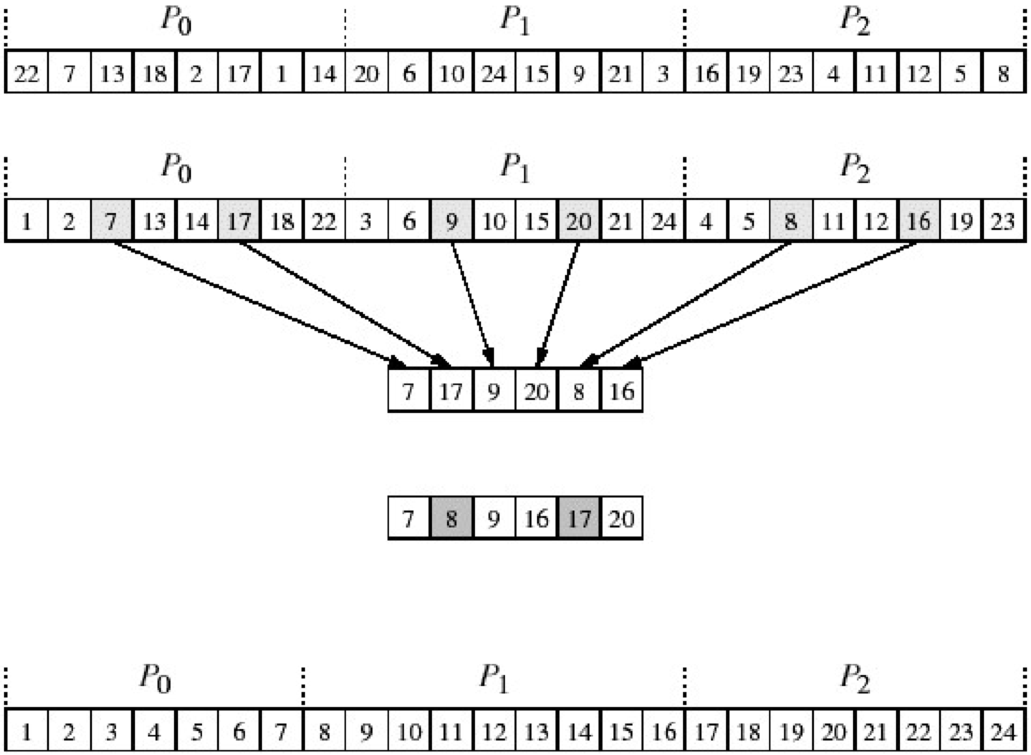
after local rearrangement: | P_2 10 9 8 12 11 | P_3 13 17 14 15 16 |

Solution: P_0 1 2 3 4 5 | P_1 6 7 | P_2 8 9 10 11 12 | P_3 13 14 15 16 17 | P_4 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	P_1	P_2
22 7 13 18 2 17 1 14	20 6 10 24 15 9 21 3	16 19 23 4 11 12 5 8

Local sort & sample selection

P_0	P_1	P_2
1 2 **7** 13 14 **17** 18 22	3 6 **9** 10 15 **20** 21 24	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

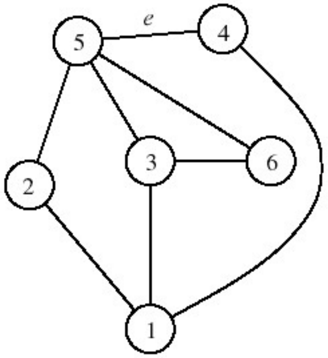
Final element assignment

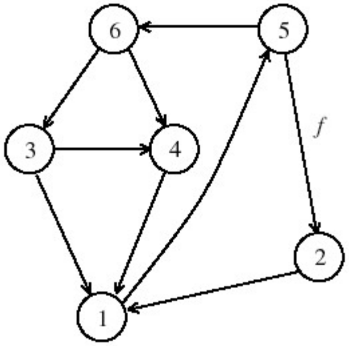
P_0	P_1	P_2
1 2 3 4 5 6 7	8 9 10 11 12 13 14 15 16	17 18 19 20 21 22 23 24

Graph Algorithms

- Graph theory is 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



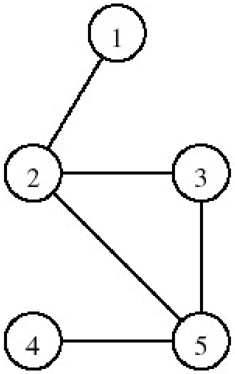


Graph Terminology

- 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 be represented as *adjacency lists*

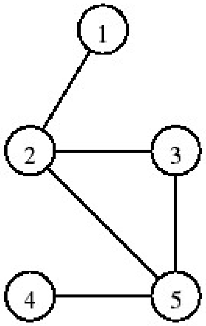
Graph Representations

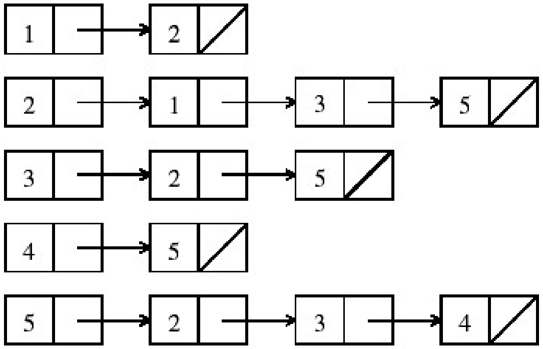
- Adjacency matrix



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

- Adjacency list





Minimum Spanning Tree

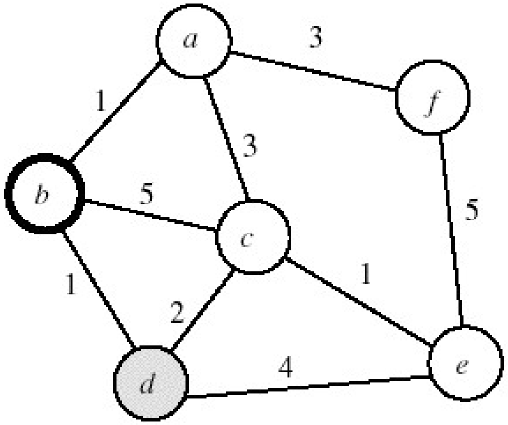
- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*
- The *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight
- Prim's algorithm can be used
  - Greedy algorithm
  - Selects an arbitrary 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

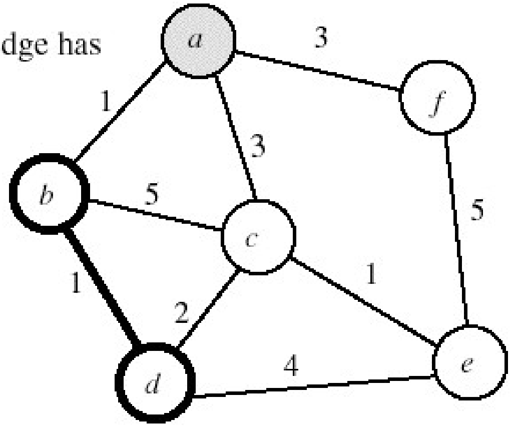
(a) Original graph



	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

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

(b) After the first edge has been selected

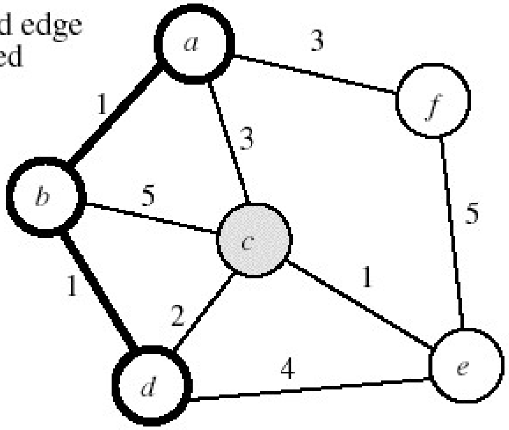


	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

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

Example: Prim’s MST Algorithm (cont’d)

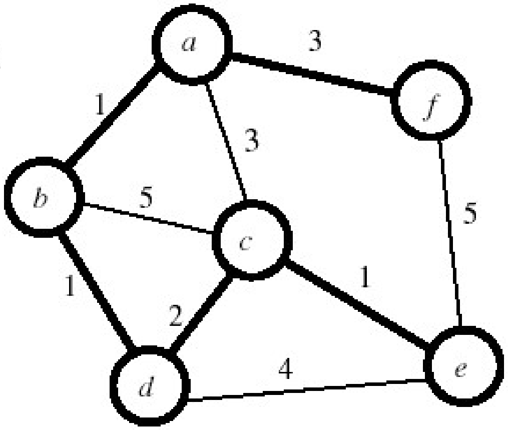
(c) After the second edge has been selected



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

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

(d) Final minimum spanning tree



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

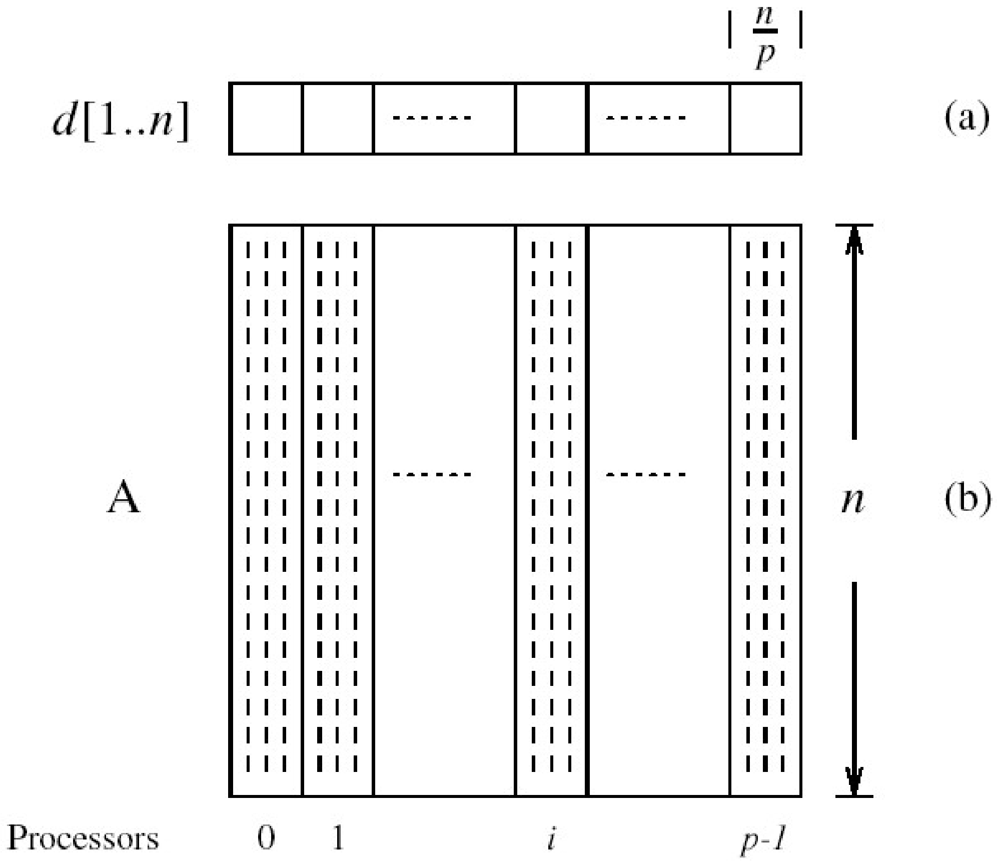
$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 2 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

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 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 Path

```
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 $i, j \le n$, 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
```

Further Readings

- **Introduction to Parallel Computing**, *by* Ananth Grama, Anshul Gupta, George Karypis, & Vipin Kumar, Addison Wesley, 2nd Ed., 2003

  http://www-users.cs.umn.edu/~karypis/parbook/

- **The Boost Graph Library (BGL)**:

  http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/index.html

- **The Parallel Boost Graph Library (Parallel BGL)**:

  http://www.boost.org/doc/libs/1_60_0/libs/graph_parallel/doc/html/index.html