Chapter 5: Probabilistic and Parallel Algorithms

—

Contents

- Probabilistic algorithms
  - Monte Carlo algorithm - Primality testing
  - Las Vegas algorithm - N-queen problem
- Parallel algorithms
  - Finding connected components in a graph
  - Parallel sorting

Parallel Computing

- **Parallelism** or **parallel computing** refers to the ability to carry out multiple calculations or the execution of processes simultaneously
- Parallel computing is commonly used to solve computationally large and data-intensive tasks
- Today parallelism is available in all computer systems
  - Multicore chips are used in essentially all computing devices
- At the larger scale, many computers can be connected by a network and used together to solve large problems.
- *Parallel computing requires design of efficient parallel algorithms*

Why Parallel Computing?

- It is not possible to increase processor and memory frequencies indefinitely
- Power consumption rises with processor frequency while the energy efficiency decreases.
- Parallelism has become a part of any computer

Parallel Execution of Algorithms

- We first need to identify the **subtasks** that can execute in parallel (not a trivial task)
- Assuming a task T can be divided into n parallel subtasks t_1, ..., t_n, these subtasks need to be **mapped/assigned** to processors of the parallel system. This process is denoted as a function from the task set T to the processor set P as $M : T \rightarrow P$
- Subtasks may have dependencies (i.e., t_i may not start before t_j)
- If we know all of the task dependencies and also characteristics such as the execution time, we can distribute tasks evenly to the processors before running them (**static scheduling**).
- In many cases, we do not have such information beforehand and thus **dynamic load balancing** is used to provide each processor with a fair share of workload at runtime.

Concepts and Terminology

Embarrassingly parallel

A computational problem is called embarrassingly parallel if it requires little or no effort to divide that problem into subproblems that can be computed independently on separate computing resources.

Parallel Algorithm Design Methods

- Data Parallelism
- Task Parallelism

Data Parallelism

- Focuses on distribution of data sets across the multiple computation programs.
- Same operations are performed on different parallel computing processors on the distributed data subset.
- Some of Big Data frameworks that utilize data parallelism are Apache Spark, Apache MapReduce and Apache YARN (it supports more of hybrid parallelism providing both task and data parallelism).

Data Parallelism Example

- Multiplying two matrices A and B of size n x n to get a matrix C can be parallelized by partitioning these matrices as n/2 x n/2 sub-matrices

$$\begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

$$C_1 = (A_1 \times B_1) + (A_2 \times B_3) \rightarrow p_1$$

$$C_2 = (A_1 \times B_2) + (A_2 \times B_4) \rightarrow p_2$$

$$C_3 = (A_3 \times B_1) + (A_4 \times B_3) \rightarrow p_3$$

$$C_4 = (A_3 \times B_2) + (A_4 \times B_4) \rightarrow p_4$$

Task Parallelism

- Covers the execution of computer programs across multiple processors on same or multiple machines on the same or different data sets.
- Focuses on executing different operations in parallel to fully utilize the available computing resources in form of processors and memory.
- The divide and conquer algorithmic method can be efficiently implemented using this model
- Some of Big Data frameworks that utilize task parallelism are Apache Storm and Apache YARN (it supports more of hybrid parallelism providing both task and data parallelism).

Task Parallelism Example

```
1: function SUM(A[1..n])
2:     if n = 1 then
3:         return A[1]
4:     else
5:         x ← Sum(A[1..n/2])
6:         y ← Sum(A[n/2 + 1..n])
7:         return x + y
8:     end if
9: end function
```

In order to have a parallel version of this algorithm, we note the recursive calls are independent as they operate on different data partitions; hence, we can perform these calls in parallel simply by performing the operations within the else statement between lines 4 and 8 of this algorithm in parallel.

Parallel Computing Architectures

1. Shared memory architectures
    a. Each processor has some local memory
    b. Interprocess communication and synchronization are performed using a shared memory that provides access to all processors
    c. Data is read from and written to the shared memory locations; however, we need to provide some form of control on access to this memory to prevent race conditions
2. Distributed memory architectures
    a. There is no shared memory
    b. Interprocess communication and synchronization are performed by sending and receiving of messages (message passing)

Parallel Computing Architectures

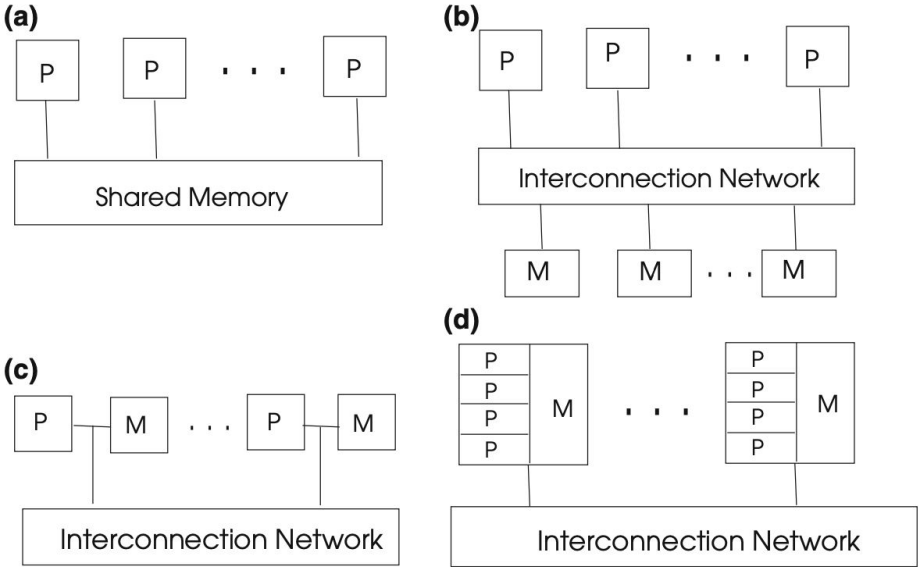


Fig. 4.1 Parallel computing architectures, **a** a shared memory architecture with a single global memory, **b** a shared memory architecture with a number of memory modules, **c** a distributed memory architecture, **d** a distributed and shared memory architecture

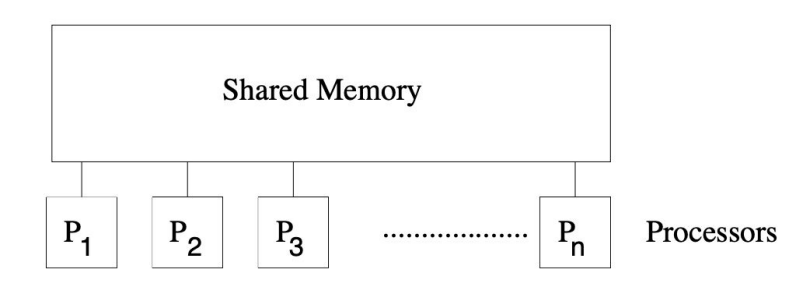
Models of Parallel Computing

Two basic models for parallel computing

1. Parallel Random Access Model (PRAM) Model
2. Message Passing Model

PRAM Model

- The PRAM model extends the basic Random Access Model (RAM) model to parallel computing
- The PRAM model has p processing units that are all connected to a common unbounded shared memory
- In a PRAM model, a processor can access any word of memory in a single step, and these accesses can occur in parallel, i.e., in a single step, every processor can access the shared memory.



PRAM Model

PRAM instructions execute in 3-phase cycles

1. Read (if any) from a shared memory cell
2. Local computation (if any)
3. Write (if any) to a shared memory cell

Processors execute these 3-phase PRAM instructions synchronously

Simultaneous access to the same location may end in unpredictable data in the accessing processing units as well as in the accessed location.

Variants of PRAM

- **Exclusive Read Exclusive Write (EREW) PRAM**
  No two processors are allowed to read or write the same shared memory cell simultaneously
- **Concurrent Read Exclusive Write (CREW)**
  Simultaneous read allowed, but only one processor can write
- **Concurrent Read Concurrent Write (CRCW)**
  Simultaneous reads and writes are allowed; different additional restrictions are imposed on simultaneous writes to avoid unpredictable effects
  - Priority CRCW: processors assigned fixed distinct priorities, highest priority wins
  - Arbitrary CRCW: one randomly chosen write wins
  - Common/Consistent CRCW: all processors are allowed to complete write if and only if all the values to be written are the same

Message Passing Model

- There is no shared memory
- The parallel tasks that run on different processors communicate and synchronize using two basic primitives: send and receive
- Particularly useful in a distributed environment where the communicating processes may reside on different, network connected, systems

PRAM	Message Passing	ALGORITHM MODEL
Threads OpenMP	MPI, PVM	PROGRAMMING
locks, semaphores	messages (send, receive)	OPERATING SYSTEM
Shared Memory	Distributed Memory	HARDWARE

Analysis of Parallel Algorithms

The **running time** of an algorithm, the number of processors it uses, and its cost are used to determine the efficiency of a parallel algorithm.

The running time of a parallel algorithm Tp is

$T_p = t_{fin} - t_{st}$

where t_{st} is the start time of the algorithm on the first (earliest) processor and t_{fin} is the finishing time of the algorithm in the last (latest) processor.

Analysis of Parallel Algorithms (Contd.)

If T_p is the worst-case runnin gtime of a particular algorithm A for a problem Q using p identical processors, and T_s is the worst-case running time of the fastest known sequential algorithm to solve Q, the **speedup** S_p is defined as

$$S_p = T_s / T_p$$

We need the speedup to be as large as possible.

Increasing the number of processors will decrease the speedup due to the increased interprocess communication.

Placing parallel tasks in fewer processors to reduce network traffic decreases parallelism.

Analysis of Parallel Algorithms (Contd.)

Efficiency of a parallel algorithm is the speedup per processor, i.e.

$E_p = S_p / p$

A parallel algorithm is said to be **scalable** if its efficiency remains almost constant when both the number of processors and the size of the problem are increased.

Analysis of Parallel Algorithms (Contd.)

The **work** of an algorithm corresponds to the total number of primitive operations performed by an algorithm.

Ignoring communication overhead from synchronizing the processors, this is equal to the time used to run the computation on a single processor

Analysis of Parallel Algorithms (Contd.)

The **span** or **depth** of an algorithm basically corresponds to the longest sequence of dependences in the computation.

It can be thought of as the time an algorithm would take if we had an unlimited number of processors on an ideal machine.

It enables analyzing to what extent the work of an algorithm can be divided among processors.

Minimizing the depth/span is important in designing parallel algorithms.

Parallel Computing with a Binary Tree

Computing sum of n integers

```
for i from lg n-1 downto 0 do
  for 2^i ≤ j ≤ 2^(i+1)-1 in parallel do
    A[j] ← A[2j] + A[2j+1]
return A[1]
```

Parallel Computing with a Binary Tree

Computing Prefix Sum

Given: array `A[0..n-1]`

Goal: Compute all prefix sums `A[0] + … + A[i] for i = 0, …, n-1`

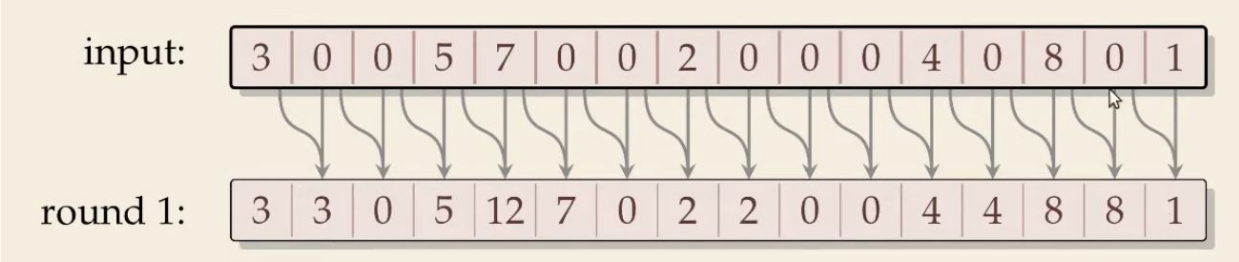
Sequential solution:

```
for i from 0 to n-1 do
  A[i] = A[i-1] + A[i]
```

Time complexity: O(n)
Cannot parallelize due to data

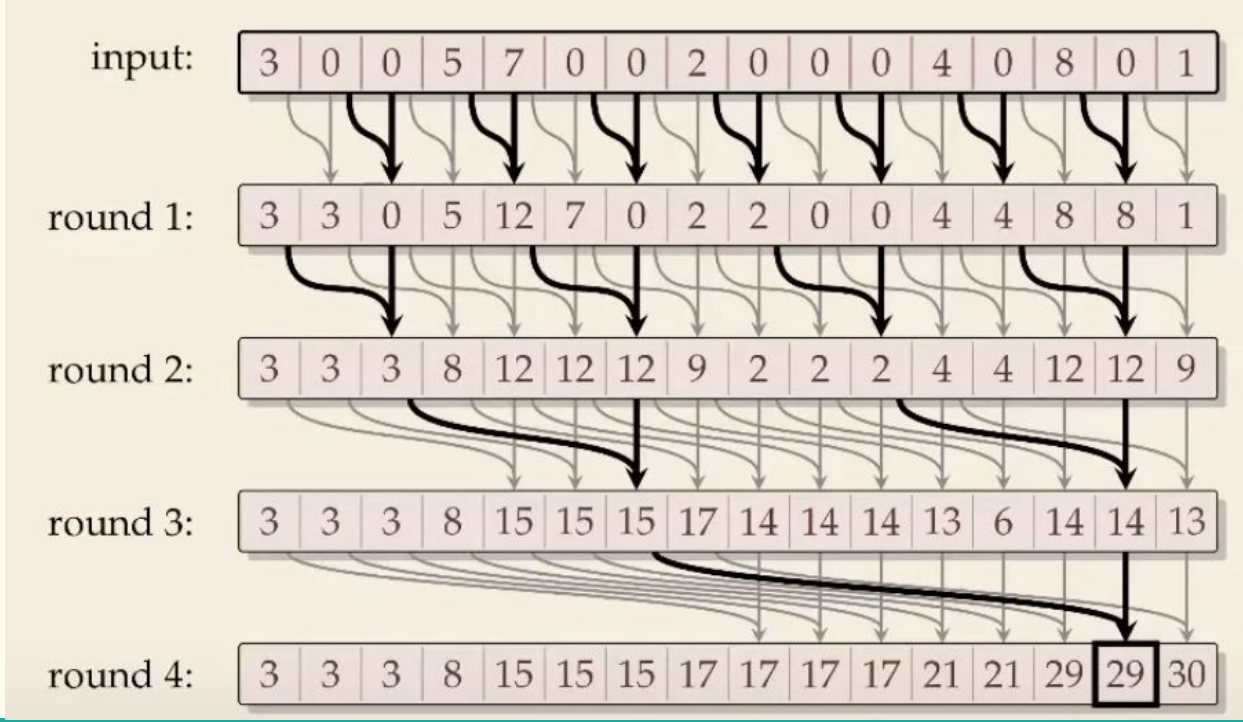
Parallel Computing with a Binary Tree (Contd.)

Parallel Prefix Sum

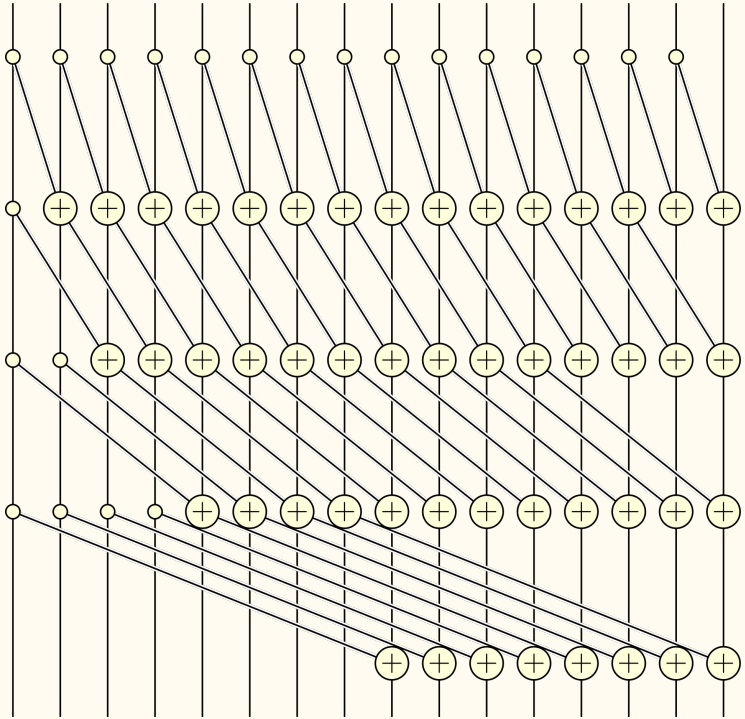


Parallel Computing with a Binary Tree (Contd.)

Parallel Prefix Sum



Parallel Computing with a Binary Tree (Contd.)



Parallel Computing with a Binary Tree (Contd.)

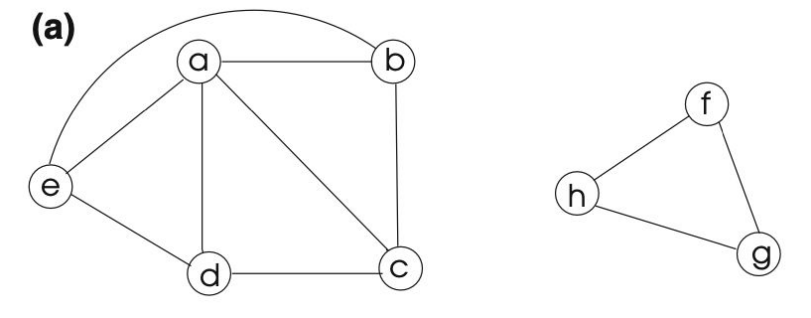
Parallel Prefix Sum

```
1 procedure parallelPrefixSums(A[0..n − 1])
2     for r := 1, . . . ⌈lg n⌉ do
3         step := 2^(r−1)
4         for i := step, . . . n − 1 do in parallel
5             x := A[i] + A[i − step]
6             A[i] := x
7         end parallel for
8     end for
```

Connected Components

A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

In connected components, all the nodes are always reachable from each other.



A graph containing two connected components

Finding Connected Components

The problem is to label all the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices.

Sequentially the connected components of a graph can easily be labeled using either depth-first or breadth-first search.

Two approaches for parallelizing the algorithm for finding connected components:

1. Partitioning the adjacency matrix
2. Contracting the graph

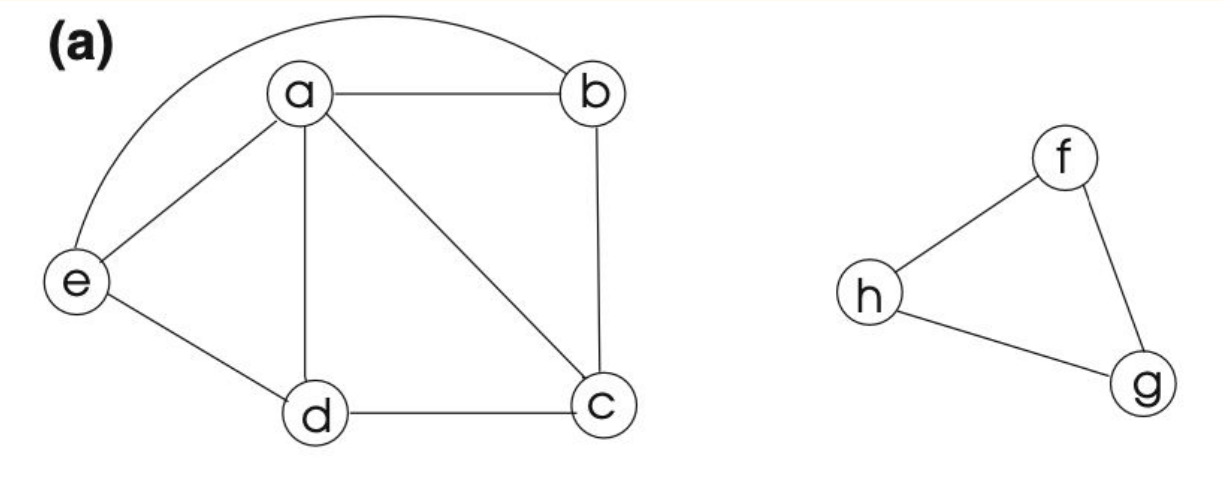
Finding Connected Components Using Adjacency Matrix

Steps

1. Row-wise partitioning of the adjacency matrix
2. Apply DFS on each partition in parallel
3. Merge the forests

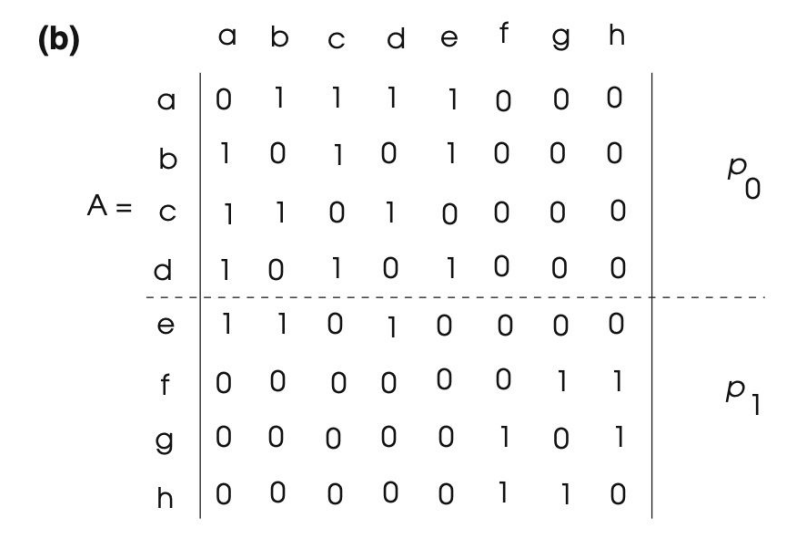
Finding Connected Components Using Adjacency Matrix

Given graph



Finding Connected Components Using Adjacency Matrix

Step 1: Row-wise partitioning of the adjacency matrix



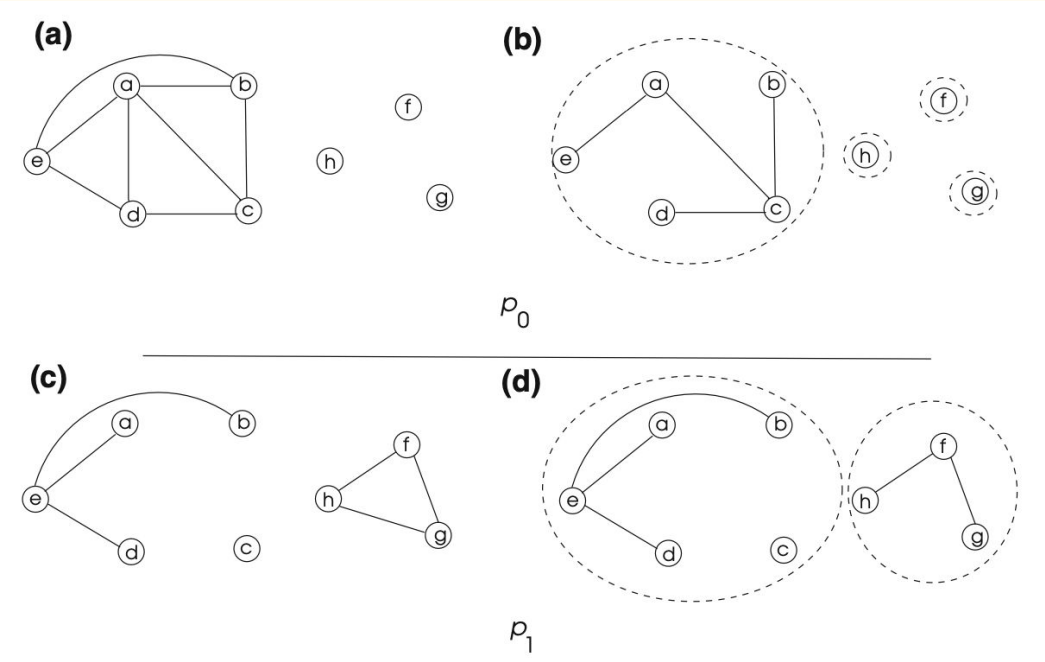
(b)

$A =$

	a	b	c	d	e	f	g	h	
a	0	1	1	1	1	0	0	0	p_0
b	1	0	1	0	1	0	0	0	
c	1	1	0	1	0	0	0	0	
d	1	0	1	0	1	0	0	0	
e	1	1	0	1	0	0	0	0	p_1
f	0	0	0	0	0	0	1	1	
g	0	0	0	0	0	1	0	1	
h	0	0	0	0	0	1	1	0	

Finding Connected Components Using Adjacency Matrix

Step 2: Apply DFS on each partition in parallel

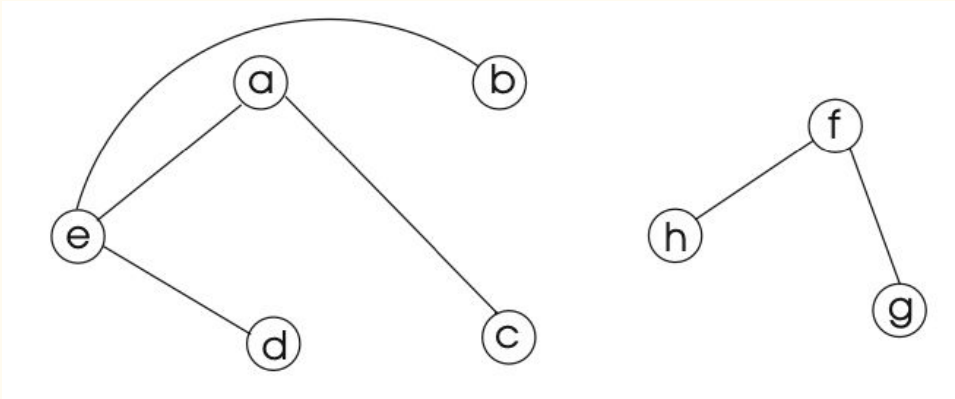


Finding Connected Components Using Adjacency Matrix

Step 3: Merge the forests

Merging can be efficiently done using the find and union operations of disjoint set data structure.

```
For given two forests A and B,
  For each edge (u, v) ∈ A:
    x = find(u) in B
    y = find(v) in B
    if x != y
      merge(u, v) in B
```



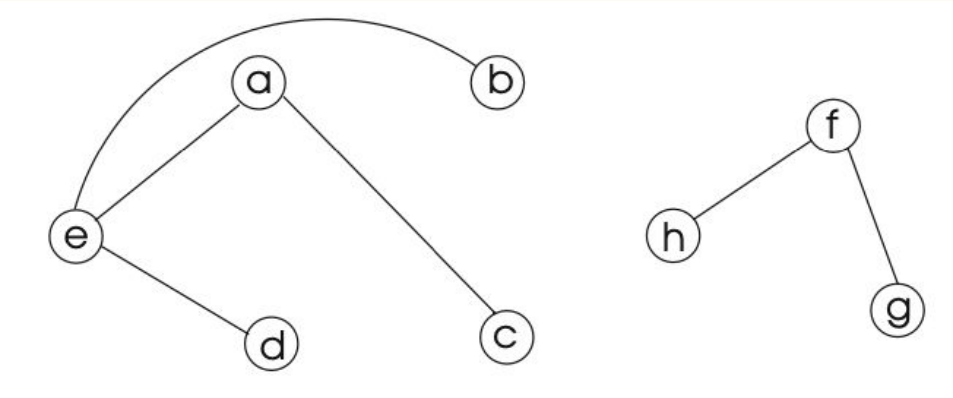
Finding Connected Components Using Adjacency Matrix

Step 3: Merge the forests

Merging can be efficiently done using the find and union operations of disjoint set data structure.

For example,
p0 returns {{a, b, c, d, e}, {f}, {g}, {h}}
p1 returns {{a, b, d, e}, {c}, {f, g, h}}

Merging will result into
{{a, b, c, d, e}, {f, g, h}}



Finding Connected Components Using Graph Contraction

Graph contraction is a technique for computing properties of graph in parallel.

The key idea behind graph contraction is to “shrink” the input graph, ideally by a constant factor in size so that we can solve the problem on a smaller graph, and then use the solution to the smaller graph to construct the solution for the input graph.

Parallel Sorting

Quick sort (sequential)

Parallel Quick Sort

- Recursive calls can run in parallel
- Partition algorithm needs to parallelize
  - Split partitioning into rounds. In each round
    - Compare all elements with pivot (in parallel), store bitvector
    - Compute prefix sums of bit vectors (in parallel)
    - Compact subsequences of small and large elements (in parallel)

Parallel Quick Sort

```
1  procedure parQuicksort(A[l..r])
2      b := choosePivot(A[l..r])
3      j := parallelPartition(A[l..r], b)
4      in parallel { parQuicksort(A[l..j − 1]), parQuicksort(A[j + 1..r]) }
5
6  procedure parallelPartition(A[l..r], b)
7      swap(A[n − 1], A[b]); p := A[n − 1]
8      for i = 0, . . . , n − 2 do in parallel
9          S[i] := ⌈A[i] ≤ p⌉    // S[i] is 1 or 0
10         L[i] := 1 − S[i]
11     end parallel for
12     in parallel { parallelPrefixSum(S[0..n − 2]); parallelPrefixSum(L[0..n − 2]) }
13     j := S[n − 2] + 1
14     for i = 0, . . . , n − 2 do in parallel
15         x := A[i]
16         if x ≤ p then A[S[i] − 1] := x
17         else A[j + L[i]] := x
18     end parallel for
19     A[j] := p
20     return j
```