DAA-Unit V
Paralle Algorithms and Concurrant Algorithms

By Prof. B.A.Khivsara
Assistant Professor,
Department of Computer Engineering
SNJB’s COE Chandwad
Outline

Sequential and Parallel Computing

RAM and PRAM Models
Amdahl’s Law
Brent’s Theorem
Parallel Algorithm Analysis and optimal parallel algorithms
Graph Problems
Concurrent Algorithms
Dinning Philosophers Problem
### Sequential Vs Parallel Computing

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>While Sequential programming involves a consecutive and ordered execution of processes one after another</td>
<td>Parallel programming involves the concurrent computation or simultaneous execution of processes or threads at the same time.</td>
</tr>
<tr>
<td>sequential programming, processes are run one after another in a succession fashion</td>
<td>while in parallel computing, you have multiple processes execute at the same time</td>
</tr>
<tr>
<td>Simple to implement</td>
<td>Complex to implement</td>
</tr>
<tr>
<td>Processor Utilization is poor</td>
<td>Processor utilization is efficient</td>
</tr>
</tbody>
</table>
Outline

Sequential and Parallel Computing

RAM and PRAM Models

Amdahl’s Law

Brent’s Theorem

Parallel Algorithm Analysis and optimal parallel algorithms

Graph Problems

Concurrent Algorithms

Dinning Philosophers Problem
Random Access Machine Model (RAM)

RAM model of serial computers:

- Memory is a sequence of words, each capable of containing an integer.
- Each memory access takes one unit of time.
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable.
- Read-only input tape, write-only output tape.
A **READ** phase in which the processor reads datum from a memory location and copies it into a register.

A **COMPUTE** phase in which a processor performs a basic operation on data from one or two of its registers.

A **WRITE** phase in which the processor copies the contents of an internal register into a memory location.
Parallel Algorithm

In computer science, a parallel algorithm, as opposed to a traditional serial algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result.
Approaches to parallelism

**shared-memory parallel processing**
- A *thread* consists of a single flow of control, a program counter, a call stack, and a small amount of thread-specific data.
- Threads share memory, and communicate by reading and writing to that memory.

**message-passing parallel processing**
- A *process* is a thread that has its own private memory.
- Threads send messages to one another.
PRAM [Parallel Random Access Machine]

PRAM composed of:

- P processors, each with its own unmodifiable program.
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer.
- A read-only input tape.
- A write-only output tape.

PRAM model is a synchronous, MIMD, shared address space parallel computer.
PRAM model of computation

Shared memory
PRAM model Characteristics

- At each unit of time, a processor is either active or idle (depending on id)
- All processors execute same program
- At each time step, all processors execute same instruction on different data ("data-parallel")
- Focuses on concurrency only
### Variants of PRAM model

<table>
<thead>
<tr>
<th></th>
<th>Exclusive Write</th>
<th>Concurrent Write</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exclusive Read</strong></td>
<td>EREW</td>
<td>ERCW</td>
</tr>
<tr>
<td><strong>Concurrent Read</strong></td>
<td>CREW</td>
<td>CRCW</td>
</tr>
</tbody>
</table>

Concurrent (C) means, many processors can do the operation simultaneously in the same memory.

Exclusive (E) not concurrent.
More PRAM taxonomy

**EREW** - exclusive read, exclusive write
- A program isn’t allowed to have two processors access the same memory location at the same time.

**CREW** - concurrent read, exclusive write

**CRCW** - concurrent read, concurrent write
- Needs protocol for arbitrating write conflicts

**CROW** – concurrent read, owner write
- Each memory location has an official “owner”
Sub-variants of CRCW

Common CRCW
- CW iff all processors writing same value

Arbitrary CRCW
- Arbitrary value of write set stored

Priority CRCW
- Value of min-index processor stored

Combining CRCW
Outline

- Sequential and Parallel Computing
- RAM and PRAM Models
- Amdahl’s Law
- Brent’s Theorem
- Parallel Algorithm Analysis and optimal parallel algorithms
- Graph Problems
- Concurrent Algorithms
- Dinning Philosophers Problem
Amdahl's law, also known as Amdahl's argument, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.
Amdahl's law

In the case of parallelization, Amdahl's law states that if $P$ is the proportion of a program that can be made parallel and $(1 - P)$ is the proportion that cannot be parallelized (serial), then the maximum speedup that can be achieved by using $N$ processors is

\[
S(N) = \frac{1}{(1 - P) + P/N}
\]

As $N$ goes to infinity, the maximum speedup $S(N) = 1/(1 - P)$
Amdahl's law Example 1

- 95% of a program’s execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

\[
\psi \leq \frac{1}{0.05 + (1-0.05)/8} \approx 5.9
\]
Amdahl's law Example 2

- 5% of a parallel program's execution time is spent within inherently sequential code. The maximum speedup achievable by this program, regardless of how many PEs are used, is

\[
\lim_{{p \to \infty}} \frac{1}{{0.05 + (1 - 0.05)/p}} = \frac{1}{{0.05}} = 20
\]
Limitations of Amdahl's law

- Amdahl's law only applies to cases where the problem size is fixed.

- Example

  - If 75% of a process can be parallelized, and there are four processors, then the possible speedup is
    \[ 1 / ((1 - 0.75) + 0.75/4) = 2.286 \]
  - But with 40 processors the speedup is only
    \[ 1 / ((1 - 0.75) + 0.75/40) = 3.721 \]

- This has led to conclude that having lots of processors won’t help very much
Brent’s Theorem

Brent’s theorem specifies that for a **sequential algorithm** with \( t \) time steps, and a total of \( m \) **operations**, that a **run time** \( T \) is definitely possible on a **shared memory machine** (PRAM) with \( p \) **processors**.

It may be possible to **implement** this algorithm faster (by scheduling instruction differently to minimise **idle** processors, for instance), but it is definitely possible to implement this algorithm in this time given \( p \) processors.

**Brent’s Theorem** : Given a parallel algorithm \( A \) that performs \( m \) computation steps in time \( t \), then \( p \) processors can execute algorithm in \( A \) in time

\[
T = t + \frac{(m-t)}{p}
\]
Brent’s Theorem Example

Say we are summing an array.

\[
\text{for (i=0; i < length(a); i++)}
\]
\[
\text{sum = sum + a(i);}
\]

Using this algorithm, each add operation depends on the result of the previous one, forming a chain of length n, thus \( t = n \). There are n operations, so \( m = n \).

\[
T = n + 0/p.
\]

so no matter how many processors are available, this algorithm will take time \( n \).
Brent’s Theorem Example

Now consider the adding the array Parallely:
\[
\text{sum}(a) = \left( (A_0 + A_1) + (A_2 + A_3) \right) + \left( (A_4 + A_5) + (A_6 + A_7) \right)
\]

We can add $A_2$ to $A_3$ without needing to know what $A_0 + A_1$ is.

For an array of length $n$, the longest chain(s) will be of length $\log(n)$. $t = \log(n)$. $m = n$.

\[
T = \log(n) + \frac{(n - \log(n))}{p}.
\]
This tells us many useful things about the algorithm:

- If we have \( n \) processors, the algorithm can be implemented in \( \log(n) \) time.

- If we have \( \log(n) \) processors, the algorithm can be implemented in \( 2\log(n) \) time (asymptotically this is the same as \( \log(n) \) time).

- If we have 1 processor, the algorithm can be implemented in \( n \) time.
Brent’s Theorem

Analysis/Importance

If we consider the amount of work done in each case, with one processor, we do $n$ work, with $\log(n)$ processors we do $n$ work, but with $n$ processors we do $n\log(n)$ work.

The implementations with 1 or $\log(n)$ processors, therefore are cost optimal, while the implementation with $n$ processors is not.

It is important to remember that Brent's theorem does not tell us how to implement any of these algorithms in parallel; it merely tells us what is possible.
Outline

- Sequential and Parallel Computing
- RAM and PRAM Models
- Amdahl’s Law
- Brent’s Theorem
- Parallel Algorithm Analysis and optimal parallel algorithms
- Graph Problems
- Concurrent Algorithms
- Dinning Philosophers Problem
Efficient and optimal parallel algorithms Analysis

- A parallel algorithm is efficient iff
  - it is fast (e.g. polynomial time) and
  - the product of the parallel time and number of processors is close to the time of at the best known sequential algorithm

\[ T_{\text{sequential}} \approx (T_{\text{parallel}} \cdot N_{\text{processors}}) \]

- A parallel algorithms is optimal iff this product is of the same order as the best known sequential time
Metrics: Speedup, Efficiency and Cost

- **Speedup:**
  \[ S = \frac{\text{Time(sequential algorithm - } T_s)}{\text{Time(parallel algorithm - } T_p)} \]

- **Efficiency:**
  \[ E = \frac{S}{N} \]
  
  \( N \) is the number of processors

- **Cost:**
  \[ C = N \times T_p \]
Metrics: Analysis

Parallel algorithm is cost-optimal:

- If parallel cost = sequential time
  - $C_p = T_1$
  - $E_p = 100$

Critical when down-scaling:

- If parallel implementation may become slower than sequential
  - $T_1 = n^3$
  - $T_p = n^{2.5}$ when $p = n^2$
  - $C_p = n^{4.5}$
Analysis a parallel algorithm

Example

Adding $n$ numbers in parallel
Analysis a parallel algorithm

Example:

Example for 8 numbers:
We start with 4 processors and each of them adds 2 items in the first step.

The number of items is halved at every subsequent step. Hence log \( n \) steps are required for adding \( n \) numbers. The processor requirement is \( O(n) \).

A parallel algorithms is analyzed mainly in terms of its time, processor and work complexities.

Time complexity \( T(n) \) : How many time steps are needed? \( T(n) = O(\log n) \)

Processor complexity \( P(n) \) : How many processors are used? \( P(n) = O(n) \)

Work complexity \( W(n) \) : What is the total work done by all the processors? \( W(n) = O(n \log n) \)
Outline

- Sequential and Parallel Computing
- RAM and PRAM Models
- Amdahl’s Law
- Brent’s Theorem
- Parallel Algorithm Analysis and optimal parallel algorithms
- Graph Problems
- Concurrent Algorithms
- Dinning Philosophers Problem
Graph Problems: Topic Overview

- General framework
- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- Bipartite Graph
Representation of Graphs

a) An undirected graph and (b) a directed graph.
Graph Problem-General Framework

- Let $M$ be an $n \times n$ matrix with nonnegative integer coefficients.
- Let $\tilde{M}$ be an matrix defines as
  
  $\tilde{M}(i,i) = 0$ for every $i$
  
  $\tilde{M}(i,j) = \min \{ M_{i_0i_1} + M_{i_1i_2} + \ldots + M_{i_{k-1}i_k} \}$ for every $i \neq j$
- Where $i_0 = i$ and $i_k = j$ and min is taken over all sequences $i_0, i_1, \ldots, i_k$
Algorithm for $\tilde{M}$

$M[i,j] = M[i,j]$ for $1 \leq i, j \leq n$

For $r = 1$ to $\log n$ do

\{
$q[i,j,k] = m[i,j] + m[j,k]$ for $1 \leq i$ and $j,k \leq n$

*In Parallel set*

$M[i,j] = \min \{ q[i,1,j], q[i,2,j], \ldots, q[i,n,j] \}$ for $1 \leq i$ and $j \leq n$
\}

Put $\tilde{M}(i,i) = 0$ for all $i$ and $\tilde{M}(i,j) = m[l,j]$ for $i \neq j$
Example

\[
\begin{array}{cccccc}
0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[M(1,5) = 0 \quad \text{since } M_{12} + M_{25} = 0\]

\[\tilde{M}(3,1) = 1 \quad \text{since for every choice of } i_1, i_2, \ldots, i_{k-1} \text{ the sum } M_{i_0i_1} + M_{i_1i_2} \ldots \text{ is } > 0\]
Graph Problems: Topic Overview

- General framework
- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- Bipartite Graph
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 $G$.

In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.

A minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight.
Minimum Spanning Tree

An undirected graph and its minimum spanning tree.
Prim's algorithm for finding an MST is a greedy algorithm.

Start by selecting an arbitrary vertex, include it into the current MST.

Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.
Minimum Spanning Tree: Prim's Algorithm

(a) Original graph

(b) After the first edge has been selected

(c) After the second edge has been selected

(d) Final minimum spanning tree
Minimum Spanning Tree: Prim's Algorithm

1. procedure PRIM_MST(V, E, w, r)
2. begin
3. \( V_T := \{r\} \);
4. \( d[r] := 0 \);
5. for all \( v \in (V - V_T) \) do
6. \( \text{if edge } (r, v) \text{ exists set } d[v] := w(r, v); \)
7. \( \text{else set } d[v] := \infty; \)
8. while \( V_T \neq V \) do
9. begin
10. find a vertex \( u \) such that \( d[u] := \min\{d[v] | v \in (V - V_T)\}; \)
11. \( V_T := V_T \cup \{u\}; \)
12. for all \( v \in (V - V_T) \) do
13. \( d[v] := \min\{d[v], w(u, v)\}; \)
14. endwhile
15. end PRIM_MST

Prim's sequential minimum spanning tree algorithm.
Prim's Algorithm: Parallel Formulation

The algorithm works in \( n \) outer iterations - it is hard to execute these iterations concurrently.

The inner loop is relatively easy to parallelize. Let \( p \) be the number of processes, and let \( n \) be the number of vertices.

The adjacency matrix is partitioned in a 1-D block fashion, with distance vector \( d \) partitioned accordingly.

In each step, a processor selects the locally closest node, followed by a global reduction to select globally closest node.

This node is inserted into MST, and the choice broadcast to all processors.

Each processor updates its part of the \( d \) vector locally.
Prim's Algorithm: **Parallel Formulation**

The partitioning of the distance array $d$ and the adjacency matrix $A$ among $p$ processes.

The partitioning of the distance array $d[1..n]$ among $p$ processes.

The adjacency matrix $A$ among $p$ processes.
Prim's Algorithm: Parallel Analysis

- The cost of a broadcast is $O(\log p)$.
- The cost of local updation of the $d$ vector is $O(n/p)$.
- The parallel time per iteration is $O(n/p + \log p)$.
- The total parallel time for $n$ vertices with $n$ iteration is $O(n^2/p + n \log p)$.
- The corresponding efficiency is $O(p^2 \log^2 p)$. 
Graph Problems: Topic Overview

- General framework
- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- Bipartite Graph
Single-Source Shortest Paths

For a weighted graph $G = (V,E,w)$, the single-source shortest paths problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in $V$.

**Dijkstra's algorithm** is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.

It grows this set based on the node closest to source using one of the nodes in the current shortest path set.
Single-Source Shortest Paths:
Dijkstra's Algorithm

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

Dijkstra's sequential single-source shortest paths algorithm.
Dijkstra's Algorithm: Parallel Formulation

The weighted adjacency matrix is partitioned using the 1-D block mapping.

Each process selects, locally, the node closest to the source, followed by a global reduction to select next node.

The node is broadcast to all processors and the l-vector updated.

The parallel performance of Dijkstra's algorithm is identical to that of Prim's algorithm.
Dijkstra's Algorithm: Parallel Analysis

The cost of a broadcast is $O(\log p)$.

The cost of local updation of the $d$ vector is $O(n/p)$.

The parallel time per iteration is $O(n/p + \log p)$.

The total parallel time for $n$ vertices with $n$ iteration is $O(n^2/p + n \log p)$.

The corresponding efficiency is $O(p^2\log^2 p)$.
Graph Problems: Topic Overview

- General framework
- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- Bipartite Graph
A Bipartite Graph

It is an undirected graph $G=(V, E)$ in which $V$ can be partitioned into two (disjoint) sets $V_1$ and $V_2$ such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or vice versa.

That is, all edges go between the two sets $V_1$ and $V_2$ but are not allowed within $V_1$ and $V_2$. 
A Bipartite Graph Example
Graph (Bipartite) Matching

Graph G = (V,E)

Graph Matching M is a subset of E

• if $e_1, e_2$ distinct edges in M
• Then they have no vertex in common

Matching in Bipartite graph is a set of edges chosen in such a way that no two edges share an endpoint.
A Bipartite Graph Matching

Match $M = \{(1,6),(3,7),(5,9)\}$

$|M| = 3$ edges

There is no edge from node 2, 4 and 8 in M. So these are free node
A Bipartite Graph Maximum Matching

consider free node 4 and add new edge (4,7) to M and edge (3,7) is removed from M, so the node 3 will be free.

consider free node and add new edge (3,9) in M and edge (5,9) is removed from M, so the node 5 will be free.

Blue Edge denote original edge in M
Green edge denote removed edge from M
Red edge denote newly added edge to M
New free node is 5, so new edge (5,8) is added in M. Now only node 2 is free but adding edge (2,6) is not feasible. No node is there in free list so stop.

There is an Alternate path from \((4,7), (7,3), (3,9), (9,5), (5,8)\)
Path 4-7-3-9-5-8 is also an Augmented path\((P)\) as it starts and ends with free node.
New \(M=\{(1,6),(3,9),(4,7),(5,8)\}\)
\(|M|= 4\) edges
Augmenting Path(P) in G given Graph Matching M

require

begins and ends at unmatched vertices

e_1, e_3, e_5, ..., e_k \quad E-M

e_2, e_4, ..., e_{k-1} \quad M
Maximum Matching Algorithm

input graph G = (V,E)
[1] M ← {} 
[2] while there exists an augmenting path p relative to M 
   do M ← M ∪ P 
[3] output maximum matching M
Parallel Algorithm for Bipartite Graph

- For parallelism, we discover multiple vertex-disjoint augmenting paths simultaneously.
- When searching for vertex-disjoint augmenting paths, we build alternating forests.
- In parallel platforms, we can search and build (vertex-disjoint) alternating forests in two ways.
  - (1) **source parallel (DFS style)**: each processor builds one alternating tree in the forest starting from an unmatched vertex.
  - (2) **level parallel (BFS style)**: every processor together builds one level of BFS forest and then move to the next level.
- The first approach employs coarse-grained parallelism because a processor builds the whole tree.
- The second approach employs fine-grained parallelism since a processor can process a single vertex.
Parallelizing Augmenting Path Searches

Coarse grained
Source parallel: DFS based

Fine grained
Level parallel: BFS based
Parallel Algorithm for Bipartite Graph

- However, note that this searching is not same as traditional BFS/DFS.
- For example, it is not a parallel single DFS because of the following differences:
  - (a) we search form multiple-source, i.e., we run multiple DFS's simultaneously
  - (b) The paths are vertex-disjoint, need extra check for that. For example, edges marked with "x" are not part of the forest to maintain vertex-disjointedness property
  - (c) and the paths are alternating, i.e., every other step is trivial
Parallelizing Augmenting Path Searches

1. Multiple source
2. Vertex-disjoint
3. Alternating DFS
Outline

Sequential and Parallel Computing
RAM and PRAM Models
Amdahl’s Law
Brent’s Theorem
Parallel Algorithm Analysis and optimal parallel algorithms
Graph Problems
Concurrent Algorithms
Dinning Philosophers Problem
Concurrent Algorithms

Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor.

Advantages of Concurrency

- Concurrent processes can reduce duplication in code.
- The overall runtime of the algorithm can be significantly reduced.
- More real-world problems can be solved than with sequential algorithms alone.
- Redundancy can make systems more reliable.
Disadvantages of Concurrency

- Runtime is not always reduced, so careful planning is required.
- Concurrent algorithms can be more complex than sequential algorithms.
- Shared data can be corrupted.
- Communications between tasks is needed.
 Achieving Concurrency

- Many computers today have more than one processor (multiprocessor machines)

Diagram:

- CPU 1
- CPU 2
- Memory

Connection:
- bus
Achieving Concurrency

- Concurrency can also be achieved on a computer with only one processor:
  - The computer “juggles” jobs, swapping its attention to each in turn
  - “Time slicing” allows many users to get CPU resources
  - Tasks may be suspended while they wait for something, such as device I/O
Concurrency vs. Parallelism

- **Concurrency** is the execution of multiple tasks at the same time, regardless of the number of processors.

- **Parallelism** is the execution of multiple processors on the same task.
Types of Concurrent Systems

- Multiprogramming
- Multiprocessing
- Multitasking
- Distributed Systems
Examples of Concurrent Algorithms

P/C Bounded-Buffer
Readers and Writers
Dining-Philosophers...
Outline

Sequential and Parallel Computing
RAM and PRAM Models
Amdahl’s Law
Brent’s Theorem
Parallel Algorithm Analysis and optimal parallel algorithms
Graph Problems
Concurrent Algorithms
Dining Philosophers Problem
Dining Philosophers Problem

Philosophers spend their lives alternating between thinking and eating.

Five philosophers can be seated around a circular table.

There is a shared bowl of rice.

In front of each one is a plate.

Between each pair of philosophers there is a chopstick, so there are five chopsticks.

It takes two chopsticks to take/eat rice, so while n is eating neither n+1 nor n-1 can be eating.
Dining Philosophers Problem

Each one thinks for a while, gets the chopsticks/forks needed, eats, and puts the chopsticks/forks down again, in an endless cycle.

Illustrates the difficulty of allocating resources among process without deadlock and starvation.
The challenge is to grant requests for chopsticks while avoiding deadlock and starvation.

Deadlock can occur if everyone tries to get their chopsticks at once. Each gets a left chopstick, and is stuck, because each right chopstick is someone else’s left chopstick.
The Dining Philosophers Problem

Philosophers
- Think
- Take Forks (One At A Time)
- Eat
- Put Forks (One At A Time)

Eating requires 2 forks

Pick one fork at a time

How to prevent deadlock?

What about starvation?

What about concurrency?

Idea is to capture the concept of multiple processes competing for limited resources
Dining Philosopher’s Problem (Dijkstra ’71)
What can go wrong?

**Starvation:** A policy that can leave some philosopher hungry in some situation

**Deadlock:** A policy that leaves all the philosophers “stuck”, so that nobody can do anything at all
An incorrect solution—**Deadlock is possible**

(→ means “waiting for this fork”)
An incorrect solution - Starvation is possible

Starvation!
Dining Philosophers Solution

- Each philosopher is a process.
- One semaphore per fork:
  - fork: array[0..4] of semaphores
  - Initialization: fork[i].count := 1 for i := 0..4

Process Pi:
repeat
  think;
  of
    wait(fork[i]);
    wait(fork[i+1 mod 5]);
    eat;
    signal(fork[i+1 mod 5]);
    signal(fork[i]);
forever

Note: deadlock if each philosopher starts by picking up his left fork!
Dining Philosophers Solution-

Possible solutions to avoid deadlock:

Allow at most four philosophers to be sitting simultaneously at the table. Allow a philosopher to pick up the forks only if both are available.

Use an asymmetric solution - an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
Dining Philosophers Solution

**A solution:** admit only 4 philosophers at a time that try to eat.

Then 1 philosopher can always eat when the other 3 are holding 1 fork.

Hence, we can use another semaphore T that would limit at 4 the number of philosophers “sitting at the table”.

Initialize: T.count := 4

Process Pi:
repeat
    think;
    wait(T);
    wait(fork[i]);
    wait(fork[i+1 mod 5]);
    eat;
    signal(fork[i+1 mod 5]);
    signal(fork[i]);
    signal(T);
forever
Thank You