MAXIMUM FLOW

- In the maximum-flow problem, we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints.
- It is one of the simplest problems concerning flow networks.

What is Network Flow ?

- Flow network is a directed graph $G=(V,E)$ such that each
- edge has a non-negative capacity $c(u,v)\geq 0$.
- Two distinguished vertices exist in G namely :
- Source (denoted by s) : In-degree of this vertex is 0.
- Sink (denoted by t) : Out-degree of this vertex is 0.

Flow in a network is an integer-valued function f defined On the edges of G satisfying $0\leq f(u,v)\leq c(u,v)$, for every Edge (u,v) in E.

What is Network Flow ?

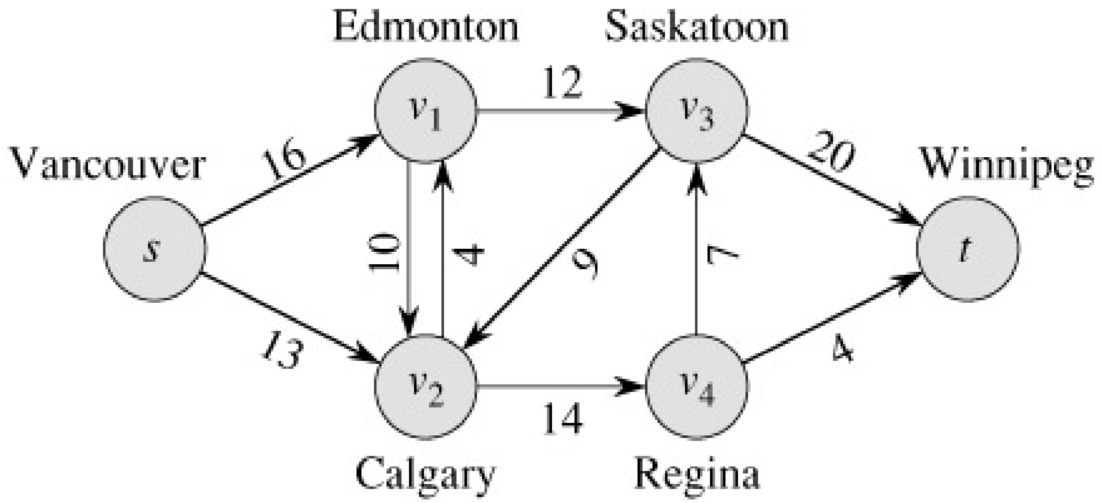
- Each edge (u,v) has a non-negative capacity c(u,v).

- If (u,v) is not in E assume c(u,v)=0.

- We have source s and sink t.

- Assume that every vertex v in V is on some path

Following is an illustration of a network flow: from s to t.



c(s,v1)=16

c(v1,s)=0

c(v2,s)=0 …

Conditions for Network Flow

For each edge (u,v) in E, the flow f(u,v) is a real valued function that must satisfy following 3 conditions :

- Capacity Constraint : $\forall$ u,v $\in$ V, f(u,v) $\leq$ c(u,v)
- Skew Symmetry : $\forall$ u,v $\in$ V, f(u,v)= -f(v,u)
- Flow Conservation: $\forall$ u $\in$ V – {s,t} $\sum_{v \in V}$ f(s,v)=0

Skew symmetry condition implies that f(u,u)=0.

The Value of a Flow.

The value of a flow is given by :

$$\boxed{|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)}$$

The flow into the node is same as flow going out from the node and thus the flow is conserved. Also the total amount of flow from source s = total amount of flow into the sink t.

Example of a flow

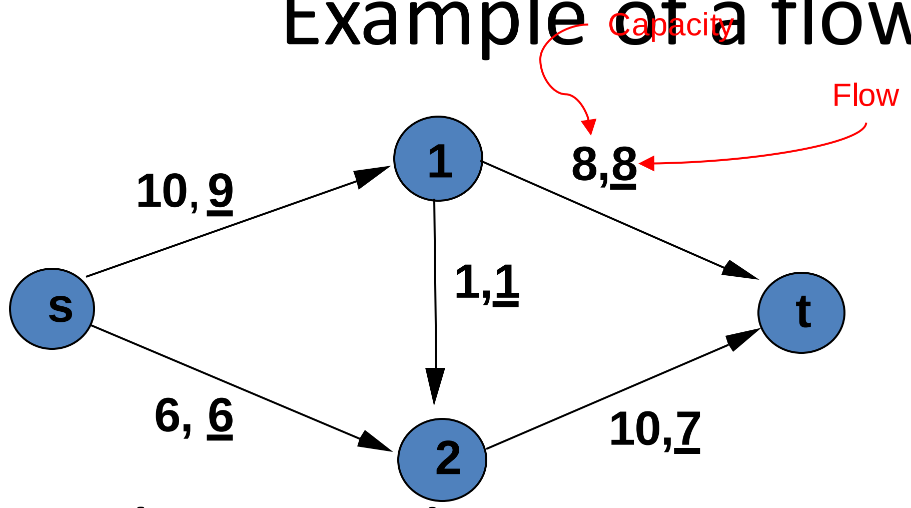


Table illustrating Flows and Capacity across different edges of graph above:

fs,1 = 9 , cs,1 = 10 (Valid flow since 10 > 9)

fs,2 = 6 , cs,2 = 6 (Valid flow since 6 ≥ 6)

f1,2 = 1 , c1,2 = 1 (Valid flow since 1 ≥ 1)

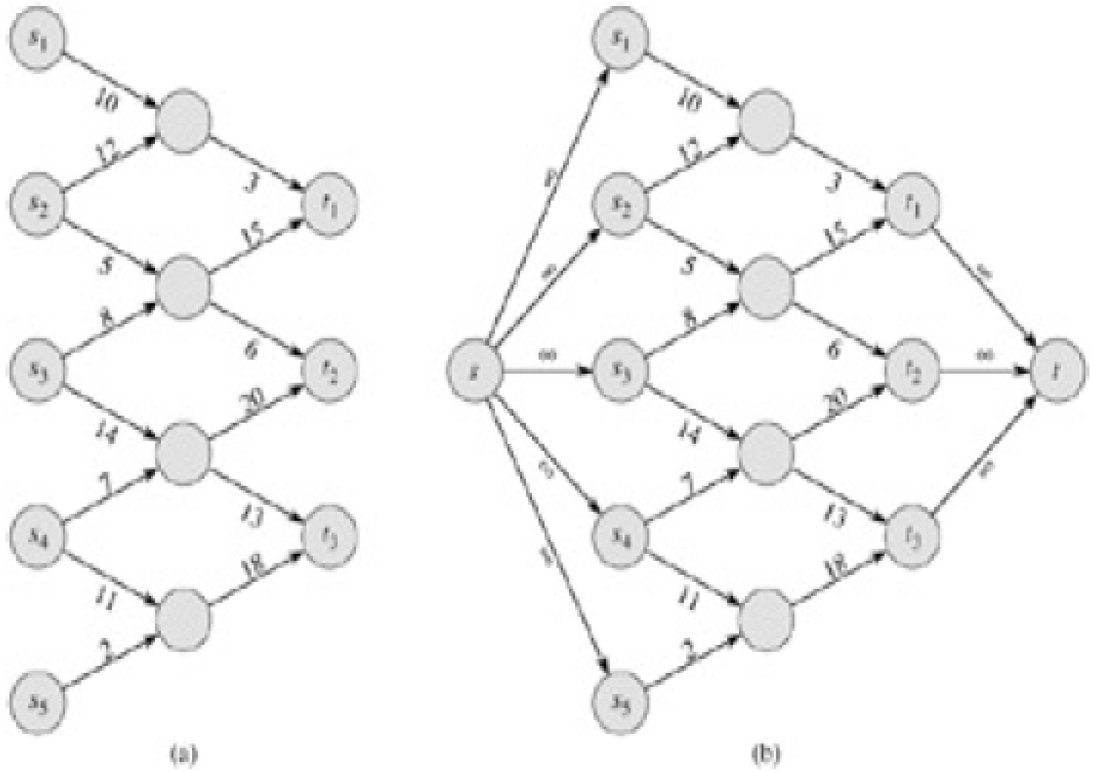
f1,t = 8 , c1,t = 8 (Valid flow since 8 ≥ 8)

f2,t = 7 , c2,t = 10 (Valid flow since 10 > 7)

The flow across nodes 1 and 2 are also conserved as flow into them = flow out

Networks with multiple sources and sinks

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Company, for example, might actually have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$.



Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. *(a)* A flow network with five sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ and three sinks $T = \{t_1, t_2, t_3\}$. *(b)* An equivalent single-source, single-sink flow network. We add a supersource s and an edge with infinite capacity from s to each of the multiple sources. We also add a supersink t and an edge with infinite capacity from each of

We can reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem. Figure 26.2(b) shows how the network from (a) can be converted to an ordinary flow network with only a single source and a single sink. We add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$. We also create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$. Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single source s simply provides as much flow as desired for the multiple sources s_i, and the single sink t likewise consumes as much flow as desired for the multiple sinks t_i. Exercise 26.1-3 asks you to prove formally that the two problems are equivalent

The Ford Fulkerson Method

The Ford-Fulkerson method is iterative. We start with $f(u, v) = 0$ for all u, v □ V, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can send more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD(G, s, t)
initialize flow f to 0
while there exists an augmenting path p
    do augment flow f along p
return f
```

Serial Algorithms & parallel algorithms

- **Serial Algorithms:** Suitable for running on an uniprocessor computer in which only one instruction executes at a time.
- **Parallel Algorithms:** Run on a multiprocessor computer that permits multiple execution to execute concurrently.

PARALLEL COMPUTERS

- Computers with multiple processing units.
- They can be:
  - **Chip Multiprocessors:** Inexpensive laptops/desktops. They contain a single multicore integrated-circuit that houses multiple processor "cores" each of which is a full-fledged processor with access to common memory.

PARALLEL COMPUTERS

- Computers with multiple processing units.
- They can be:
  - **Clusters:** Build from individual computers with a dedicated network system interconnecting them. Intermediate price/performance.

PARALLEL COMPUTERS

- Computers with multiple processing units.
- They can be:
  - **Supercomputers:** Combination of custom architectures and custom networks to deliver the highest performance (instructions per second). High price.

Models for parallel computing

- Although the random-access machine model was early accepted for serial computing, no model has been established for parallel computing.
- A major reason is that vendors have not agreed on a single architectural model for parallel computers.

Models for parallel computing

- For example some parallel computers feature **shared memory** where all processors can access any location of memory.
- Others employ **distributed memory** where each processor has a private memory.
- However, the trend appears to be toward **shared memory multiprocessor.**

Static threading

- Shared-memory parallel computers use **static threading**.
- Software abstraction of “virtual processors” or threads sharing a common memory.
- Each thread can execute code independently.
- For most applications, threads persist for the duration of a computation.

PROBLEMS OF STATIC THREADING

- Programming a shared-memory parallel computer directly using static threads is difficult and error prone.
- Dynamically partioning the work among the threads so that each thread receives approximately the same load turns out to be complicated.

PROBLEMS OF STATIC THREADING

- The programmer must use complex communication protocols to implement a scheduler to load-balance the work.
- This has led to the creation of **concurrency platforms**. They provide a layer of software that coordinates, schedules and manages the parallel-computing resources.

DYNAMIC MULTITHREADING

- Class of concurrency platform.
- It allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, etc.
- The concurrency platform contains a scheduler that load-balances the computation automatically.

DYNAMIC MULTITHREADING

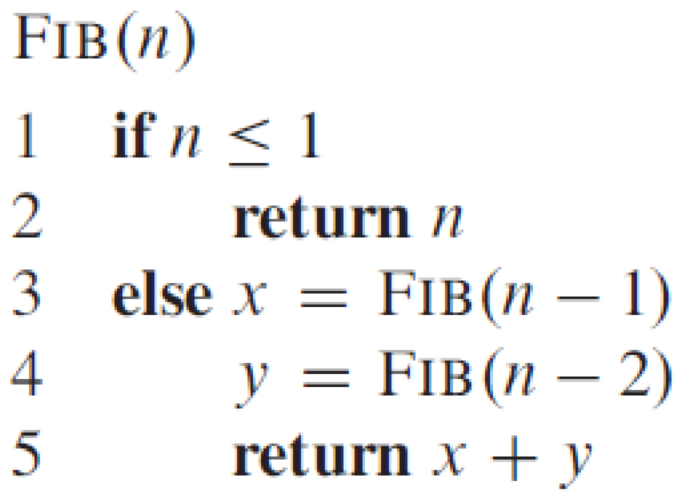
- It supports:
  - **Nested parallelism:** It allows a subroutine to be spawned, allowing the caller to proceed while the spawned subroutine is computing its result.
  - **Parallel loops:** regular for loops except that the iterations can be executed concurrently.

ADVANTAGES OF DYNAMIC MULTITHREADING

- The user only specifies the logical parallelism.
- Simple extension of the serial model with: **parallel, spawn** and **sync**.
- Clean way to quantify parallelism.
- Many multithreaded algorithms involving nested parallelism follow naturally from the Divide & Conquer paradigm.

BASICS OF MULTITHREADING

- Fibonacci Example
  - The serial algorithm: Fib(n)
  - Repeated work
  - Complexity
  - However, recursive calls are independent!
  - Parallel algorithm: P-Fib(n)



```
FIB($n$)
if $n \le 1$
    return $n$
else $x = $ FIB($n-1$)
    $y = $ FIB($n-2$)
    return $x + y$
```

Serialization

- Concurrency keywords: **spawn, sync** and **parallel**
- The serialization of a multithreaded algorithm is the serial algorithm that results from deleting the concurrency keywords.

NESTED PARALLELISM

- It occurs when the keyword **spawn** precedes a procedure call.
- It differs from the ordinary procedure call in that the procedure instance that executes the spawn - **the parent** – may continue to execute in parallel with the spawn subroutine – **its child** - instead of waiting for the child to complete.

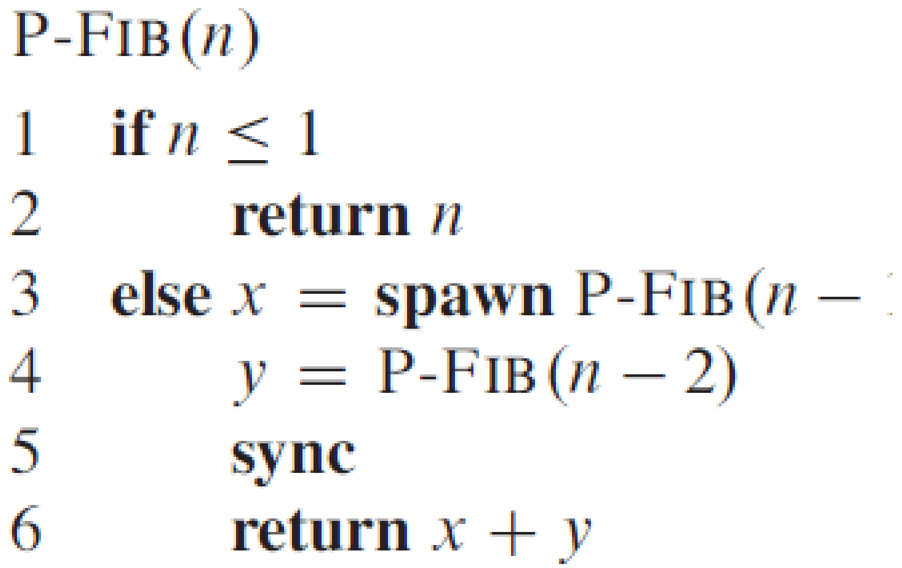
Keyword spawn

- It doesn’t say that a procedure **must** execute concurrently with its **spawned** children; only that it **may**!
- The concurrency keywords express the **logical parallelism** of the computation.
- At runtime, it is up to the **scheduler** to determine which subcomputations actually run concurrently by assigning them to processors.

Keyword sync

- A procedure cannot safely use the values returned by its spawned children until after it executes a **sync** statement.
- The keyword **sync** indicates that the procedure must wait until all its spawned children have been completed before proceeding to the statement after the sync.
- Every procedure executes a **sync** implicitly before it returns.

FIB procedure to use dynamic multithreading



```
P-FIB($n$)
if $n \leq 1$
        return $n$
else $x$ = spawn P-FIB($n -$
        $y$ = P-FIB($n - 2$)
        sync
        return $x + y$
```

Computational dag

- We can see a **multithread computation** as a directed acyclic graph G=(V,E) called a **computational dag**.
- The vertices are instructions and and the edges represent dependencies between instructions, where (u,v) є E means that instruction u must execute before instruction v.

Computational dag

- If a chain of instructions contains no parallel control (no **spawn, sync, or return**), we may group them into a single ***strand,*** *each of which represents* one or more instructions.
- Instructions involving parallel control are not included in strands, but are represented in the structure of the dag.

Computational dag

- For example, if a strand has two successors, one of them must have been spawned, and a strand with multiple predecessors indicates the predecessors joined because of a **sync**.
- Thus, in the general case, the set V forms the set of strands, and the set E of directed edges represents dependencies between strands induced by parallel control.

Computational dag

- If G has a directed path from strand u to strand, we say that the two strands are ***(logically) in series.*** *Otherwise, strands u and are* ***(logically) in parallel.***
- We can picture a multithreaded computation as a dag of strands embedded in a tree of procedure instances.
- Example!

Computational dag

- We can classify the edges:
  - ***Continuation edge :*** connects a strand u to its successor u’ within the same procedure instance.
  - **Call edges:** representing normal procedure calls.
  - **Return edges:** When a strand u returns to its calling procedure and x is the strand immediately following the next **sync** in the calling procedure.
- A computation starts with an **initial strand** and ends with a single **final strand**.