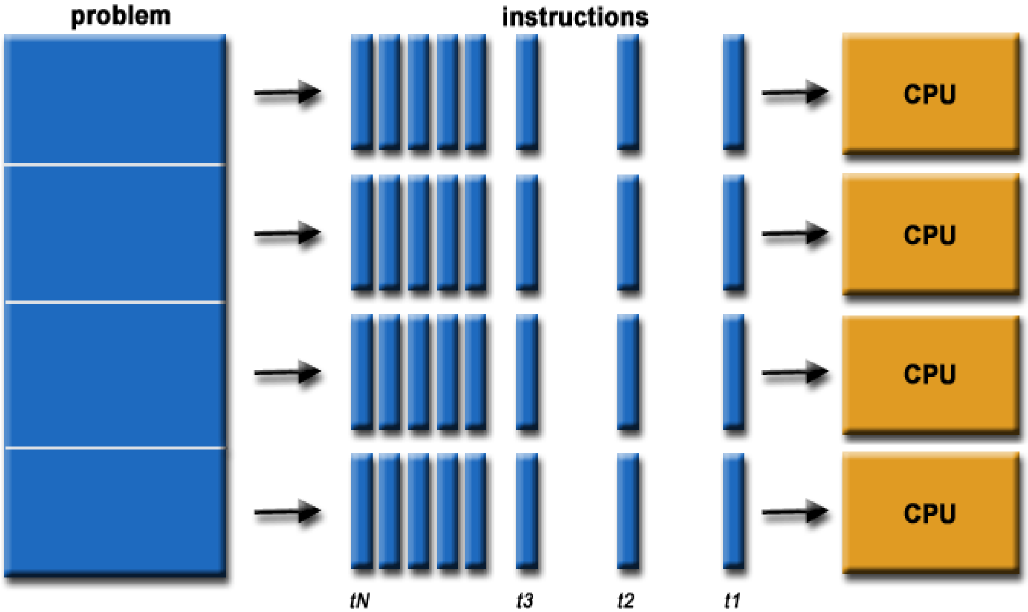
Parallel Computation

Computational Complexity
Christos H. Papadimitriou
Chapter 15

What is Parallel Computing?

Parallel computing: Many cooperating processors that are working together in order to solve the same problem instance.



Sections

I. Parallel Algorithms
II. Parallel Models of Computation
III. The Class NC

I. Parallel Algorithms

How effectively can parallelism attack to:

- Matrix Multiplication
- Graph Reachability
- Arithmetic Operations
- Maximum Flow
- The Traveling Salesman Problem

?

1. Matrix Multiplication (1/8)

The problem: Given two nxn matrices A, B compute their product C = A · B, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n$$

We wish to compute all the n^2 sums of the above form.

1. Matrix Multiplication (2/8)

Sequential Running Time:

- The *straightforward sequential algorithm* runs in $O(n^3)$ time.
- *Strassen algorithm* runs in $O(n^{2.807})$ time.
- The *Coppersmith–Winograd algorithm* is the fastest currently known sequential algorithm with running time $O(n^{2.376})$.

1. Matrix Multiplication (3/8)

Parallellization:

The straightforward sequential algorithm can be *readily parallelized.*

1. n^3 processors (i,k,j) compute $A_{ik} \cdot B_{kj}$ in 1 step.
2. n^2 processors, say (i,1,j) compute C_{ij} (by adding the n products corresponding to C_{ij}) in n-1 additional steps.

1. Matrix Multiplication (4/8)

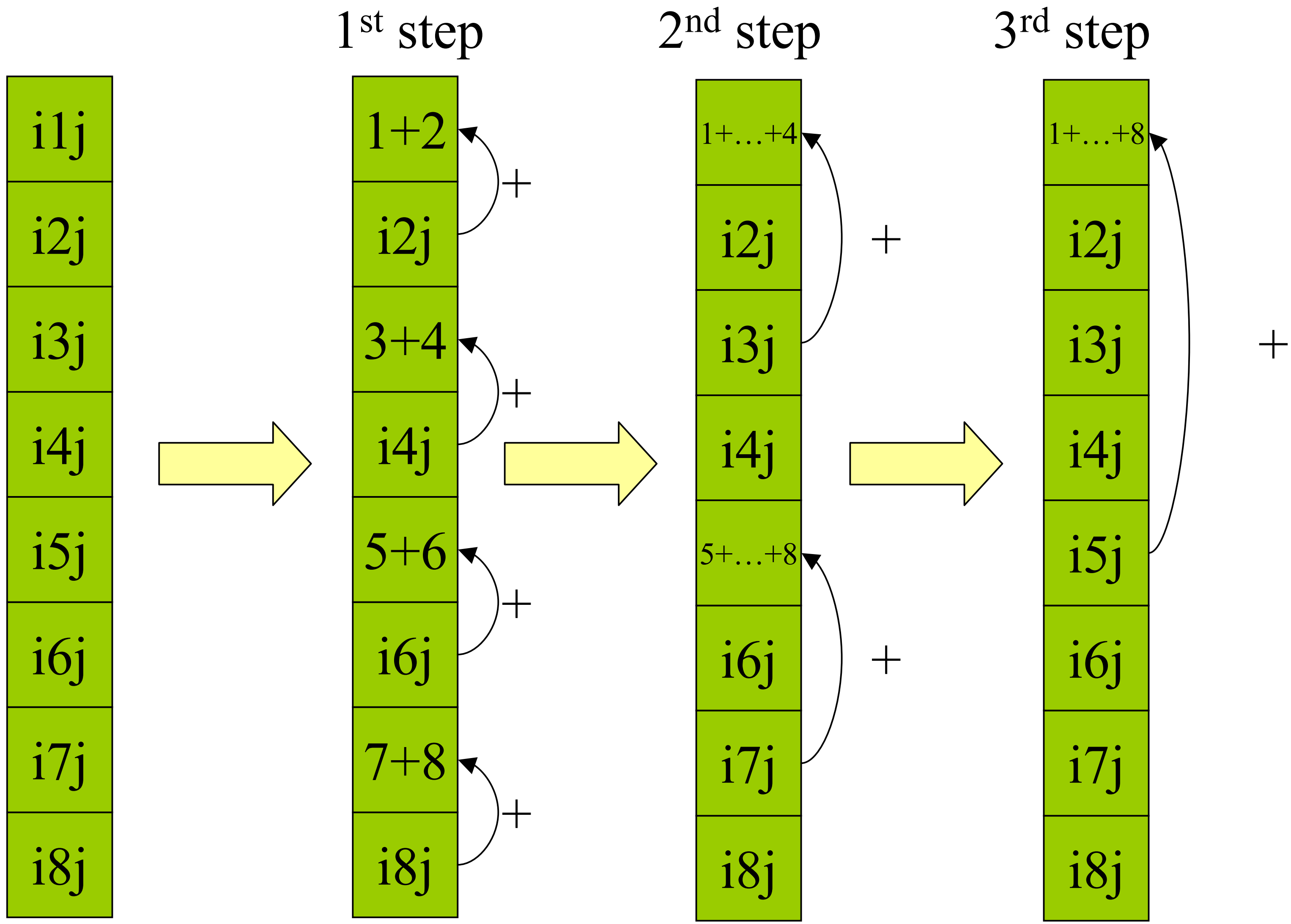
What we have done:

- The number of processors is n^3.
- The running time of the algorithm is n steps.

What we want:

- Polynomial number of processors. ✓
- An exponential drop on running time, i.e. logarithmic or polylogarithmic time (analogous to breaking the barrier between exponential and polynomial time in sequential algorithms). ✗

1. Matrix Multiplication (5/8)



1. Matrix Multiplication (6/8)

Are we satisfied?

- $\lceil \log n \rceil + 1 = O(\log n)$ time achieved. ✓
- We need at least logn steps for the additions.
- $work = \sum_{processors} {\#steps}/{processor}$
- The work should be at least $O(n^3)$ -the time complexity of the sequential algorithm-.
- A lower bound on the # of processors is $\left\lceil \frac{n^3}{\log n} \right\rceil$.

1. Matrix Multiplication (7/8)

Can we make it?

- n^3 multiplications ⇨ $\lceil n^3/\log n \rceil$ multiplications in logn *"shifts"*.
- We use (somehow???) the same $\lceil n^3/\log n \rceil$ processors to compute the first log logn parallel addition steps.
- Thus, the total running time is no more 2logn.

This technique of bringing down the processor requirement to the optimal value by using shifts of processors is known as *Brent's principle*.

1. Matrix Multiplication (8/8)

And if we don't have so many processors?…

If we have P processors (less than $n^3/\log n$), then we scale back our algorithm to the available hardware.

- P processors will execute each parallel step in $\left\lceil \frac{n^3/\log n}{P} \right\rceil$ shifts.
- The total running time is $2n^3/P$.

2. Graph Reachability (1/3)

The problem: Given a graph G(V, E) and two nodes s, t in V, examine if there exists a path from s to t.

The sequential algorithm:

- Set all nodes to the value *"unmarked"*.
- Define a set S := {s} and mark s.
- For every u in S:
  - remove u from S.
  - For every (u,v) in E add v in S and mark v.
- Stop when S is empty.
- If t is marked then answer "yes", otherwise answer "no".

2. Graph Reachability (2/3)

Parallelization

The sequential algorithm cannot be parallelized.

We can use *matrix multiplication*

- A is the adjacency matrix of G with self-loops ($A_{ii} = 1$).
- Compute $A^2 = A \cdot A$. Then $A^2_{ij} = 1$ iff there is a path of length at most 2 from i to j.
- Compute $A^4 = A^2 \cdot A^2$, then A^8 and so on.
- After $\lceil \log n \rceil$ matrix multiplications we get A^n, the adjacency matrix of the transitive closure of A.

2. Graph Reachability (3/3)

Complexity

- The running time of the algorithm is $O(\log^2 n)$. ✓
- The total work is $O(n^3 \log n)$
- By Brent's principle the # of processors is $O(n^3 / \log n)$. ✓

3. Arithmetic Operations (1/6)
(Prefix Sums)

The problem: Given a sequence x_i of n integers compute every sum of the form

$$\sum_{i=2}^{j} x_i, \quad j = 1, \ldots, n$$

The sequential algorithm:

1. Compute $x_1 + x_2$
2. Compute $x_1 + x_2 + x_3$

…

n-1. Compute $x_1 + x_2 + x_3 + \ldots + x_n$

3. Arithmetic Operations (2/6)
(Prefix Sums)

- The sequential algorithm runs in n-1 steps.
- It cannot be parallelized.

A parallel algorithm:

- Compute $x_1 + x_2$, $x_3 + x_4$, … , $x_{n-1} + x_n$ in one step.
- Solve the problem recursively for this sequence.
- Add x_{2i+1} to $x_1 + x_2 + \ldots + x_{2i}$ (i=1,…, n-1).

3. Arithmetic Operations (3/6)
(Prefix Sums)

Complexity

- $T(n) = T(n/2) + 2 = \ldots = T(n/2^i) + 2i = T(1) + 2\log n$. ✓
- $\text{Work} = n + n/2 + n/4 + \ldots \leq 2n$
- By Brent's principle # of processors needed is $n/\log n$. ✓

3. Arithmetic Operations (4/6)
(Binary Addition)

The problem: Given two binary numbers a, b, where $a = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \ldots + a_0 \cdot 2^0$ and $b = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \ldots + b_0 \cdot 2^0$, $(a_n, b_n = 0)$, find the sum $c = c_n \cdot 2^n + c_{n-1} \cdot 2^{n-1} + \ldots + c_0 \cdot 2^0$.

- The straightforward algorithm is O(n) but is tricky to parallelize.
- We will use prefix sum to get a parallel algorithm.

3. Arithmetic Operations (5/6)
(Binary Addition)

$c_i = a_i + b_i + z_{i-1}$ (with z_i we denote the carry that arises from this addition).

So, to compute c_i we must first compute the carry z_{i-1}.

We denote: $g_i = a_i \wedge b_i$ (generator) and $p_i = a_i \vee b_i$ (propagator).

So, $z_i = g_i \vee (p_i \wedge z_{i-1})$.

Also, $z_i = [g_i \vee (p_i \wedge g_{i-1})] \vee ([p_i \wedge p_{i-1}] \wedge z_{i-2})$.

Define $(a,b) \odot (a',b') = (a' \vee (b' \wedge a), b' \wedge b)$.

3. Arithmetic Operations (6/6)
(Binary Addition)

- We can compute z_i by computing

$((0,0)\odot((g_1,p_1)\odot\ldots\odot((g_{i-1},p_{i-1})\odot(g_i,p_i))\ldots))$ (HOW?)

- This can be treated as generalized prefix sums of the bit vectors $(0,0),(g_1,p_1),\ldots,(g_n,p_n)$ under the operation $\odot$.
- We can compute all carries z_i in 2logn parallel steps (6logn parallel Boolean operations since $\odot$ takes 3 operations).
- Computing the final result requires two additional parallel steps, thus the running time is O(logn). ✓
- The total work is O(n). ✓

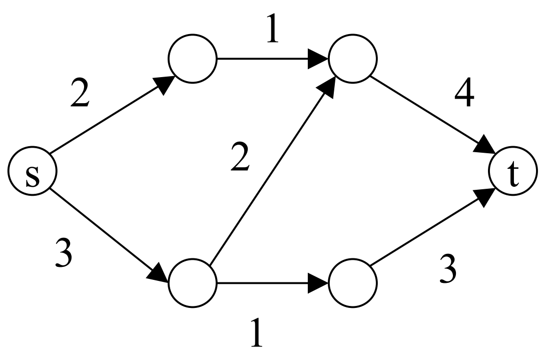
4. Maximum Flow (1/3)

The problem: Given a network N = (V, E, s, t, c), where s in V is the source, t in V is the sink and c: E → **N** is the capacity, compute the maximum flow f, where f: E → **N** and $f(i,j) \leq c(i,j)$ for every (i,j) in E.

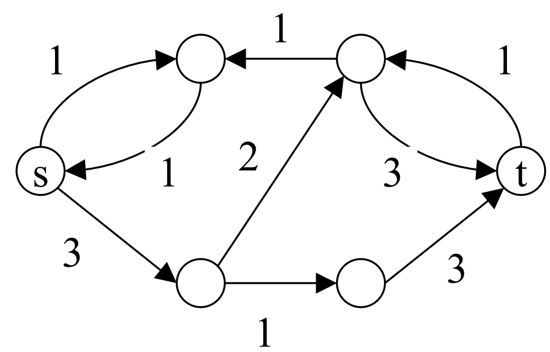
The problem can be solved sequentially in $O(n^5)$ time.

4. Maximum Flow (2/3)

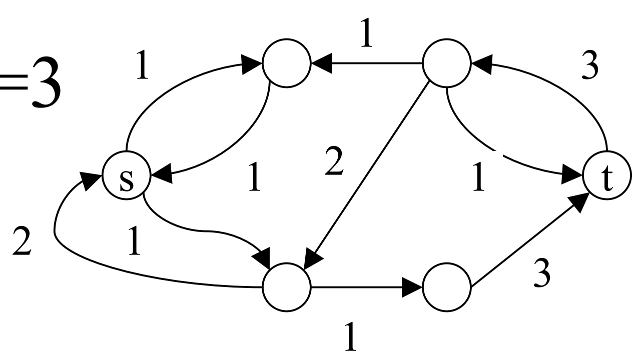
An Example



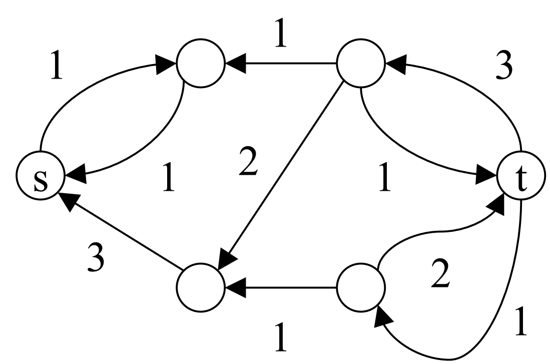
f=1



f=3



f=4



4. Maximum Flow (3/3)

What about parallelization?

- Every stage can be satisfactorily parallelized (1 step to compute N(f) and $O(\log^2 n)$ to solve *reachability*). ✓
- However stages must be built one after another and the # of stages may be large. ✗

It turns out that the problem cannot be parallelized. ☹

5. TSP (or any other NP-complete)

Parallelism: A remedy against NP-completeness?

Work = parallel time x # of processors

If the work is exponential then

- Parallel time must be exponential or (even worse…)
- # of processors must be exponential

Parallel computation ***is not*** *the answer to NP-completeness.*

II. Parallel Models of Computation

- Boolean Circuits
- The PRAM model

Boolean Circuits

Boolean circuit: an acyclic graph C(V,E) where nodes are called *gates*.

- Gates have in-degree 0, 1 or 2.
- Each gate i has a sort s(i) in {true, false, $\vee$, $\wedge$, $\neg$ } U $\{x_1, x_2, \ldots\}$.
- Nodes with in-degree 0 are called *input gates*.
- Nodes with out-degree 0 are called *output gates*.
- The *size* of C is the total number of gates.
- The *depth* of C is the number of nodes in longest path.

Boolean Circuits

Circuit Family: A sequence $C = (C_0, C_1, \ldots)$ of Boolean Circuits, s.t. C_i has i inputs.

Uniform Circuit Family: There is a logarithmic-space bounded Turing Machine which on input 1^n outputs C_n.

The *parallel time* of C is at most f(n) ($f(n)$: $\mathbf{N} \rightarrow \mathbf{N}$) if for all n the depth of C_n is at most f(n).

The *total work* of C is at most g(n) ($g(n)$: $\mathbf{N} \rightarrow \mathbf{N}$) if for all $n \geq 0$ the size of C_n is at most g(n).

PT/WK(f(n), g(n)): the class of all languages L (subset of $\{0, 1\}^*$), s.t. there is a uniform family of circuits deciding L in O(f(n)) parallel time and O(g(n)) work.

Boolean Circuits

Example: Reachability is in PT/WK($\log^2 n$, $n^3 \log n$)

Define the uniform class C as follows:

- For each n, there is a circuit Q_n with n^2 inputs (the adjacency matrix A) and n^2 outputs (A^2).
- The depth of Q_n is logn.
- C_n is the composition of logn copies of Q_n (the output of ones is the input of the next) and computes the transitive closure.

Parallel Random Access Machines

RAM program:

- a finite sequence $\Pi = (\pi_0, \pi_1, \ldots, \pi_m)$ of instructions (READ, ADD, LOAD, JUMP, etc.) with arguments standing for the contents of registers.
- Register 0 is the *accumulator* of the RAM (the result of the current operation is stored there).
- There is a program counter κ that shows the instruction to be executed.
- There is also a set of *input registers* $I = (i_0, i_1, \ldots, i_m)$.

Parallel Random Access Machines

PRAM program:

- A sequence of RAM programs $P = (\Pi_0, \Pi_1, \ldots, \Pi_q)$.
- Each of these machines executes its own program, has its own program counter and its own accumulator (Register i for the RAM i), but they all share (can both read and write) all registers.
- There is no Register 0.
- q is a function of the size m of the input I and the total length of the integers in the input I, $n = l(I)$.

Parallel Random Access Machines

How realistic are they?

- PRAM is closer to the way we are thinking and designing algorithms.
- It is extremely (unrealistic) powerful parallel computer.
- However, it comes up that it is equivalent with Boolean Circuits.

III. The Class NC

Definitions

- NC = PT/WK($\log^k n$, n^k): the problems solvable in polylogarithmic parallel time with polynomial amount of work.
- NC is the class decided by PRAMs in polylogarithmic time and with polynomially many processors.
- NC_j = PT/WK($\log^j n$, n^k).

NC and NC_j

Relations:

- NC_j is a subset of NC for every j.
- If $NC_j = NC_{j+1}$ then $NC_j = NC$.
- NC is a subset of P, since we have polynomial amount of work.
- It is unknown if P is a subset of NC.
- There are problems likely to be “inherently sequential”.
- We turn to reductions and completeness.

P-completeness

- **log-space reduction:** a reduction computable by a deterministic TM using logarithmic space.
- P-complete problems are the least likely to be in NC.
- log-space reductions preserve parallel complexity.

P-completeness

Theorem: If L reduces to L' and L' is in NC) then L is in NC.

Proof:

- Let R be a log-space reduction from L to L'.
- There exists a log-space bounded TM R' which accepts input (x,i) iff the ith bit of R(x) is 1 (i is the binary representation of an integer no larger than |R(x)|).

P-completeness

Proof (continued):

- By solving *reachability* in the configuration graph of R' on input (x, i) we can compute the ith bit of R(x).
- If we solve all these problems in parallel we can compute R(x)
- We can now use the NC circuit for L' to decide if x is in L.

P-completeness

Odd Max Flow: Given a network N(V, E, s, t, c), is the maximum flow value odd?

Circuit Value: Returns yes, if the output value is 1, else 0.

Monotone Circuit: A circuit with only AND, OR gates.

Theorem: Odd Max Flow is P-complete

P-completeness

Proof:

- Odd Max Flow is in P
- Monotone Circuit Value is reduced to OMF.

We are given a monotone circuit C. Assume that:

- The output gate is OR.
- No gate has out-degree greater than two.
- The gates have labels 0, …, n. (every gate has smaller label than its predecessor).

P-completeness

Proof (continued):

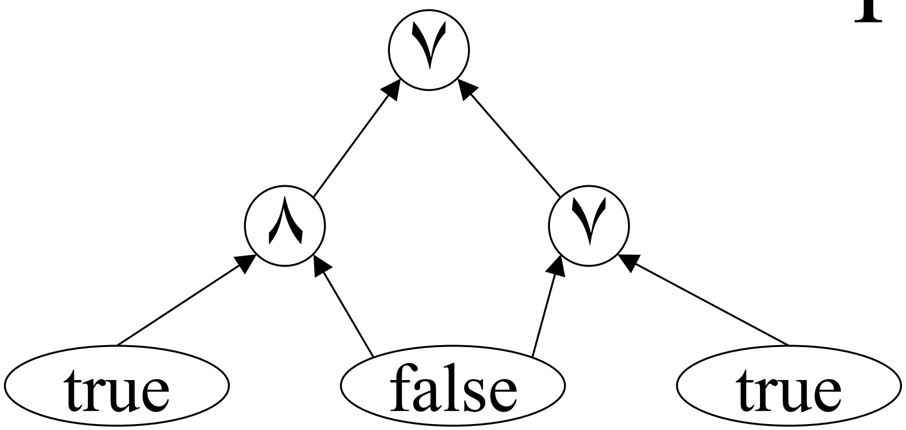
The network's N(V, E, s, t, c) construction is as follows:

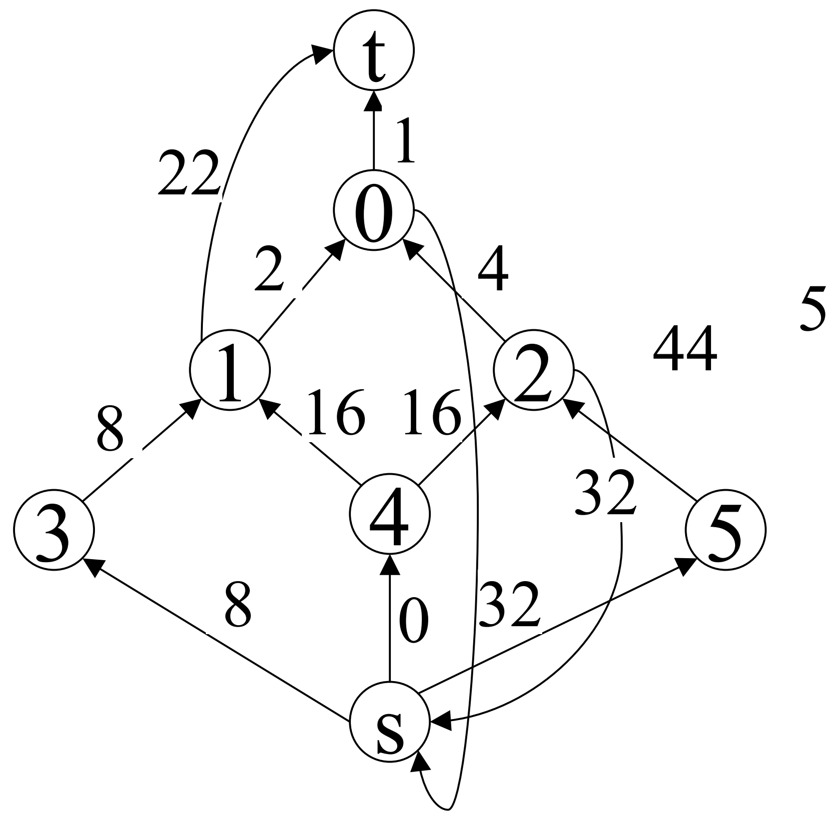
- V = the gates of C plus two nodes s, t
- s is the source and
- t the sink.

P-completeness

- For E and c, we have the following edges (and capacities):
  - From s to every **true** input gate i with capacity $d2^i$.
  - From every **true** or **false** gate i to its successor with capacity 2^i.
  - From every predecessor of an AND or OR gate i to i with capacity 2^i.
  - From the output gate to t with capacity 1.
  - From an AND gate i to t with capacity S(i) (the *surplus*).
  - From an OR gate i to s with capacity S(i).

P-completeness

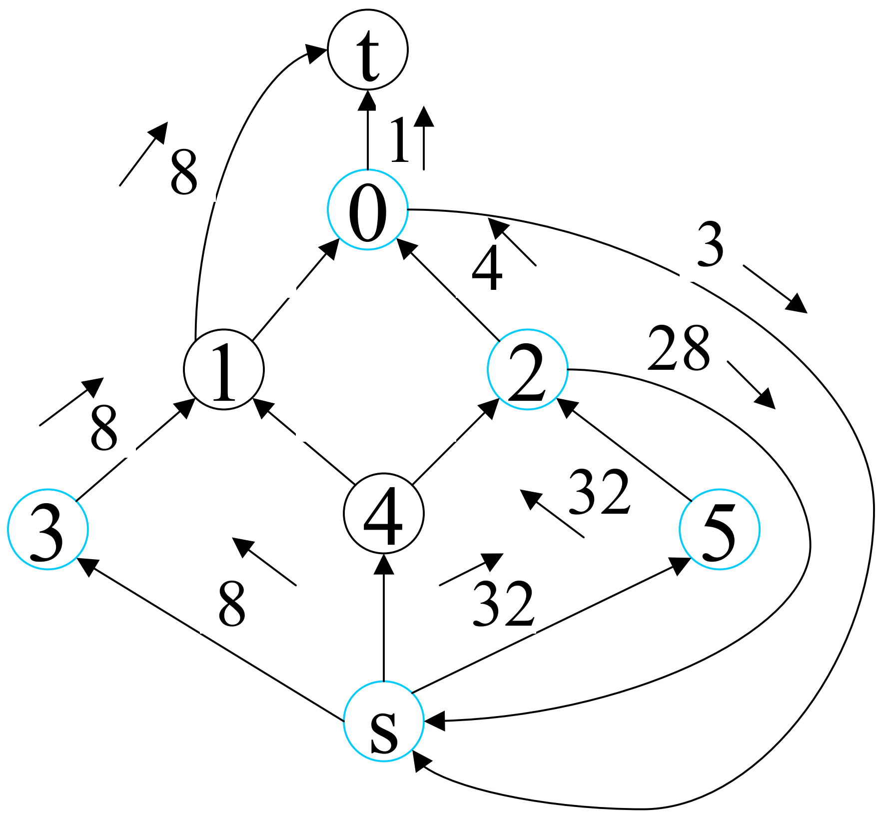




P-completeness

- A gate is full with respect to a flow f if its outgoing edges are filled to capacity
- A gate is empty if they have zero flow
- A flow is standard if all gates that have value **true** are full and all tha have value **false** are empty.

P-completeness



P-completeness

1. **Always there exists a standard flow**
2. **The standard flow is the maximum flow**

1. Fill the edges from s. Fill the others by induction on the depth of the gates.
2. If a flow f equals to a cut c then f = max f (max f $\geq$ f, c $\geq$ min cut, max f = min cut). We will find a cut that equals standard f. We denote two groups: {s, all gates with value true}, {t, all gates with value false}. Edges that are going from the first set to the second are full. So f is maximum.

P-completeness

All the flow values in the standard flow are even integers except of the edge from the output to t.

Hence:

the value of maximum (standard) flow is odd $\Longleftrightarrow$

the output gate is full $\Longleftrightarrow$

the value of the output is true.