

# **TRAFFIC FLOW OPTIMISATION USING PARALLEL PROGRAMMING**

A Major Project Report

submitted in partial fulfilment of the requirements for the degree of

**BACHELOR OF TECHNOLOGY**

**IN**

**CIVIL ENGINEERING**

By

**Shishir S Volety (17CV143)**

**Siddharth Panasa (17CV127)**

**Gokul Kumar (17CV115)**

**Rahul K (17CV133)**

Under the guidance of

**Assistant Professor. MITHUN MOHAN**



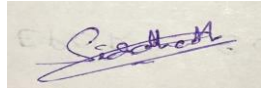
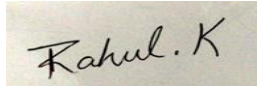
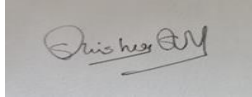
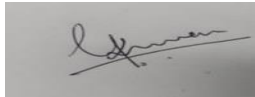
**DEPARTMENT OF CIVIL ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA  
SURATHKAL, MANGALORE - 575025**

**APRIL 2021**

# DECLARATION

We hereby declare that the Report of the B. Tech Major Project Work entitled “*Traffic Flow Optimization using Parallel Programming*”, which is being submitted to the National Institute of Technology Karnataka, Surathkal, in fulfilment of the requirements of the course titled “Major Project” as part of the Bachelor of Technology in Civil Engineering is a bonafide report of the project work carried out by us. The material contained in this Report has not been submitted to any University or Institution prior.

Register Number	Name	Signature
17CV127	SIDDHARTH P	
17CV133	RAHUL K	
17CV143	SHISHIR S	
17CV115	GOKUL KUMAR	

Department of Civil Engineering

**Place:** NITK, Surathkal

**Date:** April 2021

# CERTIFICATE

This is to certify that the Major Project (Course code CV499) entitled “Vehicle routing”, submitted by Mr. P SIDDHARTH (17CV127), RAHUL K (17CV133), Mr.GOKUL KUMAR(17CV115) and Mr. SHISHIR S VOLETY (17CV143) is accepted as the record of work carried out by them as the part of a Major Project in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology in Civil Engineering** of the Department of Civil Engineering, National Institute of Technology Karnataka, Surathkal, Mangalore.

**Asst Prof.Mithun Mohan**

Project Guide

Dept. of Civil Engineering

**Dr. B. R. Jayalekshmi**

Professor and Head

Dept. of Civil Engineering

# ACKNOWLEDGEMENT

Firstly, we would like to thank our guide and our faculty advisor Asst Prof.Mithun Mohan for his guidance in getting us acquainted with our research topic. His expert guidance and constant encouragement throughout the project played a vital role in bringing the project to good form. We express our gratitude to Asst Prof.Mithun Mohan, Professor of Civil department for extending all necessary support to present and complete the work being reported in this report.

We extend our heartfelt thanks to all the Teaching and Non-Teaching Staff of Civil Engineering Department.

**Place:** NITK, Surathkal

**Date:** April 2021

# ABSTRACT

In the report, we will be using the Push relabel algorithm in graph theory used to calculate the maximum movement of a road network. The problem of high flow is one of the most important problems in the idea of network flow. Push-Relabel algorithm is a simple algorithm for solving high flow problems and is based on the idea of finding a way to add from the starting source node to the sink area. It is widely used for similar performance problems and for various computer applications. The purpose of this report is to determine the maximum possible flow from source 's' node to target 't' node over the road network. We are using an algorithm in CUDA and the simulation results show that this same algorithm works well. To reduce the length of time, the same Push relabel algorithm is used. All arcs in a computer flow network are processed simultaneously in the same steps throughout the iteration.

# CONTENTS

1. Introduction	
1.1. Solutions we tested	8
1.2. Parallel programming	8
2. Literature review	
2.1. Concept	9
2.2. Project	9
2.3. CUDA	9
3. Methodology	
3.1. Methodology	10
3.2. Ford Fulkerson algorithm	10
3.3. Push Relabel algorithm	13
3.4. Dinic's algorithm	17
3.5. Why Push Relabel?	19
3.5.1. Time complexity	19
3.5.2. Time complexity of Push Relabel	20
3.5.3. Push Relabel over other algorithms	21
3.5.4. Differences with Ford Fulkerson & Dinic's algorithm	21
3.6. Parallel programming	22
3.7. CUDA parallel programming	22
3.8. Serial v/s Parallel processing	23
3.9. Parallel implementation of Push Relabel	23
3.10. Performance metrics	29
3.11. Validation of the Push Relabel algorithm	30
4. Data	
4.1. Data	31
5. Result	33
6. Conclusion	34
7. References	35

## Figures and Tables:

• Fig 1: Components of road network	10
• Fig 2: Ford Fulkerson road network	12
• Fig 3: Solution	12
• Fig 4: Parallel processing using CUDA	22
• Fig 5: Parallel programming	23
• Fig 6: Road network	31
• Table 1: Dataset of road network	31
• Table 2: Road capacity calculation table	32

# 1. INTRODUCTION

A road network can be modelled as a graph with a set of nodes representing crosses and a set of edges representing the sections of the road between intersections. In this report, we study the problem of traffic flow, where the capacity on the edges represents the strength of the roads.

## 1.1 Solutions tested:

- Push relabel algorithm for maximum flow
- Ford Fulkerson algorithm for maximum flow
- Dinic's algorithm for maximum flow

We used the Push relabel algorithm since the algorithm has a strongly polynomial time complexity and it is asymptotically more efficient than the other two.

**1.2 Parallel programming:** Parallel computing refers to the process of breaking down larger problems into smaller, independent, often similar parts that can be executed simultaneously by multiple processors communicating via shared memory, the results of which are combined upon completion as part of an overall algorithm. The primary goal of parallel computing is to increase available computation power for faster application processing and problem solving.

We used parallel programming to get a faster output by using CUDA.



## 2. LITERATURE REVIEW

### 2.1 Concept:

Parallel computing is a type of computation where many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

### 2.2 Project overview:

This mainly involves tweaking the already existing Data Flow Optimisation algorithms (Ford-Fulkerson, Push-Relabel, and Dinic's algorithm). Since all of these algorithms are focused on optimizing flow concerning a single source and sink nodes, the primary modification would be to make it accommodate instances of multi-source or multi-sink or both, since road networks are not as straightforward. Then, CUDA, a framework built around C-programming language will be used to run the code parallelly on the GPU (NVIDIA-GPUs only). Unlike the serial execution which instantiates a single thread per core, this instantiates up to 1024 threads (varies depending on the GPU specifications), which would lead to a faster runtime. Since traditional serial programming is a long process that does not give real-time solutions, this modified execution overcomes this limitation, thereby finding an application in general vehicular traffic flow.

### 2.3 CUDA:

CUDA is a parallel computing platform and programming model developed by Nvidia for general computing on its own GPUs (graphics processing units). CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation. CUDA has improved and broadened its scope over the years, in lockstep with improved Nvidia GPUs. As of CUDA version 9.2, using multiple P100 server GPUs, you can realize up to 50x performance improvements over CPUs. The V100 (not shown in this figure) is another 3x faster for some loads. The previous generation of server GPUs, the K80, offered 5x to 12x performance improvements over CPUs.

# 3. METHODOLOGY

## 3.1 Methodology:

This problem statement for optimizing traffic flow can be viewed as a modified variation of the max-flow min-cut problem. In the concept of computer science and efficiency, the max-flow min-cut theorem states that in a flow network, the maximum flow rate passing from the source to the sink is equal to the total weight of the edges with a small cut, i.e., the least capacity edge present leading to the sink.

This theorem has given rise to many branch algorithms used to solve the max-flow problem. In this paper we come to analyse and optimize the most suitable algorithm from this pool with regards to traffic flow. The algorithms we are considering are as follows:

- Ford-Fulkerson Algorithm
- Push-Relabel Algorithm
- Dinic's Algorithm

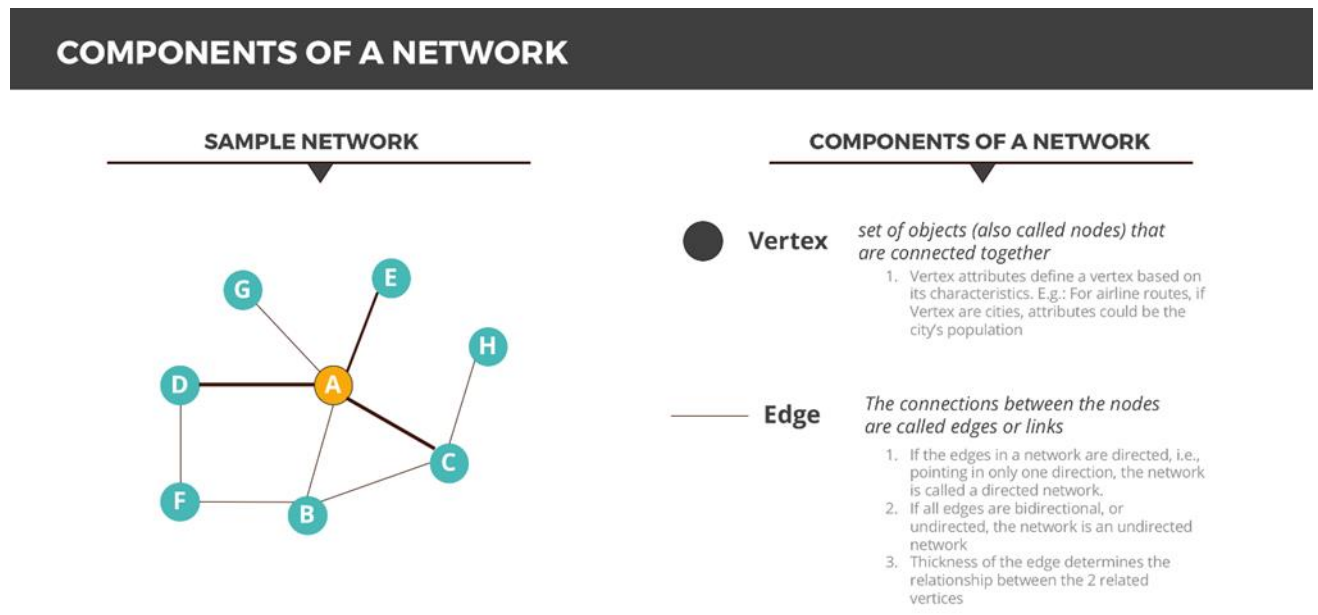


Fig 1. Components of a network

### 3.2 Ford Fulkerson Algorithm:

In the standard Ford-Fulkerson algorithm (e.g., Ford and Fulkerson, 1957, 1962; West, 2001; Gross and Yellen, 2006), it is assumed that a network can be represented by a directed graph with a maximum of one directed edge joining each pair of nodes. It is also assumed that the capacity  $C(i, j)$  of directed edge  $(i, j)$  from node 'i' to node 'j' is fixed for all pairs of nodes 'i', 'j' and that the maximum flow from a single source node 's' to a single target node 't' is required. In this section we describe the modified version of the algorithm, which can compute the maximum flow in a network with a specified set  $S$  of sources to a specified set  $T$  of sinks, where ' $S$ ', ' $T$ '. It is assumed that the edges in the network can be directed or undirected, that the capacities of an edge can be different in the two possible edge directions, and that the capacity of an edge depends on traffic speed or traffic density. A key step in the Ford-Fulkerson algorithm is the construction of an augmenting path from a given source 's' to a given target 't' in a directed graph with given flows and capacities. If no 's', 't' path exists, then the maximum flow has been found. If a 's', 't' path exists then the flow can be increased by constructing an augmenting path.

For one-way streets, there is only one possible direction of flow and therefore assigning an edge in a path as forward or backwards is always easy. For two-way streets, there are two possible directions of flow and assigning an edge as forward or backwards depends on the direction of flow through the edge. For a two-way edge  $(i, j)$ , we assign edge directions as follows. Firstly, if the flow from node 'i' to node 'j' is  $F(i, j) > 0$  then the flow from node 'j' to node 'i' is defined to be  $F(j, i) = 0$  and 'i' to 'j' is defined as the direction of the edge. If the flow on a two-way edge between nodes 'i' and 'j' is zero, i.e.,  $F(i, j) = F(j, i) = 0$ , then the direction is initially taken as not defined. However, if a two-way edge with zero flow appears in a 's', 't' path, then the direction of the edge is defined to be the "forward" s-t direction and the flow can then be increased in this forward direction. Note that the direction assigned to a two-way edge in the algorithm can be changed, but only when the flow through it is zero. In our algorithm, we use an "adjacency matrix"  $A$  to specify the state of each edge in the network before each breadth-first search for an 's', 't' path.

**Algorithm Implementation is as follows:**

1. for each edge  $(u, v) \in E [G]$
2. do  $f [u, v] \leftarrow 0$
3.  $f [u, v] \leftarrow 0$
4. while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ .
5. do  $cf (p) \leftarrow \min \{ Cf (u, v) : (u, v) \text{ is on } p \}$
6. for each edge  $(u, v)$  in  $p$
7. do  $f [u, v] \leftarrow f [u, v] + cf (p)$
8.  $f [u, v] \leftarrow -f [u, v]$

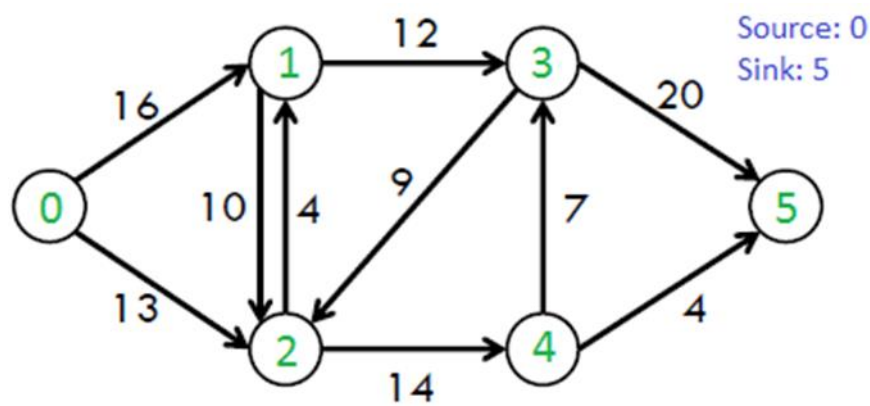


Fig 2. Ford Fulkerson road network

Solution:

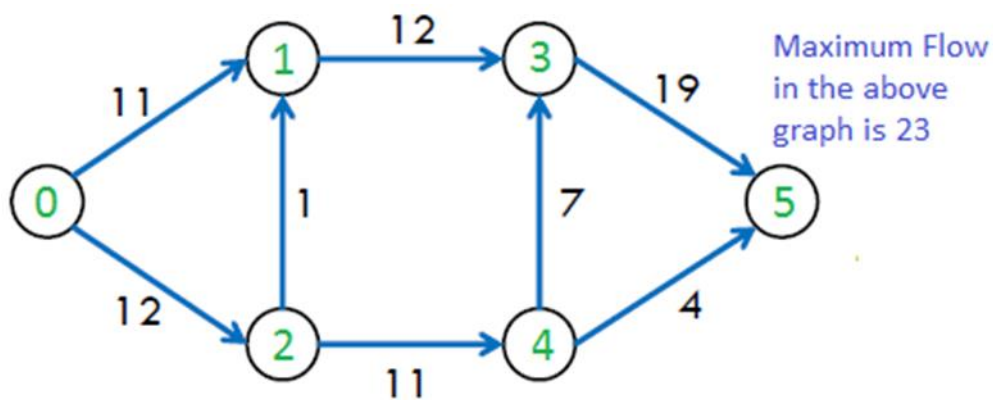


Fig 3. Solution

### 3.3 Push-Relabel Algorithm:

Assumptions or Invariants:

- $h(s) = 'n'$  always (where  $'n' = |V|$ )
- $h(t) = 0$
- for every edge  $(v, w)$  of the current residual network (with positive residual capacity),  $h(v) \leq h(w) + 1$ .

The high-level strategy of the algorithm is to maintain the three invariants above while trying to zero out any remaining excesses. Let us begin with the initialization. Since the invariants reference both a correct preflow and current vertex heights, we need to initialize both. Let us start with the heights. Clearly, we set  $h(s) = n$  and  $h(t) = 0$ . The first non-trivial decision is to set  $h(v) = 0$  also for all  $v \neq s, t$ . Moving onto the initial preflow, the obvious idea is to start with the zero flow. But this violates the third invariant: edges going out of  $s$  would travel from height  $n$  to 0, while edges of the residual graph are supposed to only go downhill by 1. With the current choice of height function, no edges out of  $s$  can appear (with non-zero capacity) in the residual network. So, the obvious fix is to initially saturate all such edges.

All three invariants hold after the initialization (the only possible violation is the edges out of  $s$ , which do not appear in the initial residual network). Also,  $f$  is initialized to a preflow (with  $\text{inflow} \geq \text{outflow}$  except at  $s$ ). Next, we restrict the Push operation so that it maintains the invariants. The restriction is that flow is only allowed to be pushed downhill in the residual network.

Every iteration, among all vertices that have positive excess, the algorithm processes the highest one. When such a vertex is chosen, there may or may not be a downhill edge emanating from  $v$ .  $\text{Push}(v)$  is only invoked if there is such an edge (in which case Push will push flow on it), otherwise the vertex is “relabelled” meaning its height is increased by one.

Neither  $s$  nor  $t$  ever get relabeled, so the first two invariants are always satisfied. For the third invariant, we consider separately a relabel (which changes the height function but not the preflow) and a push (which changes the preflow but not the height function). The only worry with a relabel at  $v$  is that, afterwards, some outgoing edge of  $v$  on the residual network goes downhill by more than one step. But the precondition for relabeling

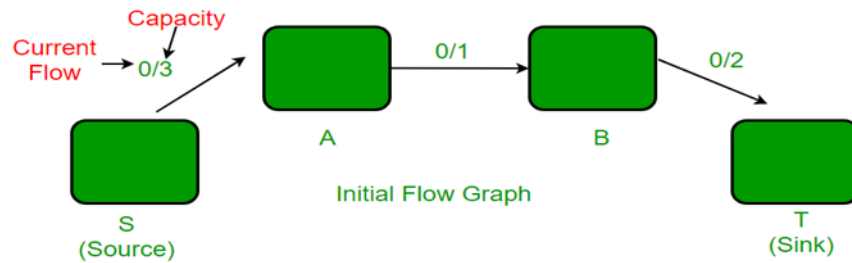
is that all outgoing edges are either flat or uphill, so this never happens. The only worry with a push from 'v' to 'w' is that it could introduce a new edge (w, v) to the residual network that might go downhill by more than one step. But we only push flow downward, so a newly created reverse edge can only go upward. The claim implies that if the push-relabel algorithm ever terminates, then it does so with a maximum flow. The invariants imply the maximum flow optimality conditions (no s-t path in the residual network), while the termination condition implies that the final preflow 'f' is in fact a feasible flow.

**Algorithm Implementation is as follows:**

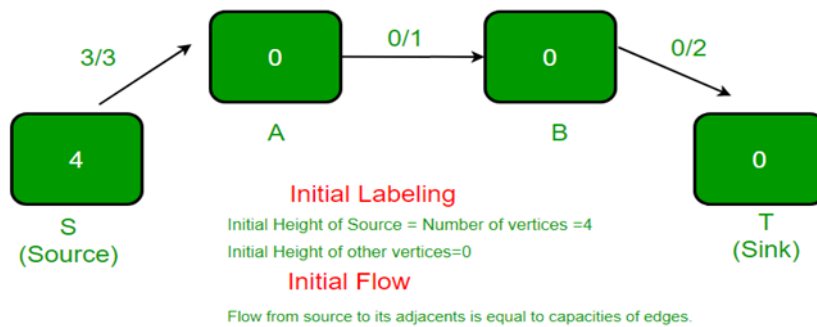
1. set  $h(s) = n$
2. set  $h(v) = 0$  for all  $v \neq s$
3. set  $f_e = u_e$  for all edges 'e' outgoing from 's'
4. set  $f_e = 0$  for all other edges
5. choose an outgoing edge (v, w) of 'v' in  $G_f$  with  $h(v) = h(w) + 1$  (if any)
6. let  $\Delta = \min\{\alpha f(v), \text{residual cap. of } (v, w)\}$
7. push  $\Delta$  units of flow along (v, w)
8. while there is a vertex  $v \neq s, t$  with  $\alpha f(v) > 0$  do :
  - a. choose such a vertex 'v' with the maximum height  $h(v)$
9. if there is an outgoing edge (v, w) of 'v' in  $G_f$  with  $h(v) = h(w) + 1$ 
  - a. then Push(v)
10. else
  - a. increment  $h(v)$

## Visual Explanation:

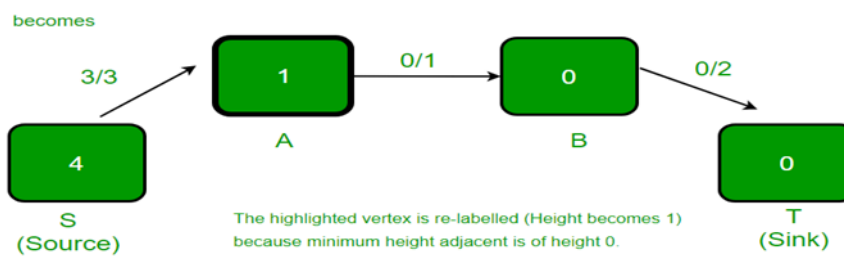
1. Initial given flow:



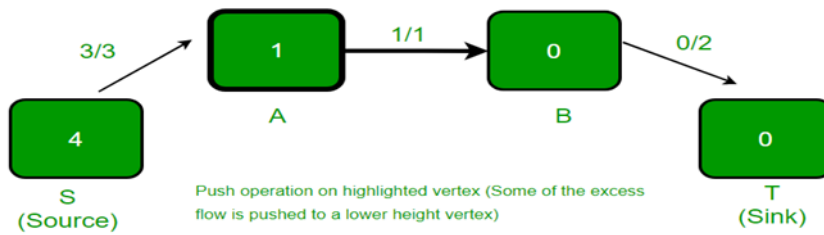
2. After Preflow:



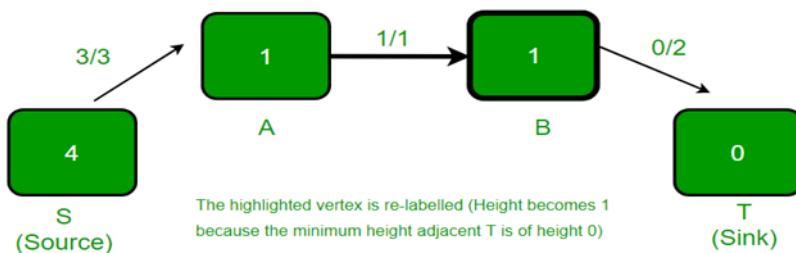
3. Height adjustment and flow transfer:



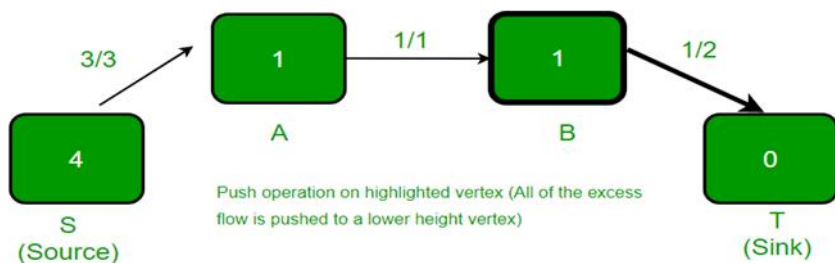
4. Push operation on highlighted edge:



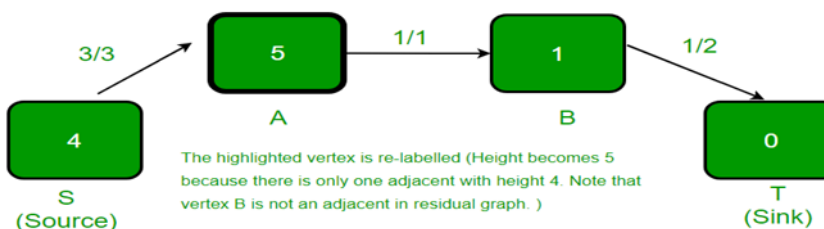
5. Highlighted edge is relabeled:



6. Excess flow at highlighted vertex is balanced:

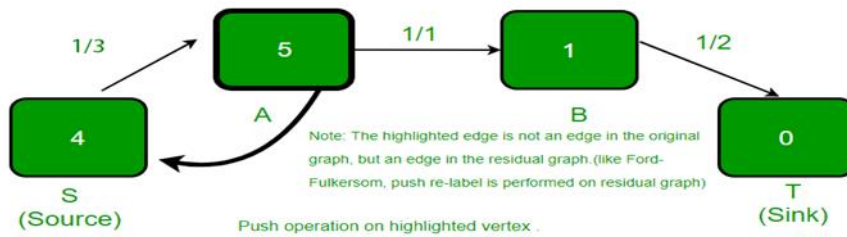


7. All edges with excess flow are consequently relabeled:

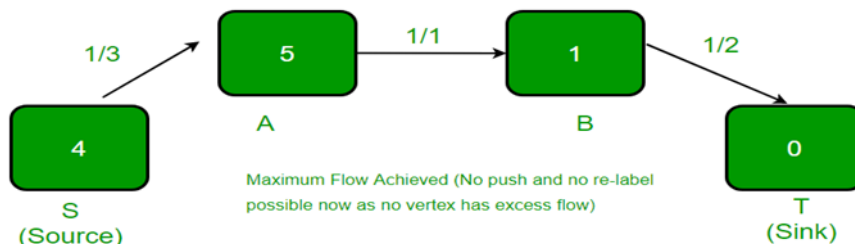




8. All excess flow sent back to source:



9. Final check for total\_excess\_flow of entire network:



### 3.4 Dinic's Algorithm:

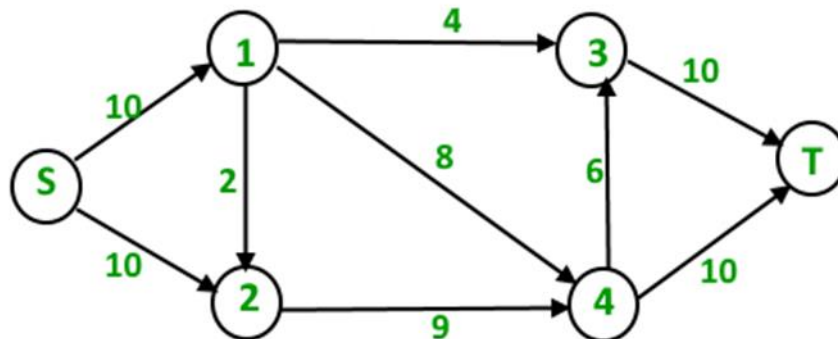
Dinic's algorithm or Dinitz's algorithm is a strongly polynomial algorithm for computing the maximum flow in a flow network, conceived in 1970 by Israeli (formerly Soviet) computer scientist Yefim (Chaim) A. Dinitz. The algorithm runs in  $O(V^2E)$  time and is like the Edmonds–Karp algorithm, which runs in  $O(VE^2)$  time, in that it uses shortest augmenting paths. The introduction of the concepts of the level graph and blocking flow enable Dinic's algorithm to achieve its performance.

**Algorithm Implementation is as follows:**

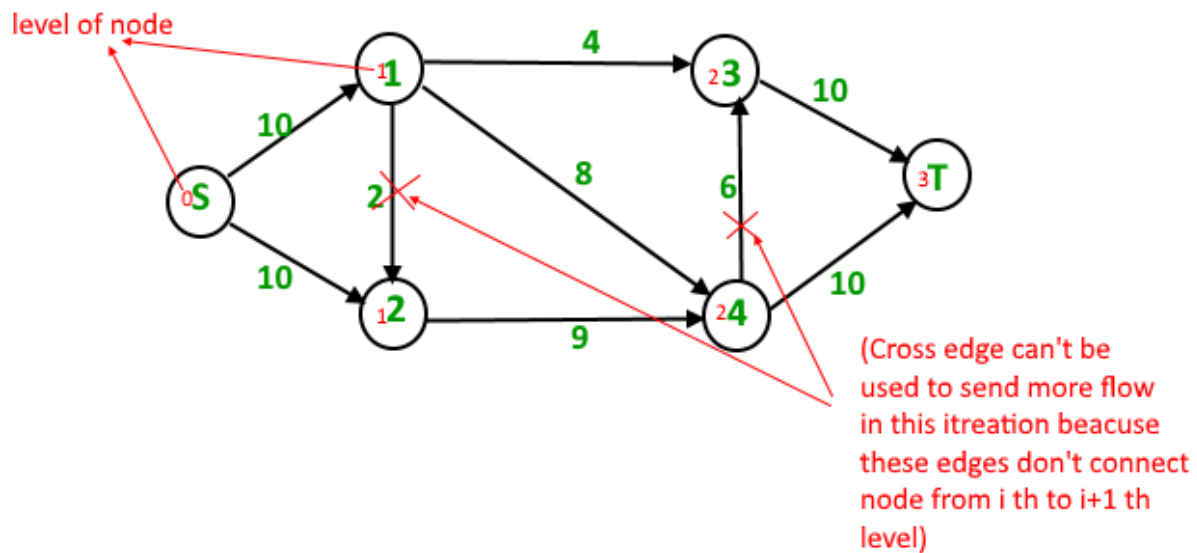
1. set  $f(e) = 0$  for each  $e$  in  $E$
2. Construct  $G_L$  from  $G_f$  of  $G$ . if  $\text{dist}(t) == \text{inf}$ , then stop and output  $f$
3. Find a blocking flow  $fp$  in  $G_L$ .
4. Augment flow  $f$  by  $fp$  and go back to step 2.

Visual Explanation:

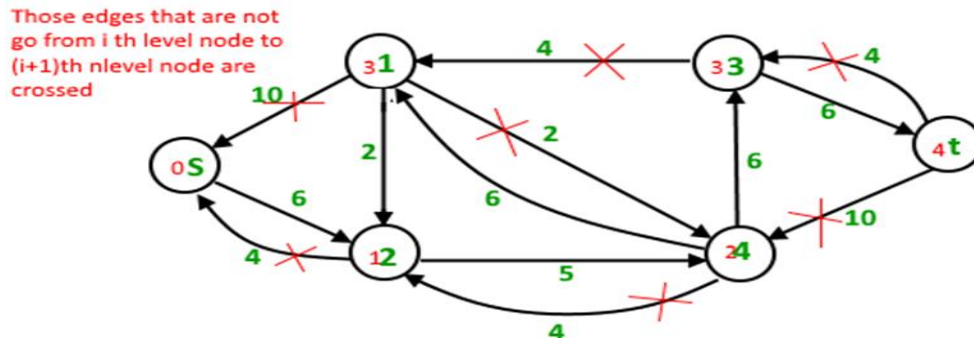
1. First iteration:



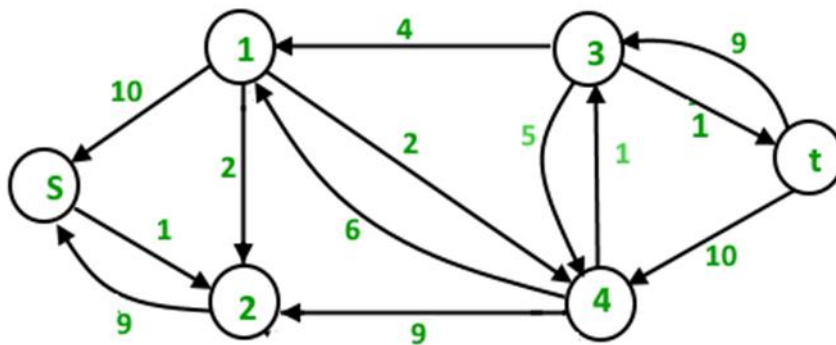
2. Second iteration: Initialization of flow from source 's'



3. Third iteration: Finding augmenting paths to the sink 't'



4. Final iteration: Balancing excess flow at each of the edges



### 3.5 Why Push-Relabel?

#### 3.5.1 Time Complexity:

Time complexity is a measure of computers that defines the amount of time it takes to create an algorithm. Time complexity is usually measured by calculating the number of basic tasks performed by the algorithm, if each basic task takes a set amount of time to perform. Therefore, the amount of time taken, as well as the amount of basic operation performed by the algorithm is quite different for a fixed object. It is important to find the most effective problem-solving algorithm. You may have many problem-solving algorithms, but the challenge here is to choose the one that works best.

Time complexity, by definition, is the amount of time taken by an algorithm to run, as a function of the length of the input. Here, the length of input indicates the number of operations to be performed by the algorithm. This gives a clear indication of what exactly time complexity tells us. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm. Yes, as the definition says, the amount of time taken is a function of the length of input only.

Finally, time complexity measures the time it takes to make each code statement in the algorithm. If a statement is set to perform multiple times when the number of times that statement is made equal to  $N$  is multiplied by the time required to perform that function each time.

### 3.5.2 Time complexity of Push-Relabel:

To bound the time complexity of the algorithm, we must analyse the number of push and relabel operations which occur within the main loop. The numbers of relabel, saturating push and no saturating push operations are analysed separately.

In the algorithm, the relabel operation can be performed at most  $(2|V| - 1)(|V| - 2) < 2|V|^2$  times. This is because the labelling  $\ell(u)$  value for any node  $u$  can never decrease, and the maximum label value is at most  $2|V| - 1$  for all nodes. This means the relabel operation could potentially be performed  $2|V| - 1$  times for all nodes  $V \setminus \{s, t\}$  (i.e.,  $|V| - 2$ ). This results in a bound of  $O(V^2)$  for the relabel operation.

Each saturating push on an admissible arc  $(u, v)$  removes the arc from  $G_f$ . For the arc to be reinserted into  $G_f$  for another saturating push, ' $v$ ' must first be relabeled, followed by a push on the arc  $(v, u)$ , then ' $u$ ' must be relabeled. In the process,  $\ell(u)$  increases by at least two. Therefore, there are  $O(V)$  saturating pushes on  $(u, v)$ , and the total number of saturating pushes is at most  $2|V| |E|$ . This results in a time bound of  $O(VE)$  for the saturating push operations.

Bounding the number of non-saturating pushes can be achieved via a potential argument. We use the potential function  $\Phi = \sum[u \in V \wedge x_f(u) > 0] \ell(u)$  (i.e.  $\Phi$  is the sum of the labels of all active nodes). It is obvious that  $\Phi$  is 0 initially and stays non-negative throughout the execution of the algorithm. Both relabels and saturating pushes can

increase  $\Phi$ . However, the value of  $\Phi$  must be equal to 0 at termination since there cannot be any remaining active nodes at the end of the algorithm's execution. This means that over the execution of the algorithm, the non-saturating pushes must make up the difference of the relabel and saturating push operations for  $\Phi$  to terminate with a value of 0. The relabel operation can increase  $\Phi$  by at most  $(2|V| - 1)(|V| - 2)$ . A saturating push on  $(u, v)$  activates  $v$  if it was inactive before the push, increasing  $\Phi$  by at most  $2|V| - 1$ . Hence, the total contribution of all saturating pushes operations to  $\Phi$  is at most  $(2|V| - 1)(2|V| \parallel E|)$ . A non-saturating push on  $(u, v)$  always deactivates  $u$ , but it can also activate  $v$  as in a saturating push. As a result, it decreases  $\Phi$  by at least  $\ell(u) - \ell(v) = 1$ . Since relabels and saturating pushes increase  $\Phi$ , the total number of non-saturating pushes must make up the difference of  $(2|V| - 1)(|V| - 2) + (2|V| - 1)(2|V| \parallel E|) \leq 4|V||E|$ . This results in a time bound of  $O(V^2E)$  for the non-saturating push operations. In sum, the algorithm executes  $O(V^2)$  relabels,  $O(VE)$  saturating pushes and  $O(V^2E)$  non-saturating pushes. Data structures can be designed to pick and execute an applicable operation in  $O(1)$  time. Therefore, the time complexity of the algorithm is  $O(V^2E)$ .

### 3.5.3 Push Relabel over other algorithms:

Push-Relabel approach is the more efficient than Ford-Fulkerson algorithm. In this post, Goldberg's "generic" maximum-flow algorithm is discussed that runs in  $O(V^2E)$  time. This time complexity is better than  $O(E^2V)$  which is time complexity of Edmond-Karp algorithm (a BFS based implementation of Ford-Fulkerson).

### 3.5.4 Differences with Ford Fulkerson and Dinic's algorithm:

- The Push-relabel algorithm works in a more localised way. Instead of exploring the rest of the network to find a way to add, Push-relabel algorithms work on one vertex at a time.
- In Ford-Fulkerson, the net difference between the total outflow and the total inflow of all vertexes (except for the source and sink) is kept to 0. Push-Relabel algorithm allows the input to pass the exit before reaching the final flow. In the last flow, the net difference is 0 for all except the source and sink.
- Time complexity wise more efficient. In addition, the timing of the Push-Relabel algorithm is closely related to the data to optimize traffic flow. Since road networks can

be compared to tightly connected graphs, the number of vertices is much lower compared to the total number of edges present in the network. Hence, this algorithm with a time complexity of  $O(V^2E)$  results in a faster execution time of the program compared to the execution time of  $O(E^2V)$  for the Ford-Fulkerson and Dinic's algorithm.

### 3.6 Parallel Programming:

Parallel computing uses multiple computer cores to attack multiple applications simultaneously. Unlike serial computing, the same configuration can tear down a function and make it more functional. Similar computer systems are well suited for modelling and mimicking real-world events.

With old-school computing (serial programming), the processor takes one step at a time, such as walking down the street individually, one after another. This is a system that does not work well compared to doing things the same way. Conversely, the parallel process is like making groups of 3 to 5, and then you all go together, covering many steps on the street at the same time.

### 3.7 CUDA-Parallel Programming:

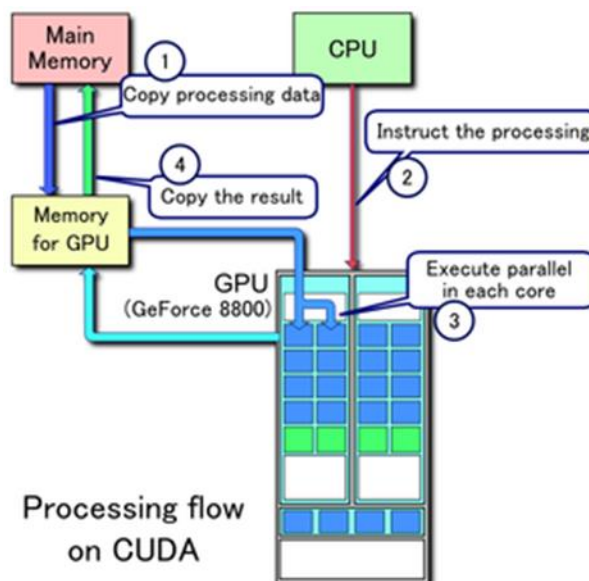


Fig 4. Parallel processing using CUDA

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (general-purpose computing on graphics processing units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

### 3.8 Serial vs parallel processing:

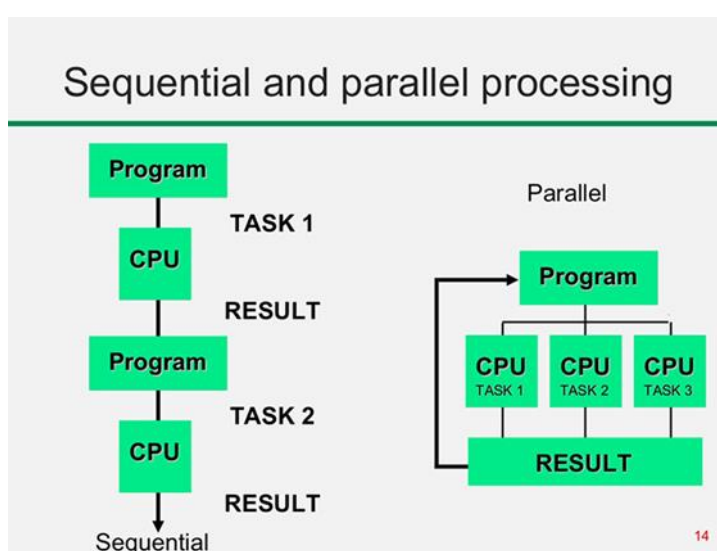


Fig 5. Parallel programming

1. In serial processing, same tasks are completed at the same time but in parallel processing completion time may vary.
2. In sequential processing, the load is high on single core processor and processor heats up quickly.
3. In serial processing data transfers in bit-by-bit form while in parallel processing data transfers in byte form i.e., in 8 bits form
4. Parallel processor is costly as compared to serial processor.
5. Serial processing takes more time than parallel processor.

### 3.9 Parallel Implementation of Push-Relabel:

Algorithm:

#### *1. Push\_relabel\_kernel*

1. 'U' is the thread ID that corresponds to the each of the different vertices.
2. While the kernel cycle is greater than 0
3. For the thread with ID 'u', if the excess flow and height of that vertex is greater than 0 and V respectively, the values of e\_dash, h\_dash, v\_dash are set.
4. For every edge (u, V), if residual flow of that edge is greater than 0, the next lowest neighbour is searched, and its height is updated.
5. If height of vertex 'u' is greater than he next lowest neighbour, the push operation can be performed (the flow can be pushed from 'u' to that neighbour)
6. The addition of flow to network and subtraction in residual network is done using atomic functions to avoid problems involving simultaneous extraction of data.
7. From 'e', if the 'IF' condition is false the height of vertex 'u' is updated based on the height of its neighbour.
8. Kernel cycle is decremented.



```

3 __global__ void push_relabel_kernel(int V, int source, int sink, int *gpu_height, int *gpu_excess_flow, int *gpu_adjmtx, int *gpu_rflowmtx)
4 {
5     //u'th node is operated on by the u'th thread
6     unsigned int u = (blockIdx.x*blockDim.x) + threadIdx.x;
7
8     //printf("u : %d\nV : %d\n",u,V);
9
10    if(u < V)
11    {
12        //printf("Thread id : %d\n",u);
13        // cycle value is set to KERNEL_CYCLES as required
14        int cycle = KERNEL_CYCLES;
15
16        /* Variables declared to be used inside the kernel :
17         * e_dash - initial excess flow of node u
18         * h_dash - height of lowest neighbor of node u
19         * h_double_dash - used to iterate among height values to find h_dash
20         * v - used to iterate among nodes to find v_dash
21         * v_dash - lowest neighbor of node u
22         * d - flow to be pushed from node u
23         */
24
25        int e_dash,h_dash,h_double_dash,v,v_dash,d;
26
27        while(cycle > 0)
28        {
29            if( (gpu_excess_flow[u] > 0) && (gpu_height[u] < V) )
30            {
31                e_dash = gpu_excess_flow[u];
32                h_dash = INF;
33                v_dash = NULL;
34
35                for(v = 0; v < V; v++)
36                {
37                    //For all (u,v) belonging to E_f (residual graph edgelist)
38                    if(gpu_rflowmtx[IDX(u,v)] > 0)
39                    {
40                        h_double_dash = gpu_height[v];
41                        //Finding lowest neighbor of node 'u'
42                        if(h_double_dash < h_dash)
43                        {
44                            v_dash = v;
45                            h_dash = h_double_dash;
46

```

```

50                    if(gpu_height[u] > h_dash)
51                    {
52                        /* height of u > height of lowest neighbor
53                         * Push operation can be performed from node u to lowest neighbor
54                         * All addition, subtraction and minimum operations are done using Atomics
55                         * This is to avoid anomalies in conflicts between multiple threads
56                         */
57
58                        //d' captures flow to be pushed
59                        d = e_dash;
60                        //atomicMin(&d,gpu_rflowmtx[IDX(u,v_dash)]);
61                        if(e_dash > gpu_rflowmtx[IDX(u,v_dash)])
62                        {
63                            d = gpu_rflowmtx[IDX(u,v_dash)];
64                        }
65                        //Residual flow towards lowest neighbor from node u is increased
66                        atomicAdd(&gpu_rflowmtx[IDX(v_dash,u)],d);
67
68                        //Residual flow towards node u from lowest neighbor is decreased
69                        atomicSub(&gpu_rflowmtx[IDX(u,v_dash)],d);
70
71                        //Excess flow of lowest neighbor and node u are updated
72                        atomicAdd(&gpu_excess_flow[v_dash],d);
73                        atomicSub(&gpu_excess_flow[u],d);
74                    }
75                }
76            }
77            else
78            {
79                /* height of u <= height of lowest neighbor,
80                 * No neighbor with lesser height exists
81                 * Push cannot be performed to any neighbor
82                 * Hence, relabel operation is performed
83                 */
84
85                gpu_height[u] = h_dash + 1;
86            }
87        }
88
89        //Cycle value is decreased
90        cycle = cycle - 1;
91    }
92 }
93

```

## 2. Print

1. This is a CPU function used to print the flow at every vertex at every iteration.
2. It accepts V, cpu\_height, cpu\_excess\_flow, cpu\_rflowmtx, cpu\_adjmtx
3. It outputs the network values of each vertex for that iteration.

```
163
164 void print(int V,int *cpu_height, int *cpu_excess_flow, int *cpu_rflowmtx, int *cpu_adjmtx)
165 {
166     printf("\nHeight :");
167     for(int i = 0; i < V; i++)
168     {
169         printf("%d ",cpu_height[i]);
170     }
171
172     printf("\nExcess flow :");
173     for(int i = 0; i < V; i++)
174     {
175         printf("%d ",cpu_excess_flow[i]);
176     }
177
178     printf("\nRflow mtx :\n");
179     for(int i = 0; i < V; i++)
180     {
181         for(int j = 0; j < V; j++)
182         {
183             printf("%d ", cpu_rflowmtx[IDX(i,j)]);
184         }
185         printf("\n");
186     }
187
188     printf("\nAdj mtx :\n");
189     for(int i = 0; i < V; i++)
190     {
191         for(int j = 0; j < V; j++)
192         {
193             printf("%d ", cpu_adjmtx[IDX(i,j)]);
194         }
195         printf("\n");
196     }
197 }
```

## 3. Readgraph

1. The adj\_mtx and residual\_flow\_mtx are initialised to 0 for each vertex.
2. The inputs are retrieved from the input file.
3. e1, e2, cp is retrieved and converted to INT from STRING.
4. Cpu\_adjmtx and cpu\_rflowmtx are updated with the above values.

```

199
200 void readgraph(int V, int E, int source, int sink, int *cpu_height, int *cpu_excess_flow, int *cpu_adjmtx, int *cpu_rflowmtx)
201 {
202     //Initialising all adjacent matrix values to 0 before input
203     for(int i = 0; i < (number_of_nodes)*(number_of_nodes); i++)
204     {
205         cpu_adjmtx[i] = 0;
206         cpu_rflowmtx[i] = 0;
207     }
208     //Declaring file pointer to read edgelist
209     FILE *fp = fopen("edgelist.txt","r");
210
211     //Declaring variables to read and store data from file
212     char buf1[10],buf2[10],buf3[10];
213     int e1,e2,cp;
214
215     // Getting edgelist input from input file
216     for(int i = 0; i < E; i++)
217     {
218         //Reading from file
219         fscanf(fp,"%s",buf1);
220         fscanf(fp,"%s",buf2);
221         fscanf(fp,"%s",buf3);
222
223         //Converting and storing as integers
224         e1 = atoi(buf1);
225         e2 = atoi(buf2);
226         cp = atoi(buf3);
227
228         /* Adding edges to the graph
229          * rflow - residual flow is also updated simultaneously
230          * So the graph when prepared already has updated residual flow values
231          * This is why residual flow is not initialised during preflow
232          */
233
234         cpu_adjmtx[IDX(e1,e2)] = cp;
235         cpu_rflowmtx[IDX(e1,e2)] = cp;
236
237     }
238
239 }

```

#### 4. Preflow

1. This function initializes all the values of the network to 0 to start off.
2. Cpu\_height of source is set to 'V'.
3. It then initiates all the flow from the source to its first neighbour vertices as follows.
4. If the flow capacity of edge (source, i), is greater than 0, its residual flow is set to 0 and a backflow equal that its capacity is added to the residual flow of edge (i, source), this operation prevents any pushes from neighbour vertices back to the source vertex, hence preventing a race around condition.
5. Excess\_flow is updated by adding the above residual flow to its previous state value.

```

240
241 void preflow(int V, int source, int sink, int *cpu_height, int *cpu_excess_flow, int *cpu_adjmtx, int *cpu_rflowmtx, int *Excess_total)
242 {
243     //Initialising height values and excess flow
244     for(int i = 0; i < V; i++)
245     {
246         cpu_height[i] = 0;
247         cpu_excess_flow[i] = 0;
248     }
249
250     cpu_height[source] = V;
251     *Excess_total = 0;
252
253     //Initiating flow in all edges going out from the source node
254     for(int i = 0; i < V; i++)
255     {
256         //For all (source,i) belonging to E :
257         if(cpu_adjmtx[IDX(source,i)] > 0)
258         {
259             // Pushing out of source node
260             cpu_rflowmtx[IDX(source,i)] = 0;
261
262             /* Updating the residual flow value on the back edge
263              * The capacity of the back edge is also added to avoid any push operation back to the source
264              * This avoids creating a race condition, where flow keeps travelling to and from the source
265              */
266             cpu_rflowmtx[IDX(i,source)] = cpu_adjmtx[IDX(source,i)] + cpu_adjmtx[IDX(i,source)];
267
268             //Updating the excess flow value of the node flow is pushed to, from the source
269             cpu_excess_flow[i] = cpu_adjmtx[IDX(source,i)];
270
271             //Update Excess_total value with the new excess flow value of the node flow is pushed to
272             *Excess_total += cpu_excess_flow[i];
273         }
274     }
275
276 }

```

## 5. Push\_relabel

1. The sum of excess\_flow of sink and source is compared to excess\_flow, if sum is lesser than excess\_flow there is at least one vertex with flow greater than its capacity.
2. Mark and scan arrays are declared of size 'V'.
3. Mark value is set to 0 initially.
4. The condition mentioned in statement 'a' is used in a 'while' loop till the excess\_flow is equal to or less than the sum of flow regarding the source and sink.
5. The push\_relabel\_kernel is invoked by passing the necessary parameters and setting the thread and block amount.
6. Height, excess\_flow and residual\_flow arrays are copied back into the CPU.

```

277 void push_relabel(int V, int source, int sink, int *cpu_height, int *cpu_excess_flow, int *cpu_adjmx, int *cpu_flowmx, int *Excess_total, int *gpu_height, int *gpu_excess_flow, int *gpu_adjmx, int *gpu_flowmx)
278 {
279     /* Instead of checking for overflowing vertices(as in the sequential push_relabel),
280      * sum of excess flow values of sink and source are compared against Excess_total
281      * If the sum is lesser than Excess_total,
282      * it means that there is atleast one more vertex with excess flow > 0, apart from source and sink
283      */
284
285     /* Declaring the mark and scan boolean arrays used in the global_relabel routine outside the while loop
286      * This is not to lose the mark values if it goes out of scope and gets redeclared in the next iteration
287      */
288
289     bool *mark,*scanned;
290     mark = (bool*)malloc(V*sizeof(bool));
291     scanned = (bool*)malloc(V*sizeof(bool));
292
293     //Initialising mark values to false for all nodes
294     for(int i = 0; i < V; i++)
295     {
296         mark[i] = false;
297     }
298
299     while((cpu_excess_flow[source] + cpu_excess_flow[sink]) < *Excess_total)
300     {
301         //Copying height values to CUDA device global memory
302         cudaMemcpy(gpu_height,gpu_height,V*sizeof(int),cudaMemcpyHostToDevice);
303
304         printf("Invoking kernel\n");
305
306         //Invoking the push_relabel_kernel
307         push_relabel_kernel<<number_of_blocks_nodes,threads_per_block>>> (V,source,sink,gpu_height,gpu_excess_flow,gpu_adjmx,gpu_flowmx);
308
309         cudaDeviceSynchronize();
310
311
312         //Copying height, excess flow and residual flow values from device to host memory
313         cudaMemcpy(cpu_height,gpu_height,V*sizeof(int),cudaMemcpyDeviceToHost);
314         cudaMemcpy(cpu_excess_flow,gpu_excess_flow,V*sizeof(int),cudaMemcpyDeviceToHost);
315         cudaMemcpy(cpu_flowmx,gpu_flowmx,V*sizeof(int),cudaMemcpyDeviceToHost);
316
317         printf("After invoking\n");
318         printf("Excess total : %d\n",*Excess_total);
319
320         //Performing the global_relabel routine on host
321         global_relabel(V,source,sink,cpu_height,cpu_excess_flow,cpu_adjmx,cpu_flowmx,*Excess_total,mark,scanned);
322
323         printf("\nAfter global_relabel\n");
324         printf("Excess total : %d\n",*Excess_total);
325     }
326 }

```

### 3.10 Performance Metrics:

Execution time is the time the CPU spends working on the task, CPU time spent on work does not include I / O waiting time or other programming. You know that the processor is not only running your system, but it may also be running other programs and if there is an I / O transfer, it may block this application and move to another application. We do not consider the time taken to perform I / O tasks and are always concerned only when performing CPU. That is the time the CPU spends on a particular program.

To determine the CPU output time of the system, you can find the total number of clock cycles that the program takes and multiply it with the duration of each cycle. Each program is done with several commands and each process takes a few clock cycles to perform. If you find the total number of clock cycles per program and if you know the time of each clock cycle, then CPU performance times can only be calculated as a product of the total number of CPU clock cycles per system of these clocks. Because of the clock cycle time and the relative clock frequency, this can also be written as clock cycles for the CPU system divided by the clock scale.

### **3.11 Validation of the Push-Relabel Algorithm:**

The above program and algorithm were validated based on trial and tested serial implementations of the Push-Relabel algorithm with modifications such as multi-source and sink capabilities, two-way path anomalies, augmenting path optimization, etc. The serial implementation was implemented to mimic the algorithm that has already been tested on real-life applications. The comparison of the output entities- Adjoint matrix, Residual-flow matrix and the max-flow were compared to its corresponding outputs from the serial implementation. This ensured that this parallel algorithm follows all the guidelines as a verified serial algorithm and its output is correct.

## 4. DATASET FOR TESTING

### 4.1 Data:

The data used primarily consists of the road network data which include the various edges that depict a road along with the start and end vertices which represent the road-joints in real life. The capacity of the road is calculated using the meta-data present that gives the number of lanes present, the width and length of each road. Using the below table, the capacity of each edge is calculated.

### Example:

Start	End	Capacity
1	2	1
1	3	7
2	3	1
2	5	2
3	5	4
4	5	1
4	6	6
5	6	2

Table 1. Dataset of below road network

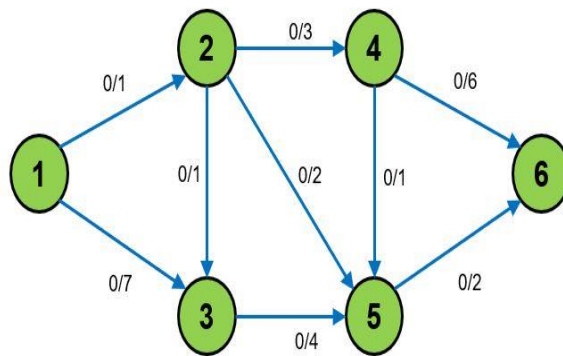


Fig 6. Road network

The data is then fed into the program as a stream of string values from a text file. The data of each edge is represented on an individual line of the text file. Each line consists of a start node, end node and capacity of that corresponding edge.

The data is processed by the **readgraph** function. This function converts the input string into integers and then populates the adjoint matrix with the capacity value of each of the edges present in the road network. If an edge between two nodes does not exist, the capacity is set to zero.

Road Type	Carriageway Width (m)	Roadside Friction	Basic Hourly Capacity (in PCU in Both Directions)
Highway	≤ 4.0	None or Light	600
Highway	4.1 - 5.0	None or Light	1,200
Highway	5.1 - 5.5	None or Light	1,800
Highway	5.6 - 6.1	None or Light	1,900
Highway	6.2 - 6.5	None or Light	2,000
Highway	6.6 - 7.3	None or Light	2,400
Highway	2 x 7.0	None or Light	7,200 (Expressway)
Urban Street	≤ 6.0	Heavy	1,200
Urban Street	6.1 - 6.5	Heavy	1,600
Urban Street	6.6 - 7.3	Heavy	1,800
Urban Street	2 x 7.0	Heavy	6,700

Table 2: Road capacity calculation table

The road capacities are calculated using the above table. The metadata helps identify the type of road and which is finally classified under one of the above categories. This in turn provides the corresponding hourly capacities of each road in the network.

[Source: <https://www.scribd.com/document/250008548/Highway-Capacity-Estimation>]



## 5. RESULT

Researchers initially debated on the CPU and GPU implementations; however, it shifted to a paradigm of combining CPU and GPU to achieve further computation gains known as heterogeneous computing. The heterogeneous computing environment displays the interconnected machines such as CPU and GPU to perform an application whose subtasks have different execution requirements. However, software development infrastructure for the parallel programming and libraries supporting the multiple vendors' hardware platforms are needed for parallel architectures and heterogeneous computing.

Compute Unified Device Architecture (CUDA) is one recently introduced framework that makes use of parallel compute engines in NVIDIA GPUs to solve complex problems efficiently. CUDA improves the performance significantly. The problem is related to the data dependency issue. It may lead to the findings of different parallelization and vectorization techniques.

Numerous researches have been performed to implement parallelization to improve the performance of sequence alignment algorithms. The Push-Relabel algorithm was implemented on a CUDA compatible GPU employing a divide-and-conquer technique. It divides the entire matrix into sub-matrices. For parallel execution, a certain number of threads and amounts of memory are allocated in each sub-matrix. Compared to the CPU execution, the calculation time was significantly times faster.

All the elements in the same column of the dynamic programming matrix were calculated in parallel independently. It enhanced the push-relabel algorithm's performance. Similarly, states that the implementation of dynamic programming-based sequence alignment in CUDA showed 1.7 times faster in computation time compared to the single-threaded CPU based implementation. In addition, to GPU implementation of the Push-Relabel algorithm, demonstrated two-dimensional array as a one-dimensional array for memory optimization, which had a positive impact on the overall performance.

## 6. CONCLUSION

Overall, the heterogeneous implementation is three times faster than the serial implementation. Both applications seemed to show equally at first based on the output; however, the difference between the two graphs increases as the input sequence length increases. The computation time difference at 4,000 sequence length was only 70ms whereas when the input is 40,000 long the heterogeneous implementation outperforms the serial implementation by 282ms. This implies that GPU is maximized on intense calculation tasks.

# REFERENCES

**Capacity estimation of edges** - <https://www.scribd.com/document/250008548/Highway-Capacity-Estimation>

**Experimental comparison of max- Jacob Mark Friis and Stefen B Olesen** flow algorithms - [jakob\\_mark\\_friis\\_steffen\\_beier\\_olesen.pdf](#)

**Great Britain traffic datasets** - GB Road Traffic Counts - [data.gov.uk](#)

**Max-flow in road networks with speed dependent capacity - Elvin J Moore, Wisut Kichainukon and Utomporn Phalavonk**

[https://www.researchgate.net/publication/285199854\\_Maximum\\_flow\\_in\\_road\\_networks\\_with\\_speed-dependent\\_capacities\\_-\\_Application\\_to\\_Bangkok\\_traffic](https://www.researchgate.net/publication/285199854_Maximum_flow_in_road_networks_with_speed-dependent_capacities_-_Application_to_Bangkok_traffic)

**Network flow study – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein** [https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6\\_046JS12\\_lec13.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec13.pdf)

**Improved algorithm about max flow using auxiliary graph theory – Pingsheng Li, Bin Li, Xiaoli Xi and Meng Wang**

<https://www.sciencedirect.com/science/article/abs/pii/S1570667208600483>.

**Dinic's algorithm** - <https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/>

**Road capacity studies – Global Security**

<https://www.globalsecurity.org/military/library/policy/army/fm/19-25/CH25.htm>

**UTD-Understanding capacity of urban networks** - <https://www.research-collection.ethz.ch/handle/20.500.11850/437802>.

**World road network datasets** - <http://networkrepository.com/road.php>

**Ford Fulkerson Algorithm** – <https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>

**Push Relabel Algorithm** - <https://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/>