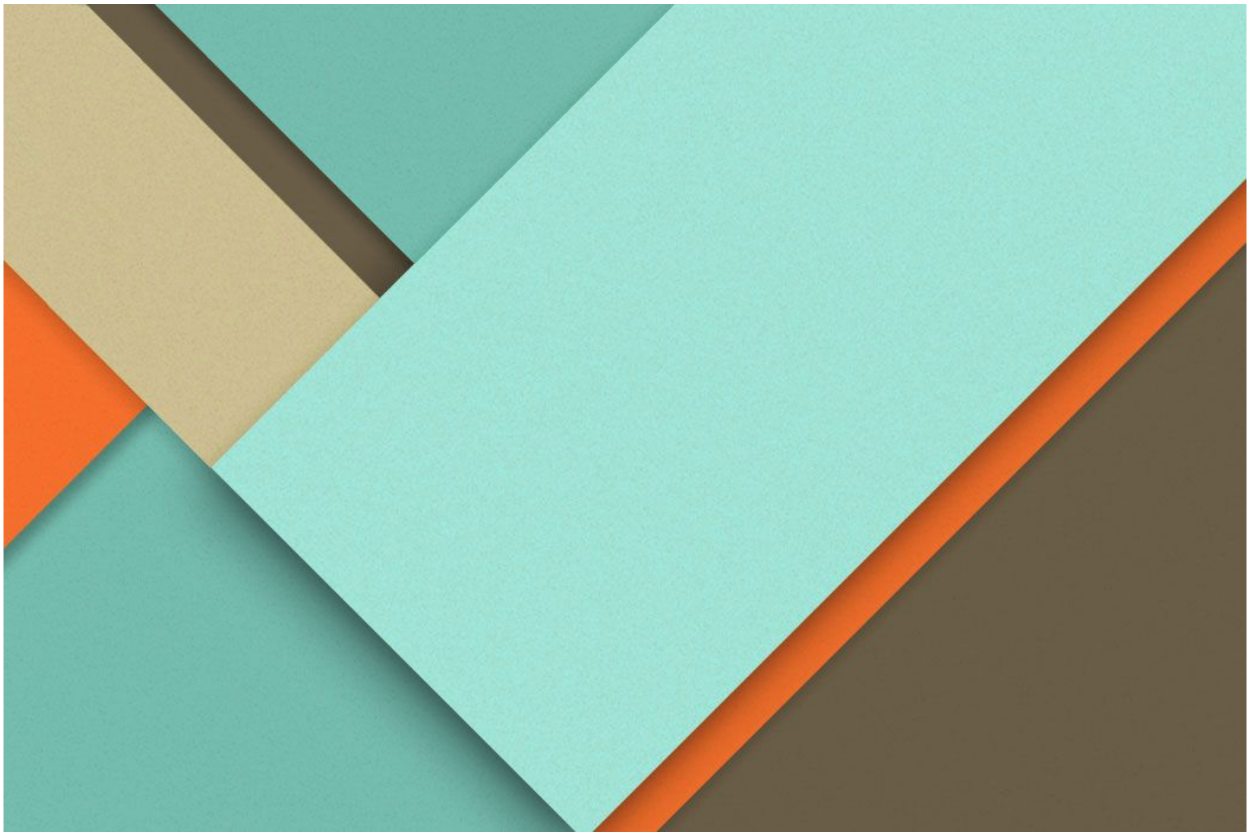


Johnson Algorithm

Parallel Implementation in Rust

IPCO - Semester 6, 2016

Killamsetty Bhagyaraj - IIT2013042
Vishnu Ks - IIT2013075

Overview

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse, edge weighted, directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist.

Goals

1. Formulate a parallel approach for Johnson algorithm and implement in Rust
2. Analyze and improve the performance.

Algorithm

```
procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
begin
    Q := V;
    for all v ∈ Q do
        l[v] := ∞;
    l[s] := 0;
    while Q ≠ ∅ do
    begin
        u := extract_min(Q);
        for each v ∈ Adj[u] do
            if v ∈ Q and l[u] + w(u, v) < l[v] then
                l[v] := l[u] + w(u, v);
    endwhile
end JOHNSON_SINGLE_SOURCE_SP
```

Implementation in Rust

```
while let Some(que_node { cost , vertex })  = heap.pop() {
    if cost > distance[vertex as usize] {
        continue;
    }

    for i in 0..nodes {
        let mut weigh =  adj_matrix[vertex as usize][i as usize];
        if(i != vertex && weigh != 0 ){
            let next = que_node { cost: cost + weigh, vertex: i };
            if next.cost < distance[next.vertex as usize] {
                heap.push(next);
                distance[next.vertex as usize] = next.cost;
            }
        }
    }
}
```

The algorithm

Parallelizing Step

```
for mut j in 0..limit  {
    for i in j..NTHREADS + j {
        let adj_list =adj_list.clone();
            children.push(thread::spawn(move || {
                Johnson(i as u32,nodes as u32,edges as u32 ,&adj_list);
        }));
    }
```

How to run the code

git clone https://github.com/praneelrathore/etig

cd etig/src

uncomment parallel::johnson(&mut gra, na);

cargo build

cargo run input_file

Make sure the main.rs contains extern crate algorithms; use algorithms::parallel;

Sample Input and Output

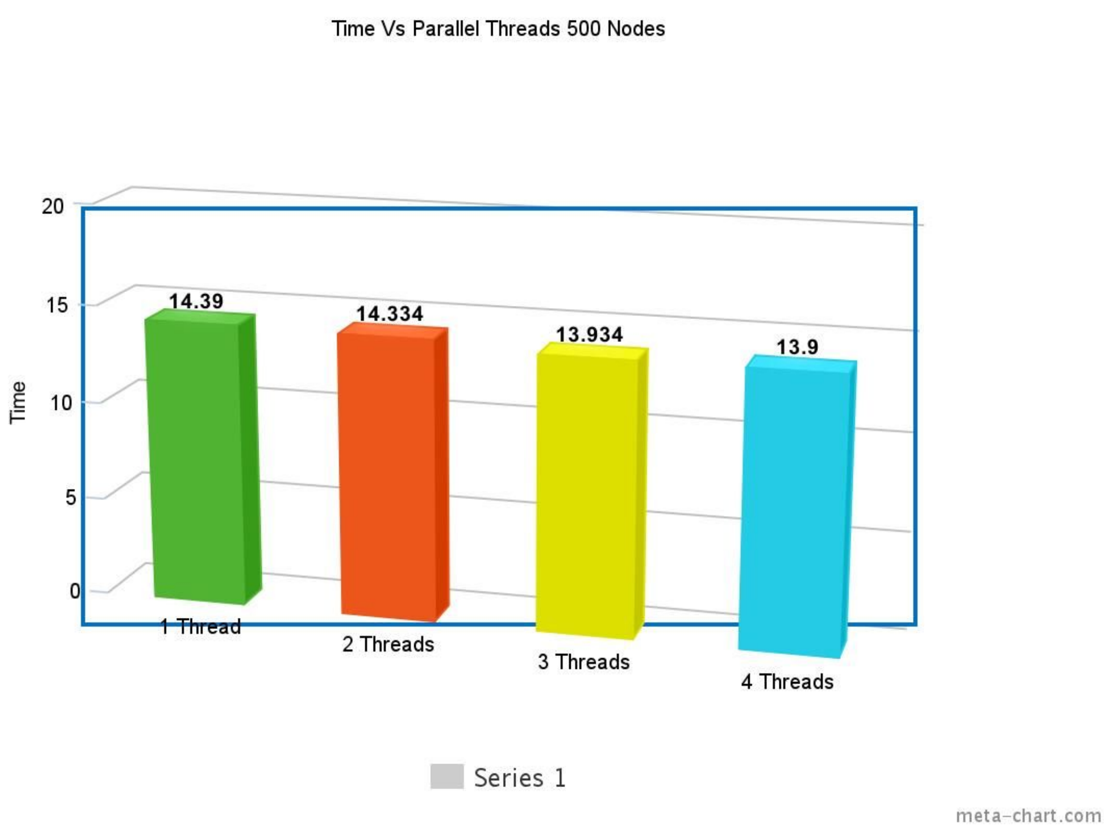
```
~/rust-algo master*
❯ more input.in
5
3 1 0 0 3
1 4 4 2 4
2 1 0 4 3
2 2 4 0 2
0 0 2 0 0

~/rust-algo master*
❯ cargo run input.in
     Running `target/debug/etig input.in`
min distance of 0 = 0
min distance of 1 = 1
min distance of 2 = 5
min distance of 3 = 3
min distance of 4 = 3
```

Analysis of Performance

Nodes = 500

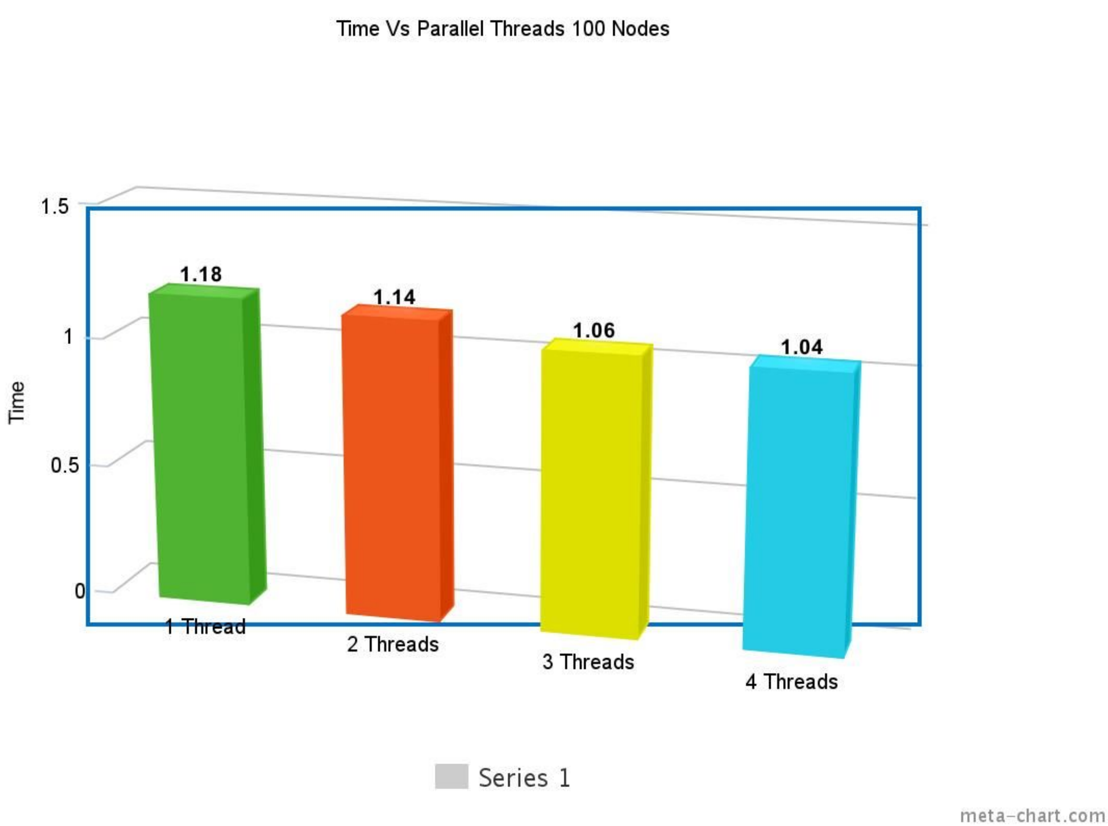
Threads	Time Taken
1	14.39
2	14.33
3	13.93
4	13.90



Analysis of Performance

Nodes = 100

Threads	Time Taken
1	1.18
2	1.14
3	1.06
4	1.04

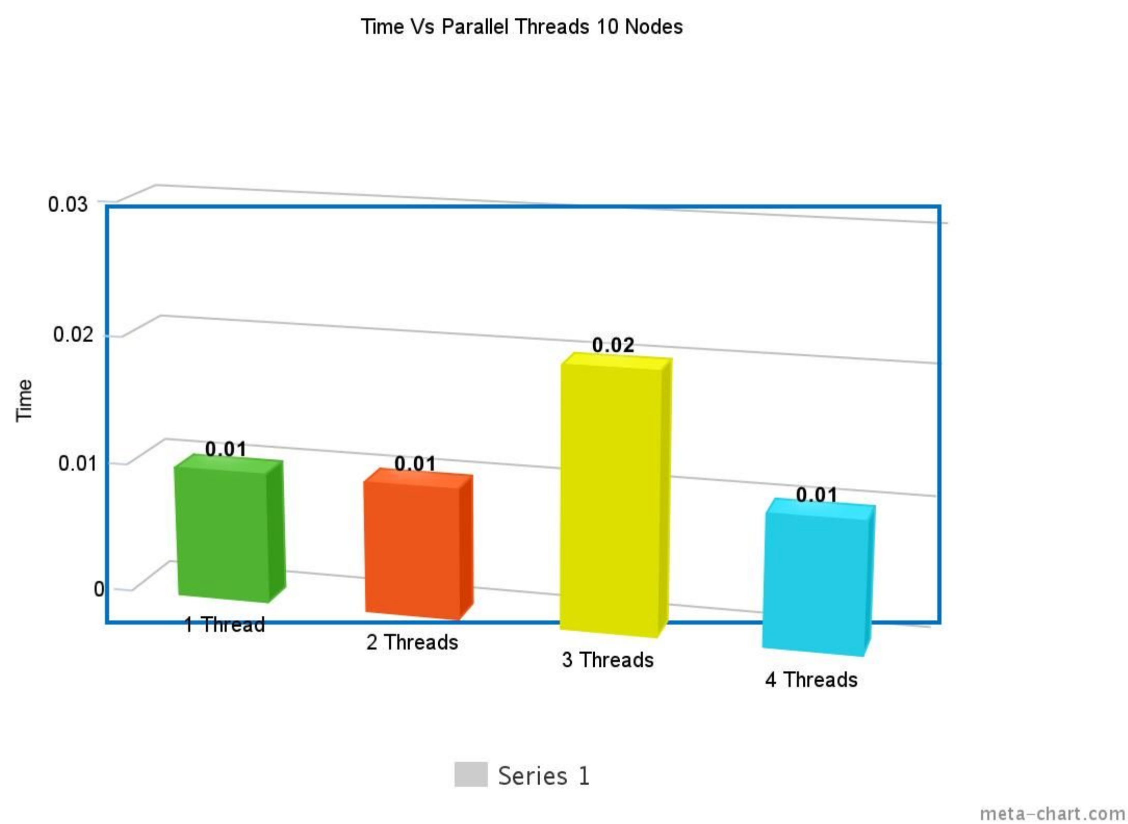


Analysis of Performance

Nodes = 10

Thread creation overhead comes to play as time for 3 threads became greater than for 2

Threads	Time Taken
1	0.01
2	0.01
3	0.02
4	0.01



Time Complexity

Without parallelization

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is $O(V2\log V + VE)$ and $O(V \log V + E)$ for each of V instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time O(V3).

After Parallelization

The time complexity after parallelization is

$$O(K2 * \operatorname{Log} V + KE) + O(K * \operatorname{Log} V + V)$$

K = V / NO_OF_THREADS

V = No of Vertices