PUSH RELABEL ALGORITHM PARALLELISATION USING OPENMP

Serial code: Vertices and edges are stored in vectors with each edge maintaining flow and capacity, and each vertex tracking height and excess flow. The source vertex is given an initial height equal to the number of vertices, and preflow is pushed along all its outgoing edges. Excess flow is pushed along admissible edges (where the source's height is higher than the destination's) and reverse edges are updated. When no push is possible, the vertex's height is increased based on the minimum height among its neighbors with residual capacity. Overflowing vertices are repeatedly processed (push/relabel) until no vertex (other than source and sink) has excess flow.

Parallel Code: A deque for active vertices is used with a boolean flag array (inActive) to track their state, along with critical sections to manage concurrent access. The push operation uses OpenMP atomic directives to safely update shared edge flow and vertex excesses. The discharge procedure is parallelized across adjacent edges of a vertex using OpenMP's parallel for loop to try pushing flow concurrently. The initial preflow push from the source is distributed among threads with critical sections ensuring safe insertion into the active queue. The code tests various thread counts using omp_set_num_threads and measures execution time to compare performance across different levels of parallelism.

Why These Changes Were Necessary

- Concurrent Updates: In a parallel environment, multiple threads may update shared data (flows, excess, active queue) concurrently; hence atomic operations and critical sections are used to prevent race conditions.
- Task Decomposition: Work is divided among threads (e.g., processing multiple adjacent edges concurrently) to improve utilization and reduce overall execution time.
- Data Structure Safety: Maintaining thread-safe structures (e.g., using a boolean flag for active vertices) is essential to ensure consistency and correctness during parallel execution.
- Load Balancing: Parallel loops help distribute the workload of push operations and discharges, which are the most computationally intensive parts of the algorithm.
- Scalability: By allowing flexible thread counts and managing synchronization overhead carefully, the parallel version aims to scale with the available cores while maintaining algorithm correctness.

Sample Input: The input graph contains around 1200 vertices and 10,00,000 edges.

1200 0 1199 – Total number of vertices, Source node, Destination node

658 991 2 – starting node, end node, edge weight

382 47 51

366 838 45

58 155 94

PARALLEL CODE:

```
#include <iostream>
#include <vector>
#include <deque>
#include <climits>
#include <algorithm>
#include <fstream>
#include <queue>
#include <omp.h>
using namespace std;
struct Edge {
   int from, to;
   long long capacity, flow;
   Edge(int u, int v, long long cap) : from(u), to(v), capacity(cap), flow(0) {}
};
class MaxFlow {
   int V;
   vector<vector<int>> adj;
   vector<Edge> edges;
   vector<int> height, excess;
   deque<int> active;
   vector<bool> inActive;
   int opCount, globalUpdateThreshold;
public:
   MaxFlow(int vertices)
      : V(vertices), adj(vertices), height(vertices, 0), excess(vertices, 0),
        inActive(vertices, false), opCount(0), globalUpdateThreshold(vertices) {}
   void addEdge(int u, int v, long long capacity) {
      edges.emplace_back(u, v, capacity);
      edges.emplace_back(v, u, 0);
      adj[u].push_back(edges.size() - 2);
```

```
    adj[v].push_back(edges.size() - 1);
}
void push(int e) {
    Edge &edge = edges[e];
    long long pushFlow = min((long long)excess[edge.from], edge.capacity - edge.flow);
    if (pushFlow > 0 && height[edge.from] == height[edge.to] + 1) {
        #pragma omp atomic
        edge.flow += pushFlow;
        #pragma omp atomic
        edges[e ^ 1].flow -= pushFlow;
        #pragma omp atomic
        excess[edge.from] -= pushFlow;
        #pragma omp atomic
        excess[edge.to] += pushFlow;
        if (edge.to != 0 && edge.to != V - 1 && excess[edge.to] > 0 && !inActive[edge.to]) {
            #pragma omp critical
            {
                active.push_back(edge.to);
                inActive[edge.to] = true;
            }
        }
    }
}
void relabel(int u) {
    int minHeight = INT_MAX;
    for (int e : adj[u]) {
        if (edges[e].capacity > edges[e].flow)
            minHeight = min(minHeight, height[edges[e].to]);
    }
    if (minHeight < INT_MAX)
        height[u] = minHeight + 1;
```

```
}

void discharge(int u, int sink) {
    bool workDone = true;  // Track if we did any push operations
    while (excess[u] > 0 && workDone) {
        workDone = false;
        #pragma omp parallel for
        for (int i = 0; i < adj[u].size(); ++i) {
            int e = adj[u][i];
            if (edges[e].capacity > edges[e].flow && height[u] == height[edges[e].to] + 1) {
                push(e);
                workDone = true; // Mark work done
            }
        }
        if (!workDone) {
            #pragma omp single
            relabel(u);
        }
    }
}
long long maxFlow(int source, int sink) {
    if (source < 0 || sink < 0 || source >= V || sink >= V) {
        cerr << "Invalid source or sink." << endl;
        return 0;
    }
    height[source] = V;
    excess[source] = 0;
    #pragma omp parallel for
    for (int i = 0; i < adj[source].size(); ++i) {
        int e = adj[source][i];
        Edge &edge = edges[e];
    if (edge.capacity > 0) {
        long long pushFlow = edge.capacity;
        edge.flow = pushFlow;
        edges[e ^ 1].flow = -pushFlow;
        excess[edge.to] += pushFlow;
        excess[source] -= pushFlow;
        #pragma omp critical
        {
            if (edge.to != source && edge.to != sink && !inActive[edge.to]) {
                active.push_back(edge.to);
                inActive[edge.to] = true;
            }
        }
    }
}
while (!active.empty()) {
    int u;
    #pragma omp critical
    {
        u = active.front();
        active.pop_front();
        inActive[u] = false;
    }
    if (u != source && u != sink)
        discharge(u, sink);
}
long long totalFlow = 0;
for (int e : adj[source]) {
    totalFlow += edges[e].flow;
}
return totalFlow;
```

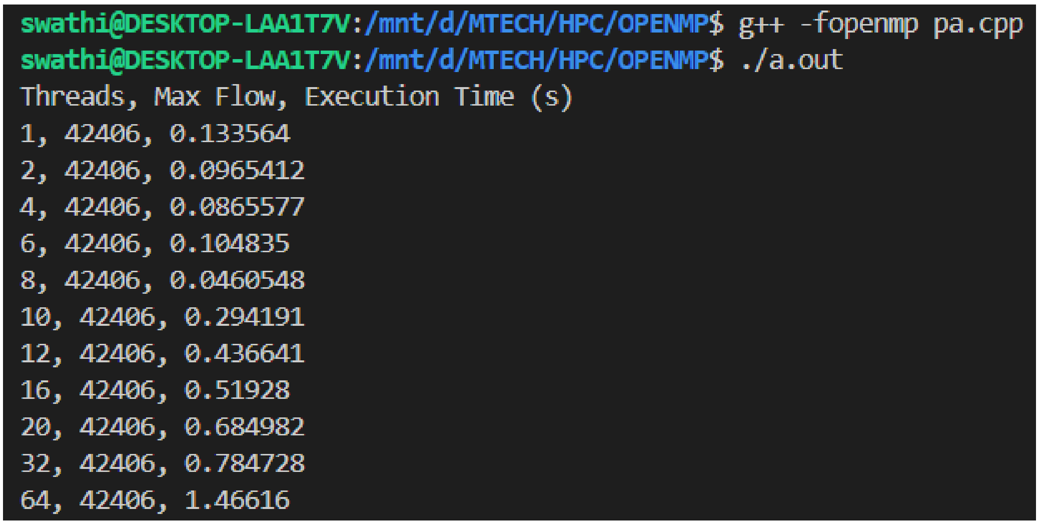
```
    }
};



int main() {
    ifstream infile("input.txt");
    if (!infile) {
        cerr << "Error opening file." << endl;
        return 1;
    }
    int V, source, sink;
    infile >> V >> source >> sink;
    MaxFlow maxFlow(V);
    int u, v;
    long long capacity;
    while (infile >> u >> v >> capacity) {
        maxFlow.addEdge(u, v, capacity);
    }
    infile.close();
    cout << "Threads, Max Flow, Execution Time (s)" << endl;
    vector<int> threadCounts = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
    for (int threads : threadCounts) {
        omp_set_num_threads(threads);
        double start = omp_get_wtime();
        long long result = maxFlow.maxFlow(source, sink);
        double end = omp_get_wtime();
        cout << threads << ", " << result << ", " << (end - start) << endl;
    }
    return 0;
}
```

THREADS VS TIME:

OUTPUT:

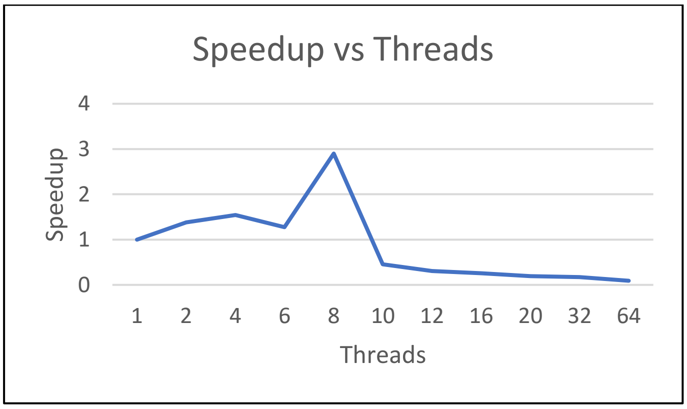


```
swathi@DESKTOP-LAA1T7V:/mnt/d/MTECH/HPC/OPENMP$ g++ -fopenmp pa.cpp
swathi@DESKTOP-LAA1T7V:/mnt/d/MTECH/HPC/OPENMP$ ./a.out
Threads, Max Flow, Execution Time (s)
1, 42406, 0.133564
2, 42406, 0.0965412
4, 42406, 0.0865577
6, 42406, 0.104835
8, 42406, 0.0460548
10, 42406, 0.294191
12, 42406, 0.436641
16, 42406, 0.51928
20, 42406, 0.684982
32, 42406, 0.784728
64, 42406, 1.46616
```

TABLE:

Processors (Threads)	**Time (s)**	**Speedup**
1	0.13356400	1
2	0.09654120	1.383492229
4	0.08655770	1.543063182
6	0.10483500	1.274040158
8	0.04605480	2.900110303
10	0.29419100	0.454004371
12	0.43664100	0.305889736
16	0.51928000	0.257209983
20	0.68498200	0.194989065
32	0.78472800	0.170204198
64	1.46616000	0.091097834

THREADS VS SPEEDUP:



PARALLELIZATION FACTOR CALCULATION:

Though a peak at 8 is observed, as the increase in speedup is uniform from 1 to 6 threads, to calculate the parallelisation fraction, the speedup obtained at 4 threads is considered as it is the maximum in that range.

Speedup = 1.543063182

By Amdalh's law, we can obtain the parallelization factor as

$$f = \frac{1-\frac{T(p)}{T(1)}}{1-\frac{1}{p}} = \frac{1-\frac{1}{S(p)}}{1-\frac{1}{p}} = \frac{1-\frac{1}{1.543063182}}{1-\frac{1}{4}} = 0.4692$$

INFERENCES:

1. Initial Parallel Gains
   - When moving from 1 to 2 or 4 threads, a clear increase in speedup can be observed. This indicates that the overhead of parallel regions and synchronization is still outweighed by the benefit of distributing push operations among multiple threads.
2. Increased Thread Counts and Diminishing Returns
   - Beyond 4 threads, the speedup curve drops below 1, indicating the parallel version is slower than the serial. Following could be the potential reasons:
     - Each push or relabel involves updating shared structures (edges, excess, active deque). In the code, atomic operations and critical sections to ensure correctness were implemented. As more threads are added, these become hotspots for contention: many threads attempt to modify the same data (e.g., the active deque) or update flows on the same edges.
     - The overhead of frequent locking (in #pragma omp critical sections) and atomic operations can balloon with more threads, reducing any parallel advantage.
3. Synchronization Bottlenecks
   - In push function, #pragma omp atomic is used for flow and excess updates. Each atomic operation serializes that memory update, so the more threads, the more they contend for these updates.
   - In the maxFlow function, #pragma omp critical is used to push newly active vertices onto the active deque. With many threads, this single critical section can become a bottleneck as only one thread at a time can safely modify the deque.
4. Work Imbalance
   - The discharge(u, sink) routine processes one vertex at a time. If certain vertices have significantly more edges or higher flow, threads might end up waiting for that work to complete. This is exacerbated by repeated attempts to grab the front of the active deque (also locked by #pragma omp critical).

5. Speedup below 1 for higher threads
    - There are several synchronization points—such as the critical section for updating the active queue and atomic updates for flow and excess—that become bottlenecks when many threads are involved. As more threads compete for these shared resources, the overhead of locking, contention, and thread scheduling increases. Once this overhead surpasses the benefit gained from parallel work, the total runtime becomes longer than the serial version, causing the speedup to drop below 1.
6. Peak at 8 threads:
    - At 8 threads, your parallel implementation reaches a performance peak because the workload is sufficiently distributed among the available cores, so the benefits of parallelizing the push and discharge operations are maximized before synchronization overheads begin to dominate. At this point:
    - Optimal Core Utilization: The hardware on which the algorithm was run has 8 physical cores, meaning each thread can run on its own core without incurring significant context-switching or resource contention.
    - Balanced Overhead vs. Work: With 8 threads, the frequency of atomic operations and critical sections is still manageable, so the overhead of synchronizing shared data (like the active deque and excess arrays) is low compared to the amount of work being done.
    - Diminishing Returns Beyond 8: Beyond 8 threads, increased contention and overhead from synchronization (due to more threads trying to access shared data) outweigh the benefits, causing performance to decline.
7. Optimal Thread Count
    - The results suggest an optimal thread count is 4 for this particular problem instance. After that point, the overhead of atomic and critical sections in the code overshadows the parallel gains. This is why drop in speedup below 1 at higher thread counts is seen.
8. Parallelization factor
    - A parallelization factor of 46.92 suggests that the parallel implementation provides significant speedup compared to the serial execution, with the algorithm benefiting from parallelizing key operations. However, the diminishing returns at higher thread counts indicate that synchronization overheads and contention limit performance beyond an optimal thread count, revealing an imbalance between parallel efficiency and execution cost for a couple of seconds.

Overall, these results show that while parallel approach does leverage OpenMP to distribute some operations (especially pushes on edges), the high degree of synchronization required in push–relabel dampens performance gains at larger thread counts.