PUSH RELABEL SERIAL CODE PROFILING

ABOUT THE ALGORITHM: Push-Relabel algorithm is an efficient method for computing the maximum flow in a flow network by maintaining a preflow and adjusting flow locally at each node. Unlike augmenting path algorithms like Ford-Fulkerson, it allows nodes to temporarily store excess flow and redistributes it using two operations: push and relabel. The algorithm assigns a height label to each node, ensuring that flow moves from higher to lower nodes. Initially, the source node is given a height equal to the number of nodes, and flow is pushed to its neighbors. If a node with excess flow cannot push further, it increases its height to enable further movement. This process continues until no active nodes remain, at which point the total flow reaching the sink represents the maximum flow. With an average time complexity of $O(V^2\sqrt{E})$, the Push-Relabel method is particularly efficient for dense graphs.

INPUT SAMPLE:

1200 0 1199
658 991 2
382 47 51
366 838 45
58 155 94
377 575 8
316 562 28
251 894 42
40 1145 53
705 558 72
907 1110 66
862 729 72

The first line contains the number of vertices the graph has, source node and sink node details. From the second line, the input contains the edge starting node, ending node and its weight. Around 1,000,000 edges are present in the graph.

OUTPUT:
Maximum flow is 42406.

FUNCTIONAL PROFILING

FLAT PROFILE:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 32.41      0.17     0.17 469070802     0.00     0.00  std::vector<Edge, std::allocator<Edge> >::operator[](unsigned lon
g)
 30.56      0.34     0.17 467282346     0.00     0.00  std::vector<Edge, std::allocator<Edge> >::size() const
 12.96      0.41     0.07     1544     0.05     0.25  Graph::push(int)
 12.96      0.48     0.07      603     0.12     0.25  Graph::relabel(int)
  9.26      0.53     0.05      941     0.05     0.18  Graph::updateReverseEdgeFlow(int, int)
  1.85      0.54     0.01   262143     0.00     0.00  void std::__relocate_object_a<Edge, Edge, std::allocator<Edge> >(E
dge*, Edge*, std::allocator<Edge>&)
  0.00      0.54     0.00  1725394     0.00     0.00  std::vector<Vertex, std::allocator<Vertex> >::operator[](unsigned
long)
  0.00      0.54     0.00  1062415     0.00     0.00  Edge&& std::forward<Edge>(std::remove_reference<Edge>::type&)
```

1. **std::vector<Edge, std::allocator<Edge>>::operator[](unsigned long)** (32.41%, 469,078,092 calls) – This function is used to access elements in a vector indicating frequent edge lookups in adjacency lists. Its high call count suggests that edge access is a performance bottleneck. Instead of storing the edge data as a vector using an adjacency list would bring the time down.

2. **Graph::push(int)** (12.96%, 1,544 calls) – Implements the push operation in the push-relabel algorithm, moving excess flow from a node to its lower-height neighbors. The number of calls is relatively low, but since it consumes a significant portion of execution time, optimizing how flow is pushed could improve performance.

3. **Graph::relabel(int)** (12.96%, 603 calls) – This function increases a node's height when no valid push operation is possible. It has fewer calls compared to push(), but since it takes equal execution time, it suggests that relabeling is computationally expensive. Optimizing the relabel operation or reducing unnecessary calls might help.
4. **Graph::updateReverseEdgeFlow(int, int)** (3.70%, 262,143 calls) – Updates reverse edge capacities in the residual graph after a flow push. While it is called less frequently, its impact on performance suggests that optimizing residual graph updates could be beneficial.

CALL GRAPH:

```
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 1.85% of 0.54 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.54                 main [1]
                0.00    0.53       1/1           Graph::getMaxFlow(int, int) [2]
                0.00    0.01  179082/179082      Graph::addEdge(int, int, int) [15]
                0.00    0.00       1/1           Graph::Graph(int) [83]
                0.00    0.00       1/1           Graph::~Graph() [84]
-----------------------------------------------
                0.00    0.53       1/1           main [1]
[2]     98.2    0.00    0.53       1         Graph::getMaxFlow(int, int) [2]
                0.07    0.31    1544/1544        Graph::push(int) [3]
                0.07    0.08     603/603         Graph::relabel(int) [7]
                0.00    0.00       1/1           Graph::preflow(int) [16]
                0.00    0.00    3089/3089        overFlowVertex(std::vector<Vertex, std::allocator<Vertex> >&) [37]
                0.00    0.00       1/601         std::vector<Vertex, std::allocator<Vertex> >::back() [48]
-----------------------------------------------
                0.07    0.31    1544/1544        Graph::getMaxFlow(int, int) [2]
[3]     70.5    0.07    0.31    1544         Graph::push(int) [3]
                0.05    0.12     941/941         Graph::updateReverseEdgeFlow(int, int) [5]
                0.07    0.00 190384508/469070802     std::vector<Edge, std::allocator<Edge> >::operator[](unsigned long)
 [4]
                0.07    0.00 189425750/467282346     std::vector<Edge, std::allocator<Edge> >::size() const [6]
                0.00    0.00  639265/1725394     std::vector<Vertex, std::allocator<Vertex> >::operator[](unsigned long)
 [24]
                0.00    0.00     941/941         int const& std::min<int>(int const&, int const&) [43]
```

```
-----------------------------------------------
                0.00    0.00  181098/469070802     Graph::preflow(int) [16]
                0.04    0.00 108992906/469070802     Graph::relabel(int) [7]
                0.06    0.00 169512290/469070802     Graph::updateReverseEdgeFlow(int, int) [5]
                0.07    0.00 190384508/469070802     Graph::push(int) [3]
[4]     32.4    0.17    0.00 469070802         std::vector<Edge, std::allocator<Edge> >::operator[](unsigned long) [4]
-----------------------------------------------
                0.05    0.12     941/941         Graph::push(int) [3]
[5]     32.0    0.05    0.12     941         Graph::updateReverseEdgeFlow(int, int) [5]
                0.06    0.00 169512290/469070802     std::vector<Edge, std::allocator<Edge> >::operator[](unsigned long)
 [4]
                0.06    0.00 169229397/467282346     std::vector<Edge, std::allocator<Edge> >::size() const [6]
                0.00    0.00     936/180306      Edge::Edge(int, int, int, int) [31]
                0.00    0.00     936/936         std::vector<Edge, std::allocator<Edge> >::push_back(Edge const&) [44]
-----------------------------------------------
```

In the main function[1], the function to calculate the maxflow is called.
In the getMaxFlow function[2], push, relabel, preflow and overFlowVertex functions are called multiple times.
Whenever the push function[5] and preflow[4] is called, the updateReverseEdgeFlow function is called.
So, trying to parallelise these functions would increase the performance of the algorithm.

LINE PROFILING

Hotspots and Frequency Analysis

1. Preflow Initialization (Graph::preflow, lines 82–105)
    - The preflow function is called once and, within it, loops over all edges.
    - Within preflow, for each edge starting at the source, it sets the initial flow and pushes a corresponding reverse edge (lines 89–102).

```
        1:   82:void Graph::preflow(int s)
        -:   83:{
        -:   84:    // Making h of source Vertex equal to no. of vertices
        -:   85:    // Height of other vertices is 0.
        1:   86:    ver[s].h = ver.size();
call    0 returned 1
call    1 returned 1
        -:   87:
        -:   88:    //
   179371:   89:    for (int i = 0; i < edge.size(); i++)
call    0 returned 179371
branch  1 taken 179370
branch  2 taken 1 (fallthrough)
```

    Inference:

    - Although this loop goes over every edge, it runs only once at the start. Its cost is fixed and not as dominant compared to the iterative push–relabel operations.

2. The OverFlowVertex Function (lines 108–116)
    - This function scans through all vertices (except the source and sink) to find a vertex with positive excess flow.
    - It is called repeatedly within the main while loop of getMaxFlow (line 215 and again at line 217).

```
     3089:  108:int overFlowVertex(vector<Vertex>& ver)
        -:  109:{
   901935:  110:    for (int i = 1; i < ver.size() - 1; i++)
call    0 returned 901935
branch  1 taken 901934
branch  2 taken 1 (fallthrough)
   901934:  111:        if (ver[i].e_flow > 0)
call    0 returned 901934
branch  1 taken 3088 (fallthrough)
branch  2 taken 898846
     3088:  112:            return i;
        -:  113:
        -:  114:    // -1 if no overflowing Vertex
        1:  115:    return -1;
        -:  116:}
```

    Inference:

    - Since the while loop (in getMaxFlow) runs until no vertex has excess flow, this function is a key hotspot. Every iteration of the main loop starts by scanning almost all vertices.

3. Graph::push Function (lines 138–179)
    - Inside push, there is a loop over all edges (line 142) with very high iteration counts shown (e.g., 189,425,750 iterations in one of the reported lines).

```
     1544:  138:bool Graph::push(int u)
        -:  139:{
        -:  140:    // Traverse through all edges to find an adjacent (of u)
        -:  141:    // to which flow can be pushed
189425750:  142:    for (int i = 0; i < edge.size(); i++)
call    0 returned 189425750
branch  1 taken 189425147
branch  2 taken 603 (fallthrough)
        -:  143:    {
        -:  144:        // Checks u of current edge is same as given
        -:  145:        // overflowing vertex
189425147:  146:        if (edge[i].u == u)
call    0 returned 189425147
branch  1 taken 318688 (fallthrough)
branch  2 taken 189106459
```

- Within this loop, for each candidate edge from the overflowing vertex, the function checks if a push can be made (comparing flows and capacities) and, if so, performs the push and calls updateReverseEdgeFlow.
  Inference:
- This function is one of the most frequently executed parts of the algorithm and represents the “workhorse” of the push–relabel method.

4. Graph::updateReverseEdgeFlow Function (lines 119–135)
   - This function is called whenever a push is performed, and it itself loops over all edges (line 123) to locate the reverse edge.

```
      941:  119:void Graph::updateReverseEdgeFlow(int i, int flow)
        -:  120:{
      941:  121:    int u = edge[i].v, v = edge[i].u;
call    0 returned 941
call    1 returned 941
        -:  122:
169229397:  123:    for (int j = 0; j < edge.size(); j++)
call    0 returned 169229397
branch  1 taken 169228461
branch  2 taken 936 (fallthrough)
        -:  124:    {
169228461:  125:        if (edge[j].v == v && edge[j].u == u)
call    0 returned 169228461
branch  1 taken 281942 (fallthrough)
branch  2 taken 168946519
call    3 returned 281942
branch  4 taken 5 (fallthrough)
branch  5 taken 281937
branch  6 taken 5 (fallthrough)
branch  7 taken 169228456
```

   Inference:
   - Because it is invoked during every successful push, and its inner loop can run over many edges, it is another hotspot. Optimizing this lookup (for example, by maintaining an index or using a more efficient data structure) could yield performance improvements.

5. Graph::relabel Function (lines 182–207)
   - This function loops over all edges (line 188) to determine the minimum height among adjacent vertices, which is then used to update the height of the overflowing vertex.

```
      603:  182:void Graph::relabel(int u)
        -:  183:{
        -:  184:    // Initialize minimum height of an adjacent
      603:  185:    int mh = INT_MAX;
        -:  186:
        -:  187:    // Find the adjacent with minimum height
108447752:  188:    for (int i = 0; i < edge.size(); i++)
call    0 returned 108447752
branch  1 taken 108447149
branch  2 taken 603 (fallthrough)
        -:  189:    {
108447149:  190:        if (edge[i].u == u)
call    0 returned 108447149
branch  1 taken 181523 (fallthrough)
branch  2 taken 108265626
```

Inference:

- Although it is called less frequently than push, it still is a significant part of the overall loop when no push is possible.

INFERENCES FROM FUNCTIONAL AND LINE PROFILING

1. **push(int):**
   This is the workhorse of the push–relabel algorithm. Its inner loop iterates over all outgoing edges from a vertex to determine possible flow pushes. Since each edge check is independent, the scan can be divided among multiple threads. Parallelizing the evaluation of candidate edges (while carefully synchronizing updates to excess flows) could yield significant performance gains.
2. **updateReverseEdgeFlow(int, int):**
   Every time a push occurs, this function is invoked to locate and update the reverse edge by scanning the entire edge vector. The search itself is a hotspot due to the high frequency of calls. Parallelizing the inner loop or, better yet, restructuring the data to allow direct reverse-edge lookups (for instance, by maintaining an index or using a hash map) can reduce the overhead substantially.
3. **relabel(int):**
   In scenarios where no push is possible, the relabel function is called to update the vertex's height. This function loops over all edges to find the minimum height among adjacent vertices. The minimum computation in this loop can be parallelized using a reduction approach, where each thread computes the minimum over a partition of edges before combining the results.
4. **OverFlowVertex Function:**
   This function scans the list of vertices to identify one with positive excess flow. Since each vertex check is independent, the search can be performed concurrently over subsets of vertices. A parallel reduction can then be used to quickly find a candidate vertex, which is especially beneficial given its frequent invocation within the main while loop of the algorithm.

PERFORMANCE COUNTER PROFILING

L2CACHE:

Metric	Core 0
Runtime (RDTSC) [s]	1.6254
Runtime unhalted [s]	2.1624
Clock [MHz]	4573.3451
CPI	0.3124
L2 request rate	0.0177
L2 miss rate	0.0066
L2 miss ratio	0.3706

L3CACHE:

Metric	Core 0
Runtime (RDTSC) [s]	1.7672
Runtime unhalted [s]	2.3409
Clock [MHz]	4565.9252
CPI	0.3382
L3 request rate	0.0001
L3 miss rate	4.322450e-06
L3 miss ratio	0.0704

L2 miss rate was observed to be 0.0066 and miss ratio is 0.3706 whereas L3 miss rate is comparatively very low and miss ratio is 0.0704. The high L2 miss rate is most likely because of the way data is stored using arrays. Changing this will reduce the misses further more.

FLOPS DP:

Metric	Core 0
Runtime (RDTSC) [s]	1.7622
Runtime unhalted [s]	2.3366
Clock [MHz]	4559.9396
CPI	0.3376
DP [MFLOP/s]	3.121051e-05
AVX DP [MFLOP/s]	0
Packed [MUOPS/s]	0
Scalar [MUOPS/s]	3.121051e-05
Vectorization ratio	0

FLOPS SP:

Metric	Core 0
Runtime (RDTSC) [s]	1.7413
Runtime unhalted [s]	2.3237
Clock [MHz]	4588.2665
CPI	0.3357
SP [MFLOP/s]	0
AVX SP [MFLOP/s]	0
Packed [MUOPS/s]	0
Scalar [MUOPS/s]	0
Vectorization ratio	-

The algorithm primarily involves integer-based operations (e.g., flow updates and height adjustments), so floating-point optimizations are not a priority. However, vectorization techniques (SIMD) may still be explored for integer operations to speed up edge processing.

CLOCK:

Metric	Core 0
Runtime (RDTSC) [s]	1.7005
Runtime unhalted [s]	2.2606
Clock [MHz]	4588.0510
Uncore Clock [MHz]	0
CPI	0.3266
Energy [J]	64.0550
Power [W]	37.6676
Energy DRAM [J]	2.3118
Power DRAM [W]	1.3594

The total energy consumed is 64.05J whereas DRAM consumed 2.31J which is indicating that the main memory is not accessed very much.

ICACHE:

Metric	Core 0
Runtime (RDTSC) [s]	1.7194
Runtime unhalted [s]	2.2684
Clock [MHz]	4554.1092
CPI	0.3278
L1I request rate	0.1704
L1I miss rate	3.004138e-06
L1I miss ratio	1.762587e-05
L1I stalls	426269
L1I stall rate	1.807261e-05

L1 instruction cache miss rate, miss ratio and stall rate are very low.

CYCLE STALLS:

Metric	Core 0
Runtime (RDTSC) [s]	1.7502
Runtime unhalted [s]	2.3205
Clock [MHz]	4575.0070
CPI	0.3353
Total execution stalls	926547652
Stalls caused by L1D misses [%]	88.2860
Stalls caused by L2 misses [%]	12.8842
Stalls caused by memory loads [%]	97.4047
Execution stall rate [%]	11.7165
Stalls caused by L1D misses rate [%]	10.3440
Stalls caused by L2 misses rate [%]	1.5096
Stalls caused by memory loads rate [%]	11.4124

The high CPU cycle count suggests that the algorithm is not fully utilizing execution units and may be memory-bound. Multi-threading or parallelization of key functions (push, relabel, updateReverseEdgeFlow, overFlowVertex) could help improve CPU utilization.