Parallelizing Dinic's Algorithm for Max Flow in Flow Networks

By Benedict Ozua & Matei Budiu

15-418 Final Project

Summary

Our project aims to speed up Dinic's algorithm for finding max flow in a flow network. We first created a simple sequential version to analyze bottlenecks, so that we could focus our optimization work where it could provide the greatest speedups. We wrote our algorithms in C++, using OpenMP to parallelize on the CPU and CUDA to parallelize on the GPU. We experimented with several approaches with parallelism on the CPU, including using atomics, critical sections, and lock free data structures. We evaluated a variety of graph sizes to understand performance characteristics and memory bounds of the new algorithms.

BACKGROUND

Max Flow Algorithms

Flow networks are directed graphs with a source and sink node. Every edge has a nonnegative capacity assigned to it. A flow in a flow network is an assignment of values to each edge such that all values are between 0 and the edge's capacity, and such that for all nodes (except the source and sink) the sum of values for in-edges equals that for out edges. The total capacity of the flow is defined as the sum of the out-edge values minus the sum of the in-edge values of the source.

Dinic's algorithm is one method to find the maximum flow of a flow network. The algorithm operates on an augmented graph - a copy of the input graph where each directed edge from A to B is paired with a reverse edge of capacity zero. As a unit of flow is pushed through an edge, its capacity gets reduced by 1, while its reverse edge gains a capacity of 1, allowing the algorithm to send a unit of flow backwards on an edge.

Dinic’s algorithm involves repeating two steps: computing a layered/level graph inside of a flow network’s augmented graph (using BFS to get the distances of each node from the source), and then finding a blocking flow in that layered graph (repeatedly invoking DFS to find paths in the graph to add to the flow value).

Sequential Design

At a high level, Dinic’s algorithm is composed of two repeating phases:

Phase 1 creates a layered graph of depth *d* from a source vertex to the sink vertex, using breadth-first search. This layered graph is a directed acyclic subset of the original graph, including only edges that increase in BFS depth.

Phase 2 finds a blocking flow inside this layered graph from the source to the sink vertex. A blocking flow is a flow where all paths from source to sink in the layered graph have an edge that is saturated in the graph after applying the blocking flow to the graph. In other words, there is an edge in each path from source to sink that will no longer exist (ie, have remaining capacity zero) after the blocking flow is added to the graph. The blocking flow is created through a breadth-first search traversal that is *d* layers deep.

The breadth first algorithm will add an edge to the layered graph if the edge connects layer *i* to layer *i +1*. Edges going beyond the depth of the sink, d, are not included in the level graph because the BFS stops once it has reached the sink. This layered graph will then be used in the second phase of Dinic’s.

Algorithm Breadth First Search

```
$visited \leftarrow \{v_i \mid 0 \leq i < n\}$
$frontier =$ queue of $\{SOURCE\}$
set $SOURCE$ layer to 0
$foundSink \leftarrow false$
$sinkLayer \leftarrow$ UNSET
**while** frontier is not empty and foundSink $==$ false **do**
    $v \leftarrow frontier.front()$
    pop front of frontier
    **if** $v == SINK$ **then**
        $foundSink \leftarrow true$
        $sinkLayer \leftarrow v.layer$
        **continue**
    **end if**
    **if** $sinkLayer$ not UNSET and $v.layer \geq sinkLayer$ **then**
        **continue**
    **end if**
    **for** neighbors Neigh of v **do**
        **if** Neigh.layer is UNSET or Neigh.layer $==$ v.layer $+ 1$ **then**
            add edge $(v, Neigh)$ to layered graph
            **if** Neigh is UNSET **then**
                $Neigh.layer \leftarrow v.layer + 1$
                append Neigh to frontier
            **end if**
        **end if**
    **end for**
    $visited[Neigh] \leftarrow true$
**end while**
**return** foundSink
```

In the second phase, depth-first search is repeatedly run on the layered graph, starting at the source. When a path from the source to sink is found, the maximum flow that can be pushed through the graph is determined by finding the minimum edge capacity along the path (the bottleneck). Then, that much flow is pushed through every edge on the path, reducing the edge capacities by the flow amount, and increasing backward edge capacities accordingly (the backward edges are not part of the layered graph, so flow will not be able to run backward through them until the next iteration).

As the DFS traverses edges, it will remove edges that lead to dead ends (dead edges). These can appear due to either dead ends in the initial level graph or because an edge was saturated by a previous flow pushed through a graph. We keep track of dead edges by assigning each mode in the level graph a "next neighbor" counter that is used for deciding which edge to traverse in DFS next and incremented once the edge to that neighbor is deemed "dead".

Algorithm Depth First Search (Dead Edge)

```
$visited \leftarrow \{v_i | 0 \leq i < n\}$
stack as empty
while stack is not empty do
    $v = stack.top()$
    $visited[v] = true$
    if v has no neighbors then
        $stack.pop()$
        discard $(v.parent, v)$
        continue
    end if
    $neigh \leftarrow$ neighbor of v
    if visited[neigh] or capacity of $(v, neigh) == 0$ then
        discard $(v, neigh)$
        continue
    end if
    $neigh.parent \leftarrow v$
    if $neigh == SINK$ then
        return $true$
    end if
end while
return $false$
```

Using these modified BFS and DFS implementations, there is enough information to implement the Dinic's algorithm. This "skeleton" for our algorithm is maintained across all of our optimized versions too, with changes being done to the BFS and DFS parts of the algorithm.

Algorithm Dinics Algorithm

```
while BFS() do//Algorithm 1
    while DFS() do//Algorithm 2
        iterate from SINK to SOURCE going through each node's parent
        $min\_capacity \leftarrow$ min capacity from each node $(v, v.parent)$
        capacity $(v.parent, v) -= min\_capacity$
        capacity $(v, v.parent) += min\_capacity$
    end while
end while
return sum of flow leaving SOURCE
```

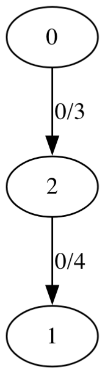
The algorithm runs in *O(n^2m)* time, since each BFS is guaranteed to have a depth greater than the previous, up to a maximum depth of n (for at most n iterations of the outer for loop), and since each runs in O(m) time, there will be at most *O(mn)* time spent in BFS. There will be at most m DFSs per each inner loop, since every DFS will saturate at least one edge. Each DFS will cover at most n nodes since the level graph is acyclic and directed, and be *O(n)*, so at most *O(n^2m)* time will be spent in DFS. Further analysis about the time complexity can be found from our reference, Daniel Anderson and David Woodruff. In practice, despite the complexity bounds, we find that for most graphs much less time is spent in DFS than in BFS.

Key Structures

For BFS, the key data structures keep track of the frontier and the neighbors of each edge. Our DFS implementation uses a stack to track the current path from the source. We use a vector (indexed by source vertex) of unordered maps (indexed by destination vertex) to store edges (as structs which keep track of capacities). This vector is read often to navigate the graph, but it is not modified (apart from the edge capacities).

Input format

Our program takes in a file describing the number of vertices. Each line from 1 to *n* afterward contains information about its directed edges and its capacity to those nodes.



```
3
1  2  3
0
1  1  4
```

The example above shows a graph of 3 vertices. The first line is the source, the second is the sink, and the rest are other vertices. This file says that the source has an edge of capacity 3 to the vertex 2. Vertex 1 (the sink) has 0 directed edges. The last vertex has 1 directed edge to vertex 1 with capacity 4. All vertices are number 0 to *n*-1.

Bottlenecks

From benchmarks taken on the sequential version of the code, BFS takes most of the computational time. For some graphs, BFS takes more than 90% of the computational time. Thus, it is most beneficial to look at how to parallelize the BFS implementation before thinking about parallel DFS implementations. Additionally, graph algorithms have inconsistent data access problems which can be problematic for hardware prefetchers to optimize.

Limitations

It should be noted that BFS has a natural parallelization plan. Each node in a frontier can independently look to add its next valid neighbors to the next frontier. Afterwards, duplicates can be removed or kept (depending on which is less expensive). DFS on its own is naturally sequential as past searches influence the future searches making it hard to create independent tasks. Additionally, DFS in this implementation is very unlikely to run into a dead end. Each non dead node takes it further down the tree making it a good graph structure for DFS to operate on. Additionally, after the DFS, a thread must modify the graph which is inherently sequential and unlikely to be parallelizable.

APPROACH

Initially, we tested the sequential algorithm on four files:

- bigInput1 has 50,000 vertices and 999,752 edges.
- bigInput3 has 50,000 vertices and 49,502,662 edges.
- verydensegraph has 4000 verts and 20,227,196 edges.
- ginormousgraph has 50,000 vertices and 99,005,324 edges.

Time measurements	bigInput1	bigInput3	verydensegraph	ginormousgraph
Compute Time	0.494581	27.289853	2.077583	26.86948
Time in BFS	0.441977	27.172728	1.969035	26.75314
Time in DFS	0.036268	0.098355	0.102571	0.09772

From these results, we decided to parallelize the BFS portion of the algorithm. These test cases were run on the Gates machines. Given how small the DFS portion of the algorithm is, our goal is to find a BFS implementation that has a small portion of the compute time relative to the DFS time.

Tools

We used OpenMP and CUDA with C++ to parallelize the BFS portion of the code. We plan to use the Gates Machines to run our analysis.

Implementation strategies

Atomic Queue with OpenMP capture

The initial attempt to make ParallelBFS was through making the frontier a parallel data structure. The original idea came from the 15210 lecture notes on parallel BFS that uses a similar idea https://www.diderot.one/courses/154/books/677/chapter/9633. However, basic tests on bigInput1 demonstrated that the overall time was much higher than the sequential version. Additionally, their overall time increased with more threads. Manual time measurements demonstrated that long portions of code were in the Critical and capture OpenMP pragmas. The critical pragma is used when inserting edges into the layered graph, to prevent the same vertex from having its layered edges vector modified at the same time by multiple threads. The atomic capture pragma helps to develop an atomic way of inserting elements into the frontier queue.

Optimized frontiers and layered graphs (Fast BFS)

One technique to speed up BFS was to change how local frontiers from each thread were merged into a greater frontier. Originally, we used an atomic operation to reserve enough space in the merged frontier by incrementing a counter, and then copied to the reserved space asynchronously. It turns out that the actual time required to copy to merge frontiers is negligible (0.000255s of the 3.34s of BFS is spent merging frontiers (in parallel) in the ginormousgraph test case), and that both using a sequential scan to calculate start indices then copying in parallel and just sequentially merging are fast alternatives.

Splitting up layered graph neighbor calculation into separate buffers instead of using a critical section to prevent adding at the same time also provided a great increase in speed. Critical sections are very expensive to use, especially for high-contention objects like the layered graph neighbor vectors. When running on the verydensegraph test case with 8 threads, we found that each thread was spending around 0.5s of the 0.69s of BFS time in the critical section which added edges to the layered graph. Using separate buffers and removing the critical marker made the same section of code only take around 0.011s per thread, improving BFS time to 0.22s. Our DFS was modified to loop through all these separate buffers, since it was already sequential.

Our approach also did not use locking to verify that vertices hadn't already been visited, so sometimes duplicates could find their way into the frontier. That said, this was not an issue for the algorithm's correctness since our DFS was entirely sequential, and having duplicates would make no difference since exploring the same edge again would not allow for more flow to be pushed along it than would normally be allowed.

No frontier BFS (Per Vertex)

Building up from the atomic capture, one idea was to get rid of the frontier as it can be complicated to code around. Additionally, appending to arrays and resetting them do not lend themselves well to independent workloads that are good for parallel programs. An easier idea is to notate the level that each node is at. Then, loop through the vertices in parallel and see if it is in the frontier (rather than figuring out the frontier and looping through the subset). This approach makes it easier to parallelize, involves fewer writes to shared memory and demonstrates better performance. Instead of iterating through the frontier, the algorithm iterates through all of the vertices, and checks if their depth is the same as the step count that the algorithm is on. Because of its data-parallel structure, we adapted this idea in our CUDA approach.

C/C++
```
bool MGraph::bfsStep(std::vector<bool> &visited, std::vector<int> &sizes,
                    int step) {
 bool sz = false;
#pragma omp parallel for reduction(| : sz)
 for (int i = 0; i < visited.size(); i++) {
   if (this->vertices[i].layer != step)
     continue;
   for (int neigh = 0; neigh < this->adj_list[i].size(); neigh++) {

     int neighInd = this->adj_list[i][neigh];
     Vertex &dstVert = this->vertices[neighInd];
     Vertex &srcVert = this->vertices[i];
     if (dstVert.layer <= step || !isLayerReachable(srcVert, dstVert)) {
       continue;
     }

     sizes[i]++;
     visited[neighInd] = true;
     dstVert.layer = step + 1;
     srcVert.layered_dst.push_back(neighInd);
     sz |= true;
   }
 }
 return sz;
}
```

Double loop BFS

Building up from the no frontier BFS idea, we were thinking about iterating through all the possible sources and all possible destinations, testing if the source-destination pair is an edge, and if source has the proper step value. The idea with doing a double for loop is to improve on the 50% cache miss rate that is explained in more detail in the results section. This idea would lead to predictable access patterns for the hardware when looping and the two for loops are completely independent, and thus both loops can be collapsed into a single for loop through the OpenMP collapse parameter. However, this loop runs in *O(n^2)* time and proved that many graphs tried on this implementation ran for a long time. The sheer amount of volume showed that this method would not be effective in a faster BFS algorithm.

CUDA implementation

As BFS is well-known to be somewhat parallelizable, several others have explored the idea of a GPU implementation of BFS. We decided to take a look at Matthias Springer's implementations of BFS in CUDA and adapt their idea of a simple frontier-using CUDA BFS to work with creating a level graph. Our approach involved sending the GPU data about the graph and its edge capacities, and repeatedly running a kernel that advanced the frontier one level deeper.

We parallelized per vertex, with each vertex (1) figuring out if it is part of the frontier, and if not, exiting, (2) looping through all its neighbors, (3) checking if the neighbors have already been discovered, and (4) updating their depth and adding them to the frontier and layered edges list.

C/C++

```c
__global__ void bfsKernel(...) {
 int vert = blockIdx.x * blockDim.x + threadIdx.x;

 if (!frontier[vert] || vert >= count) // 1
   return;

 int es = edgesStart[vert];
 int ec = edgesCount[vert];
 int dist = level;
 int lc = 0;

 for (int i = 0; i < ec; i++) { // 2
   int dest = edges[es + i];

   if (edgeCapacities[es + i] > 0 && vertDists[dest] >= dist + 1) { // 3
     if (dest == 1)
       *foundSink = true;

     vertDists[dest] = dist + 1; // 4
     newFrontier[dest] = true;
     layeredEdges[es + lc] = dest;
     *progressed = true;
     lc++;
   }
 }

 layeredEdgesCount[vert] = lc;
}
```

To be able to easily use CUDA, we had to change the format we stored our data in. Our new approach is more compact:

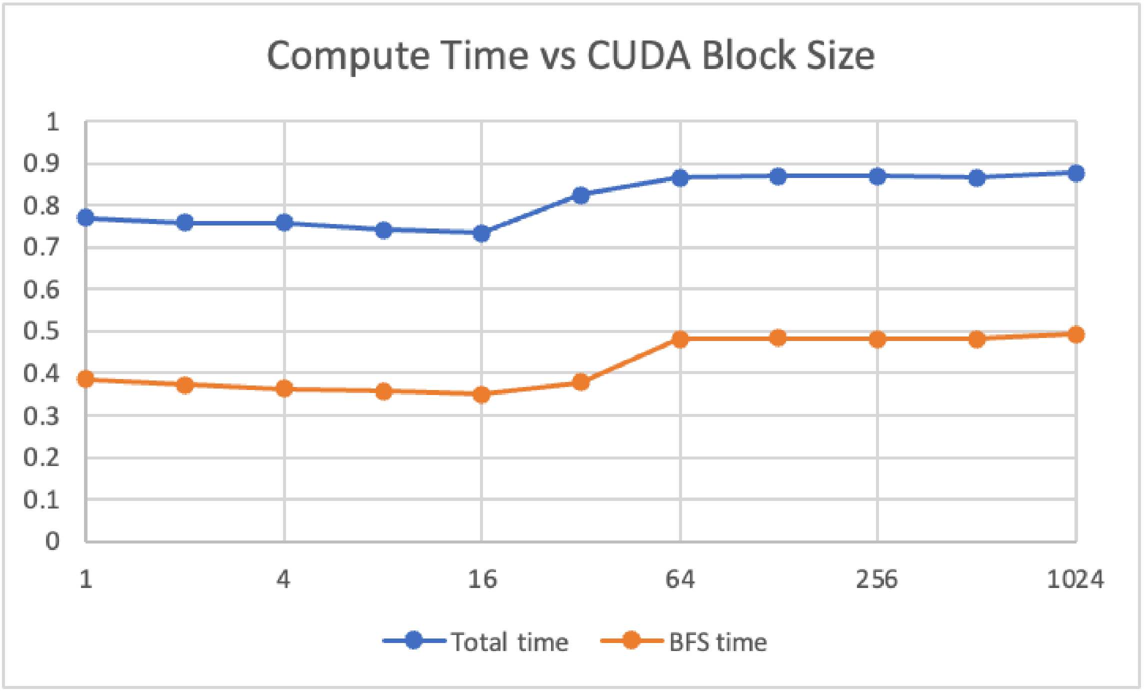
Edges are stored in a compact array:

- The edges of each vertex are in a contiguous block in the array
- An array indexed by vertex is used to store the starting index in the edge array for that vertex's edges, and a different similarly-indexed array is used to store the number of edges per vertex
- Edge capacities and edge destinations are separate arrays but indexed the same way
- The layered graph edges array (written to by the GPU) is indexed with the same starting points as the other edge arrays, but has a separate edge count array because layered graph edges are a subset of the graph edges. This means that there can be some unused space in this array.

Originally, we would convert between formats and upload the data to the GPU for every BFS search. This proved very slow, since the data conversion was sequential and took a lot of time. In the ginormous graph test case, our original implementation took 23.5s to run BFS, which we optimized down to 0.48s. Out of these 23.5s, 22.1s was spent converting the data, 0.12s was spent in cudaMalloc and cudaFree calls which we changed to only happen once at the start and end of the program, and 0.15s was spent copying edge destination, index, and count data to the GPU, which was done every BFS call and which we changed to only happen once at initialization.

In fact, the vast majority of time in the BFS call was spent transferring data, rather than computing. In the ginormous graph test case, out of the 0.48s spent in BFS, we found that 0.000033s was spent in the kernel call itself, and 0.17s was spent in the BFS loop which continually spawned kernels, updated the level counter, and switched the frontier and next frontier data. 0.15s was spent sending data to the GPU for a BFS call, and 0.16s was spent transferring results back from the GPU.

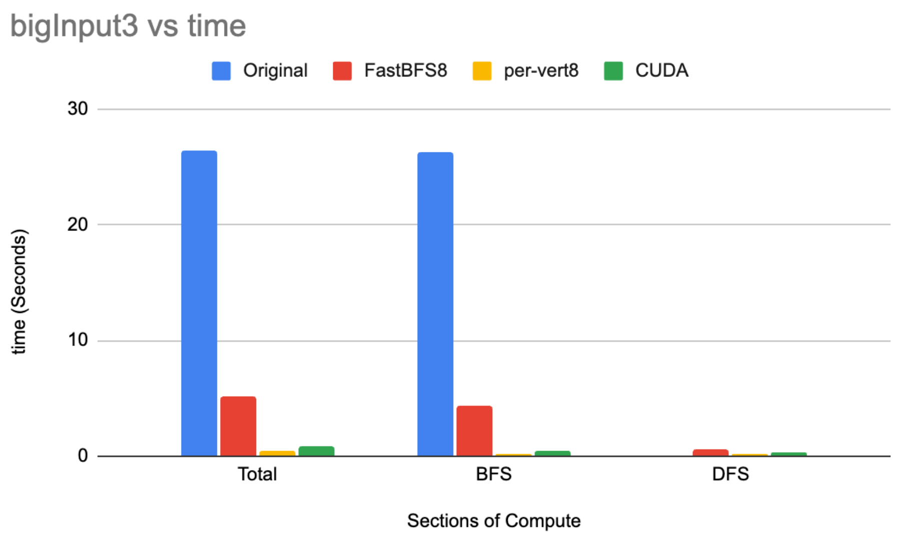
We also experimented with the block size of CUDA calls. We found that the optimal block size was 16, and it allowed us to improve our BFS time from 0.48s with our original block size of 512 to 0.35s, on ginormous graph. We suspect that this may be because BFS involves a lot of scattered memory accesses and early returns, and having too wide a block size may cause poor load balancing. Only vertices in the frontier do computation, but all vertices get their own thread, with other vertices returning early, so having a larger block size probably results in more threads that have returned early waiting for other threads in their block to finish before they can finish, due to the data-parallel nature of the GPU.

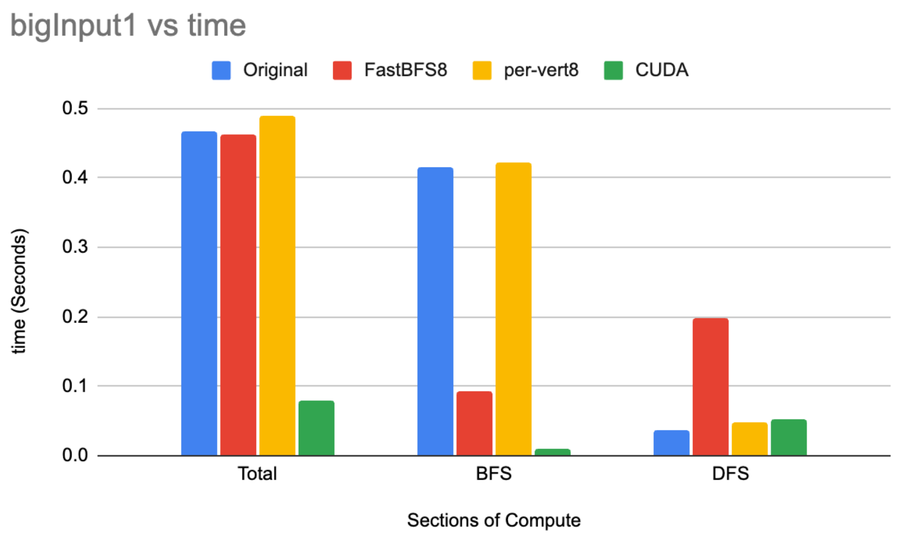


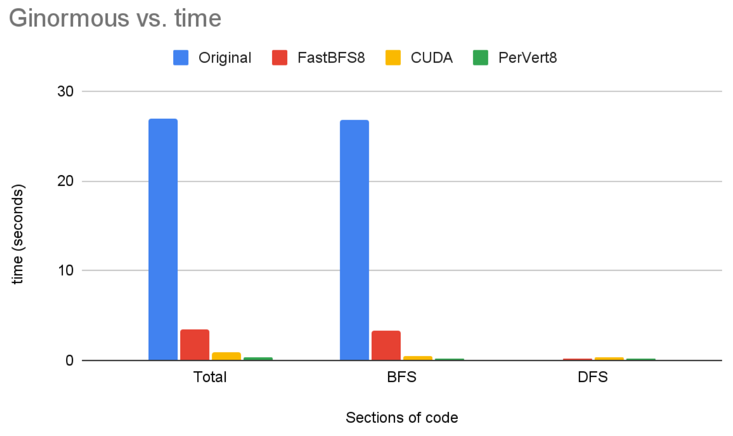
RESULTS

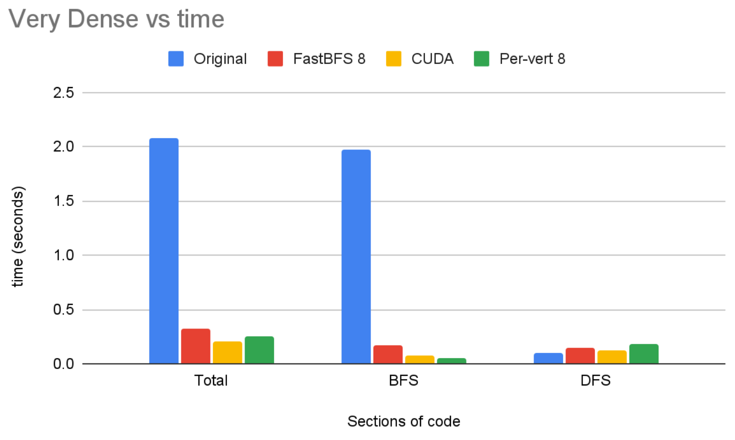
Note, for these analysis, we do not test the performance of the atomic queue and double loop implementations because they were strictly worse and/or didn't complete on the bigger graphs that we were working with in a reasonable time. They provided a stepping stone to the future, more performant implementations which were CUDA, FastBFS, and per vertex BFS.

Wall-clock graph time breakdown









Graph breakdown

BigInput1 is a graph with 50K vertices and 999752 edges, approximately a 20:1 edge to node ratio. BigInput3 is a graph with 50K vertices and 49502662 edges, approximately a 990:1 edge to node ratio. Ginormous contains 50K vertices and 41667556 edges. VeryDense has 4K vertices and 20227196. We used our graph generator to generate them, to analyze how our algorithms would perform with high vertex and edge counts.

For the CPU threaded programs, we graphed the 8 threaded version of the program under OpenMP. The CUDA version uses 512 CUDA threads. We found that the CUDA algorithm performed best for the smaller verydense and bigInput1 graphs, and per-vertex performed best for the larger bigInput3 and ginormous graphs. The CUDA algorithm always outperformed the FastBFS algorithm with 8 threads.

Result goals

Our goals were to find a BFS implementation such that the BFS does not become the bottleneck in timing, thus reducing the total Dinic's algorithm compute time. A secondary goal included making sure that the parallel BFS scales close to linearly with processors added - we will talk about this in the speedup portion of our paper. Our program makes sure that the resulting flow outputs are consistent across implementations. The tests are run on the GHC machines, which have an 8-core Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and an NVIDIA GeForce RTX 2080 GPU.

Using the per-vertex approach for different graphs

For bigInput1, the per-vertex approach performs the worst when compared to the original sequential algorithm. For a graph like this, the implementation spends a lot of time in the BFS iterating through the vertices but not seeing many edges. It seems like this implementation performs poorly for sparse graphs. In this scenario, the CUDA and the FastBFS (which uses local frontiers and a merge) implementations perform much better.

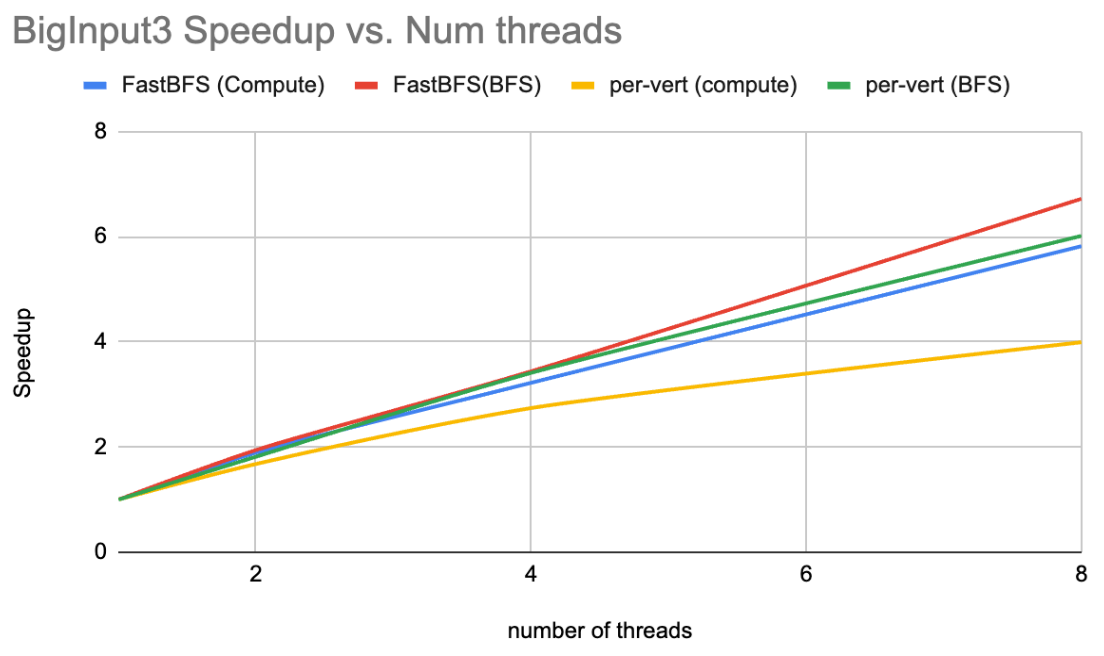
However, in other graphs with higher ratio of edges to vertices, we see the per-vertex algorithm perform better as there is more work to be done for each vertex than before, where it would've done better iterating through the vertices. In these other denser graphs, the FastBFS performs worse than the CUDA and per-vertex approach. Here, the work for doing local frontier and merge starts to be more of a bottleneck as there are more nodes to put onto the frontier than in a sparse graph. A relationship can be loosely seen in these graph results.

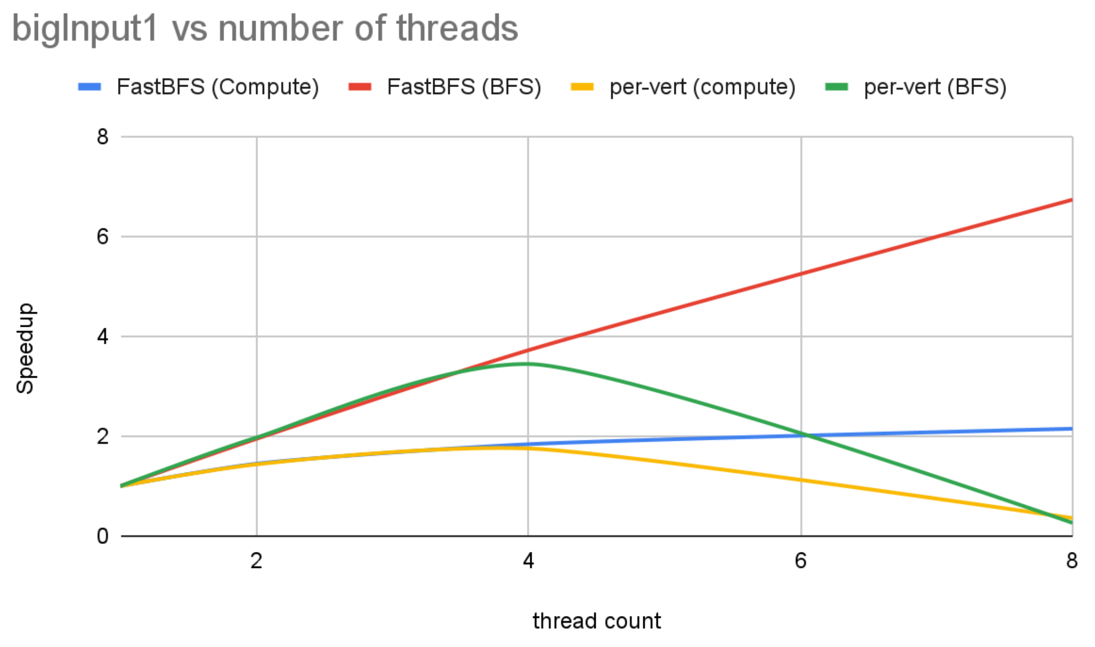
What algorithms perform best for different graph types?

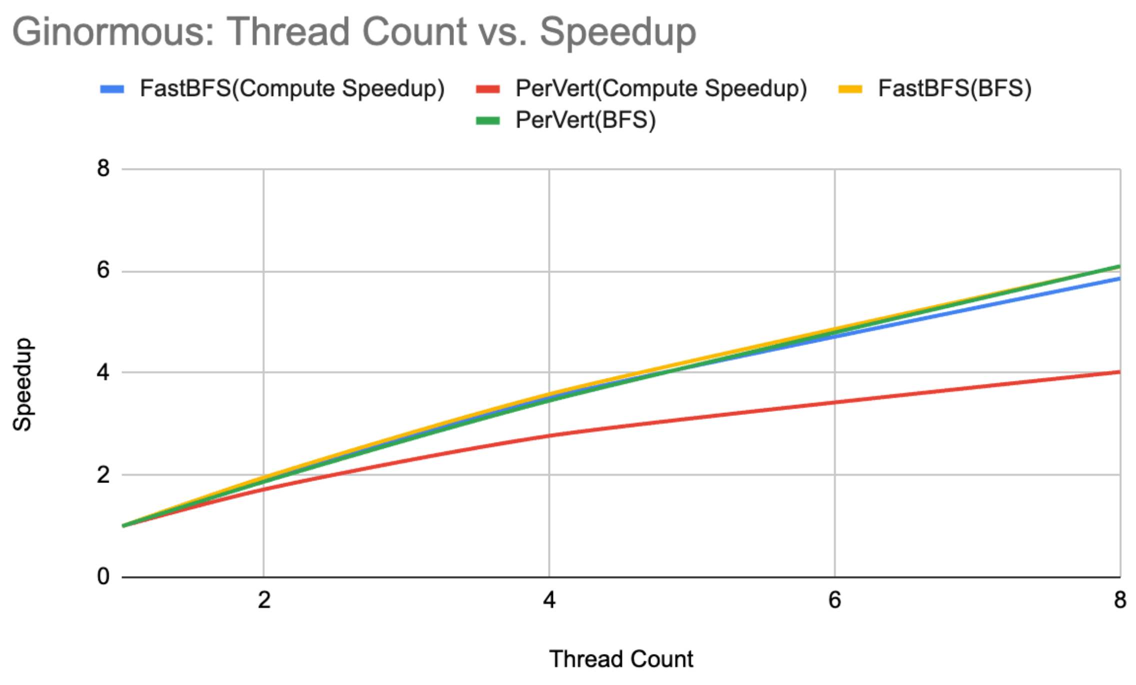
For sparse graphs, a local frontier and a merge performs very well. However, for denser graphs, doing a per-vertex approach obtains better wall timing.

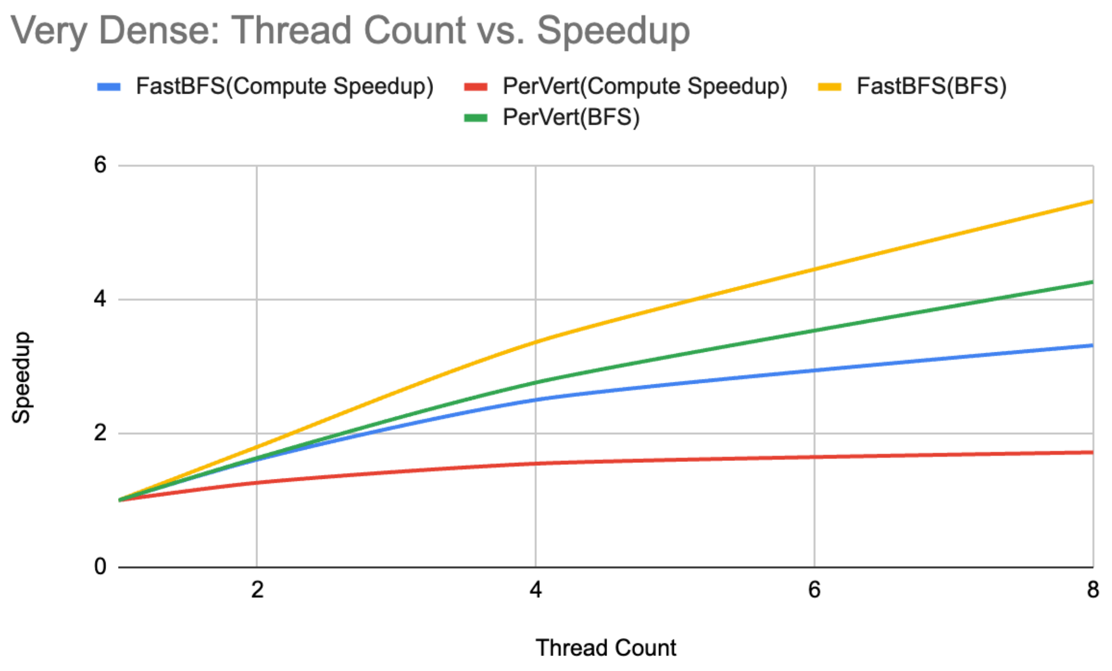
Although the CUDA implementation also does a similar per-vertex idea, it possesses better cache locality when accessing elements. We will discuss cache locality more later in the report.

Speedup graph breakdown









Speedup Analysis

For most of these graphs, the algorithms show a sub-linear trend. The relative speedup appears to grow less with the number of threads for each CPU implementation. For bigInput1, the CPU implementation of the per vertex algorithm falters at 8 threads and becomes worse than the 1 CPU thread implementation. It would appear that 4 threads is the ideal thread count for this implementation and the current implementation suffers from communication overhead.

By Amdahl's law, the total speedup is always worse than the BFS speedup in both implementations because of the sequential portion - the DFS and the sequential graph modification. For dense graphs, the per-vertex approach has the worst speedup with the total computation speedup barely reaching 2x speedup with 8 threads. For thinner ratios, note that the BFS implementations perform similarly in terms of BFS time. Since ginormous and bigInput3 have the same node count and similar edge counts, the overall connectedness of the graph seems to play a role in the computation speedup of the bigInput3 and ginormous graphs. Again, the per-vertex approach plays better when the work is more evenly distributed among the nodes. The local frontier strategy is likely to have worse speedup for graphs that have big frontiers, such as bigInput3 and ginormous.

Cache miss and hit rate

We used perf to analyze performance characteristics for our implementations. When initially running on the sequential implementation, we originally noticed that there was a roughly 50% cache miss rate occurring. A cause for these cache misses is accessing the directed edge between two vertices, which can largely be random. Our graphs are large and sometimes it is not possible to fit all neighbors of a node into a single cache line. Additionally, a perf report showed details that edge accesses, done via a vector and then a map access, accrued around 50% of the cache misses. After trying to parallelize the BFS portion, the miss rate stayed the same. Since the graph has random access into the edge data structure and the frontier is fairly random, it is probably quite difficult to make large graphs much more cache friendly. However, when using perf on the CUDA implementation, we noticed that in some graphs like the very dense graph, had only a 25% cache miss which is half as many as before. The cache miss rate for larger graphs was around 60%, which is on par with the other algorithms. This could be one of the reasons why the CUDA algorithm performed best on the smaller graphs in terms of vertex and edge count - bigInput1 and verydense.

Limitations

A fundamental bottleneck is the pushing of flow through a path in the graph, which involves writes. No other work can realistically be done while this is occurring because there is a writer on the graph data structure, and because the new structure of the graph will change future behavior. Another limitation is the DFS algorithm. The sequential version is already pretty fast due to the optimization of dead edges. DFS algorithms are already likely to make progress in each step to reach the sink. Although DFS is naturally sequential, the graph the BFS algorithm generates is a DAG, and parallel algorithms do exist for this algorithm. However, algorithms really only stand to benefit from DFS when a parallel BFS becomes enough of a bottleneck.

Appendix: Raw Time Data

Very dense

	Original	FastBFS 1	FastBFS 2	FastBFS 4	FastBFS 8	CUDA	Per-vert 1	Per-vert 2	Per-vert 4	Per-vert 8
Total	2.077583	1.071977	0.666003	0.428885	0.323669	0.205395	0.4292	0.339841	0.277348	0.250144
BFS	1.969035	0.915312	0.508651	0.272213	0.167504	0.073206	0.239513	0.146529	0.08683	0.056244
DFS	0.102571	0.148476	0.148596	0.147187	0.14643	0.124442	0.183244	0.186478	0.183848	0.187053
Speedup	0.5159731	1	1.6095678	2.4994509	3.3119545	5.2190998	1	1.2629436	1.5475143	1.7158117
BFS Speedup	0.4648531	1	1.7994892	3.3624845	5.4644188	12.503237	1	1.6345775	2.758413	4.2584631
DFS Speedup	1.4475437	1	0.9991924	1.0087576	1.0139725	1.1931342	1	0.9826575	0.9967147	0.9796368

Ginormous

	Original	FastBFS 1	FastBFS 2	FastBFS 4	FastBFS 8	CUDA	Per-vert 1	Per-vert 2	Per-vert 4	Per-vert 8
Total	26.869484	20.547189	10.572107	5.840379	3.511313	0.734276	1.63534	0.949259	0.590156	0.406659
BFS	26.753141	20.388352	10.412492	5.68181	3.35013	0.3505	1.470639	0.784541	0.424927	0.241317
DFS	0.09772	0.134375	0.134758	0.133467	0.13588	0.36615	0.146422	0.146074	0.146499	0.14661
Speedup	0.7647035	1	1.9435283	3.518126	5.851711	27.982923	1	1.7227543	2.77103	4.0214037
BFS Speedup	0.7620919	1	1.9580665	3.5883551	6.0858391	58.169335	1	1.8745215	3.4609215	6.0942205
DFS Speedup	1.3751023	1	0.9971579	1.0068032	0.9889241	0.3669944	1	1.0023824	0.9994744	0.9987177

Note: speedup in these graphs is calculated with respect to the 1-thread version of the implementation, with CUDA and Original being compared to FastBFS 1

BigInput1

	Original	FastBFS1	FastBFS2	FastBFS4	FastBFS8	per-vert1	per-vert2	per-vert4	per-vert8	CUDA
Total	0.467347	0.99554	0.687795	0.541663	0.463487	0.175472	0.122217	0.100079	0.488506	0.080194
BFS	0.415965	0.620575	0.319051	0.166848	0.092178	0.111361	0.05651	0.03234	0.422092	0.010041
DFS	0.035905	0.197384	0.196337	0.199568	0.198187	0.046114	0.047276	0.049451	0.047816	0.052968

BigInput3

	Original	FastBFS1	FastBFS2	FastBFS4	FastBFS8	per-vert1	per-vert2	per-vert4	per-vert8	CUDA
Total	26.444116	30.180239	15.931629	9.36761	5.186051	1.615869	0.962907	0.589468	0.404734	0.872151
BFS	26.329705	29.373532	15.119924	8.538653	4.370536	1.451875	0.799385	0.425724	0.241352	0.482263
DFS	0.096133	0.616035	0.619857	0.631723	0.624178	0.145436	0.144774	0.145014	0.14456	0.371894

References

Ozua, Benedict, and Matei Budiu. “Bozebro/Dinics-OpenMP: Built with the Help of Mateib.” *GitHub*, github.com/BozeBro/Dinics-OpenMP. Accessed 4 May 2024.

Anderson, Daniel, and David Woodruff. “Network Flow II.” *Design and Analysis of Algorithms*, www.cs.cmu.edu/~15451-s23/lectures/lec12-flow2.pdf. Accessed 4 May 2024.

Springer, Matthias. “Breadth-first Search in CUDA.” Tokyo Institute of Technology. https://m-sp.org/downloads/titech_bfs_cuda.pdf. Accessed 4 May 2024.

CONTRIBUTIONS OF EACH TEAM MEMBER

Work Split

Benedict 50%

Matei 50%

Task Completed	Completed by Member
DINICS: CUDA BFS and associating DFS	Matei
Initial Sequential DINICS algorithm	Matei + Benedict
DINICS: Lock free queue BFS working	Matei + Benedict
DINICS: frontier-less BFS impl	Benedict
DINICS: Localized frontier + merge	Matei
Final Report	Matei + Benedict
Optimizing Sequential DINICS	Matei + Benedict
TestCase Generator	Matei
Perf Performance analysis	Benedict